



Algoritmizálás és a C# programozási nyelv használata

ELŐSZÓ

Ez a jegyzet a Digitális kultúra tantárgy 11. évfolyamos tananyagából az Algoritmizálás és programozási nyelv használata témát dolgozza fel. Tematikájában, legtöbb feladatában és megjelölésében az állami tankönyv¹, valamint annak online kiegészítése képezi az alapját. Sokszor 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.

Először áismételjük a 9-es jegyzetben ([Digi09 Cisz *.pdf²](#)) részletezett témákat, a C# nyelv és a programozás technikájának alapjait. Ezután a 10-es jegyzetben ([Digi10 Cisz *.pdf](#)) tárgyalt eljárások, függvények és elemi típusalgoritmusok következnek, előtérbe helyezve a strukturált programozásban használatos formákat. Az új programozási feladatok előtt a fájlkezelés kerül sorra. Ebből a szövegfájlok kezelésére szükség lesz több feladatban, a bináris fájlok kezelése a tananyagon túlmutat. Az algoritmusok közül erősebb hangsúlyt kapnak a középiskolai szintű feladatokban gyakrabban előfordulók. Egy-egy feladattípushoz a tankönyvhöz képest jóval több algoritmus található a jegyzetben. A sokféle megoldás célja a gondolkodási módok bemutatása, hogy legyen miből kiválasztani a legszimpatikusabbat. A típusalgoritmussal történő megoldások mellett a jegyzet kitér a rekurzív algoritmusokra és megtalálhatók a C# előregyártott függvényei és funkcionális nyelvi elemei is; tanulmányozható a Lambda kifejezések használata és a LINQ is. Az algoritmusok mellett ebben az évben az összetett adattípusok használata is rutinná kell váljon. Az ismétlés során minden megoldás az alapnak tekinthető tömb és struct típusokkal készült, de végig használható a List és a class is. Egyes feladatoknál a tankönyvben preferált szótár használatára (Dictionary) is van példa.

A jegyzet elején jellemző a kódhoz írt szövegbuborék, de ezt később felváltja a kódba írt megjegyzés, illetve a kód tördelése. A sorok számozása ebben a jegyzetben még fontosabb, mint korábban, mert a kódrészleteket a program különböző területeire kell írni. Bár a struct helyett bátran használható a class, az objektumorientált programozás paradigmái messze túlmutatnak a tananyagon. Az OOP és a grafikus felület programozása témákban egy-két feladat részletesebb megoldásán keresztül ismerkedünk a szoftverkészítés problémamegoldási attitűdjével és módszerkészletével.

Az előző két jegyzethez hasonlóan, a bemutatott megoldások példák, a lehetséges jó megoldások száma sokszorosa egy tanulócsoporthoz létszámának.

Sikerekben gazdag tanulást kívánok:

Budapest, 2022.

Szalayné Tahy Zsuzsanna

¹Digitális kultúra 11. tankönyv (https://www.tankonyvkatalogus.hu/pdf/OH-DIG11TA_teljes.pdf) Oktatási Hivatal 2022.

² A jegyzetek (frissített verziók is) itt érhetők el: <https://sztzs.infokatedra.hu/public/prog/>

TARTALOM

Vittük valamire – Kódolás, futtatás, tesztelés, fejlesztés	5
Vittük valamire – Szekvenciák, elágazások és a feltételes ciklus	8
1. példa: Duma (Szekvencia)	8
2. példa: Vélemény (Elágazás: if, else if, else)	8
3. példa: Cica vagy kutya (switch, char).....	9
4. példa: Egér (while).....	10
5. példa: A harmincéves háború (do-while, int)	10
Vittük valamire – Összetett adatok, léptető ciklusok, függvény, eljárás.....	10
Azonos típusú adatok sorozata	11
6. példa: Szököévek I. Ferenc József uralkodásától napjainkig (számláló ciklus)	11
Szöveg (string)	12
7. példa: Szó visszafelé (tömb, iterátor, index, string és char[]).....	12
Tömb (Array): azonos típusú adatok tárhelye	12
8. példa: Javuló(?) eredmények (string[], szövegdarabolás)	12
Lista (List<>).....	13
9. példa: Dolgozatjegyek hiányzókkal (lista, foreach, számjegyek kódja).....	13
Feladat.....	14
10. példa: A három kismalac háza	14
Objektumok tulajdonságai helyett a rekord (struktúra)	14
11. példa: A három kismalac (struct).....	15
Kiegészítés Python-utánozóknak, Pythonról áttérőknak (Dictionary)	16
12. példa: Kedvenceink (List<struct>)	17
Függvény és eljárás	18
Kész kódból eljárás, függvény készítése: Refactoring.....	18
Új függvény, új eljárás írása	19
Változók láthatósága függvényen belül és kívül	20
13. példa: Osztály szintű változók használata	20
14. példa: Függvény paraméterezése, lokális változók átadása	20
Vittük valamire – Típusalgoritmusok.....	22
Sorozatszámítás	23
15. példa: Hónapok napjaiból az év hossza	23
Eldöntés	23
16. példa: Van-e 28 napos hónap az évben?	23
Kiválasztás	25
17. példa: Hányadik az első 30 napos hónap?	26
Keresés.....	26
18. példa: Ha van 28 napos hónap az évben, akkor hányadik hónap ilyen?	26
Kiegészítés Pythonról áttérőknak és C# nyelvészeknek	28
Megszámolás	28
19. példa: Hány 30 napos hónap van az évben?	29
Szélsőérték-kiválasztás: maximum- és a minimumkiválasztás	29
20. példa: Hányadik hónap a legrövidebb?	29
Feladatok	30
Vittük valamire – Típusalgoritmusok kétdimenziós tömbbel és rekordok tömbjével	30
21. példa: Hiányzók	31
22. példa: Hazánk legmagasabb hegycsúcsai	34
23. példa: Kiegészítés: Kutyaoltás (szótárral)	38

Fájlok a kérészeletű adatok helyett.....	39
<i>Elmélkedés:</i> Szöveges és bináris fájlok – Bájtok és bitek értelmezése.....	40
<i>Ismétlés:</i> Szöveges fájlban tárolt adatok felhasználása konzolon keresztül	41
Fájlok hozzáadása a megoldáshoz	42
Fájl beolvasása, adatok tárolása	44
24. példa: Fájlból be, konzolra ki	44
25. példa: Fájlból beolvasott adatok tárolása	44
26. példa: Fájlból beolvasott adatok felhasználása – legöregebb állat	50
Szövegfájlok írása	50
27. példa: Legöregebb állat adatainak fájlba írása	50
Ékezetes betűket tartalmazó szöveg beolvasása és fájlba írása	51
28. példa: Ékezetes állatok olvasása fájlból és kiírása fájlba	51
<i>Kitekintés:</i> Bináris fájlok olvasása és írása	52
Nézzük meg egy bitmap fájl kódolását!	53
Kísérlet.....	55
Írjunk programot bájtok vizsgálatára!	55
29. példa: Fájlból olvasás karakterenként	55
A bináris olvasó	56
30. példa: Szöveges fájl olvasása binárisan	57
Bináris fájl – *.bmp – olvasása, módosítása, írása	58
31. példa: A bmp . bmp olvasása, módosítása és kiírása	58
32. példa: A bmp fejléc értelmezett tárolása és kiírása.....	60
Kópiakészítés és digitális Hamupipőke – Másolás, ki- és szétválogatás típusalgoritmussal ...	61
Másolás	61
33. példa: Másolás hagyományos, strukturált módon	62
34. példa: Kiegészítés: Másolás haladó nyelvi eszközökkel	63
35. példa: Taxis adózott bevételei.....	64
36. példa: Libatömegek farkas előtt és farkas után.....	64
37. példa: A tanya állathangjai	65
Kiválogatás és szétválogatás	66
38. példa: A farkas és a róka libalakomája	67
39. példa: Hőségriadós napok	68
Feladatok a kilenc típusalgoritmusra és a fájlok használatára	69
40. példa: Gyalogtúra	69
41. példa: E terkes mecske leberetette e tejfelt	70
42. példa: Kutya és macskaoltások.....	70
43. példa: Tojásrakók	71
44. példa: Jók és rosszak	71
45. példa: Hajónapló	72
46. példa: Távirat.....	73
Mintamegoldások.....	74
Mindent bele! – Összefüggő feladatsor megoldása.....	82
Feladatok	82
Megoldás praktikus, minimális nyelvi eszközzel.....	82
Megoldás Lambdával és egyéb virtuóz eszközökkel.....	90
Gyakran használt összetett algoritmusok	92
Egyesítés.....	92
Metszetképzés.....	92
47. példa: Tömegközlekedés Százszorszépvárosban	92
Unió	94
48. példa: Fricska és Kökény első szavainak jegyzése.....	94
És a többi halmazművelet	95

Rend a lelünk – A rendezés algoritmusai	96
Mezítlábas rendezés	96
A csere algoritmus	97
Helyben szétválogatás	98
49. példa: Rosszak előre, jók hátra	98
Egyszerű cserés rendezés	100
Szélsőérték-kiválasztásos rendezések	102
Buborék rendezés	105
Beszűrő és Törpe rendezés	106
A Sort(), a Reverse(), az OrderBy() és az OrderByDescending()	108
És a többi	110
Rendezés a gyakorlatban	110
50. példa: A tanyán élő állataink neve és kora	110
Típusalgoritmusok rendezett sorozaton	117
Bináris keresés	118
51. példa: Merre van?	118
52. példa: Felénk járó kíváncsi postás – melyik állatok vannak	119
Összefuttatás	122
53. példa: Fricska és Kökény szótárainak közös része (metszete)	122
54. példa: Fricska és Kökény szótárainak egyesítése (unió)	123
Ó, ió, Rekurzióóó!	124
55. példa: Hello, itt vagyok	125
56. példa: Faktoriális számítás: ciklus vs. rekurzió	126
Feladatok	126
Megoldások	127
Gyorsrendezés	129
57. példa: Tömb gyors rendezése (nincs ismétlődő érték)	130
58. példa: Tömb gyors rendezése (ismétlődő értékek is vannak)	132
Csoportnapló – Tapogatózás az objektumorientált programozás irányába	134
59. példa: Csoportnapló	134
Feladatok	135
Megoldás	136
Adatszerkezetek tervezése és megvalósítása objektumosztályokkal	142
Objektum inicializálása	142
60. példa: A tanuló osztálya a Tanulo osztály	143
Feladat	145
Megoldás	145
Objektumaink működni kezdenek – Függvények az objektumok belsejében	146
61. példa: A macskák fejlődésének nyomon követése	146
Feladatok	150
Sok objektum, mindegyikben sok adat	150
62. példa: Csoportnapló Diak osztályból	151
Kiegészítések a grafikus felhasználói felületen programozás előtt	157
Modulok, több fájlból álló programok	157
Nem class, nem struct, mégis türkíz színű – mi az?	157
63. példa: Mérés és értékelés – (enum és modul)	159
Modul készítése – Csak a mindent tudni akaróknak	162
Grafikus felhasználói felületű alkalmazást fejlesztünk	163
Kiegészítések tárgymutatója	164

VITTÜK VALAMIRE – KÓDOLÁS, FUTTATÁS, TESZTELÉS, FEJLESZTÉS

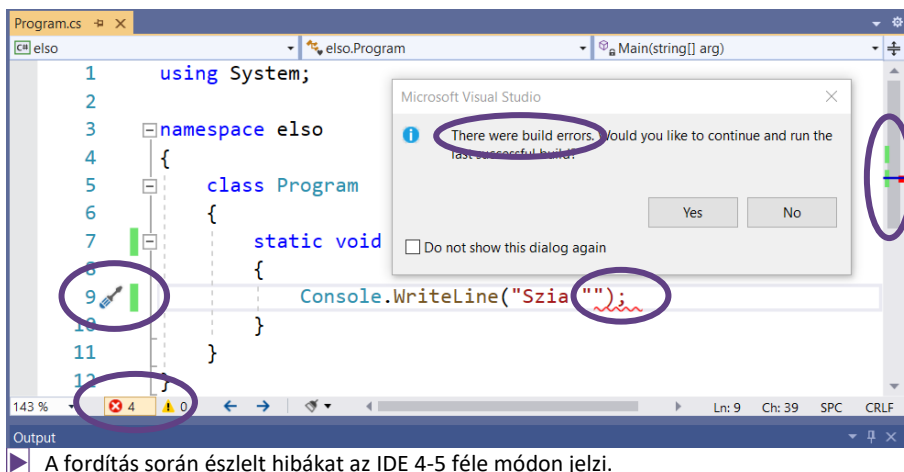
A kilencedik és tizedik évfolyamon jócskán belekóstoltunk a programozásba, és már egészen klassz dolgokra tudjuk rávenni a számítógépet – ebben és a soron következő három leckében először ezeket az ismereteket ismételjük át.

A bennünket körülvevő rengeteg számítógép közül a digitáliskultúra-órákon minket épp azok érdekelnek leginkább, amelyekről látszik, hogy számítógépek: a laptopok és az asztali számítógépek. Mindazonáltal tudjuk, hogy éppúgy programok működtetik a háztartási gépeinkben, járműveinkben, egyéb eszközeinkben lévő számítógépeket is, és az sem titok, hogy mára csak a legegyszerűbb gépeinkben nincs számítógép.

A programokat sokféle nyelven írhatjuk – itt a könyvben történetesen C# nyelven kódoljuk a programjainkat. Legyen azonban szó bármelyik programozási nyelvről, ha elég messziről nézzük őket, igencsak hasonlóak. Az előző mondatnak csak látszólag mond ellent az a megjegyzésünk, hogy az egyes nyelvek bizonyos feladatok elvégzésére alkalmasabbak lehetnek a többinél. Az újabb és újabb programozási nyelvek, illetve környezetek sokszor a visszatérő problémák kész megoldását nyújtják függvények, eljárások formájában.

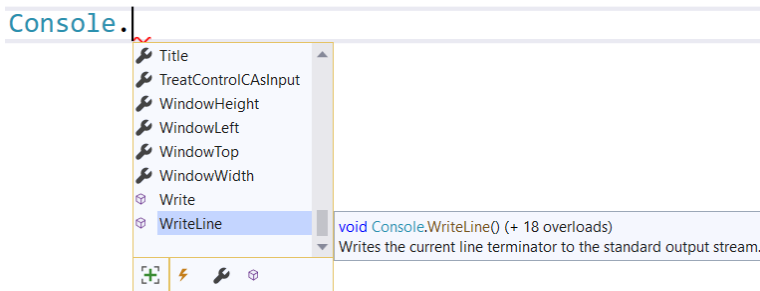
A programok mindig utasításokból állnak, az utasításokat az adott programozási nyelven kódoljuk. A program futtatása előtt a nyelvhez írt fordítóprogram a kódot a processzor számára értelmes utasításokká alakítja. Az interpreter típusú fordítóprogram a kódot értelmezi és azonnal futtatja is. A compiler az általunk írt kódból előállítja a futtatható bináris állományt, amit ezután már a programozási környezettől és kódtól függetlenül futtathatunk. Visual Studio környezetben a C# nyelven írt kódból – a futtatás ikonra kattintáskor vagy az F5 billentyű lenyomásakor – a beépített `cs.exe` compiler készíti el a programot – ha megérti – majd futtatja.

Ha a kódunkban szintaktikai (helyesírási) hiba van, akkor a fordítóprogram megszakítja a fordítást és jelzi, hogy hol ütközött akadályba.



A Visual Studio – egy integrált fejlesztő környezet (integrated development environment, röviden IDE) – az ilyen típusú hibák elkerüléséhez többféle segítséget, támogatást ad. Ezért a programkód írásakor a monitoron folyamatosan ellenőrizzük a beírt kódunkat, figyelünk az IDE jelzéseire.

A leggyakrabban használt segítség a kódkiegészítés. Egy kódrészlet első néhány karakterének a begépelése után az IDE felajánlja az általa ismert kiegészítéseket.



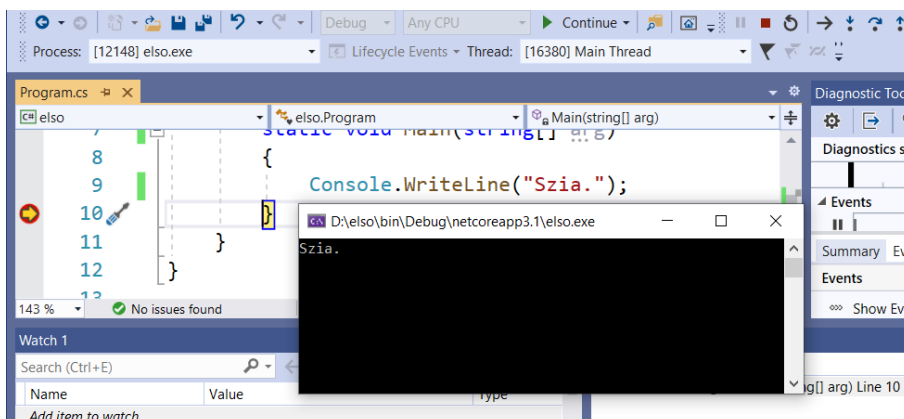
Így nemcsak a helyes gépelést segíti, de a kód megfogalmazása is könnyebb lesz. Az egyre újabb Visual Studio verziókban ez a funkció egyre intelligensebb, a 2022-es verzióban előfordul, hogy nemcsak a megkezdett kifejezést egészíti ki, hanem a megelőző sorok alapján a folytatásra is javaslatot tesz.

Gyakran használt kódrészletek helyes beírását a Visual Studio rövidítéssel, úgynevezett code snippet-tel segíti. Ezeket érdemes megtanulni és használni.

A kiírás kódolása a cw snippettel: `Console.WriteLine()` helyett elegendő beírni: `cw` és kétszer leütni a TAB billentyűt.

Ha még ez sem elég a kód helyes begépeléséhez, akkor a hibát piros hullámos aláhúzással jelzi az IDE. Ahogy a futtatás előtti ellenőrzésnél, ebben az esetben is a jelzett hely fölé vitt egér esetén szövegbuborékban kapunk rövid leírást a hiba természetéről.

Attól, hogy a programunk futtatható, még nem biztos, hogy azt csinálja, amit szeretnénk. Ilyenkor szemantikai hiba van a programban, a kóddal nem a szándékunknak megfelelő utasítást adtuk meg. A programunk többszöri futtatásával teszteljük a munkánkat, a kapott eredményből következtethetünk a félreértés, a félreértelmezés okára. Ha nem látjuk, hogy mi okozza a félreértést, akkor Debug módban, akár lépésenként futtathatjuk a programunkat, lassítva figyelhetjük meg a működését. Egy hosszú program lépésenkénti futtatása nagyon időigényes, de breakpointtal jelezhetjük, hogy melyik kódrészletnél szeretnénk elkezdni a megfigyelést. A kódsor elé kattintva tehetjük ki a piros megállító pontokat, amelyeknél a futtatás során megáll a programunk.



Egy program írása során a szemantikai hiba szándékos is lehet. Összetett feladatok megoldásához részfeladatok megoldásán keresztül vezet az út. A jó programozó egy feladatnak mindig csak egy kicsi, elemi részét oldja meg, a megoldását teszteli, majd a kívánt irányba továbbfejleszti. Egy C# program első állapota jellemzően 10–11 sor. Ezt teszteljük először.

```

1. using System;
2. namespace kiir
3. {
4.     class Program
5.     {
6.         static void Main()
7.         {
8.             Console.WriteLine("Idén is lesz programozás.");
9.         }
10.    }
11. }

```

A nagy programok kódja is – általában több fájlban, de azonos névtérben (`namespace`) található. Más névtérben írt kódokat a `using` utasítással lehet a programhoz csatolni. A `System` névtérben található a `Console`, ezért a konzolra íráshoz erre szükségünk van. Egy program nagyon sok egységből állhat, ezek kódolása történhet egy vagy több fájlba, az egységek sorrendje tetszőleges. A fordítóprogramnak – és a programozónak – tudnia kell, hogy hol van az elsőnek végrehajtandó feladat. Ezért a `Program` nevű osztálynak (`class`) kitüntetett szerepe van. Ezt fogja elsőként értelmezni, ezen belül pedig a `Main()` eljárást keresi meg a fordítóprogram, itt kezdjük a programkód írását.

Ahhoz, hogy a kódunkat a fordítóprogram értelmezni tudja, a kód kezdetének jól felismerhetőnek kell lennie. Ezért a kódunk elég sok kötelező sort tartalmaz. Sok programozási nyelvben – így a C# .Net 6-os verziójában – ezeket a sorokat nem kell kiírni, a hiányos kódot megérti és kiegészíti a fordítóprogram. A jegyzetbéli kódrészleteknél sem szerepelnek a „magától értőddő” sorok.

```

1. Console.WriteLine("Idén is lesz programozás.");

```

A fejlesztés során a már megírt kód módosul, kiegészül. A meglévő kódrészlet elé kerülnek újabb részletek, az egyszerű feltételekből összetett kifejezések lesznek, az egyszerű struktúrákból összetett kódszerkezetek lesznek. A `Main()` elé és mögé más eljárások és függvények kerülnek, a `Program` osztály elé és mögé osztályokat, struktúrákat hozhatunk létre.

A programkód nem mese, amit az elejétől a végéig kell olvasni, inkább egy rejtvényhez hasonlít, amiben fel kell fedezni a belső összefüggéseket. Ebből következik, hogy a programozást nem kész programkódok olvasásával és lemásolásával tanuljuk, hanem programunk fejlesztéséhez keresünk mintát, kódrészletet. Ezek kipróbálása, tesztelése során szerzett tapasztalatok adják programozási ismereteinket. A programkód írása – az olvasáshoz hasonlóan – nem fogalmazásírás, hanem inkább építkezés. A kódrészleteket, a struktúrákat egymás után vagy egymásba (jellemzően nem keresztbe) ágyazva írjuk. A kód-egységeket kapcsolószerűjelekkel `{ }` határoljuk.

VITTÜK VALAMIRE – SZEKVENCIÁK, ELÁGAZÁSOK ÉS A FELTÉTELES CIKLUS

A programok mindig utasításokból állnak. A legtöbb nyelv használatakor – ha másképp nem kérjük – a számítógép ezeket az utasításokat szépen sorra veszi, és egymás után – programozóul úgy mondjuk, hogy **szekvenciálisan** – végrehajtja őket.

1. példa: Duma (Szekvencia)

```
1. Console.WriteLine("Idén is lesz programozás.");
2. Console.WriteLine("Hát nem remek? ;)");
3. Console.ReadKey();
```

Már a legegyszerűbb programok is képesek lehetnek a felhasználóval való kommunikációra. A felhasználó válaszait (is) változóknak tároljuk. A változó létrehozásakor megadjuk, a típusát és a nevét. A program a típustól függően megfelelő memóriaterületet foglal le a változó számára. A változó értéke – a számára fenntartott memóriaterület tartalma – a változó nevének említésével kiolvasható. A változó típusa nem csak a lefoglalt memóriaterület méretéhez szükséges, hanem a tartalom értelmezéséhez is. Ugyanaz a bitsorozat a típustól függően lehet szöveg (karakterek sorozata), egész szám vagy lebegőpontos szám is. A változó létrehozásakor (csak ott) a típus megadásával azt is meghatározzuk, hogy hogyan használjuk az adatot. A sokféle változótypusból használtuk a `char`, az `int`, a `double` és a `bool` egyszerű típusokat, valamint a karakterek sorozatát, a `string` típust.

A felhasználó a billentyűleütésekkel karaktereket küld, ami az ENTER leütésekor egyetlen `string` adatként kerül be a programba. A beírt adat eltárolásához az adatsorozatból a kívánt adattípust elő kell állítani. Számok esetén erre használhatjuk a megfelelő `.Parse()` vagy `Convert.To...` függvényeket, karakter esetén a sorozatok egy elemét kijelölő szelektort: `[]`. Ha egy sorban több adatot adunk meg, akkor a `string`-et a `.Split(' ')` függvénnyel bonthatjuk fel több `string`-re, azaz `string[]` – szöveg tömb – típusú adatra.

2. példa: Vélemény (Elágazás: if, else if, else)

Gyakori, hogy egy változó értékétől függően mást és mást csinál a programunk. Ilyenkor **elágazás** van a programban, aminek csak az egyik ágát szeretnénk végrehajtani.

```
6. static void Main()
7. {
8.     Console.WriteLine("Idén is lesz programozás.");
9.     Console.Write("Örülsz? (i/n) ");
10.    string vélemény = Console.ReadLine();
11.    if (vélemény == "i")
12.    {
13.        Console.WriteLine("Hát én is!");
14.        Console.WriteLine("Jaj de jó!");
15.    }
16.    else if (vélemény == "n")
17.        Console.WriteLine("Hüpp.");
18.    else //Minden más válasz esetén
19.    {
20.        Console.WriteLine("Nem értem. Pedig igazán próbálkoztam.");
21.    }
22.    Console.WriteLine("Pápá.");
23. }
```

Szöveg típusú változót hozunk létre és elhelyezzük benne a felhasználó választ.

NINCS pontosvessző!!!
Egy (1 db) utasításhoz nem kell { }

Ez a két sor akkor fut le, ha a felhasználó „i”-t választ

Ez akkor fut le, ha „n” a válasz...

...ez minden egyéb esetben

Ez mindig lefut, mert az elágazás után van.

A fenti elágazásnak három ága van, de más esetekben lehet bővebb és hiányos is. Az első – az **if** utáni feltételnek megfelelő esetekre végrehajtandó igaz-ág – utasítás vagy blokk kötelező, a többi elhagyható. A tipikus elágazásban az igaz-ág után a példabéli harmadik ág, **else** kulcsszóval kezdődő hamis-ág következik. A közbenső – **else if** – ág a további feltételek szétválasztását teszi lehetővé. Egyes programozási nyelvekben kissé eltérő formában jelenhet meg, folyamatábrán nem használható. Ha létezik, akkor tetszőleges számú ág létrehozható a használatával. Ha a nyelvben ez a kifejezés nem szerepel, akkor a hamis ágon belül lehet újabb egy- vagy kétágú elágazásokkal helyettesíteni.

Ha az elágazás ágait egy változó felsorolható értékeitől függően kell kiválasztani, akkor használhatunk **switch**-et. Ez nem csak áttekinthetőbb, de módot ad a program folytatásának belső átadására is. Ha elágazásként használjuk, akkor minden ág végén egy **break** jelzi, hogy az ág végrehajtása után a kapcsoló utáni utasítással kell folytatni.

A kódolást segítik a snippetek:

if, else, switch, while vagy do után két TAB kiegészíti a kódot.

3. példa: Cica vagy kutya (switch, char)

Írjuk meg azt a programot cicakutya néven, amelyik megkérdi a felhasználót, hogy a program cica vagy kutya legyen-e! Ha a felhasználó cicát szeretne, akkor a program kérjen tejet, és írja ki, hogy „Nyaú!”. Kutya esetén persze csont lesz szükség, a megnyilvánulás pedig „Vaú!”.

```

6. static void Main()
7. {
8.     Console.WriteLine("Cica vagy kutya legyek? (c/k): ");
9.     char állat = Console.ReadLine()[0];
10.    switch (állat)
11.    {
12.        case 'c':
13.            Console.WriteLine("Tejet, ha lehet.");
14.            Console.WriteLine("Nyaú".PadRight(8, 'ú') + "!");
15.            break;
16.        case 'k':
17.            Console.WriteLine("Egyet mondok, adjál csontot!");
18.            Console.WriteLine("Vaú!");
19.            break;
20.        default:
21.            break;
22.    }
23. }
```

A beírt szöveg első karaktere.

Egész vagy karakter típusú változó

'c' esetén

8 karakter hosszúra 'ú'-val kiegészítve

'k' esetén

minden egyéb esetben

Ide ugrik a break-nél. (Máshova is ugratható.)

Ha valamilyen ismétlődő feladatot szeretnénk végeztetni a számítógéppel, ciklust szervezünk. Az első ciklusunk az előtesztelő feltételes ciklus vagy **while**-ciklus. Ebbe a ciklusba akkor lépünk be, ha a belépés feltétele igaz, és addig maradunk benne, amíg ez a feltétel teljesül.

4. példa: Egér (while)

```
8. Console.Write("Üdv én egér vagyok. Cincogjak neked? (i/n) ")
9. string válasz = Console.ReadLine();
10. while (válasz == "i")
11. {
12.     Console.Write("Cin-cin \tMég? (i/n)");
13.     válasz = Console.ReadLine();
14. }
15. Console.WriteLine("Szia.");
```

Ha a felhasználó „i”-t válaszol, belépünk a ciklusba

Ha nem „i”-t válaszolunk, akkor a 14. után a 10. sorban nem teljesül a feltétel, onnan a 15. sorra ugrik.

5. példa: A harmincéves háború (do-while, int)

Írjunk programot, amely addig kérdegeti, hogy hány évig tartott a harmincéves háború, amíg a felhasználó ki nem találja! Ne felejtünk megfelelő típusúra alakítani a felhasználó választát! Segítségül itt az algoritmus mondatszerű leírása:

```
program haboru30
ciklus
    be: válasz
    amíg válasz <> 30
    ciklus vége
    ki: dicséret
program vége
```

Amikor már működik a programunk, itt az ideje továbbfejleszteni. Ha a felhasználónk tippje hibás, írjuk ki, hogy kisebb vagy nagyobb-e a megfelelő érték! Ha pedig harmadjára sem találta el, megsúghatjuk neki, hogy mettől meddig tartott a szóban forgó háború.

```
8. int válasz;
9. int próba = 0;
10. do
11. {
12.     if (próba >= 3)
13.         Console.WriteLine("Súgok: 1618-tól 1648-ig tartott.");
14.     Console.Write("Hány évig tartott a harmincéves háború? ");
15.     válasz = int.Parse(Console.ReadLine());
16.     próba += 1;
17.     if (válasz > 30)
18.         Console.WriteLine("Rövidebb volt.");
19.     else if (válasz < 30)
20.         Console.WriteLine("Hosszabb volt.");
21. } while (válasz != 30);
22. Console.WriteLine("Ügyes! Nem gondoltam volna...");
```

VITTÜK VALAMIRE – ÖSSZETETT ADATOK, LÉPTETŐ CIKLUSOK, FÜGGVÉNY, ELJÁRÁS

Az előző leckében kétféle feltételes ciklussal foglalkoztunk, itt másik két ciklustípus kerül sorra: az általánosított számlálós-ciklus, azaz a for-ciklus és a listákat bejáró foreach-ciklus. Mindkét esetben a ciklusnak adnunk kell egy azonos típusú elemeket tartalmazó adatsorozatot (vagy objektumok sorozatát), amelynek az elemein végiglépdelhet, azaz, amit bejárhat.

A kódolást segítik a snippetek:

for vagy foreach után két TAB beírja a kód vázlatát és mutatja, hogy hol kell kiegészíteni

Azonos típusú adatok sorozata

Az első összetett adattípus, amivel megismerkedtünk, az a szöveget tároló `string`. Ebben karakterek sorozatát tároljuk. Ezt követte a tömb típus, aminek a létrehozáskor meg kell adnunk a méretét is. A tömb egy olyan tároló, amelyen belül bárhova tehetünk adatot, de nem bővíthető. Az egyes adatokat a `[]` szelektorban megadott sorszámmal – tömbön belül elfoglalt helyükkel – adhatjuk meg. Az első sorszám a 0.

```
típusnév[] változónév = new típusnév[tárolható_adatok_száma];
változónév[0] = adat;
```

Van, aki jobban szereti a méretbéli szabadságot. A lista bővíthető adatsorozat. Ennek használatához szükségünk van az adattípust leíró `System.Collections.Generic` névtérre. Egy lista létrehozásakor csak azt adjuk meg, hogy milyen típusú adatokat fogunk majd beletenni. Később az adatokat a lista végéhez tudjuk hozzáadni (hozzáfűzni). A lista egyik – változó – tulajdonsága a tárolt adatok száma.

```
using System.Collections.Generic;
...
List<típusnév> változónév = new List<típusnév>();
változónév.Add(adat);
```

Természetesen adódik, hogy egy tömb vagy lista szöveg típusú adatokat tároljon. Egy karakter elérése `változónév[adatsorszám][karakterorszám]` formában lehetséges. Ez a karakterekre nézve kétdimenziós adatszerkezet jelent. Ízléstől – és a megoldandó feladat természetétől – függ, hogy hogyan kombináljuk a tömböt és a listát egymással, erre a 10-es jegyzetben láthatók példák. Fontos szerepe van azonban a táblázatnak, a kétdimenziós tömbnek. (Lehet több dimenziós is, de akkor táblázat helyett inkább mátrix a köznapi neve.)

```
típusnév[,] változónév = new típusnév[sorok_száma,oszlopok_száma];
változónév[0,0] = adat;
```

A tömb, a lista, és az ezekből képzett többdimenziós változatok közös jellemzője, hogy azonos típusú adatokat tartalmaznak, ezek az adatok egymásután felsorolhatók, sorra vehetők. Ezt használjuk ki az indexeléskor, amivel sorszámot rendelünk az adathoz. A tömb és a lista közös jellemzője, hogy bármelyik adatuk bármikor elérhető és módosítható az indexen keresztül. A `string` karaktersorozatában bármely karakter elérhető, de nem módosítható.

6. példa: Szököévek I. Ferenc József uralkodásától napjainkig (számlálós ciklus)

A számsoron, a számok egy intervallumán (esetleg a karakterek kódértékein) a for-ciklus hagyományos, számlálós formájával tudunk végigmenni – és a számokkal egyenként műveletet végezni. Ilyenkor a ciklusváltozó értéke a kezdőértékről indul, a feladatok elvégzése után a megadott lépésközzel módosul az értéke addig, amíg el nem éri a végső értéket (amíg igaz a ciklusváltozó értékére megadott kifejezés).

Írjunk programot, amely megadja I. Ferenc József trónra lépésétől az idei évig tartó időszak szököéveit!

```

8. int ideiév = 2022;
9. for (int év = 1848; év <= ideiév; év++)
10. {
11.     if (év % 4 == 0 && (év % 100 != 0 || év % 400 == 0))
12.         Console.Write(ev + ". ");
13. }

```

Szöveg (string)

A leggyakrabban használt összetett adattípus a **string**, ami speciális karaktersorozat. Az egyes karakterek kiolvashatók, de – C# nyelvben – nem módosíthatók. Specialitása, hogy számos függvényt használhatunk a szöveg (karaktersorozat) átalakításra, két szöveg karakterenként összehasonlítható az '==' és a '!=' relációkkal, továbbá a '+' jellel összefűzhetők.

Fontos megjegyezni, hogy az ábécé szerinti összehasonlításához a '<' és '>' relációs jelek nem használhatók, mert különböző nemzetek ábécéiben a sorrend különböző. Ezt a **CompareTo()** függvény tudja figyelembe venni. Az **astring.CompareTo(bstring)** értéke 1, ha **astring** az ábécé szerinti rendezés esetén hátrébb szerepel mint **bstring**. Fordított esetben az érték -1, azonosság esetén 0.

7. példa: Szó visszafelé (tömb, iterátor, index, string és char[])

Kérjünk be egy szót, állítsuk elő a karaktereit fordított sorrendben tartalmazó szót!

```

8. Console.Write("Írd be a megfordítandó szót: ");
9. string szo = Console.ReadLine();
10. string vissza = "";
11. for (int i = 0; i < szo.Length; i++)
12.     vissza = szo[i] + vissza;
13. Console.WriteLine("A szó megfordítva: " + vissza);

```

Számláló (iterátor) kezdőértékkel

Indexével kiválasztott karakter

Ugyanebből a szóból állítsuk elő megfordított sorrendben a karaktereinek a tömbjét!

```

14. char[] betuk = new char[szo.Length];
15. for (int v = szo.Length - 1; v >= 0; v--)
16. {
17.     int e = szo.Length - 1 - v;
18.     betuk[e] = szo[v];
19. }
20. for (int i = 0; i < betuk.Length; i++)
21.     Console.Write(betuk[i]);

```

Karakterek tömbjének létrehozása

Visszafelé számlálás

{ }, ha a ciklusmagban több utasítás van.

Tömb (Array): azonos típusú adatok tárhelye

8. példa: Javuló(?) eredmények (string[], szövegdarabolás)

Tároljuk el egy vizsgára felkészülés próbadolgozatainak pontszámait! A pontszámokat egy sorban, szóközzel elválasztva kapjuk meg. Írjunk ki a képernyőre '+' jelet, ha javult, '-' jelet, ha stagnált vagy rosszabb lett az előző dolgozathoz képest az eredmény!

Elemezzük az alábbi – szintaktikailag helyes – kódot, amely háromféle megoldást ad a problémára!

```

6. static void Main()
7. {
8.     string adatsor = "9 19 57 73 100";
9.     string[] adatok = adatsor.Split(' ');

```

```

10. /*karakterek összehasonlítása*/
11. for (int i = 1; i < adatok.Length; i++)
12.     if (adatok[i][0] > adatok[i - 1][0]) Console.Write('+');
13.     else Console.Write('-');
14.     Console.WriteLine();
15. /*int tömbbe másolás, számok összehasonlítása*/
16. int[] pontszámok = new int[adatok.Length];
17. for (int i = 0; i < adatok.Length; i++)
18.     pontszámok[i] = int.Parse(adatok[i]);
19. for (int i = 1; i < pontszámok.Length; i++)
20.     if (pontszámok[i] > pontszámok[i - 1]) Console.Write('+');
21.     else Console.Write('-');
22.     Console.WriteLine();
23. /*szövegek összehasonlítása*/
24. for (int i = 1; i < adatok.Length; i++)
25.     if (adatok[i].CompareTo(adatok[i - 1])>0) Console.Write('+');
26.     else Console.Write('-');
27.     Console.WriteLine();
28. }

```

Lista (List<>)

9. példa: Dolgozatjegyek hiányzókkal (lista, foreach, számjegyek kódja)

Egy csoport dolgozatjegyeit szeretnénk tárolni. Az eredményeket (névsor szerint) elválasztás nélkül, számjegyek sorozataként kapjuk, a hiányzó jegye helyett 'H', 'h' vagy kötőjel szerepel. Például: „5543521H3545h555-4H5”. Készítsük el a dolgozatjegyek listáját!

```

1. using System;
2. using System.Collections.Generic;
3. namespace jegyek
4. {
5.     class Program
6.     {
7.         static void Main()
8.         {
9.             string dolgozat = "5543521H3545h555-4H5";
10.            Console.WriteLine("Csoportlétszám: {0} fő", dolgozat.Length);
11.            List<int> jegyek = new List<int>();
12.            foreach (char c in dolgozat)
13.            {
14.                if (c >= '1' && c <= '5')
15.                {
16.                    int jegy = c - '0';
17.                    jegyek.Add(jegy);
18.                }
19.            }
20.            foreach (int jegy in jegyek)
21.                Console.Write(jegy + " ");
22.            Console.WriteLine();
23.            Console.WriteLine("A dolgozatot megírta: {0} fő", jegyek.Count);
24.        }
25.    }
26. }

```

Feladat

Keressük meg az interneten – illetve a korábbi jegyzetekben, hogy mi közös a lista, a tömb és a szöveg típusok kezelésében! Nevezzünk meg néhány eltérést is! Keressünk a listához hasonló, de másképp használható adattárolókat (a Collections további eszközeit), derítsük ki, hogyan használhatók!

10. példa: A három kismalac háza

Nézzük azt a kétdimenziós tömböt, amely a három kismalac nevét és házának anyagát tárolja.

```
8. string[,] malacok = new string[3, 2] {
9.     { "Töfi", "szalma" },
10.    { "Röfi", "fa" },
11.    { "Döfi", "kő" } };
```

Írjuk ki egy-egy sorba az egyes malacok nevét és a házuk anyagát! Amelyik malacnak „kő”-ből készül a háza, amellé írjuk oda azt is, hogy a farkas nem tudja bántani!

```
12. for (int m = 0; m < 3; m++)
13. {
14.     for (int i = 0; i < 2; i++)
15.     {
16.         Console.Write(malacok[m, i] + " ");
17.         if (i == 1 && malacok[m, i] == "kő")
18.             Console.Write("A farkas nem tudja bántani {0}t.", malacok[m, 0]);
19.     }
```

Megoldható a feladat listába tett listával és foreach-ciklus használatával is! A listákat egyenként, konstruktorral kell létrehozni (Ez a méretbéli szabadság ára.)

```
8. List<List<string>> malacok = new List<List<string>>() {
9.     new List<string>() { "Töfi", "szalma" },
10.    new List<string>() { "Röfi", "fa" },
11.    new List<string>() { "Döfi", "kő" } };
```

Így a külső ciklusban egy-egy listát vizsgálunk, a belső ciklusban ennek az adatait. Az adat sajnos nem ismeri a környezetét, ezért a „kő” adatból nem tudunk hivatkozást a malac nevére.

```
12. foreach (List<string> malac in malacok)
13. {
14.     foreach (string adat in malac)
15.     {
16.         Console.Write(adat + " ");
17.         if (adat == "kő")
18.             Console.Write("A farkas nem tudja bántani {0}t.", malac[0]);
19.     }
```

Az „adatot megelőző adat” helyett a listában szereplő indexével adjuk meg a nevet.

A lista használata lehetővé teszi, hogy újabb kőházas malacokat vegyünk fel. Egészítsük ki ezzel a programunkat, kérjük be Turi és Coci adatait a felhasználótól, és teszteljük a programunk működését!

Objektumok tulajdonságai helyett a rekord (struktúra)

Bár malacelemző programunk remekül működik, egy idő után nehéz lehet követni, hogy mit is jelent a `malac[0]` és a hasonló jelölések. Olyan összetett adattípus kell, amiben az egyes dolgoknak (idegen szóval entitásoknak) különféle adatait, tulajdonságait tárolhatjuk. Ilyen célt

szolgál az adatbázis-kezelésben a rekord, a Python nyelvben a szótár (dictionary), a C-típusú nyelvekben használható a struktúra (**struct**). Ezeknél haladóbb nyelvi elem az objektum orientált programozás alapvető adateleíró (pontosabban objektumot leíró) eszköze az osztály (**class**), amiről kiegészítő ismeretként volt már szó és a **struct** helyett minden esetben használható.

Az eddig tárgyalt összetett adattípusok esetében egy névvel hivatkoztunk több azonos típusú adatra, az összetett adattípus elemeire. A rekord, a struktúra és az osztály másképp összetett: nem felsorolt elemei, hanem részei, összetevői, tulajdonságai vannak. Egy ilyen összetétel egy komponensének kiválasztása – szelekciója – a ponttal (.) történik. Malacelemző programunkban a malac első adata `malac[0]`, ehelyett sorrendiségtől függetlenül, `malac.név` – a malacnak a neve – formában hivatkozhatunk rá.

11. példa: A három kismalac (struct)

A C# **struct** eszköze nem adattípus, hanem adattípus létrehozására szolgál. Készítsünk Malac néven összetett adattípust, amelyben tárolható egy kismalac neve és házának anyaga, ezt követően ilyen Malac típusú adatokból hozzunk létre tömböt!

Az első megoldásunkban a **struct** definíció csak a legfontosabb nyelvi részeket tartalmazza. Ilyenkor az értékadás hosszabb:

```

8. struct Malac
9. {
10.     public string Név;
11.     public string Házanyag;
12. }
13.
14. static void Main()
15. {
16.     Malac[] malacok = new Malac[3];
17.     malacok[0].Név = "Töfi";
18.     malacok[0].Házanyag = "szalma";
19.     malacok[1].Név = "Röfi"; malacok[1].Házanyag = "fa";
20.     malacok[2] = new Malac() { Név = "Döfi", Házanyag = "kő" };
21. }
```

A `malac[0]` két adata egymás alatt szerepel, de ettől csak vizuálisan különbözik `malac[1]` adatainak megadása. A `malac[2]` adatainak beírásakor viszont más módszer látható: a default konstruktorral – azaz adatok nélkül – új malacot teszünk a harmadik elem helyére, de azonnal felül is írjuk az adatokat: kapcsos-zárójelben megadjuk, hogy mi a megfelelő szöveg.

A struktúra használatát megkönnyíti, ha a struktúrán belül megírjuk, hogy hogyan kapjanak értéket az adattagok. Ez a leírás a feladatra szabott konstruktor, amit a **new** kulcsszó után írva használunk. Ekkor a leírásunknak megfelelően, a megadott paramétereket a konstruktor átadja az egyes adatoknak.

```

8. struct Malac
9. {
10.     public string Név;
11.     public string Házanyag;
12.     public Malac(string név, string házanyag)
13.     {
14.         Név = név;
15.         Házanyag = házanyag;
16.     }
17. }
18.
19. static void Main()
20. {
21.     Malac[] malacok = new Malac[3] {
22.         new Malac("Töfi", "szalma"),
23.         new Malac("Röfi", "fa"),
24.         new Malac("Döfi", "kő") };
25.
26.     for (int ez = 0; ez < 3; ez++)
27.     {
28.         Console.WriteLine($"{malacok[ez].Név} {malacok[ez].Házanyag} ");
29.         if (malacok[ez].Házanyag == "kő")
30.             Console.WriteLine($"A farkas nem tudja bántani {malacok[ez].Név}t");
31.         Console.WriteLine();
32.     }

```

Program osztályon belül, a Main előtt!

Adattagok (publikusak)

Konstruktor: innentől megadjuk, hogy a bemeneti paraméterekből hogyan lesz adattag

Tömb konstruktor

Malac konstruktorok

PONT

A `malacok[ez]` egy malac, ha listaelem lenne, akkor a `foreach` ciklusban `malac.Név` szerepelhetne a `malacok[ez].Név` helyett.

Kiegészítés Python-utánozóknak, Pythonról áttérőknek (Dictionary)

A kétféle – adatsorozat és adattagok – összetétel között van a Python szótár adattípusa, a `malac['név']` formával. Hasonló megoldás más programozási nyelven is létezik, de a tulajdonságok felsorolhatósága általában felesleges, ezért a megoldások nem egyszerűek. Elrettentésként íme a C# `Dictionary`-val történő megoldás:

```

8. List<Dictionary<string,string>> dicmalac =
   new List<Dictionary<string,string>>() {
9.     new Dictionary<string,string>() {{"név", "Töfi"}, {"ház", "szalma"}},
10.    new Dictionary<string,string>() {{"név", "Röfi"}, {"ház", "fa"}},
11.    new Dictionary<string,string>() {{"név", "Döfi"}, {"ház", "kő"}}
12. };
13. foreach (Dictionary<string,string> malac in dicmalac)
14. {
15.     Console.WriteLine($"{malac["név"]} házának anyaga: {malac["ház"]}");
16.     if (malac["ház"] == "kő")
17.         Console.WriteLine($"A farkas nem tudja bántani {malac["név"]}t.");
18.     Console.WriteLine();
19. }

```

A `Dictionary` szintén bejárható objektum. Bejárható úgy, hogy a kulcsait kapjuk meg (az előző példában „név” és „ház”), de bejárható érték szerint vagy épp mindkettőt elkérve. Az egyetlen megszorítás a Python szótárához képest, hogy a Key és a Value típusa kötött, ha az egyik Value érték string, akkor a többi sem lehet más. Lássuk erre a tizenegyedikes tankönyv példájának C# nyelvű tükörfordítását:


```

8. Dictionary<string, string> ember = new Dictionary<string, string>();
9. ember.Add("név", "Debora");
10. ember.Add("magasság", "167");
11. ember.Add("IQ", "131");
12. foreach (string kulcs in ember.Keys)
13.     Console.WriteLine(kulcs + " értéke: " + ember[kulcs] + "; ");
14. Console.WriteLine();
15. foreach (string ertekek in ember.Values)
16.     Console.WriteLine(ertekek + " ");
17. Console.WriteLine();
18. foreach (KeyValuePair<string, string> adat in ember)
19.     Console.WriteLine(adat.Key + " értéke: " + adat.Value);

```

A kulcsok egyediek – nem lehet két „név”. Az értékek lehetnek egyenlők (131 helyett 167)

A `Dictionary` valódi felhasználási területe a szópárok tárolása (például egy alkalmazás magyarítása – lokalizációja – esetén), illetve, kihasználva, hogy nem lehet két azonos kulcs, a `Value` tárolhatja a `Key` előfordulásainak a számát (például előforduló keresztnévek és számosságuk). Ismét hangsúlyozzuk, hogy a `Dictionary` nem tananyag, egyszerűbb eszközökkel helyettesíthető.

12. példa: Kedvenceink (List<struct>)

Írjunk egy programot kedvencek néven, amely egy-egy rekordban tárolja három kedvenc vloggerünket, előadónkat, tanárunkat vagy sakknagymesterünket! Egy adatnak három tulajdonsága legyen:

- a tárolt ember neve,
- legjelentősebb teljesítménye a szakmájában és
- az említett teljesítmény évszáma.

A három kedvencünk adatait helyezzük el egy listában (lehet tömbben is), majd írjuk ki, amit az egyes emberekről tudunk!

```

9. struct Ember
10. {
11.     public string Név;
12.     public string Teljesítmény;
13.     public int Év;
14.     public Ember(string név, string teljesítmény, int év)
15.     {
16.         Név = név;
17.         Teljesítmény = teljesítmény;
18.         Év = év;
19.     }
20. }
21. static void Main()
22. {
23.     List<Ember> nagymesterek = new List<Ember>(){
24.         new Ember("Polgár Judit", "Garri Kaszparov legyőzése", 2002),
25.         new Ember("Kempelen törökje", "Napóleon legyőzése sakkban", 1809),
26.         new Ember("Beth Harmon", "TV-sorozattá lett az \\'élete\\'", 2020) };
27.     foreach (Ember ember in nagymesterek)
28.         Console.WriteLine($"{ember.Név}: {ember.Teljesítmény} ({ember.Év});");
29. }

```

Függvény és eljárás

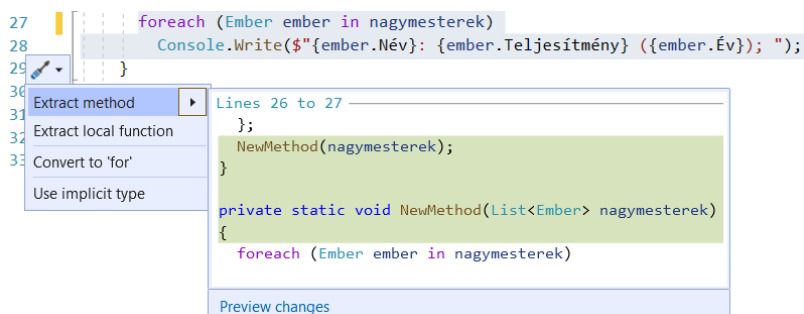
Ha a programunk szépen működik, akkor úgy bővítjük, hogy a felhasználónak legyen lehetősége új embereket felvenni a listába. A listát az új emberek felvétele előtt és után is ki akarjuk írni. Ha egy programban több helyen csinálnánk ugyanazt, akkor eljárást vagy függvényt készítenénk a feladat elvégzésére. A kettő között az a különbség, hogy a függvénynek van visszatérési értéke, az eljárásnak nincs. Sok nyelv, így a C# sem különbözteti meg az eljárást a függvénytől: az eljárás olyan függvény, amelynek a visszatérési értéke `void` (azaz üres).

Az adatok kiírására többször lehet szükség, ezért alakítsuk át a mintaprogram 27–28. sorát!

Kész kódból eljárás, függvény készítése: Refactoring

Megírt kódrészletből a Visual Studioban két kattintással és két ENTER leütésével készíthetünk eljárást vagy függvényt. Ehhez...

- jelöljük ki a kódrészletet és helyi menüből (1. – jobb – klikk)
- válasszuk ki a „Quick Actions and Refactoring...” lehetőséget (1. – bal – klikk),



- ENTERrel kiválasztjuk az „Extract method” lehetőséget,
- ENTERrel elfogadjuk a felajánlott módosítást.

Ezután értelemszerűen átnevezzük a még kijelölt nevet és ellenőrizzük – ha nem ismerte fel elég pontosan, akkor módosítjuk – a visszaadott érték és a paraméterek típusát.

```
27. Nagymestereket_listáz(nagymesterek);
```

Az eljárást a Main függvény után találjuk, de a sorrend igény szerint módosítható.

```
30. private static void Nagymestereket_listáz(List<Ember> nagymesterek)
31. {
32.     foreach (Ember ember in nagymesterek)
33.         Console.WriteLine($"{ember.Név}: {ember.Teljesítmény} ({ember.Év}); ");
34. }
```

Kódszerkesztés Refactoring-gal

Eljárás vagy függvény írásához, kiemeléséhez jobb klikk után „Quick Actions and Refactorings...”

A `Program` osztályon belül a `static` jelző szükséges, a `privat` jelző viszont elhagyható, mert jelzés nélkül is csak osztályon belül használható. A `public` jelzővel ellátott függvényt tagfüggvénynek hívjuk. Például több típusnak a `.Parse()`, a `string` típusú változóknak a `.Split()` a tagfüggvénye.

Új függvény, új eljárás írása

A módosítást követően tesztelünk... Ha minden szépen működik, akkor átgondoljuk, hogy egy új ember megadásához mit kell tenni:

- három adatot be kell kérni,
- az évszámot számmá kell alakítani,
- az adatokból egy `Ember`-t készíteni és
- az `Ember`-t a lista végére fűzni.

Az első három feladat megoldását egy függvénybe szervezzük. A függvényt csak egyszer fogjuk lefuttatni, de olvashatóbb lesz így a kód, jobban elkülönül, hogy a program melyik része mit csinál. A függvényünknek most nincs paramétere, de van visszatérési értéke – hiszen ettől függvény –, mégpedig az új `Ember`.

Ha nem vagyunk biztosak a szintaktikában, akkor kezdjük a `Main()` eljárás módosításával:

```
28. Ember új_ember = Adatokat_bekér();
29. nagymesterek.Add(új_ember);
30. Console.WriteLine("A nagymesterek listája ilyen lett:");
31. Nagymestereket_listáz(nagymesterek);
```

A hiányzó `Adatokat_bekér()` függvényre a „Quick Actions and Refactorings...” most a „Generate Method ’Program.Adatokat_bekér’” javaslatot teszi, amit nyugodtan elfogadhatunk. Az eredmény egy függvény váz, benne egyetlen sorral: `throw new NotImplementedException();`. Ha így futtatjuk a programunkat, akkor adatbekérés helyett egy figyelmeztetést kapunk, hogy a programnak ez a része még nincs megírva. Töröljük ezt a sort és írjuk meg a helyére az adatok bekérését.

```
36. private static Ember Adatokat_bekér()
37. {
38.     Console.Write("Mi a neve? ");
39.     string név = Console.ReadLine();
40.     Console.Write("Mi a legjelentősebb teljesítménye? ");
41.     string kunszt = Console.ReadLine();
42.     Console.Write("Melyik évben érte ezt el? ");
43.     int év = int.Parse(Console.ReadLine());
44.     return new Ember(név, kunszt, év);
45. }
```

Már csak annyi dolgunk maradt, hogy egy ciklussal újra meg újra megkérdezzük, hogy akar-e még a felhasználónk újabb embert megadni. Az utolsó kiegészítés:

```
28. string bővítjük;
29. do
30. {
31.     Console.Write("Akarsz új nagymestert megadni? (i/n) ");
32.     bővítjük = Console.ReadLine();
33.     if (bővítjük != "n")
34.         nagymesterek.Add(Adatokat_bekér());
35. } while (bővítjük != "n");
36. Console.WriteLine("A nagymesterek listája ilyen lett:");
37. Nagymestereket_listáz(nagymesterek);
```

Egy kicsit szépíthetünk is a megoldáson: Az eredeti 28. sorban létrehozunk egy `Ember`-t, amit csak a 29. sorban használunk. Ehelyett kód és helytakarékosabb az új megoldás 34. sora, az `Add()` függvény paramétere helyére írjuk be a függvényünket: `Add(Adatokat_bekér())`

Változók láthatósága függvényen belül és kívül

Az első szabály, hogy egy változó azon a blokken belül látható – és használható – amin belül megadtuk a típusát és nevét. A blokk határai mindig kapcsos-zárójelek.

Ha egy eljáráson, függvényen belül létrehozunk egy változót, akkor ez csak a függvényen belül használható. Kivétel: a függvények `return` utasítással megadott visszatérési értéke. Ez az utasítás egyben a függvény futásának befejezését is jelenti.

Eljáráson, függvényen kívül – de jelen tudásunkkal a Program osztályon belül – elnevezett változót minden osztályon belül lévő eljárásban és függvényben tudunk használni. Ezeket a változókat statikusan – `static` jelzővel kezdve – kell deklarálni (azaz megadni típusát és nevét). Amennyiben nem adunk azonnal kezdőértéket, vagy nem hozzuk létre (`new`) konstruktorral, akkor figyelni kell arra, hogy a program futása során először ez történjen meg – esetleg valamelyik meghívott függvényben – és csak ezt követően használjuk fel.

13. példa: Osztály szintű változók használata

```
6. static int MaxN=1000;
7. static int[] tomb = new int[MaxN];
8. static int N;
9. static double[] t;
10. static void Main()
11. {
12.     tomb[0] = 3;          /*a tömb létezik, elemének kezdőértéket adunk.*/
13.     N = tomb[0] + 1;      /*tömbérték felhasználása kezdőértékkadásra*/
14.     Console.WriteLine(N);
15.     Másikeljárás();
16.     Console.WriteLine(t.Length);
17. }
18.
19. static void Másikeljárás()          /*nincs paramétere*/
20. {
21.     t = new double[N];              /*adott méretű tömb létrehozása*/
22. }
```

Ha egy változót nem abban az eljárásban vagy függvényben használunk – hozunk létre, olvasunk be, módosítunk – amelyikben megneveztük, akkor argumentumként a függvény megfelelő típusú paraméterében adjuk át.

14. példa: Függvény paraméterezése, lokális változók átadása

```
6. struct R {public string N; public int K;}
7. static void Main()
8. {
9.     int n = 3; R r = new R {N="r", K=0}; int[] t = new int[2] {5, 4};
10. }
```

- Ha csak olvasni akarjuk a változó értékét, akkor elegendő a típust és az argumentum nevét megadni. Ekkor egyszerű vagy `struct`-tal létrehozott adattípusú változónk másolatát fogja használni a függvény, a `string`, `List<>` és tömb adatoknál a változóra mutató hivatkozást kapja meg.

```

11. int x = Felhasznál(n, r, t);
12. Console.WriteLine($"{x} {n} {r.N} {r.K} {t[0]} {t[1]}");
13. /*      Eredmény:      11 3      r      0      9      4      */
14. t[0] = 5; //Vissza, mert t[0] így is módosult, a többi nem.
15.

```

Az átadott változók felhasználására paraméterezett függvény:

```

26. static int Felhasznál(int adat, R rekord, int[] tomb)
27. { /*adatban és rekordban másolat, tombben hivatkozás*/
28.     tomb[0] += tomb[1];
29.     rekord.K--;
30.     adat += tomb[0] + rekord.K;
31.     return adat;
32. }
33.

```

- Ha módosítani akarjuk a kapott változót, akkor – az egyszerű adattípusok és `struct`-tal megadott típusok esetén – a `ref` kulcsszóval kell a függvény megadásakor és használatkor is jelezni, hogy az eredeti példánnyal szeretnénk dolgozni, nem a másolattal. Ilyenkor ezeket az adatokat is az eredeti helyükre mutató hivatkozáson keresztül érjük el, így módosíthatjuk is. Ha csak egy ilyen változót szeretnénk használni, akkor praktikusabb olyan függvényt írni, aminek az adott változó eredménye a visszatérési értéke.

```

16. int y = Módosít(ref n, ref r, t);
17. Console.WriteLine($"{y} {n} {r.N} {r.K} {t[0]} {t[1]}");
18. /*      Eredmény:      11 11      r      -1      9      4      */
19.

```

Az átadott változók módosítására paraméterezett függvény:

```

34. static int Módosít(ref int adat, ref R rekord, int[] tomb)
35. { /*adatban, rekordban és tombben hivatkozás*/
36.     tomb[0] += tomb[1];
37.     rekord.K--;
38.     adat += tomb[0] + rekord.K;
39.     return adat;
40. }
41.

```

- Ha az eljárás, függvény során létrehozás, kezdőértékkadás is szükséges, akkor a `ref` helyett az `out` jelzőt használjuk. Itt is sokszor praktikusabb függvény visszatérési értéként átadni a függvényen belül létrehozott változót.

```

20. int a; R elem; int[] sor;
21. Létrehoz(out a, out elem, out sor);
22. Console.WriteLine($"{a} {elem.N} {elem.K} {sor[0]} {sor[1]}");
23. /*      Eredmény:      1      név      0      3      4      */
24. }
25. /*Lásd még: bool int.TryParse(string s, out int result)*/

```

Az átadott változók létrehozására paraméterezett függvény:

```
42. static void Létrehoz(out int adat, out R rekord, out int[] tomb)
43. {
44.     adat = 1;
45.     rekord.N = "név"; rekord.K = 0;
46.     tomb = new int[2] { 3, 4 };
47. }
```

Egy eljárásban vagy függvényben megadott kódot úgy építhetünk be a meghívás helyére, hogy a paramétereit az eljáráson belül átírjuk a hívásnál adott argumentumokra és a teljes kódot átmásoljuk. A függvénybéli **return** helyére értékadást írhatunk vagy kiírhatjuk az eredményt. Egy paraméter nélküli eljárásban megadott kód változtatás nélkül tehető más eljárás részének. (Ez lényegében a Refactoring visszafelé történő elvégzése.)

VITÜK VALAMIRE – TÍPUSALGORITMUSOK

A tizedik évfolyamos könyvünkben láttuk, hogy a típusalgoritmusokat – más néven programozási tételeket – azért érdemes ismerni, mert gyakran felmerülő problémákra adnak kipróbált megoldást. Az eddig megismert – úgynevezett „egyszerű” – típusalgoritmusok közös jellemzője, hogy egy adatsorozatot vizsgálnak, és egy egyszerű értékkel térnek vissza. Az egyszerű érték lehet például egy szám, a sorozat egy eleme vagy logikai érték: igaz vagy hamis.

Vannak olyan programozási nyelvek – ilyen a C# is –, ahol a típusalgoritmusoknak van rövid formájuk, függvényük is, ezek a System.Linq névtérből érhetők el. Ha nem lennének ilyen rövid formák, akkor a programozók saját kis kód-tárában lennének elmentve a gyakori feladatokra a megoldások és azt használnák, saját készítésű névtereket adnának a programjukhoz. A rövid formákat nagyon szeretjük (gyorsabban lehet beírni őket, könnyen olvashatók), de sokszor olyan, összetettebb eset van, amikor az előre elkészített függvény „konzerv” helyett jobb az általános forma. Fontos tehát, hogy a megoldás elvét is megismerjük. Így mindig pontosan az épp felmerülő problémának megfelelően tudjuk alkalmazni, kódolni a típusalgoritmusokat.

Mivel a tizedikes jegyzetben részletesen szerepelnek a típusalgoritmusok függvényes megoldásai, ezeket – a tankönyvtől eltérően – itt nem ismételjük meg. Abban is eltérünk a tankönyvtől, hogy a megoldásokat csak tömb típusra adjuk meg, a listák használatát ismerők számára az átírás nem jelenthet gondot. Az egyes algoritmusokra külön függvényeket írunk, aminek bemeneti paraméterként adjuk át a feldolgozandó tömböt, függetlenül a konkrét feladat adataitól. Az általunk írt függvény és A **Main()** függvény – ahol felhasználjuk függvényünket – a **Program** osztályon belül vannak, ezért a **static** jelzőt kiírjuk, de sorrendjük tetszőleges, ezért a jegyzetben a sorok számozása csak egy lehetséges állapotot mutatnak, a **Main()** függvénybe írt kódrészletnél sokszor nincs számozás.

A tanult típusalgoritmusok többségében az elágazás és ciklus magja csak egy utasítást tartalmaz. Ha snippettel írjuk a kódot, akkor a ciklusmagot a kapcsos-zárójelek közé kell írni, de ez a zárójelpár egy utasítás esetén felesleges. A jegyzetben a kód olvasását nehezíti az oldaltörés, ami sokkal gyakrabban bekövetkezik, ha két sor helyett négy, 3 sor helyett hét sor hosszú a kódrészlet. Ezért a jegyzetben, ha csak egy utasítás van a ciklusmagban, akkor nem tesszük ki a kapcsos-zárójelet. A feladatok kódolásakor – főleg, ha snippettel írjuk a vezérlési struktúrát, akkor érdemes megtartani a zárójeleket, mert egy későbbi továbbfejlesztés ebből az állapotból könnyebb.

{ } elhagyható for, if... után, ha csak 1 utasítást tartalmaz (ahogy a jegyzetben van) de nem hagyjuk el, ha snippettel írjuk a kódot.

Sorozatszámítás

A sorozatszámítás algoritmusára arra jó, hogy egy bejárható objektum elemeit adjuk össze, szorozzuk össze, vagy írjuk egymás mellé. Azaz

ebből	ilyen művelettel	ilyet készít:
[1,2,5]	összeadás	8
['ma', 'j', 'om']	összefűzés	'majom'
[2,3,4]	átlagolás	3
[1,2,3,4]	összeszorzás	24

15. példa: Hónapok napjaiból az év hossza

Adjuk meg az év hosszát az egyes hónapok napjainak száma alapján!

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     Console.WriteLine("Összesen: {0}", Sorozatszámítás(hónapnapok));
6. }
```

A sorozatszámítás függvénye:

```
1. static int Sorozatszámítás(int[] tomb)
2. {
3.     int szum = 0; /*műveletfüggő kezdőérték*/
4.     for (int i = 0; i < tomb.Length; i++)
5.         szum += tomb[i]; /*kumuláció, halmozás művelete*/
6.     return szum;
7. }
```

Eldöntés

Az eldöntés algoritmusára arra a kérdésre válaszol, hogy van-e adott tulajdonságú elem a bejárható objektumunkban.

16. példa: Van-e 28 napos hónap az évben?

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     if (Van28(hónapnapok))
6.         Console.WriteLine("Van 28 napos hónap.");
7.     else
8.         Console.WriteLine("Nem találtunk 28 napos hónapot.");
9. }
```

A tizedikes jegyzetben szerepel egy kitekintés arról, hogy az egyszerű típusalgoritmusokban van valami közös... Nulladik megoldásunk – gondolatébresztőnek – egy trükkös sorozatszámítás, aminek a lényege, hogy az eredmény akkor legyen `true`, ha egyik vagy másik vagy harmadik vagy... hónap 28 napos. Azaz az összegzés itt most „összevagyolás”.

```

1. static bool Van28(int[] tomb) /*1. megoldás*/
2. {
3.     bool van_28 = false;
4.     for (int i = 0; i < tomb.Length; i++)
5.         van_28 |= (tomb[i] == 28); /*true lesz, ha van legalább egy true*/
6.     return van_28;
7. }

```

Hogyan?
false || false || true
|| false || ... -> true

Ezzel a megoldással egyenértékű, ha a halmozás műveletét egyágú elágazással fogalmazzuk meg. Aki nehezen használja a logikai változót, az használhat helyette egész számot: a **false** helyett **0**, a **true** helyett **1** vagy bármilyen pozitív szám megfelelő. Ekkor a „vagy” helyett az összeadás használható.

```

1. static bool Van28(int[] tomb) /*2. megoldás*/
2. {
3.     bool van_28 = false; /*Lehetne: */
4.     for (int i = 0; i < tomb.Length; i++) /*int db = 0 */
5.         if (tomb[i] == 28)
6.             van_28 = true; /* db += 1; */
7.     return van_28; /*return db > 0; */
8. }

```

Mindkét megoldással van azonban egy kis bibi: *nem hatékony*. Működik, de a szükségesnél több energiát használ és lassúbb. Ha lépésenként futtatjuk a programunkat, akkor látszik, hogy felesleges az első 28-as tömbelem megtalálása után folytatni a ciklust. Ezért figyeljük ezt is, a ciklusba csak akkor lépünk be, ha a **van28** értéke **false** (azaz a tagadása igaz).

```

1. static bool Van28(int[] tomb) /*3. megoldás*/
2. {
3.     bool van_28 = false;
4.     for (int i = 0; i < tomb.Length && !van_28; i++)
5.         if (tomb[i] == 28)
6.             van_28 = true;
7.     return van_28;
8. }

```

igaz, ha van_28 == false,
hamis, ha van_28 == true

Találatkor true lesz. Utána a 4. sorban
NEM true -> false
ezért nem lép be a ciklusba

Kis változtatás a kódban, nagy lépés a hatékonyságban... Most már csak a kód *olvashatóságán* kellene egy kicsit javítani. Erre két lehetőségünk van, de közös bennük, hogy a **van_28** változó felesleges. Ezt abból lehet látni, hogy az értéke egyetlen módon változhat, akkor, ha a **tomb[i] == 28**, azaz lényegében **van_28 == (tomb[i] == 28)**.

```

1. static bool Van28(int[] tomb) /*4. megoldás*/
2. {
3.     int i;
4.     for (i = 0; i < tomb.Length && !(tomb[i] == 28); i++)
5.     {}
6.     return i < tomb.Length;
7. }

```

!(tomb[i] == 28) másképp: tomb[i] != 28

Szövegesen: Úgy döntöttem el, hogy van-e 28 napos hónap, hogy elindulok a tömb elejétől és amíg van tömbelem, *de ez nem* 28, addig továbblépek. Ha a tömb vége előtt fejezem be a továbblépést, akkor találtam (ha **i** a tömb utolsó eleme utánra mutat, akkor nem találtam). A megoldás során figyelni kell arra, hogy ha a végeredmény **false**, akkor ott nem szabad vizsgálni a **tomb[i]**-t, mert az az utolsó utáni tömbelem lenne. Ugyanezért ökölszabály, hogy több

feltétel esetén először mindig azt nézzük, hogy létezik-e egy változó és csak ezt követően vizsgáljuk az értékét – mindig az `i < tomb.Length` az első. A programozásban nem kommutatívák a logikai műveletek.

A strukturált programozás elvei alapján – bár a nyelv megengedi – for-ciklust csak akkor használunk, ha bejárjuk a teljes adatsort. Ezért az algoritmus feltételes (while-) ciklussal dolgozik – és ebben az esetben már nem lesz üres a ciklusmag.

```

1. static bool Van28(int[] tomb) /*5. megoldás*/
2. {
3.     int i = 0;
4.     while (i < tomb.Length && !(tomb[i] == 28))
5.         i++;
6.     return i < tomb.Length;
7. }

```

(1) létezik, (2) ÉS, (3) NEM, (4) jó

kérem a következőt (mert ez nem jó)

Létezik jó: `i < hossz`. Nem létezik jó: `i == hossz`.

Másik módja a logikai változótól való megszabadulásnak, ha megszakítjuk a ciklus futását. Ennek a megoldásnak előnye, hogy ember számára könnyebben ellenőrizhető, ha azt kell vizsgálni, hogy valami igaz-e, mintha a tagadását.

```

1. static bool Van28(int[] tomb) /*6. megoldás*/
2. {
3.     int i;
4.     for (i = 0; i < tomb.Length; i++)
5.         if (tomb[i] == 28)
6.             break;
7.     return i < tomb.Length;
8. }

```

Nem a 4. sorra ugrik ellenőrzésre, hanem a 7. sorra, pánikszzerűen.

Ez nem strukturált megoldás, mivel programozási nyelvtől függ, hogy mi lesz egy megszakítás járulékos következménye. Az eldöntés típusalgoritmusban a cikluson belül csak egy elágazás van, de a nyelv lehetővé teszi, hogy előtte elkezdjünk egy másik feladatot, amit utána fejezünk be. Megszakításkor az elágazásban lévő, `break` utáni utasítások és a ciklusban az elágazás utáni utasítások elvégzése elmarad.

Ha függvényként írjuk meg az eldöntést, akkor tovább szépíthetünk a kódon:

```

1. static bool Van28(int[] tomb) /*7. megoldás*/
2. {
3.     for (int i = 0; i < tomb.Length; i++)
4.         if (tomb[i] == 28)
5.             return true;
6.     return false;
7. }

```

Az eldöntés algoritmusát sokféle formában meg lehet írni. A szabványos típusalgoritmus az 5. megoldás, sokan használják a 6. vagy 7. megoldást. Ha ezek nem érthetőek, akkor bármelyik másik megoldás is helyesen működik és amíg nem szakember írja a kódot, addig csak az a lényeg, hogy tudjuk: mit miért írtunk a kódban.

Kiválasztás

A kiválasztás algoritmus akkor használható, ha tudjuk, hogy van adott tulajdonságú elem az adatsorozatunkban és kíváncsiak vagyunk, hogy hányadik sorszámu ez az elem.

17. példa: Hányadik az első 30 napos hónap?

Tudjuk, hogy van 30 napos hónap, csak azt nem tudjuk, hogy hányadik. Ezt az algoritmust csak nagy körültekintéssel szabad használni. Például, ha a 28 napos hónapot szeretnénk kiválasztani egy szököévben, az programhibát okozna. A strukturált programozás követelményének eleget téve – előtesztelés feltételes ciklust használunk.

```
1. static int Harminc(int[] tomb)
2. {
3.     int i = 0;
4.     while (tomb[i] != 30)
5.         i++;
6.     return i;
7. }
```

NEM jó

kérem a következőt

A visszaadott indexnél 1-gyel nagyobb a hónap sorszáma, mert a tömböket 0-tól indexeljük, de a hónapokat 1-től sorszámozzuk.

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     Console.WriteLine("A(z) {0}. hónap 30 napos.", Harminc(hónapnapok) + 1);
6. }
```

A while- helyett használható for-ciklus is, de annak nem lesz látható ciklusmagja.

Keresés

A keresés algoritmus az eldöntés és a kiválasztás egybeépítése. Van-e adott tulajdonságú elem, és ha igen, akkor hol?

18. példa: Ha van 28 napos hónap az évben, akkor hányadik hónap ilyen?

A megoldás bármelyik eldöntésre írt megoldás módosításával lehetséges, de a statikusan típusos nyelvek (mint a C# is) esetén gondot jelent, hogy a visszatérés típusa találat esetén egy szám, ellenkező esetben logikai adat. Az alábbiakban az eldöntésre adott megoldások átalakításainál az első sorban az eldöntés sorszáma látható.

Az első megoldásban a kétféle eredményt azzal védjük ki, hogy eljárásként írjuk meg az algoritmust. Ez a megoldás akkor jó, ha a talált adattal nincs további feladatunk.

```
1. static void Melyik28(int[] tomb) /*3. megoldás átalakítva*/
2. {
3.     int melyik = -1; /*bool van_28 = false;*/
4.     for (int i = 0; i < tomb.Length && melyik == -1; i++) /*!van_28*/
5.         if (tomb[i] == 28)
6.             melyik = i; /*van28 = true;*/
7.     if (melyik > -1) /*return van_28;*/
8.         Console.WriteLine("A(z) {0}. hónap 28 napos.", melyik + 1);
9.     else
10.        Console.WriteLine("Nem találtunk 28 napos hónapot.");
11. }
```

Felhasználása:

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     Melyik28(hónapnapok);
6. }
```

A második megoldásban a keresés kiválasztás jellegét emeljük ki, a függvény felhasználójára bízunk, hogy figyeljen arra, hogy a visszaadott index létezik-e. A „nem létező index” kétféle lehet, vagy kisebb, mint bármelyik lehetséges – tipikusan -1 –, vagy nagyobb a lehetségeseknél – tipikusan a tömb hossza –. Most az utóbbi megoldást valósítjuk meg.

```
1. static int Melyik28(int[] tomb) /*5. megoldás*/
2. {
3.     int i = 0;
4.     while (i < tomb.Length && !(tomb[i] == 28))
5.         i++;
6.     return i; /*< tomb.Length*/
7. }
```

A felhasználásnál kell figyelni!

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     int melyik = Melyik28(hónapnapok);
6.     if(melyik < hónapnapok.Length)
7.         Console.WriteLine("A(z) {0}. hónap 28 napos.", melyik + 1);
8.     else
9.         Console.WriteLine("Nem találtunk 28 napos hónapot.");
10. }
```

A következő megoldás úgy lesz használható, mint a számok `.TryParse()` függvényei: A függvény logikai értéket ad vissza, de futása során – ha tudja – átadja a keresett indexet egy paraméterén keresztül.

```
1. static bool Melyik28(int[] tomb, out int ez) /*7. megoldás*/
2. {
3.     for (int i = 0; i < tomb.Length; i++)
4.         if(tomb[i] == 28)
5.         {
6.             ez = i;
7.             return true;
8.         }
9.     ez = -1;
10.    return false;
11. }
```

A hívás helyén lévő változóban megadjuk a választ. vagy egy helyettesítő értéket.

Felhasználása:

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     int melyik;
6.     if(Melyik28(hónapnapok, out melyik))
7.         Console.WriteLine("A(z) {0}. hónap 28 napos.", melyik + 1);
8.     else
9.         Console.WriteLine("Nem találtunk 28 napos hónapot.");
10. }
```

Kiegészítés Pythonról áttérőknek és C# nyelvészeknek

Ha nagyon megőröltetnénk magunkat, készíthetnénk egy struktúrát a két eredmény tárolására. Az így létrejövő adattípus lehetne a függvényünk kimenete, de elég elrettentően kényszeredett a talál.van és talál.ez forma használata csak azért, hogy egy függvény két értéket tudjon visszaadni. Több más hasonló adatstruktúra (például egy [Dictionary](#) elem) után, a [Tuple](#) adattípus a C# 7.0 verziójában, 2017-ben jelent meg. Később, mint a Pythonban és nem teljesen azonos funkciókkal, de a fő felhasználási terület azonos. A [Tuple](#) egy átmeneti struktúra, változók csoportja. A felhasználás oldaláról indulva:

```
1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     (bool van, int melyik) = Melyik28(hónapnapok);
6.     if(van)
7.         Console.WriteLine("A(z) {0}. hónap 28 napos.", melyik + 1);
8.     else
9.         Console.WriteLine("Nem találtunk 28 napos hónapot.");
10. }
```

Két változóba várjuk a függvény eredményét

Talán nem meglepő, hogy a függvény kimenetének hasonló a szintaktikája. Ráadásul az eldöntés típusalgoritmushoz képest minimális kiegészítéssel kapjuk meg a keresés algoritmusát:

```
1. static (bool, int) Melyik28(int[] tomb) /*5. megoldás*/
2. {
3.     int i = 0;
4.     while (i < tomb.Length && !(tomb[i] == 28))
5.         i++;
6.     return (i < tomb.Length, i);
7. }
```

Két kimenete van.

Egyszerre két eredményt ad át.

Ha eldöntést sokféleképpen lehet írni, akkor a lineáris keresésnek hatványozottan sokféle megoldása van. Ezek közül legalább egyet érteni kell és tudni kell mindenféle feladatkörnyezetben alkalmazni.

Megszámolás

A megszámlálás algoritmus megmondja, hogy hány adott tulajdonságú elem van a bejárható objektumban.

19. példa: Hány 30 napos hónap van az évben?

A 30-at tekintjük paraméternek, így egy kicsit rugalmasabban írjuk meg a megoldást, a használat helyén döntjük el, hogy a 30, 31 vagy esetleg a 29 napos hónapok számára vagyunk-e kíváncsiak.

```

1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     Console.WriteLine("{0} 30 napos hónap van.", Darab(hónapnapok, 30));
6. }
7.
8. static int Darab(int[] tomb, int ertekek)
9. {
10.     int db = 0;
11.     for (int i = 0; i < tomb.Length; i++)
12.     {
13.         if (tomb[i] == ertekek)
14.         {
15.             db++;
16.         }
17.     }
18.     return db;
19. }

```

jó megoldások még:
 $db = db + 1;$
 $db += 1;$
 $++db;$

Szélsőérték-kiválasztás: maximum- és a minimumkiválasztás

A maximum- vagy a minimumkiválasztás algoritmus az adatsorozat legnagyobb vagy legkisebb elemét keresi meg. Nem lényegtelen, hogy kiválasztás és nem keresés, mivel tudjuk, hogy létezik megoldás. Tényleg mindig létezik megoldás? Majdnem... A nulla db adatot tartalmazó adatsorozatnak nincs szélsőértéke. Ha ez előfordulhat, akkor a szélsőértéket keresni kell és nem kiválasztani. Ilyen esetben az a legjobb megoldás, ha először ellenőrizzük, hogy van-e adat a sorozatunkban és ha van, akkor kiválaszthatjuk a szélsőértéket, ha nincs, akkor szélsőérték sincs.

A szélsőérték kiválasztásának algoritmusában a másik kényes kérdés, hogy mi legyen a válasz: a leg... érték vagy az az index, ahol ez az érték található. Általában az indexet, a szélsőérték helyét érdemes kiválasztani, mert ebből azonnal meg tudjuk mondani, hogy mennyi az értéke az adott adatnak. Ha a szélsőérték értékét adjuk meg, akkor annak megadása, hogy ez hol van – vagy a későbbiekben valami más, hozzá kapcsolódó érték megadása – újabb feladat, egy kiválasztás megírását igényli. Ez dupla mennyiségű kód és átlagosan 1,5-szeres processzoridő.

20. példa: Hányadik hónap a legrövidebb?

```

1. static void Main()
2. {
3.     int[] hónapnapok = new int[12]
4.         { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
5.     Console.WriteLine("A legrövidebb hónap {0} napos.",
6.         hónapnapok[Minhely(hónapnapok)]);
7. }

```

```

1. static int Minhely(int[] tomb)
2. {
3.     int mini = 0; /*int ertekek = tomb[0];*/
4.     for (int i = 1; i < tomb.Length; i++)
5.         if (tomb[mini] > tomb[i]) /*ertekek > tomb[i]*/
6.             mini = tomb[i]; /*ertekek = tomb[i]*/
7.     return mini; /*ertekek*/
8. }

```

A fenti kódban megjegyzésként olvasható, hogy mit kellene írni, ha nem a minimum helyét, hanem az értéket adnánk meg.

Maximumkiválasztáshoz egyetlen helyen kell a kódot módosítani: az 5. sorban meg kell fordítani a relációs jelet (nagyobb helyett kisebb kell). Ha a szélsőérték helyét adjuk meg és több egyforma értékünk van, akkor a fenti kód az első minimum helyét adja meg. Az utolsó minimum hely megadásához a nagyobb helyett nem kisebb reláció kell.

Feladatok

A következő kérdések egy tíznapos periódus hajnali hőmérsékleteit tartalmazó listára vonatkoznak.

Adjuk meg, hogy az egyes kérdésekre a válaszhoz melyik típusalgoritmust használjuk! Legalább három megoldást kódoljuk is!

hőmérsékletek = [7, 5, -2, 0, -4, 3, 3, 3, 4, 4]

1. Hány fok volt a legmelegebb hajnalon?
2. Volt-e olyan hajnal, amikor fagyott? (A nulla fokos víz vagy megfagyott már, vagy még éppen nem. Menjünk biztosra, nézzük a nulla foknál hidegebb hajnalokat!)
3. Volt-e olyan hajnal, amikor három fokot mértek?
4. Hány nulla fokos hajnal volt?
5. Mikor volt először nulla fokos hajnal?
6. Hányadik hajnalon volt először fagy?
7. Hányadik hajnalon volt utoljára fagy?
8. Hányszor fagyott hajnalban?
9. Volt-e olyan, hogy egymás utáni hajnalokon ugyanazt a hőmérsékletet mérték?
10. Hányadik hajnalon volt először melegebb, mint előző nap?

VITTÜK VALAMIRE – TÍPUSALGORITMUSOK KÉTDIMENZIÓS TÖMBBEL ÉS REKORDOK TÖMBJÉVEL

Többdimenziós adatsorozatok vizsgálatakor általában csak az egyik dimenzióra tudjuk használni a típusalgoritmusokat, de – akár többször ismételve – a kapott eredményekre ismét használhatunk egy típusalgoritmust.

21. példa: Hiányzók

Ismerjük egy négyhetes időszak egyes napjain a 11. zs osztály tanulóinak a hiányzásait, amit egy kétdimenziós tömbben tárolunk. Az adatok alapján a következő kérdésekre fogunk válaszolni:

1. Hány hiányzás volt összesen?
2. Volt-e olyan hét, amikor nem volt hiányzó?
3. Volt-e olyan hét, amikor ötnél kevesebb hiányzás volt?
4. Melyik héten volt a legtöbb hiányzás?
5. Hányadik héten volt egyetlen hiányzás?
6. A hétfői vagy a keddi napokon hiányoztak-e többen a négy hét alatt?

A feladatokra a választ a `Main()` függvény tartalmazza, ahol a szöveges kiírásban felhasználjuk a függvény vagy eljárás formában készített megoldásokat. Az egyes feladatokra készített függvények és eljárások a példaadatoktól függetlenek, a tömböt paraméterként veszik át. Ezért a feladatok megoldása során használt adatok kétdimenziós tömbjét a `Main()` függvényen belül adjuk meg.

```
6. static void Main()
7. {
8.     int[,] hianyzok = new int[4, 5]{ /*táblázat eleje*/
9.         /*1. sor*/ { 3, 4, 0, 0, 1 },
10.        /*2. sor*/ { 2, 3, 0, 0, 0 },
11.        /*3. sor*/ { 4, 3, 2, 3, 4 },
12.        /*4. sor*/ { 0, 0, 1, 0, 0 } /*táblázat vége*/};
```

Például a második hét keddjén három fő hiányzott.

```
13. Console.WriteLine("Heti hiányzás statisztikája");
14. Console.WriteLine("1a. Össz hiányzás: {0}.", TombSzum(hianyok));
15. Console.WriteLine("1b. Összes hiányzás: {0}.", SzumSzum(hianyok));
16. if (VoltHet0(hianyok))
17.     Console.WriteLine("2. Volt hiányzásmentes hét.");
18. else
19.     Console.WriteLine("2. Nem volt hiányzásmentes hét.");
20. if (VoltHetKevesebb(hianyok, 5))
21.     Console.WriteLine("3. Volt 5-nél kevesebb hiányzásos hét.");
22. else
23.     Console.WriteLine("3. Ötnél mindig több volt a hiányzás.");
24. Console.WriteLine("4. A(z) {0}. héten volt a legtöbb
        hiányzás.", MaxHet(hianyok) + 1);
25. Console.WriteLine("5. A {0}. héten volt 1 hiányzás.",
        Kivalaszt1(hianyok));
26. Console.WriteLine("6. ");
27. HetfoKedd(hianyok);
28. }
29.
```

Az eldöntés vagy keresés típusalgoritmussal megoldható feladatokban gyakori, hogy az eredményt csak egy értékadásra használjuk. A 16–19. sorban a válaszokban a „Volt” és „Nem volt” ez az érték, amit egy elágazásban adtunk meg, de helyette használhatjuk a **háromoperandusú operátort**: `op1?op2:op3`, ahol a kérdőjel előtt az `op1` feltételt adjuk meg, utána a feltétel teljesülése esetén érvényes `op2` értéket, a kettőspont után a feltétel nem teljesülése esetén érvényes `op3` értéket írjuk.

```
16. Console.WriteLine("2. {0} hiányzásmentes hét.",
    VoltHet0(hianyzo) ? "Volt" : "Nem volt");
```

Ezzel négy sor helyett egyetlen sorban adtuk választ. A megoldásban elrejtettünk egy értékadást, amit azért nem kell kiírnunk, mert csak egyszer használjuk fel:

```
string válasz = VoltHet0(hianyzo) ? "Volt" : "Nem volt");
```

Hasonlóan átírható a 20–23. sor is, de nem érdemes, mert a teljes szöveg módosul. A háromoperandusú operátor csak értékadásra használható, az értékek helyén nem lehet eljárás, csak két azonos típusú érték, változó vagy függvényérték.

1. Hány hiányzás volt összesen?

Első megoldásunkban a sorozatszámítás algoritmusát úgy módosítjuk, hogy a sorokon belül a napokon is végigmegyünk. A két ciklus használatakor – a későbbiek miatt – kiemeljük, hogy két dimenzióról van szó, az `int[4, 5]` tömb esetén a `GetLength(0)` értéke 4, a `GetLength(1)` értéke 5. A végrehajtás szempontjából ez olyan, mintha az adatokat egyetlen sorba írtuk volna fel, egyetlen ciklusban a `tomb.Length` lehetne a lépések száma.

```
30. private static int TombSzum(int[,] tomb)
31. {
32.     int szum = 0;
33.     for (int tr = 0; tr < tomb.GetLength(0); tr++) /*soronként 4-ig*/
34.         for (int td = 0; td < tomb.GetLength(1); td++) /*naponként 5-ig*/
35.             szum += tomb[tr, td];
36.     return szum;
37. }
38.
```

A megoldás egy másik megfogalmazása: kiszámítjuk a heti hiányzások összegét, majd ezekből meghatározzuk a négyheti összeget. Ehhez először egy olyan függvényt írunk, ami egy megadott sorra számol összeget. Ez azért is praktikus, mert később több feladat megoldásához szükséges lesz a sorokra számolt összeg.

```
39. private static int SorSzum(int[,] tomb, int sor)
40. {
41.     int szum = 0;
42.     for (int i = 0; i < tomb.GetLength(1); i++)
43.         szum += tomb[sor, i];
44.     return szum;
45. }
46.
```

A paraméterben megadott sorban.

A `SorSzum()` majdnem tisztán sorozatszámítás, csak az indexelés speciális. Az alábbi `SzumSzum()` szintén tipikus, csak az érték helyett függvényértékeket összegzünk.

```
47. private static int SzumSzum(int[,] tomb)
48. {
49.     int szum = 0;
50.     for (int i = 0; i < tomb.GetLength(0); i++)
51.         szum += SorSzum(tomb, i);
52.     return szum;
53. }
54.
```

Függvényhívás, a számított eredménnyel nő az szum értéke.

2. Volt-e olyan hét, amikor nem volt hiányzó?

Ezúttal is szükségünk van a `SorSzum()`, heti összegekre. A kérdés megválaszolásához logikai értéket visszaadó függvényt írunk: eldöntjük, hogy van-e a heti összegek között 0 érték. Ehhez az eldöntés (strukturált programozásnak megfelelő) típusalgoritmusát használjuk:

```
55. private static bool VoltHet0(int[,] tomb)
56. {
57.     int ez = 0;
58.     while (ez < tomb.GetLength(0) && SorSzum(tomb, ez) != 0)
59.         ez++;
60.     return ez < tomb.GetLength(0);
61. }
62.
```

3. Volt-e olyan hét, amikor ötnél kevesebb hiányzás volt?

Ugye látjuk, hogy egy relációs jel módosítása a lényegi változtatás az előző feladat kódjához képest?

```
63. private static bool VoltHetKevesebb(int[,] tomb, int korlat)
64. {
65.     int ez = 0;
66.     while (ez < tomb.GetLength(0) && SorSzum(tomb, ez) >= korlat)
67.         ez++;
68.     return ez < tomb.GetLength(0);
69. }
70.
```

4. Melyik héten volt a legtöbb hiányzás?

Ismét jó hasznát vesszük a `SorSzum()` függvénynek. Illik arra is figyelni, hogy ha a feladat megoldása során többször számolnánk ki ugyanazt a függvényt, akkor praktikus az eredményeket eltárolni. Egy tipikus szélsőérték kiválasztás feladatban a tömb indexeket, vagy az indexek által mutatott értékeket használjuk. Itt az indexszel paraméterezett függvényértéket használnánk. Többször. Az ismételt kiszámítást elkerülendő, az első használat alkalmával változóban tároljuk az értéket.

```
71. private static int MaxHet(int[,] tomb)
72. {
73.     int maxi = 0;
74.     int maxe = SorSzum(tomb, 0);
75.     for (int i = 1; i < tomb.GetLength(0); i++)
76.     {
77.         int akt = SorSzum(tomb, i);
78.         if (maxe < akt)
79.         {
80.             maxe = akt;
81.             maxi = i;
82.         }
83.     }
84.     return maxi;
85. }
86.
```

Kétszer használt i-edik összeg

Minden cikluslépésben kétszer használt maxi-edik összeg

5. Hányadik héten volt egyetlen hiányzás?

A kérdés alapján feltételezhető, hogy tudjuk, volt egy olyan hét, amikor csak egy hiányzás volt. A példa adatok között valóban van is egy ilyen hét, ezért – valamint azért, mert a tankönyvben kiválasztás van és nem keresés – a kiválasztás típusalgoritmust használjuk.

```
87. private static int Kivalaszt1(int[,] tomb)
88. {
89.     int i = 0;
90.     /*hibás, ha nincs megfelelő sor!*/
91.     while (SorSzum(tomb, i) != 1)
92.         i++;
93.     return i;
94. }
95.
```

6. A hétfői vagy a keddi napokon hiányoztak többen a négy hét alatt?

A feladatok közül ez az egyetlen, amelyik nem a hetekről szól, hanem a napokról, ezért most nem használható a `SorSzum()`. Mivel csak két napra kell összegezni, erre rövidebb a belső ciklus nélküli megoldás. Helyette két kezdőértékkel és két értékmódosítással a két sorozatszámítás „párhuzamosítjuk”.

A feladatnak az is specialitása, hogy az eredménytől függően háromféle szöveges válaszunk lehet, ami a feladatnak lényeges része, ezért nem célszerű a főprogramra hagyni. Ezért, és a változatosság kedvéért, ez a megoldás nem függvény, hanem eljárás lesz.

```
96. private static void HetfoKedd(int[,] tomb)
97. {
98.     int hetfo = 0;
99.     int kedd = 0;
100.    for (int het = 0; het < tomb.GetLength(0); het++)
101.    {
102.        hetfo += tomb[het, 0];
103.        kedd += tomb[het, 1];
104.    }
105.    string valasz;
106.    if (hetfo > kedd)
107.        valasz = "Hétfői napokon hiányoznak többen.";
108.    else if (hetfo < kedd)
109.        valasz = "Keddi napokon hiányoznak többen.";
110.    else
111.        valasz = "Ugyanannyi a hiányzás a hét két napján.";
112.    Console.WriteLine(valasz); /*ha függvény: return valasz;*/
113. }
```

22. példa: Hazánk legmagasabb hegycsúcsai

A következő feladatok során egy olyan adatsorozattal foglalkozunk, melynek elemei összetettek, többféle adattípusból álló rekordok, másnéven struktúrák. Példánkban a sorozat elemei országunk legmagasabb hegycsúcsai közül néhánynak az adatait – a hegycsúcs és a hegység nevét, valamint a csúcs magasságát – tartalmazzák.

A hegycsúcsokat tartalmazó tömböt az előző példától eltérően, nem a főprogramban, hanem előtte, statikusan hozzuk létre. A megalkotására két lehetőséget nézünk meg. Az első módszerben a legegyszerűbb módon írjuk meg a hegycsúcs struktúrát. Ezzel – bár kicsit hosszán, de – létrehozható olyan tömb, amiben több hegycsúcs adatait el tudjuk tárolni:

```

115. struct HCs
116. {
117.     public string Név;
118.     public string Hegys;
119.     public int Magas;
120. }
121.
122. static HCs[] cs = new HCs[7] {
123.     new HCs { Név = "Kékes", Hegys = "Mátra", Magas = 1014 },
124.     new HCs { Név = "Hidas-bérc", Hegys = "Mátra", Magas = 971 },
125.     new HCs { Név = "Galya-tető", Hegys = "Mátra", Magas = 964},
126.     new HCs { Név = "Szilvási-kő", Hegys = "Bükk-vidék", Magas = 961 },
127.     new HCs { Név = "Péter hegyese", Hegys = "Mátra", Magas = 960 },
128.     new HCs { Név = "Kettős-bérc", Hegys = "Bükk-vidék", Magas = 958 },
129.     new HCs { Hegys = "Bükk-vidék", Név = "Istállós-kő", Magas = 958 }
130. };

```

A kódból látható, hogy a `HCs` típusú elemekből álló tömb létrehozása után, minden elemet külön létre kell hozni. Szakszerűbben: a tömb konstruktorának futtatása után az egyes elemek konstruktorainak is le kell futnia. A `HCs`-nek nem írtunk konstruktort, így az alapértelmezett konstruktora üres szöveget és 0 értéket állítana be. De ettől eltérő adatokat szeretnénk megadni, ezért meg kell mondanunk az egyes adattagok értékét. A megadás során nem okoz fordítási problémát, ha egyes adattagoknak nem adunk értéket és a megadás sorrendje sem számít, csak az olvashatóságot rontja, ezért nem illik összevissza vagy hiányosan írni.

Látható, hogy az adattípus nevét minimálisra rövidítettük, az adattagok neve is az érthetőség határát súrolja, mert ezeket sokszor kell leírni és jelentősen rontja az átláthatóságot a képernyőn túlra lógó sor, a többsoros írás. A második megoldásban ezért a struktúrát egy kicsit fejlesztjük – írunk hozzá egy konstruktort – ezzel a tömbben hétszer megspóroljuk az adattagok megnevezését.

```

6. struct Csucs
7. {
8.     public string Név;
9.     public string Hegység;
10.    public int Magasság;
11.    public Csucs(string név, string hegység, int magasság)
12.    {
13.        Név = név;
14.        Hegység = hegység;
15.        Magasság = magasság;
16.    }
17. }
18.
19. static Csucs[] csúcsok = new Csucs[7]{
20.     new Csucs( "Kékes", "Mátra", 1014),
21.     new Csucs( "Hidas-bérc", "Mátra", 971),
22.     new Csucs( "Galya-tető", "Mátra", 964),
23.     new Csucs( "Szilvási-kő", "Bükk-vidék", 961),
24.     new Csucs( "Péter hegyese", "Mátra", 960),
25.     new Csucs( "Kettős-bérc", "Bükk-vidék", 958),
26.     new Csucs( "Istállós-kő", "Bükk-vidék", 958)
27. };
28.

```

Bár a `Hcs` és a `Csucs` adattípus nagyon hasonló, nem azonosak, különböző „fajta” adatok, ezért nem használhatók vegyesen. A feladatok megoldásában ezután a `Csucs` típusú csúcsok tömböt fogjuk használni (de használható a `Hcs` típusú `cs` tömb is). Mivel a tömböket a `Main()` eljárásen kívül, statikusan hoztuk létre, a `Program` osztályon belül bárhol olvasható. Példánkban a következő kérdésekre fogunk válaszolni:

1. Hány csúcs található a Bükk-vidéken?
2. Mennyi a mátrai csúcsok magasságának átlaga?
3. Van-e ezer méternél magasabb csúcs a listában?
4. Melyik a Bükk-vidék legmagasabb csúcsa?

Az első két kérdésre a `Main()` eljárásen belül válaszolunk, a megoldás tagolásához régiókat alakítunk ki.

1. Hány csúcs található a Bükk-vidéken?

```
29. static void Main()
30. {
31.     #region Bükk-vidék csúcsainak száma
32.     int db = 0;
33.     for (int i = 0; i < csúcsok.Length; i++)
34.         if (csúcsok[i].Hegység == "Bükk-vidék")
35.             db++;
36.     Console.WriteLine($"A Bükk-vidék {db} csúcsa van a listában.");
37.     #endregion
```

2. Mennyi a mátrai csúcsok magasságának átlaga?

```
39. #region Mátrai csúcsok átlagos magassága
40.     double szum = 0;
41.     db = 0; //újrahasznosított
42.     for (int i = 0; i < csúcsok.Length; i++)
43.         if (csúcsok[i].Hegység == "Mátra")
44.         {
45.             szum += csúcsok[i].Magasság;
46.             db++;
47.         }
48.     Console.WriteLine($"A mátrai csúcsok átlagosan {(szum / db):F0}
49.         métereseek.");
49.     #endregion
```

3. Van-e ezer méternél magasabb csúcs a listában?

Ennek és a következő kérdésnek a megválaszolásához függvényt írunk. Az itt bemutatott megoldás egyik érdekessége, hogy a két lehetséges válasz egy szóban tér el egymástól, ezért érdemes a háromoperandusú feltételes értékadást használni az elágazás kiírása helyett.

```

50.
51. Console.WriteLine("{0} 1000 méternél magasabb csúcs",
                       VanEzres() ? "Van" : "Nincs");
52. Console.WriteLine("A Bükk-vidék legmagasabb csúcsa: " +
                       BukkLeg("Bükk-vidék").Név);
53. }/*Main() vége*/
54.
55. private static bool VanEzres()
56. {
57.     int i = 0;
58.     while (i < csúcsok.Length && !(csúcsok[i].Magasság > 1000))
59.         i++;
60.     return i < csúcsok.Length;
61. }
62.

```

A következő kérdésre paraméteres megoldást írunk, de ezt is könnyen paraméteressé tehetjük. Bővítsük ezt a programrészt úgy, hogy a felhasználó adhassa meg, hogy hány méteres magasságot vizsgáljon a program!

4. Melyik a Bükk-vidék legmagasabb csúcsa?

A főprogramban a függvényhívást könnyen módosíthatjuk úgy, hogy a felhasználó adja meg a hegység nevét, ez azonban a megoldásban komolyabb megfontolásokat igényel, mivel a felhasználó által megadott hegység nevét nem kiválasztani kell, hanem keresni. Mi van, ha csak annyit hibázik a felhasználó, hogy két szóba írja a Bükk-vidék-et? Ezért a megoldás három részből áll. Először megkeressünk egy (első) adott hegységben található hegycsúcsot. Ezután, ha van ilyen, kiválasztjuk a legmagasabbat, végül – hiányzó hegységnev esetén is értelmesen – válaszolunk.

```

63. private static Csucs BukkLeg(string hegység)
64. {
65.     /*első csúcs keresése*/
66.     int maxi = 0;
67.     while (maxi < csúcsok.Length && csúcsok[maxi].Hegység != hegység)
68.         maxi++;
69.     if (maxi < csúcsok.Length) /*Ha van egy Bükk-vidéki csúcs,*/
70.     /*A legnagyobb kiválasztása*/
71.         for (int i = maxi + 1; i < csúcsok.Length; i++)
72.             if (csúcsok[i].Hegység == hegység &&
                  csúcsok[i].Magasság > csúcsok[maxi].Magasság)
73.                 maxi = i;
74.     /*Ha van Bükk-vidéki csúcs, akkor a legmagasabb nevének megadása*/
75.     if (maxi < csúcsok.Length)
76.         return csúcsok[maxi];
77.     else /*Hibajelzéshez értékek*/
78.         return new Csucs("#Nincs csúcs#", hegység, 0);
79. }

```

Itt a 75–78. sor helyett szintén alkalmazhatjuk a háromoperandusú értékadást. Ez azért is lenne szebb megoldás, mert így a függvényünk jelenlegi két befejezését egyetlen `return`-nel helyettesíthetjük.

```

return (maxi < csúcsok.Length) ? csúcsok[maxi]
                                : new Csucs("#Nincs csúcs#", hegység, 0);

```

23. példa: Kiegészítés: Kutyaoltás (szótárral)

Az egyik tipikus eset, amikor szótárat érdemes használni az, amikor sok „melyik-mennyi” párosításban tárolunk adatokat. Ekkor lényegében nevet kapnak az adatok. Ilyen eset például, hogy kinek hányas a dolgozata, melyik napon hányan néztek meg egy videót, vagy melyik napon hány kutyát oltott be az állatorvos. Ez utóbbiról szól ez a példa. Az állatorvostól kapott, egy hétre vonatkozó adatokat szótárban tárolták:

```
1. using System;
2. using System.Collections.Generic; /*KELL*/
3. namespace kutyaoltas {
4. class Program {
5. static void Main()
6. {
7.     Dictionary<string, int> oltasok = new Dictionary<string, int>()
8.     {
9.         {"hétfő", 8}, {"kedd", 14}, {"szerda", 2}, {"csütörtök", 3}, {"péntek", 13}
10.    };
11. }
```

Később jön a hír, hogy pénteken és szombaton is volt még egy-egy kutyus oltásért, ezeket még fel kellene jegyezni és válaszolni kellene a kérdésekre.

1. Hány kutyát oltott be a héten az állatorvos?
2. Volt-e olyan nap, amikor legalább tíz kutyát oltott be a doki?
3. Melyik napon oltotta be a legkevesebb állatot az állatorvos?

A pénteki és szombati plusz oltások egyenkénti bejegyzése során meg kell nézni, hogy az adott nap szerepel-e már a szótárban. Ha már szerepel, akkor nem tudjuk hozzáadni a napot, csak az értéket kell megnövelni. Ha még nem szerepel, akkor nem tudjuk növelni az aznapi oltások számát; az adott napot hozzá kell adnunk 1 értékkel a szótárhoz.

```
12. if (oltasok.ContainsKey("péntek")) /*Ha van ilyen Key érték*/
13.     oltasok["péntek"]++;           /*akkor növeljük a Value-t*/
14. else
15.     oltasok.Add("péntek", 1);      /*különben betesszük 1 értékkel*/
16. if (oltasok.ContainsKey("szombat"))
17.     oltasok["szombat"]++;
18. else
19.     oltasok.Add("szombat", 1);
```

A kérdésekre a főprogramban adunk választ, a megoldásokra írt függvények felhasználásával.

```
20. Console.WriteLine($"1. A héten {Osszes(oltasok)}
    kutya kapott oltást");
21. Console.WriteLine($"2. {(Volt(oltasok, 10) ? "Volt" : "Nem volt")}
    legalább tízkutyás nap.");
22. Console.WriteLine($"3. A legkevesebb kutya ezen a napon
    kapott oltást: " + Min(oltasok));
23. }
```

A függvényekben foreach-ciklust használunk, mert a `Dictionary` lista jellegű adatsorozat. Eből következik, hogy az eldöntés/keresés típusalgoritmusokban `break`; megszakítás kell.

1. Hány kutyát oltott be a héten az állatorvos?

```

24. private static int Osszes(Dictionary<string, int> oltasok)
25. {
26.     int szum = 0;
27.     foreach (KeyValuePair<string, int> oltas in oltasok)
28.         szum += oltas.Value;
29.     return szum;
30. }
31.

```

2. Volt-e olyan nap, amikor legalább tíz kutyát oltott be a doki?

```

32. private static bool Volt(Dictionary<string, int> oltasok, int minimum)
33. {
34.     bool volt = false;
35.     foreach (KeyValuePair<string, int> oltas in oltasok)
36.         if (oltas.Value >= minimum)
37.         {
38.             volt = true; /*38-39. sor helyett lehetne: return true;*/
39.             break;
40.         }
41.     return volt; /*itt pedig return false; és a 34. sor nem kell*/
42. }

```

3. Melyik napon oltotta be a legkevesebb állatot az állatorvos?

A bejárás ciklus miatt az első adat lekérdezése nem idevaló. Helyette adjunk meg egy lehetetlenül nagy „minimális” értéket. A legnagyobb lehetséges érték az egész típusnak a `MaxValue` tulajdonsága. (Ez a tulajdonság statikus, azaz nem valamelyik `int` típusú számnak, hanem magának az `int`-nek a tulajdonsága.)

A `KeyValuePair<string, int>` adattípus hosszú és nehezen jegyezhető meg. Enélkül írjuk meg a függvényt. A listaelem típusának kitalálását bízzuk a fordítóprogramra a `var` jelöléssel. Ugyanezt a trükköt az első minimumjelölt esetén nehézkes alkalmazni, ezért két változóban tároljuk a napot és a hozzá tartozó értéket.

```

43. private static string Min(Dictionary<string, int> oltasok)
44. {
45.     string minnap = "Hiányzik";
46.     int minertek = int.MaxValue; /*az int legnagyobb lehetséges értéke*/
47.     foreach (var oltas in oltasok)
48.         if (oltas.Value < minertek)
49.         {
50.             minnap = oltas.Key;
51.             minertek = oltas.Value;
52.         }
53.     return minnap; /*csak a nap neve kell*/
54. }
55. }}

```

FÁJLOK A KÉRÉSZÉLETŰ ADATOK HELYETT

A szemfülesebbeknek bizonyára feltűnt már, hogy az igazi programokra nem jellemző, hogy az adatokat az `.exe` programfájl tartalmazza. Valamivel gyakoribb, hogy néhány adatot egy program bekér a felhasználótól, de általában a beírhatónál sokkal több adattal dolgozik egy

program, amit vagy a háttértárról tölt be, vagy valamilyen más csatornán – például interneten, Bluetooth kapcsolaton keresztül vagy szenzortól kap. Az is jellemző, hogy a program által előállított adatokat ugyanezen csatornákon továbbítani lehet vagy el lehet menteni háttértárra. Mi eddig tárolt adatokat felhasználó programot nem írtunk, de épp itt az ideje ezen változtatni!

Elméletkedés: Szöveges és bináris fájlok – Bájtok és bitek értelmezése

Ha elég messziről – vagy inkább egy jegyzetombben – tekintünk a számítógépeken tárolt fájlokra, akkor alapvetően kétfélét különböztetünk meg. Az egyikben szöveg van, még hozzá formázatlanul; olyan, amely csak számokat, betűket, szóközöket, írásjeleket tartalmaz. Az ilyen fájlok kiterjesztése nagyon gyakran `txt`, az angol text, azaz 'szöveg' szó alapján. A `txt` kiterjesztésűeken kívül ebbe a csoportba tartozik még többek között a táblázatos adatokat tárolni képes `csv` fájl típus, valamint a weboldalak kódját tartalmazó `html` és `css` kiterjesztésű állomány és még jó néhány fájl típus. Szöveget tartalmazó fájlokat készíthetünk az úgynevezett egyszerű szerkesztőkkel, például a Windows Jegyzetömbjével, de ilyen fájlt ír minden IDE. Visual Studiot használva ilyen a `Program.cs`, `*.sln`, `*.csproj` fájl is.

A „másik” fájl típus – jegyzetombben olvasás szempontjából – az összes többi. Kiterjesztéseik és a bennük tárolt adatok rendkívül változatosak, de nagyon gyakori, hogy a tárolt adatok között – sokszor pont a legelején – találunk útmutatást a kódolás típusára. Idetartoznak a lefordított programok (bináris fájlok). Például a MS `exe` és `dll` fájljainak elején MZ az első két bájt, az ezt követő kódokat viszont már csak a processzor érti meg. A videók, a kép- és a hangfájlok elején szintén gyakori a formátumra jellemző jelzés, a signature. Több alkalmazás binárisnak látszó szöveges fájlt készít. Van, ahol titkosítással, van, ahol tömörítéssel teszik olvashatatlanná a kódot. Például a `micro:bit` `*.hex` fájlja tizenhatos számrendszerben kódolva tartalmazza a programot, végrehajtható kódhoz képest egész olvashatóan. A zip tömörítés eredménye olvashatatlan akkor is, ha `txt`-t tömörítünk, de az első két bájtja PK árulkodik a csomagolás módjáról. Ránézésre – PK kezdetű – bináris fájl, de a „lelke mélyén” szövegfájlokból áll a `*.docx`, `*.xlsx`, `*.pptx`: a (másolat) kiterjesztését zip-re módosítva mappákba rendezett `xml` fájlok, esetleg képfájlokat találunk bennük.

Eddigi programjainkban konzolról kértünk be adatokat, ez mindig billentyűleütések sorozatát jelentette, azaz karaktersorozat, szöveg formájában kapta meg a program az adatokat. Az adatsorozat olvasása és (konzolra, képernyőre) írása során a program így-úgy kezeli az ékezetes karaktereket, beállítható, hogy melyik karakterkódolást – ASCII, UTF-8... – használja. Az első példában – és az összes vizsgán, programozási versenyen – ékezetmentes adatokkal dolgozunk és az alapértelmezett beállításokat használjuk, de az életben adódó problémák megoldásához szükséges lehet a kódtábla módosítása.

Szöveges fájl kezelésekor kikerülhetetlen az ENTER többféle kódjának az értelmezése. Például Windows esetén a 13-as ('`\r`', CR, Carriage Return) és 10-es ('`\n`', LF, Line Feed) binárisan kódolt, nem nyomtatható (két) karakter jelenti az ENTER-t, de, ha egy fájlt – például hálózaton keresztül – több operációsrendszerben szeretnénk használni, akkor lehet, hogy csak a két kód egyike jelzi az ENTER-t, esetleg a két kód fordított sorrendben van. Szöveges fájl feldolgozásához nem csak az ékezetes karaktereket, hanem az ENTER négyféle alakját – '`\r\n`' (Win), '`\n`' (Unix, Mac), '`\r`' (régí Mac), '`\n\r`' (régí mikrokontroller) – is kezelnie kell a fájl olvasását vagy írását végző eszköznek. Első közelítésben egyszerű lenne a két leggyakoribb eset kezelése, de a '`\n`' Windows rendszerekben – például Wordben – a sortörés kódja és így egy Windowson

futó program nem tudja eldönteni, hogy a kapott karaktersorozatban azért van `'\n'`, mert Linux típusú operációs rendszeren készült a szöveg vagy azért, mert egy Windowson futó programban írtak sortörést. A szövegkezelő eszköz a keletkezés módjának ismerete nélkül fog dönteni, hogy átalakítja-e `\r\n` alakúra. Hasonlóan visszafelé, a `\r\n` helyettesíthető egy `'\n'`-nel, de egy ősbib jelölési mód miatt lehet, hogy `\n\n` lesz belőle.

A szövegbeolvasó eszközök a beolvasás során értelmezik a kódot, ebből következik, hogy a fájlban és a program változóiban nem pontosan ugyanaz a bitsorozat lesz.

A fejezet további részében szövegfájlokkal, az ezekben tárolt adatok felhasználásával foglalkozunk, de kitekintésként a legegyszerűbb binárisan kódolt kép fájl, a `*.bmp` konzol alkalmazásból történő módosítását is ki lehet próbálni. A jegyzet legvégén, a grafikus alkalmazás készítése során pedig természetes lesz a képek felhasználása a programunkban.

A leckénknek minden példájában szöveges fájlként az alábbi, `allatok.txt` fájlt használjuk, amely megtalálható a tankönyv weblapjáról letöltött fájlok között. (Ha nem találjuk, akkor gépeljük be txt fájlba.) A fájlban kedvenc tanyánk háziállatai vannak felsorolva. Érdekes tudni, hogy a fájl tizenöt sorból áll, és Latyak kacsza sora után nem kezdünk új sort, nem nyomunk Entert. Az egyes sorokban találjuk az állat nevét, azt, hogy miféle állatról van szó (a későbbiekben az egyszerűség kedvéért sokszor és pontatlanul fajtának nevezzük ezt a tulajdonságot), és azt is, hogy ez az állat hány éves.

```
Totyak kacsza 1
Lakli kutya 12
Hektor kutya 4
Puha birka 3
Nyeldekel kecske 5
Csinibaba szarvasmarha 3
Nyakas liba 2
Dinka malac 1
Coca malac 1
Smafu malac 3
Tarajos kakas 1
Csillag macska 2
Zokni macska 3
Pamacs birka 4
Latyak kacsza 2
```

Ismétlés: Szöveges fájlban tárolt adatok felhasználása konzolon keresztül

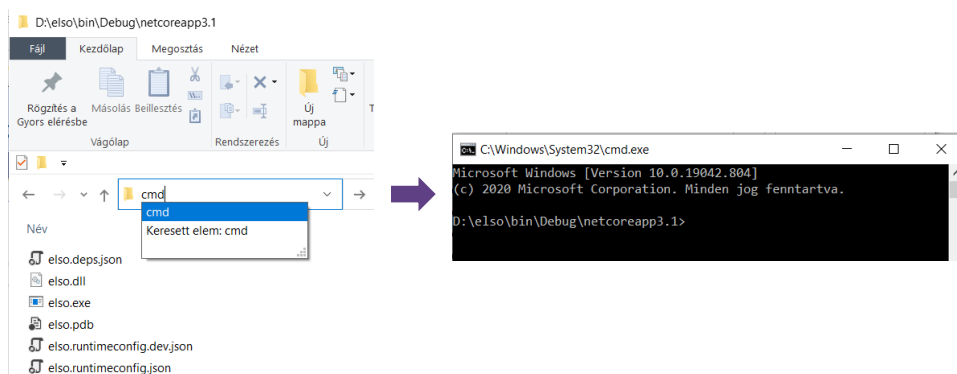
A kilencedikes jegyzetben már volt szó arról, hogy hogyan lehet hosszabb (szöveges) adatokat felhasználni a programunkban. Az egyik módszer a fájlból konzolra másolás, a másik a konzol átirányítása. Lényeges részletek:

A fájlból konzolra másoláshoz egy egyszerű szöveges fájlban eltároljuk, amit majd be szeretnénk írni. A program adatsorokat olvas, ezért a beíráskor ennek megfelelően, enterrel töröljük sorokra az adatainkat. A program futtatásakor a szöveget kimásoljuk és a konzolra beillesztjük: az egérrel jobb-klikk, Beillesztés lehetősége vagy a `CTRL+V` billentyűkombináció is alkalmas. Programunk nem fog sokkot kapni 1000 adatsortól még akkor sem, ha csak egyetlen számot vár – főleg, ha a bemásolt adatok első sorában valóban egy szám van –, mert a bájtok kiolvasását az első enterig végzi. A többi adat a konzol csatornában vár – pont úgy, mint a

leütött billentyűk sorozata –, amíg sorra kerül, vagy amíg befejeződik a program. A program bezárásakor a csatorna is megszűnik, így a benne maradt adatok is törlődnek.

A konzol átirányításához előzetesen el kell készítenünk a programot (az exe kiterjesztésű fájlt), célszerű mellé másolni az adatokat tartalmazó szöveges fájlt. Az átirányítást az operációs rendszer végzi, ezért nem az IDE-ből futtatjuk ilyenkor a programunkat, hanem a fordítás után, egy konzolablakban (terminál ablakban) indítjuk el a programunkat:

- Keressük meg fájlkezelőben az .exe fájlt, de ne kattintsunk rá, csak ellenőrizzük, hogy a .txt is a mappában van!
- A fájlkezelő címsorába az elérési út helyére írjuk be: `cmd` és üssünk ENTER-t. A `cmd` parancs megnyit egy konzolablakot és – mivel az elérési út helyére írtuk – épp a megadott mappában várja az utasításunkat.



3

- A program fájl nevének beírásával indítható.

A fájl „becsatornázása” a programba a ‘<’ jellel oldható meg. Például így indítjuk a programot:

```
program.exe < allatok.txt
```

A kiírást is átirányíthatjuk, ekkor nem a képernyőre, hanem a fájlba ír a programunk:

```
program.exe > eredmeny.txt
```

Lehet egyszerre mindkettőt használni, a bemenő adatokból előállítani az eredményfájlt:

```
program.exe < allatok.txt > eredmeny.txt
```

Lényegében ezt használják ki, amikor egy programot tesztelnek.

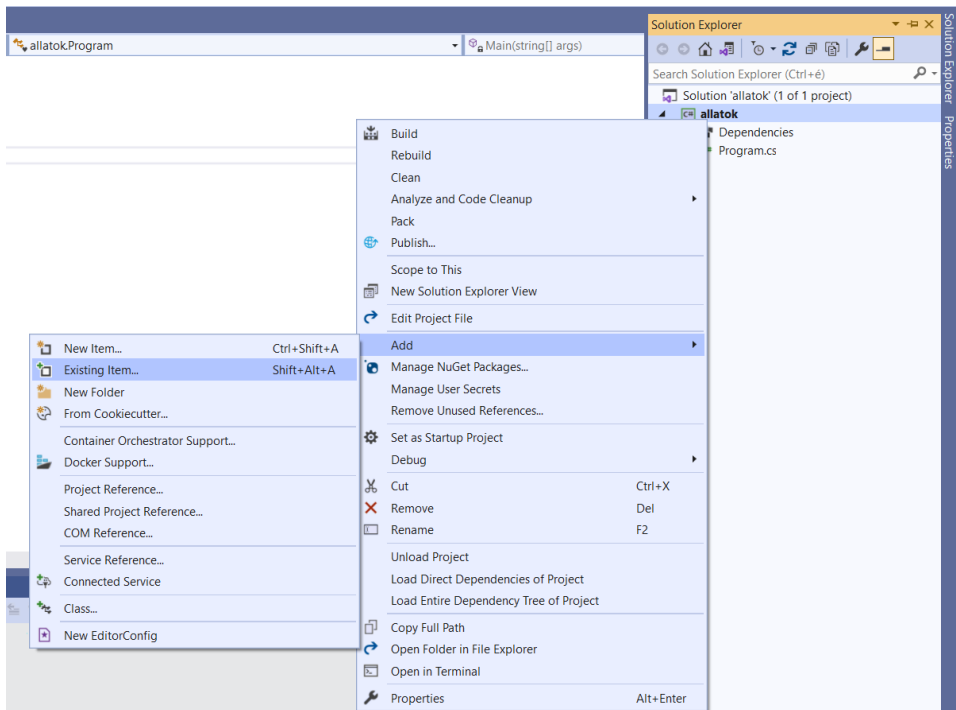
Fájlok hozzáadása a megoldáshoz

Az előző két módszernél az adatokat a felhasználó, illetve az operációsrendszer adta át a programnak, a konzol csatornán keresztül. Fájlok beolvasásáról akkor beszélünk, ha a program nyúl a fájlért, a program nyitja meg és kezdeményezi a fájl betöltését a háttértárról. A művelet kritikus része, hogy a program megtalálja-e a fájlt. Tipikus hibalehetőség, ha a fájl elérési út-vonalát a gyökérkönyvtártól elindulva adjuk meg, mert így csak az adott gépen működhet a programunk (vagy ott se, egy mappanévv módosítása után). Relatív, a programhoz képest leg-egegyszerűbb elérési útvonala az, amikor nem kell útvonalaat írni, csak a fájl nevét. Ezt úgy érjük

³ A kilencedikes jegyzetben szereplő kép.

el, ha az .exe fájl mellé másoljuk a szöveges fájlt. Apró probléma, hogy kezdetben nincs .exe fájl, esetleg a mappája is hiányzik, hiszen az csak az első futtatás során jön létre. Ezért hasznos az IDE által generált – még semmit sem tartalmazó – programkódot egyszer lefuttatni.

További hibalehetőséget jelent, ha a futtatás módján változtatunk, az alapértelmezett Debug módról Release módra váltunk. Ilyenkor új, Release mappák jönnek létre ezen belül lesz a programfájl is. A Visual Studio – és általában a programot projektként vagy solution-ként kezelő IDE – képes arra, hogy a program által használt segédfájlokat kezelje. Ehhez az kell, hogy a felhasználandó fájlt a projekt részévé tegyük.

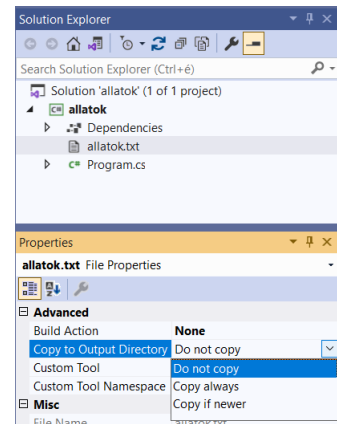


A fenti képen jobbról balra haladva:

- a Solution Explorerben kiválasztjuk a (félkövér) projektet;
- A projekten jobb-klikkel megnyitott menüből az Add menün belül...
- az Existing Item... lehetőséget választjuk;
- a megjelenő ablak alján beállítjuk, hogy minden fájlípust listázzon: Fájlnev... All files (*.*);
- kiválasztjuk a forrásfájlt, ami így a Program.cs fájl mellé lesz másolva.

Ezután a Solution Explorerben a szöveges fájlt kiválasztva megnyitjuk a tulajdonság ablakát (Properties) és beállítjuk, hogy a fájlt az IDE másolja oda, ahova a program is kerül. (Copy to Output Directory). A célnak bármelyik másolás (Copy...) lehetőség megfelel.

Ha rendelkezésre áll a fájl, akkor nekiállhatunk első fájlbeolvasó programunk megírásának.



Fájl beolvasása, adatok tárolása

24. példa: Fájlból be, konzolra ki

Az első programunk mindössze annyit tesz, hogy megnyitja a fájlt, beolvassa és kiírja konzolra a beolvasott sorokat, majd bezárja a fájlt. Ehhez a `StreamReader` típusú eszközre lesz szükségünk, ami a `System.IO` névtérben található. Az IO – input, output – névtér tartalmazza a meg-hajtók, mappák és fájlok eléréséhez és kezeléséhez szükséges eszközöket. Ezek közül a `StreamReader` – folyam-olvasó – a karaktersorozatok, azaz a szöveges fájlok beolvasását szolgálja. Működése a `Console` beolvasó részéhez hasonló: a fájl elejétől sorban olvassa a sorokat, karaktereket. Nem tud sem kihagyni, sem visszalépni. Plusz képessége, hogy felismeri a fájl végét – az `EndOfStream` tulajdonsága ekkor igaz lesz –, még mielőtt nemlétező adatot próbálna kiolvasni. A `StreamReader` működése idején a szövegfájl más programok használatban lévőnek – foglaltnak – látják, ezért a beolvasást követően a foglalást meg kell szüntetni, a kapcsolatot a `Close()` függvénnyel le kell zárni.

```
1. using System;
2. using System.IO;           /*Nem szabad elfelejteni, hogy ez is kell !*/
3. namespace allatok
4. {
5.     class Program
6.     {
7.         private static void Beki()
8.         { /*az olvaso nevű StreamReader az allatok.txt-t megnyitja*/
9.             StreamReader olvaso = new StreamReader("allatok.txt");
10.            while (!olvaso.EndOfStream) /*amíg nincs 'vége' jelzés*/
11.                Console.WriteLine(olvaso.ReadLine());
12.            olvaso.Close(); /*végül bezárja*/
13.        }
14.    }
15. }
```

A Console kiírja a képernyőre azt, amit az olvasó a fájlból beolvas

A `StreamReader` létrehozásakor csak a fájl nevét adtuk meg, mert az `allatok.txt` az `allatok.exe` mellett található. Ha a fájl elhelyezésénél vagy a megnevezésében hiba van, akkor futtatáskor `System.IO.FileNotFoundException` hibát jelez a Visual Studio.

25. példa: Fájlból beolvasott adatok tárolása

Ha a fájlban lévő adatokat tárolni szeretnénk, akkor a beolvasás rész nagyon hasonló, de az eltárolásra több megoldásunk lehet, amelyek különböző megfontolásokat igényelnek. A legjellemzőbb kérdés, hogy tudjuk-e, hogy hány sort kell beolvasnunk, eltárolnunk.

1. A fájl első sorában található a beolvasandó adatok száma

Mivel a fájl végéig olvasást az egyes programozási nyelvek nagyon eltérő módon kezelik, tipikus adattárolási módszer, hogy az adatsorokat tartalmazó fájl első sorában szerepel az adatsorok száma. Az első megoldásunkhoz ezért a `15allat.txt` fájlt használjuk, aminek az első sorába beírjuk, hogy 15.

Az adatok számára az első sor beolvasása után hozzuk létre a megfelelő méretű (15) tömböt. Az eredmény ellenőrzéseként, a fájl lezárása után az adatokat képernyőre írjuk.

```

8. private static void Be15()
9. {
10.     StreamReader olvaso = new StreamReader("15allat.txt");
11.     int N = int.Parse(olvaso.ReadLine());
12.     string[] sorok = new string[N];
13.     for (int i = 0; i < N; i++)
14.         sorok[i] = olvaso.ReadLine();
15.     olvaso.Close();
16.     for (int i = 0; i < N; i++)
17.         Console.WriteLine(sorok[i]);
18. }
19.

```

2. Egyszerre beolvassunk mindent, majd a programban sorokra tördeljük

A `StreamReader` nem csak sort tud olvasni, hanem teljes fájlt is, a `ReadToEnd()` függvénnyel. Az eredmény egyetlen `string`, amit ezután a `Split()` függvénnyel feloszthatunk. A függvény elintézi, hogy pont jó méretű legyen a tömb. A megoldás egyszerűnek látszik, de nem biztonságos, mert nem tudjuk, hogy mi határolja a sorokat.

```

20. private static void BeEgybe()
21. {
22.     StreamReader olvaso = new StreamReader("allatok.txt");
23.     string fajl = olvaso.ReadToEnd();
24.     Console.WriteLine("A beolvasott fajl");
25.     Console.WriteLine(fajl);
26.     Console.WriteLine();
27.     string[] teszt = fajl.Split('\n');
28.     for (int i = 0; i < teszt.Length; i++)
29.         Console.WriteLine(teszt[i]);
30. }
31.

```

- Ha a `Split('\n')`-t használjuk, akkor a sor végén ottmaradhat a `\r` (13-as kódú „kocsi vissza”), ettől minden sor egy láthatatlan karakterrel hosszabb lesz, amire figyelni kell. Például a `Write()`-tal kiíráskor egymás mögé írás helyett egymást felülírják a sorok.
- Ha a `Split("\r\n")`-t használjuk, akkor lehet, hogy nem történik semmi mert a fájlt olyan programmal is készíthették, ami csak a `\n`-t írja a sor végére.
- Ha a `Split()` paraméter nélküli alakját használjuk, akkor minden szóköznél is felbontja a szöveget, így nem 15, hanem 45 soros lesz az eredmény.

Hogy éppen melyik megoldás jó? Ez további felhasználás módjától függ.

3. Nem tudjuk, hány sor? Tegyük az adatokat listába.

Jó és egyszerű megoldás, ha jól tudjuk használni a `List<>` adattípust.

```

32. private static void BeTarba()
33. {
34.     List<string> tesztl = new List<string>();
35.     StreamReader olvaso = new StreamReader("allatok.txt");
36.     while (!olvaso.EndOfStream)
37.         tesztl.Add(olvaso.ReadLine());
38.     olvaso.Close();
39.     foreach (string s in tesztl)
40.         Console.WriteLine(s);
41. }
42.

```

4. Összetett adatok statikus tömbjének feltöltése fájlból

A fájl egy-egy sorában egy-egy állat neve, faja és kora szerepel. Ennek megfelelően a tárolását két `string` és egy `int` típusú adatot tartalmazó struktúrában illik megoldani. Minél összetettebb egy adat, annál nagyobb az futási időbeli különbség a tömb és a lista között, a tömb javára. Ha a tömb méretét nem tudjuk, akkor időn úgy spórolhatunk, hogy már a program legelején lefoglalunk egy kellően nagy tömböt (persze ez memóriában nem lesz olyan jó, de általában nem műholdra írjuk a programunkat, ahol minden bájtt lefoglalása számít).

```

43. struct Allatka
44. {
45.     public string Nev;
46.     public string Faj;
47.     public int Kor;
48. }
49. static Allatka[] allatos = new Allatka[100];
50.

```

Az itt következő megoldásban az adatok tárolására egy kellően nagy méretű, minden más programrész által használható tömböt hozunk létre, ami a program indulásától rendelkezésre áll. Ebbe írjuk be a fájlból az adatokat. Később majd tudnunk kell, hogy meddig vannak a tömbben adatok – a többi üres, hibát okozhat, ha azokkal is számolunk –, ezért a megoldásunk egy függvény, aminek az eredménye a beolvasott és eltárolt sorok száma lesz.

```

51. private static int BeMinimal()
52. {
53.     StreamReader olvaso = new StreamReader("allatok.txt");
54.     int db = 0;
55.     while (!olvaso.EndOfStream)
56.     {
57.         string sor = olvaso.ReadLine();
58.         string[] adatok = sor.Split(' ');
59.         allatos[db].Nev = adatok[0];
60.         allatos[db].Faj = adatok[1];
61.         allatos[db].Kor = int.Parse(adatok[2]);
62.         db++;
63.     }
64.     olvaso.Close();
65.     return db;
66. }
67.

```

A függvény felhasználása a `Main()` eljárásban:

```
int NMinimal = BeMinimal(); /*közben módosítja az allatos tömböt is*/
for (int i = 0; i < NMinimal; i++)
    Console.Write(allatos[i].Nev + " ");
Console.WriteLine();
```

5. Összetett adatok helyi tömbjének feltöltése fájlból

Ez a megoldás a beolvasás szempontjából nem tartalmaz újdonságot, de a sikerességet erősen befolyásolja, ha a feltöltendő tömb nem a konkrét programrészben (mint az első három megoldásban) és nem is globálisan, a függvényeken kívül (mint a 4. megoldásban) található, hanem egy másik függvény vagy eljárásban. Jelen esetben a `Main()` eljárásban hozzuk létre a 100 elemű `allatkak` tömböt és ebbe töltjük be az adatokat a `BeApro()` eljárással.

```
Allatka[] allatkak = new Allatka[100]; /*van 100 adat, ezért ref kell*/
int NApro; /*Nincs kezdőérték, ezért out kell*/
BeApro(allatkak, out NApro);
for (int i = 0; i < NApro; i++)
    Console.Write(allatkak[i].Nev + " ");
Console.WriteLine();
```

Mivel a sorok számának a másolata kerülne az eljárás paraméterébe, fontos jelezni, hogy értéket szeretnénk adni a darabszámnak – ezt jelöli az `out`. Enélkül is futtatható a programunk, csak éppen nem lesz eredménye. Az eljáráson belül módosulnak az adatok, de a módosulás az eljárás végén törlődő másolaton történik, nem az eredeti példányon.

A `BeApro()` eljárásban az előző megoldáshoz képest alig van változás:

```
68. /*kisallatok: megfelelő méretű Allatka adatokra készített tömb*/
69. private static void BeApro(Allatka[] kisallatok, out int db)
70. {
71.     StreamReader olvaso = new StreamReader("allatok.txt");
72.     db = 0;
73.     while (!olvaso.EndOfStream)
74.     {
75.         string sor = olvaso.ReadLine();
76.         string[] adatok = sor.Split(' ');
77.         Allatka ez;
78.         ez.Nev = adatok[0];
79.         ez.Faj = adatok[1];
80.         ez.Kor = int.Parse(adatok[2]);
81.         kisallatok[db] = ez;
82.         db++;
83.     }
84.     olvaso.Close();
85. }
86.
```

6. Összetett adat létrehozása fájlból beolvasott sorból, egylépésben.

Az előző két megoldásban a beolvasások ciklusának maga öt sor. Ezt rövidíthetjük le azzal, ha praktikus konstruktort írunk a struktúrához. A korábbi – ismétlődő – feladatokban a konstruktor szignatúrájában (bemenő adatainál) az adattagoknak megfelelő típusok sorát adtuk meg. Most olyan konstruktor lenne praktikus, ami vagy az adatsort – egy `string`-et – vagy a szavakra bontott adatsort – `string[]` tömböt – használja az adat előállítására. Alábbiakban az első megoldást választjuk.

```

87. struct Allat
88. {
89.     public string Nev;
90.     public string Faj;
91.     public int Kor;
92.     public Allat(string sor)
93.     {
94.         string[] adatok = sor.Split(' ');
95.         Nev = adatok[0];
96.         Faj = adatok[1];
97.         Kor = int.Parse(adatok[2]);
98.     }
99. }
100.

```

Ezután a beolvasás jelentősen lerövidül. A változatosság kedvéért most statikusan létrehozott listába olvassuk be az adatokat. A kód a legelső listás megoldásunkhoz hasonlóan egyszerű, de az eredmény egy sokkal jobban használható adattömb:

```

101. static List<Allat> allatList; /*Csak megnevezés, nincs létrehozva*/
102. private static void BeListbe()
103. {
104.     allatList = new List<Allat>();
105.     StreamReader olvaso = new StreamReader("allatok.txt");
106.     while (!olvaso.EndOfStream)
107.         allatList.Add(new Allat(olvaso.ReadLine()));
108.     olvaso.Close();
109. }
110.

```

A `Main()` eljárásban, miután ezzel az eljárással beolvastuk az adatokat, ki tudjuk írni az állatok neveit:

```

BeListbe();
foreach (Allat allat in allatlist)
    Console.WriteLine(allat.Nev);

```

7. Beolvasás fájlból paraméteres eljárással

Van, hogy egy programban a felhasználó adhatja meg a beolvasandó fájl nevét. Az is lehet, hogy egy programban több fájlból több tömbbe kell adatokat beolvasnunk. Most olyan eljárást írunk, amelyben az adatok forrásának és végső tárolásának a helyét is paraméteresen adjuk meg. A felhasználás helyén hozzuk létre a tömböt és beállítjuk a rekordok számát nullára

```

const int MaxN = 100; /*A program legelején szokott lenni.*/
Allat[] allataim = new Allat[MaxN];
int Ndb = 0; /*komoly galibát okoz, ha ez nem 0*/
BeNdb("allatok.txt", allataim, ref Ndb);
for (int i = 0; i < Ndb; i++)
    Console.Write(allataim[i].Nev + "; ");
Console.WriteLine();

```

Az `Ndb`-nek értékadás így nem egészséges, mert ha elfelejtjük, vagy rossz értéket adunk, akkor meghal a programunk. Figyeljünk meg a kódban, hogy miért. Miért jobb az out?


```

111. private static void BeNdb(string fn, Allat[] allatok, ref int N)
112. {
113.     StreamReader olvaso = new StreamReader(fn);
114.     while (!olvaso.EndOfStream)
115.     {
116.         allatok[N] = new Allat(olvaso.ReadLine());
117.         N++;
118.     }
119.     olvaso.Close();
120. }
121.

```

8. Hasznos trükk ismeretlen mennyiségű adat egyszerre beolvasására.

A System.IO egy másik eszköze a `File`. Ezzel lehet fájlokat létrehozni, törölni, másolni, áthelyezni... és – meglepő módon – a fájlból minden sort beolvasni szövegtömbbe. A 2. megoldásunkat 1 sorban – és hibamentesen – helyettesíti a `ReadAllLines()`.

```

122. static Allat[] allattombs;           /*itt még nincs mérete*/
123. private static void BeFileSorok(string fajlnev)
124. {
125.     string[] sorok = File.ReadAllLines(fajlnev);
126.     allattombs = new Allat[sorok.Length]; /*static, itt jön létre*/
127.     for (int i = 0; i < sorok.Length; i++)
128.         allattombs[i] = new Allat(sorok[i]);
129. }
130.

```

A megoldás egyetlen hátránya, hogy a függvény futásakor az összes adatot két példányban tároljuk. Előnye, hogy a létrejövő tömb pont olyan hosszú, amilyen az adatok számára kell.

9. Tömb létrehozása fájlból beolvasott adatokból.

Végül jöjjön egy kompakt megoldás, amiben a bemeneten megadjuk a beolvasandó fájl nevét, az eredmény pedig egy tömb, a beolvasott adatokkal. A tömb mérete megegyezik a beolvasott adatok számával, ezért foreach-ciklussal is kezelhető.

```

Allat[] allattomb = BeOlvas("allatok.txt");
foreach (Allat allat in allattomb)
    Console.Write(allat.Nev + ", ");
Console.WriteLine("\b\b.");

```

A `BeOlvas()` kódjában kétszer olvassuk be a fájlt. Először csak az adatsorok száma miatt, majd újra megnyitva, beolvassuk a sorokat.

```

131. private static Allat[] BeOlvas(string fajlnev)
132. {
133.     int N = File.ReadAllLines(fajlnev).Length;
134.     Console.WriteLine(N); /*nem kell, csak a tesztelés miatt*/
135.     Allat[] allatok = new Allat[N];
136.     StreamReader olvaso = new StreamReader(fajlnev);
137.     for (int i = 0; i < allatok.Length; i++)
138.         allatok[i] = new Allat(olvaso.ReadLine());
139.     olvaso.Close();
140.     return allatok;
141. }
142.

```

Így nem tároljuk kétszer az adatokat, viszont kétszer olvasunk be a háttértárról. Egy SSD háttértár esetén talán nem számít, annyira, de általában nagyságrendekkel több idő kell a háttértárról olvasáshoz, mint a memóriában tárolt adat eléréséhez. Ismét láthatjuk, hogy a lefoglalt memória mérete és a futási idő egymás rovására csökkenthető.

26. példa: Fájlból beolvasott adatok felhasználása – legöregebb állat

A típusalgoritmusainkat természetesen a fájlokból beolvasott adatokkal is tudjuk használni. Ha például a beolvasást követően ki szeretnénk írni a legöregebb állat nevét és fajtáját, akkor a fenti megoldások közül bármelyiket nagyjából hasonlóan kell kiegészítenünk.

Az 1–3. megoldásokban az eljáráson belül folytathatjuk a kód írását, vagy az eljárások kódját a `Main()` eljárásba tesszük és ott folytatjuk a programunkat. A továbbiakban nehézséget okozhat, hogy minden vizsgálatot meg kell előzze az adatsor felbontása és a kor egész számmá alakítása.

A 4–9. megoldásokat lefuttatása után a főprogramban is megírhatjuk legidősebb kiválasztását, vagy bármelyik adatsorozatot átadhatjuk egy függvénynek.

Figyeljünk arra, hogy a 4., 5. és 7. beolvasás során nem csak a tömböt kell átadnunk, hanem a beolvasott adatok számát is. Ezekben az adattároló jóval nagyobb, mint a beolvasott adatok száma.

A 3. és 6. megoldásban listát használtunk, a többiben tömböt. Bár az adatok bevitele után a lista és a tömb használata között csekély az eltérés, nem mindegy, hogy milyen adattípusra írjuk meg a megoldást. Az eredmény típusánál is meg kell különböztetni az 1–3. feladatok `string`-jét, a 4–5. feladatok `Allatka` struktúráját és az 5–9. feladatok `Allat` struktúráját.

A példamegoldást az utolsó beolvasáshoz írjuk, a maximumkiválasztás típusalgoritmusának alkalmazásával:

```
143. private static Allat Legoregebb(Allat[] allatok)
144. {
145.     int maxi = 0;
146.     for (int i = 0; i < allatok.Length; i++)
147.         if (allatok[i].Kor > allatok[maxi].Kor)
148.             maxi = i;
149.     return allatok[maxi];
150. }
151.
```

Szövegfájlok írása

27. példa: Legöregebb állat adatainak fájlba írása

```
Fajlbair(Legoregebb(allattomb), "oreg.txt");
```

A szövegfájlok írása egyszerűbb az olvasásuknál. A fájl megnyitásához `StreamWriter` kell, ennek konstruktorában megadjuk a fájl nevét. Ha a fájl már létezik, akkor az törlődni fog, a `StreamWriter` új fájlt hoz létre. A fájlba írás eljárása és tulajdonságai is pontosan megegyeznek a `Console`-ra történő kiírással. Fontos azonban, hogy a fájl lezárásáig csak a memóriába kerülnek be az adatok, a `Close()` a bezárás előtt – és csak ekkor – elmenti a fájlt, majd felszabadítja a többi alkalmazás számára. Ha elfelejtjük a `Close()`-t, akkor nem lesz kimeneti fájl.

```

152. private static void Fajlbair(Allat allat, string fajlnev)
153. {
154.     StreamWriter iro = new StreamWriter(fajlnev);
155.     iro.WriteLine(allat.Nev + " " + allat.Kor);
156.     iro.Close();
157. }

```

Ebben a fejezetben a fájlból olvasásra az `olvaso` változót használtuk, az írásra az `iro`-t. Ha interneten vagy szakkönyvben olvasunk fájlműveletekre C# kódokat, akkor a fájlból olvasásra az `sr`, a fájlba írásra az `sw` változóneveket használják. Ez majdnem annyira gyakori, mint a ciklusváltozónak az `i` változónév. A modern programozó csapatban dolgozik, a közös munkát könnyíti meg, ha az elnevezések szabványosak. A `struct`, `class` és a függvénynevek nagybetűvel kezdődnek, a többszavas elnevezéseket nagy kezdőbetűvel, szóközök nélkül írják. Egy `struct` vagy `class` objektum neve, ha más hasonló nincs, akkor a típusának kisbetűs rövidítése lesz. Így a `StreamReader` típusú változó `sr`, a `StreamWriter` `sw`. Egyszavas megnevezések esetén gyakori, hogy a változónév a típus kisbetűs formája. Például `Allat allat`, egyszerű adat esetén az első betű, például `string s`, `char c`. Érdekes ezekhez a jelölésekhez alkalmazkodni, mert megkönnyíti a programkód értelmezését.

Ékezetes betűket tartalmazó szöveg beolvasása és fájlba írása

A különböző operációsrendszerek, kódolások és online alkalmazások körében az a biztos, ha a szöveg nem tartalmaz ékezetes betűket, a fájlnev (sőt, az elérési út sem) tartalmaz semmilyen extra karaktert az angol ábécé betűin és számokon kívül. De érdemes kipróbálni, hogy az éppen használt fejlesztési környezetben tudunk-e ékezetes karakterekkel dolgozni.

28. példa: Ékezetes állatok olvasása fájlból és kiírása fájlba

Készítsünk egy új szöveges fájlt `állatok.txt` néven (á-val):

```

Hápi kacs a 1
Morgó kutya 12
Hekus kutya 4
Bégő birka 3
Fűtfal kecske 5
Múzós szarvasmarha 3
Hínár liba 2
Különc malac 1

```

Olvassuk be a fájlt, írjuk ki az állatok nevét képernyőre vesszővel felsorolva és írjuk ki soronként a neveket és a fajtájukat fájlba.

A megoldást – mivel csak tesztelés a cél – az adatok eltávolítása nélkül adjuk meg. Ehhez egyszerre nyitjuk meg a fájlt olvasásra és egy másik fájlt írásra. Mivel a kor értéke nem számít a megoldásban, ezért ezt nem alakítjuk át számmá.

A megoldáshoz készítsünk új programot, ne felejtjük el, hogy a fájl elérhetővé kell tennünk és a `System.IO` névtérre is szükségünk lesz.

```

7. private static void ÉkezetesOlvasIr()
8. {
9.     StreamReader sr = new StreamReader("állatok.txt");
10.    StreamWriter sw = new StreamWriter("névsor.txt");

```

```

11. while (!sr.EndOfStream)
12. {
13.     string[] sor = sr.ReadLine().Split(' ');
14.     Console.Write(sor[0] + ", ");
15.     sw.WriteLine(sor[0] + " " + sor[1]);
16. }
17. sr.Close();
18. sw.Close();
19. Console.WriteLine("\b\b.");
20. }

```

Ellenőrizzük, hogy a `névsor.txt` létrejött-e, és helyesen szerepelnek-e benne a nevek.

Kitekintés: Bináris fájlok olvasása és írása

Ez a fejezet tényleg nagy kitekintés lesz a tankönyvi anyaghoz képest. Érdekes azonban elolvasni, mert választ ad arra a kérdésre, hogy miért nem használjuk a `Console.Read()` és a `sr.Read()` függvényeket a szövegfájlok olvasásához, majd – akár megértve, akár csak átmásolva a megfelelő kódrészleteket – „feltörünk” és „meghackelünk” egy `bmp` formátumú képet.

Amikor egy fájlban binárisan tárolunk adatokat (adatot, nem programéli utasításokat), akkor a tárolt bájtok a programban használt formátumban őrizik meg az adatot. A `byte` típusú `12` a `00001100` bitsorozat lesz. Az `int` típusú `12` négy bájtot foglal el, ebből 3 bájt csak 0 bitet tartalmaz, továbbá vagy az első vagy a negyedik bájt lesz `00001100`. A helyes értelmezéshez tudni kell, hogy a tárolás milyen bájtrendben történt: `little-endian` esetén az első bájt, `big-endian` esetén a negyedik bájt tartalmazza az 1-es biteket. Ha a `"12"`-t szöveges formában tároljuk, akkor két bájtot foglal el, az `'1'` bináris alakja `00110001` (ASCII kódja 49), a `'2'`-é `00110010` (ASCII kódja 50). Az adatokat csak úgy tudjuk helyesen értelmezni, ha tudjuk, hogy honnantól, hány bájtot kell venni, azaz milyen bájtrendben vannak a bájtok (`little-endian` vagy `big-endian`) és milyen típusuként kezeljük (`int`, `double`...).

A különböző karakterkódolások esetén – azaz bármely kódtáblát választhatjuk is – egyértelmű a `[0; 127]` tartomány kódjainak értelmezése. Ezek az ASCII kódok. Ezen belül, a `[32; 126]` tartományban nyomtatható karakterek találhatók. Ezeket az egybájtos értékeket (nyomtatható karaktereket) egy binárisan kódolt fájlból karakterként kiolvastva ugyanazt kapjuk, mintha szöveges fájlból olvasnánk ki. Ezért egy binárisan kódolt fájl jegyzettömbben megnyitva láthatunk értelmes szövegrészleteket. A `[0; 32]` és a `127-es` kódú karakterek vezérlő karakterek, amit egyes szövegszerkesztők (például a Notepad++) a „rejtett karakterek megjelenítése” nézetben jeleznek (például a `13-as` kód a `CR`). A `[128; 255]` tartományban lévő értékek szöveggént történő értelmezése környezetfüggő.

Például, ha binárisan kódoljuk a `65-ös int` értéket, akkor Notepad++-ban ez `A[NUL][NUL][NUL]` formában lesz olvasható. A `130-nak` például `☐[NUL][NUL][NUL]` lehet a szöveges megjelenése, a `260-nak` `[EOT][SOH][NUL][NUL]`.

⁴ A `little-endian` és `big-endian` szakkifejezések, egyben utalások a Gulliver utazásaiban olvasható háborúra, amely arról szólt, hogy melyik végén kell a főtt tojásokat feltörni (elkezdeni a pucolását).

Nézzük meg egy bitmap fájl kódolását!

A képfájloknak kétféle típusát ismertük meg: a pixelgrafikust és a vektorgrafikust. Vektorgrafikus formátum például az `svg`, ami szöveges formában tárolja a kép adatait. Pixelgrafikus a `png`, `jpg`, `gif`, `bmp`. Ezek közül a `gif` és a `jpg` veszteséges tömörítéssel menti a bitmap adatait, a `png` veszteségmentesen tömörít. Akárhogy is, egy tömörített állományból ránézésre nagyon nehéz lenne megállapítani, hogy mi volt a bitmap-en. A `bmp` csak akkor tömörít, ha csökkentett színekészlettel mentjük el (fekete-fehér, 16 színű vagy 256 színű képként). A 24 bites bitkép a képernyőn látható kép adatait változtatás nélkül tartalmazza. Ezért `bmp` képeket fogunk vizsgálni.

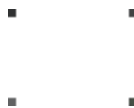
Bármelyik képünket elmenthetjük `bmp` formátumban, de minél nagyobb és színesebb egy kép, annál nehezebb kiismerni magunkat a rengeteg bájt között. Ezért vizsgálódásunkhoz készítünk egy kicsi, vélhetően jellemző jeleket tartalmazó képecskét `bmp.bmp` néven.

A kép szélessége 16 pixel, magassága 12 pixel. A méret meghatározásához praktikus 4-gyel osztható szélességet megadni, mert a mentést 4 bájtonként végzi processzor akkor is, ha az adatok 3 bájtosok. Ha a mentendő bájtok száma nem osztható négygel, akkor lyukat hagy a mentéskor. Ugyancsak praktikus viszonylag kis méretet beállítani, hogy kevés kódot kelljen értelmeznünk.

A képünk fehér lesz, csak a négy sarkában színezzük át 1-1 pixelt. Ehhez a lehető legnagyobb nagyítást és ceruzát használunk.

Mivel szövegszerkesztőben szeretnénk majd megnyitni és látni a bájtok értékét, színekomponeenseknek válasszuk nyomtatható karakterek kódjait. Például legyen a bal-felső pixel RGB kódja (48, 49, 50), ezek a 0, 1, 2 ASCII kódjai; a jobb-felső pixel (65, 66, 67) – az ABC –, bal alsó (97, 98, 99) – az abc – és a jobb-alsó sarokban (69, 78, 68) – az END kódjai. Ilyen lesz a kép:

400-szoros nagyításban kicsit több látszik belőle, de a pixeleket elmossa a megjelenítő. A színnek sem mondanak sokat, sötétszürke pontok...



Jegyzetömbben megnyitva a fájlt (és az ablakot megfelelő méretűre állítva) ezt láthatjuk:



Az értelmezhető karakterek:

- BM: minden bmp így kezdődik
- v ASCII kódja 118
- 6 ASCII kódja 54
- (ASCII kódja 40
- @ ASCII kódja 64

Megvan a 012, ABC, abc és END, de fordított sorrendben és fel van cserélve a lent és a fent. Az első és utolsó sorban 42 pötty van, a közbelső sorokban 48 és összesen 12 ilyen sor van. Ez a pötty a fehér szín (255, 255, 255) kódja, három karakter ad ki egy pixelt.

A fejléc értelmezéséhez olyan szövegszerkesztőt érdemes választani, ami megjeleníti a nem nyomtatható (vezérlő) karaktereket is. Ilyen a Notepad++. Ebben a `bmp.bmp`-t ezt láthatjuk:



- STX:
Start of Text kódja 2
- NUL:
NULL kódja 0
- DLE:
Data Link Escape kódja 16
- FF:
Form Feed kódja 12
- SOH:
Start of Heading kódja 1
- CAN:
Cancel kódja 24

A bitmap ugyanúgy néz ki és a nyomtatható karakterek is azonosak a jegyzetombben láthatókkal.

- A bitmap formátum specifikációjából (leírásából) tudjuk, hogy a BM bájtokat a fájl mérete követi, ami 4 bájtos: `v[STX][NUL][NUL]`. Ez lenne 630 – amit a fájlkezelőben a fájl tulajdonságaként látunk. De hogyan? $v = 118$, $630 - 118 = 512$. Alakul... $512 = 2 * 256$ tehát a négy bájt little-endian kódolással adja ki a méretet: $0 + 0 + 2 * 256 + 118$.
- A `6[NUL][NUL][NUL]` értéke ugyanígy számolva 54. Ez a teljes fejléc mérete, másképp: az 54. bájtton kezdődik a bitmap. Ez Notepad++-ban ellenőrizhető, ha a 'c' elé kattintunk, a státuszsorban láthatjuk, hogy az 55. karakter következik 1-től számolva.
- A `A[DLE][NUL][NUL][NUL][FF][NUL][NUL][NUL]` egymás után két szám, a kép szélességének és magasság értéke.
- A `A[SOH][NUL][CAN][NUL]` nem egy négybájtos szám, hanem két darab kétbájtos érték. Az első értéke $1 + 0$, az alapértelmezett kimeneti perifériát határozza meg, a második pedig a felbontás, ami itt 24.
- Egy 4 bájtos 0 érték után jön a `@[STX][NUL][NUL]`. azaz $64 + 2 * 256 + 0 + 0 = 576$. – akkor is ennyi, ha csak kétbájtos adat lenne. A fájlformátum leírása alapján ez a bitmap mérete bájtokban. Szóval $16 * 12 * 3$ vagy, a fájl méret és a fejléc méretének a különbsége: $630 - 54$. Valóban?

Kísérlet

Most, hogy már tudjuk, hogy hol vannak a megjelenítés szempontjából fontos adatok és hol, hogyan jelennek meg a képpontok a szöveges nézetben, próbáljuk ki, mi történik, ha módosítjuk a képet.

1. Készítsünk több példányban más-más néven mentést a `bmp.bmp`-ről!
2. Nyissuk meg jegyzetombben vagy Notepad++ programmal az egyes példányokat, írjuk át a bitmap néhány értékét! Mentés után nézzük meg, hogyan változott meg a kép!

Jóslat az eredményre: Lesz olyan alkalmazás, amiben a legcsekélyebb módosítás is elrontja a képet, a mentés után megnyitáskor hibát jelez a társított alkalmazás. Talán lesz olyan szövegszerkesztő is, amelyikben nem okoz fájl sérülést a módosítás, megtekinthető az eredmény. Az alábbi képbe Notepad++ programban a „Kutya”, a „MACSKA” és a „Barátság” szó lett beírva:



Mi lehet az oka annak, hogy egyes alkalmazásokban a módosítás során elromlik a képfájl? Mivel csak a bitmap színkomponenseit módosítottuk, ott kell keresni a magyarázatot, de a fehér pixelek bájttjait jelölő pöttyökről nem látjuk, hogy valóban milyen karakterek.

Írjunk programot bájtok vizsgálatára!

A `StreamReader Read()` függvénye karakterenként olvassa az adatokat, az eredményt `int` (egész) típusú változóban tudjuk tárolni. Olvassuk be így az `állatok.txt` adatait!

29. példa: Fájlból olvasás karakterenként

Az előző példában szereplő `állatok.txt` fájlt olvassuk be a `StreamReader Read()` függvényével. Az olvashatóság érdekében, ha a beolvasott karakter nagybetű, akkor új sort kezdünk a konzolon. Ha a karakter az angol ábécé betűje vagy szám, akkor karakterként írjuk ki – mert ennek helyesnek kell lennie –, egyéb esetben szóközők között a kódját jelenítjük meg.

```

7. private static void Karakterolvasas()
8. {
9.     StreamReader sr = new StreamReader("állatok.txt");
10.    while (!sr.EndOfStream)
11.    {
12.        int c = sr.Read();
13.        if ('A' <= c && c <= 'Z')
14.            Console.WriteLine(); /*nagybetű előtt új sort kezd*/
15.        if (('A'<=c && c<='Z') || ('a'<=c && c<='z') || ('0'<=c && c<='9'))
16.            Console.Write((char)c);
17.        else
18.        {
19.            Console.Write(" " + (int)c + " ");
20.        }
21.    }
22.    sr.Close();
23. }
```

Az állatok.txt tartalma:

```
Hápi kacs 1
Morgó kutya 12
Hekus kutya 4
Bégó birka 3
Fűtfal kecske 5
Múzos szarvasmarha 3
Hínár liba 2
Különc malac 1
```

```
Microsoft Visual Studio Debug Console
állatok.txt fájlból olvasás karakterenként
H 225 pi 32 kacs 32 1 13 10
Morg 243 32 kutya 32 12 13 10
Hekus 32 kutya 32 4 13 10
B 233 g 337 32 birka 32 3 13 10
F 369 tfal 32 kecske 32 5 13 10
M 250 z 243 s 32 szarvasmarha 32 3 13 10
H 237 n 225 r 32 liba 32 2 13 10
K 252 l 246 nc 32 malac 32 1
```

A képernyőképen látható, hogy a sorvégét a Windowsra jellemző 13-as és 10-es karakterek jelzik. A szóköz karakterkódja 32, ezt is ellenőrizhetjük bármelyik kódtáblában. Az ékezetes karakterek kódja 127-nél nagyobb. Ez is rendben lenne, de Bégó birka és Fűtfal kecske karakterei között van olyan, amelyiknek a kódja 255-nél is nagyobb. Ez necces... Egy bájton a legnagyobb tárolható érték a 255. Hogyan lesz az 'ő' kódja 337 és az 'ű' kódja 369? Ez 82-vel, illetve 114-gyel nagyobb a megengedettnél.

Módosítsuk a programunkat úgy, hogy ne `int`, hanem `byte` legyen a beolvasott karakterkód!

```
26. byte c = (byte)sr.Read();
```

Ebben az esetben az 'ő' helyett 'Q', az 'ű' helyett 'q' jelenik meg. Ez azt mutatja, hogy a `Read()` eredménye egy olyan 4 bájtos érték, aminek nem csak az első bájton van értéke: a Q kódja 81, éppen 256-tal kisebb a 337-nél.

Talán a legérdekesebb az, ha a `c`-nek `char` az értéke:

```
char c = (char)sr.Read();
B 233 g 81 32 birka 32 3 13 10
F 113 tfal 32 kecske 32 5 13 10
```

Az eredmény azt mutatja, hogy az 'ő' kódja 81, az 'ű' kódja 113. De a 81 (Q) és a 113 (q) a normál karakterek között van, ezért karakterként kellene kiírnia – ahogy azt a byte típus esetén meg is teszi. Itt mégis csak a kódot írja ki, nem a karaktert.

Az ellentmondás feloldása a karakterkódolás fejlődésében keresendő. A manapság használt ékezetes karaktereket is ismerő kódkészletek nem egybájtosok. Amikor kiírjuk vagy beolvasunk a szöveget, akkor – takarékosági okokból „tömörítheti” a karakterkódot, lehetőleg 1 bájtosra. A kódolási módszert vagy az operációsrendszer vagy a program beállításai tartalmazzák. Vagy, például html esetén megadjuk a fájl elején: `lang="hu"`.

Ha tényleg binárisan, az eredeti bájtokat akarjuk a fájlból kiolvasni, akkor a `StreamReader` – a szövegolvasó – nem jó, mert a beolvasott karaktert a beállításoktól függően átalakítva, 4 bájtos egész értéként adja át a programnak és ettől kezdve annyiféleképpen látjuk a kód értékét, ahány módon megpróbáljuk megjeleníteni.

A bináris olvasó

Látható, hogy a `StreamReader` két feladatot végez el: egyrészt eléri a fájlt és ebben sorra veszi a bájtokat, másrészt a kiolvasott adatot valamilyen kódtábla alapján karakterként értelmezve továbbadja a programnak. Az értelmezés nélküli olvasáshoz, csak a fájl eléréséhez és ebben lépkedéshez egy másik eszközre lesz szükségünk, ez a `FileStream`. Azonban ez szinte csak

bitsorozat olvasó, ezért szükség van egy másik eszközre, ami a C# változóinak megfelelően értelmezi a beolvasott biteket. Erre való a `BinaryReader`, aminek számos függvénye van arra, hogy az egyszerű adattípusok, illetve ezek tömbjének megfelelő formában adja tovább a bitsorozatot. Láthattunk már hasonlót: `string`-ből a `Convert` osztály készíti el a megfelelő adattípust, vagy – az adattípus felől megközelítve – az egyes típusok `Parse()` függvényei is ugyan- ezt a célt szolgálják. Csak éppen nem fájlhoz, hanem szöveghez kell őket használni.

30. példa: Szöveges fájl olvasása binárisan

Nincs más dolgunk, mint a `StreamReader`-t lecserélni `FileStream`-re és `BinaryReader`-re, kitá- lálni, hogy hogyan vehetjük észre a fájl végét és melyik függvény olvas bájtot. Ez utóbbi két feladathoz a két eszköz függvényei között kell keresni a megoldást.

Az új, most már tényleg binárisan olvasó programunk:

```

7. private static void Binarisolvasas()
8. {
9.     FileStream fs = new FileStream("állatok.txt", FileMode.Open);
10.    BinaryReader br = new BinaryReader(fs);
11.    for (int i = 0; i < fs.Length; i++) /*hossza csak az fs-nek van*/
12.    {
13.        byte c = br.ReadByte(); /*adattípust csak a br ismer*/
14.        if ('A' <= c && c <= 'Z')
15.        {
16.            Console.WriteLine();
17.        }
18.        if (('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z') || ('0' <= c && c <= '9'))
19.            Console.Write((char)c);
20.        else
21.        {
22.            Console.Write(" " + (byte)c + " ");
23.        }
24.    }
25.    br.Close();
26.    fs.Close();
27. }

```

Az eredmény:

```

c# Microsoft Visual Studio Debug Console
bináris olvasás az állatok.txt fájlból

H 195 161 pi 32 kacsa 32 1 13 10
Morg 195 179 32 kutya 32 12 13 10
Hekus 32 kutya 32 4 13 10
B 195 169 g 197 145 32 birka 32 3 13 10
F 197 177 tfal 32 kecske 32 5 13 10
M 195 186 z 195 179 s 32 szarvasmarha 32 3 13 10
H 195 173 n 195 161 r 32 liba 32 2 13 10
K 195 188 l 195 182 nc 32 malac 32 1

```

Így már látszik, hogy mi „motiválta” a `StreamReader`-t. UTF-8 kódolásban ékezetes karakterek mindegyike valójában kétbájtos. Az első bájt – 195 vagy 197 – alapján lehet tudni, hogy a másodikat hogyan kell értelmezni. Mindkét szám 127-nél nagyobb, ezért a 127 vagy annál kisebb értékek egyértelműen a [0; 127] tartományból való karakterek kódjai, ebben az egy bájtot módosítás nélkül olvassa be.

Bináris fájl – *.bmp – olvasása, módosítása, írása

Most már az is érthető, hogy miért okozott problémát a bitmap átírása. A 255-ös kódot – nagyobb, mint 127 – a beolvasás során átértelmezi a szövegszerkesztő. Amikor sikerült módosítani a képet, akkor nem UTF-8 formátumban, hanem ANSI kódolással értelmezte a szövegszerkesztőnk a fájlt. Az ANSI egybájtos amerikai szabvány, a 7 bites ASCII (szintén amerikai szabvány) kiegészítése. A Notepad++ programban választható mintegy 50 szabvány közül ez az egyik.

A `bmp.bmp` fájl `StreamReader Read()` függvényével történő beolvasásakor a fejléc és a négy sarok jó lesz, mert ott minden bájt értéke 127-nél kisebb (szerencsés a méret kiválasztása), de a fehér szín 255-ös kódját hibásan adja vissza a `Read()`: a 255 lehet a nem-törhető szóköz kódja; a pötty a szövegszerkesztőnkben 02D9 (729); UTF-8 kódként a 00FF az Ÿ; a `Read()` szerint 65 533, aminek utolsó bájtja kiírva vagy mentve 253.

31. példa: A `bmp.bmp` olvasása, módosítása és kiírása

Elsőként nézzük a beolvasást. Nem karaktereket, hanem pixeleket akarunk kétdimenziós tömbben tárolni, ezért el kell készíteni a `Pixel`-t. A beolvasás során most lazán vesszük a fejléc adatok beolvasását, csak elraktározzuk egy 54 bájtos tömbben, hogy a végén ugyanazt tudjuk kiírni. Ezt követően figyelve a két szabályra – sorok fordítva és RGB fordítva – beolvassuk a `Pixel` tömbbe a színtkomponensek értékét. Csak ellenőrzésképpen néhány adatot kiírunk konzolra. Végül nem felejtjük el bezárni az olvasókat.

```
1. using System;
2. using System.IO;
...
7. struct Pixel          /*ebből van a bitmap*/
8. {
9.     public byte R;
10.    public byte G;
11.    public byte B;
12. }
13.
14. static void Main()
15. {
16.     Console.WriteLine("*.bmp fájl olvasása, Red kiírása");
17.     Console.WriteLine();
18.     FileStream fs = new FileStream("bmp.bmp", FileMode.Open);
19.     BinaryReader br = new BinaryReader(fs);
20.     byte[] fej = br.ReadBytes(54); /*ha nem fog változni, így is jó*/
21.
22.     Pixel[,] bitmap = new Pixel[12, 16];          /*tudjuk, hogy ekkora*/
23.     for (int sor = 11; sor >= 0; sor--)          /*első sor az alsó...*/
24.     {
25.         for (int hely = 0; hely < 16; hely++)
26.         {
27.             bitmap[sor, hely].B = br.ReadByte();
28.             bitmap[sor, hely].G = br.ReadByte();
29.             bitmap[sor, hely].R = br.ReadByte();
30.             Console.Write(bitmap[sor, hely].R);    /*csak az R kiírása*/
31.             Console.Write(" ");
```

```

31.     }
32.     Console.WriteLine();
33. }
34. br.Close();
35. fs.Close();
36.

```

A bitmap létrehozásakor – 21. sor – konkrét értéket adunk meg, mert a fejléc beolvasásakor nem bontottuk fel az 54 bájtot adatokra, így nem tudjuk kiolvasni belőle. Erre a következő példában látunk profibb megoldást.

A 29. sorban az R értékeket azért érdemes kiírni, mert így tudjuk ellenőrizni, hogy tényleg a piros kapta a piros komponens értékét. Ebből már lehet következtetni a másik kettő helyességére is, vagy ugyanitt az R átírásával könnyen ellenőrizhetjük a többi szín helyességét is.

Mindegyik pixelből célszerű kiírni egy adatot és a méretnek megfelelően sorokra bontani, mert így ellenőrizhető, hogy jók-e a ciklusok paraméterezése. A 32. sor csak a soronkénti kiírás miatt kell.

A kép módosítása ezután már egyszerű, át kell írni a megfelelő pixelek R, G és B értékét.

```

37. for (int sor = 1; sor < 5; sor++)
38.     for (int hely = 1; hely < 9; hely++)
39.     {
40.         bitmap[sor, hely].R = 255; bitmap[sor, hely].B = 255; /*marad*/
41.         bitmap[sor, hely].G = 0;
42.         if (sor != 3 || hely < 3 || hely > 6)
43.             bitmap[sor, hely].B = 128;
44.     }
45.

```

Mivel tudjuk, hogy kezdetben a pixelek nagyrésze fehér, ezért a 40. sor felesleges. Másrészt, egy tetszőleges kép módosításánál sohasem szabad elfelejteni, hogy egy pixel 3 bájtjának kell értéket adni.

Azzal, hogy az adott területen a zöld komponens 0 lesz, a pixelek színe lila lesz a 3. sor 4–5. helyén. A többi helyen kevesebb lesz a kék, így ott pirosabb lesz a téglalap,

Az eredmény fájlba írása – a beolvasás után – már gyerekjáték. `FileStream` és `BinaryWriter` létrehozása – ki gondolta volna, hogy így hívják? – adatok kiírása, eszközök lezárása.

```

46. fs = new FileStream("piros.bmp", FileMode.Create);
47. BinaryWriter bw = new BinaryWriter(fs);
48. bw.Write(fej);
49. for (int sor = 11; sor >= 0; sor--)
50.     for (int hely = 0; hely < 16; hely++)
51.     {
52.         bw.Write(bitmap[sor, hely].B);
53.         bw.Write(bitmap[sor, hely].G);
54.         bw.Write(bitmap[sor, hely].R);
55.     }
56. bw.Close();
57. fs.Close();
58. }

```

És ezt követően az eredményt is láthatjuk (itt 400%-os nagyításban):



Talán érdemes megjegyezni, hogy a 46. sorban ugyanazt az `fs` nevű változót használhatjuk, mint amit a 18. sorban, ezért a típusát nem adjuk meg újra, de a konstruktorral újra inicializáljuk, ezért egy másik fájlra fog mutatni és másmilyen, fájlletheozás jogai lesznek.

32. példa: A bmp fejléc értelmezett tárolása és kiírása

Az előző példában a fejléc adatokat csak eltároltuk. Ha érdemben szeretnénk használni az adatokat, akkor megfelelő típusú változókba kell tenni. Ehhez az előző megoldás 19. sorát az itt látható 12 sorral helyettesíthetjük:

```
char[] bm = br.ReadChars(2); //BM
int fajlmeret = br.ReadInt32(); //630 v.STX.NUL.NUL
byte[] szabad = br.ReadBytes(4); //0
int bitmapkezd = br.ReadInt32(); //54 6.NUL.NUL.NUL
int infomeret = br.ReadInt32(); //40 (.NUL.NUL.NUL
int Szeles = br.ReadInt32(); //16, DLE.NUL.NUL.NUL
int Magas = br.ReadInt32(); //12, FF.NUL.NUL.NUL
Int16 kimenet = br.ReadInt16(); //1 SOH.NUL
Int16 pixelbit = br.ReadInt16(); //24 CAN.NUL
int tomere = br.ReadInt32(); //0
int bitmapmeret = br.ReadInt32(); //576 @.STX.NUL.NUL
byte[] egyeb = br.ReadBytes(16); //0
```

A megoldás inkább munkaigényes, mint nehéz. Pontosán kell követni a BMP fájl specifikációját. Ezt követően a 21. sorban a méret megadása és a tömb bejárásaihoz a ciklus határa paraméterezhető.

```
Pixel[,] bitmap = new Pixel[Magas, Szeles];
```

Használni lehet, de módosítani veszélyes a méretet, mert elronthatja a fájlt, mivel a mérettel együtt a `fajlmeret` és a `bitmapmeret` is változik. További veszélyt rejt, ha a módosítás során nem osztható 4-gyel az egy sorban lévő bitek száma, mert ilyenkor a sorvégeket megfelelő számú 0 kódú bittel ki kell egészíteni, ami szintén hatással van a fájl méretre. Emiatt egy 90°-os forgatás is módosíthatja a fájl méretét. Ha a színek számán is változtatnánk, akkor az `egyeb` után a használt színek palettája következne, 2-szer, 16-szor vagy 256-szor négy bájt, így már `bitmapkezd` 54-es értéke is módosulna.

A beolvasott fejléc adatainak megtartásával is rengeteg érdekes képmódosításra van módunk. Például: nagyon egyszerű színszűrőt alkalmazni, egy-egy komponens értékét 0-ra vagy 255-re módosítani. Lehet tükrözni a képet. Egy kisebb képet betöltve, össze lehet másolni képeket. Sőt, fekete-fehér képet el is rejthetünk (ez a szteganográfia) például úgy, hogy a piros komponens értékét legfeljebb 1-gyel módosítva párosra, a fekete értékét páratlanra állítjuk. Aki tudja az elrejtés szabályát, az ki tudja nyerni az információt.

A módosított kép mentésekor a fejlécet pontosan kell kiírni. A 48. sor helyett ez újabb 12 sor:

```

bw.Write(bm);
bw.Write(fajlmeret);
bw.Write(szabad);
bw.Write(bitmapkezd);
bw.Write(infomeret);
bw.Write(Szeles);
bw.Write(Magas);
bw.Write(kimenet);
bw.Write(pixelbit);
bw.Write(tomore);
bw.Write(bitmapmeret);
bw.Write(egyeb);

```

KÓPIAKÉSZÍTÉS ÉS DIGITÁLIS HAMUPIPŐKE – MÁSOLÁS, KI- ÉS SZÉTVÁLOGATÁS TÍPUSALGORITMUSSAL

Tizedik évfolyamon megismerkedtünk a típusalgoritmusok – ha úgy tetszik, programozási tételek – egy csoportjával, mostanra pedig fel is elevenítettük használatukat. Az eddig megismertek az „egyszerű” típusalgoritmusok. Közös tulajdonságuk, hogy egy adatsorozat sok eleméhez egyetlen értéket – az összegüket, az átlagukat, a legnagyobbat, egy kiválasztott elem értékét, egy logikai értéket – rendelünk velük.

E mostani leckével indulóan olyan típusalgoritmusokat ismerünk meg, amelyek eredménye nem egy, hanem több érték, egy új adatsorozat. Valójában már találkozhattunk ezekkel az algoritmusokkal, hiszen a fájlból beolvasás is, és a szöveges adatsorozatból egész típusú adatok előállítás is másolás, a kiválogatás pedig kiegészítő tananyag volt a megszámlálás után a 10-es jegyzetben.

Másolás

A másolás típusalgoritmusának lényege a következő:

1. járjuk be egy adatsorozat elemeit,
2. valamilyen módon alakítsuk át az egyes elemeket, és az átalakítást követően
3. gyűjtsük újabb adatsorozatba az így kapott elemeket.

Az elemek átalakítását matematikusmegfogalmazással mondhatjuk úgy is, hogy az adatsorozat elemeihez valamilyen hozzárendelési szabállyal, hozzárendelő függvénnyel másik elemet rendelünk.

Mindez mondszerű leírásban:

```

program
    sorozat = valamilyen adatsorozat
    másolat = üres adatsorozat
    ciklus a sorozat minden elem-ére
        átalakított_elem = hozzárendelő_függvény(elem)
        másolat-hoz hozzáfűz(átalakított_elem)
    ciklus vége
program vége

```

Megjegyezzük, hogy amikor a „hozzárendelő függvény” feladata kellőképp egyszerű, nem feltétlenül írunk külön függvényt, hanem helyben kiszámítjuk a hozzárendelés értékét.

Az egyik leggyakoribb másolási feladat a nagyon gyakran előforduló típusátalakítás, például amikor szöveg formátumban tárolt számokat alakítunk – egész vagy lebegőpontos – számokká, vagy fordítva. Három módszert is bemutatunk.

33. példa: Másolás hagyományos, strukturált módon

Az első a leghagyományosabb kód a másolás tételre teljesen megfelel a fenti mondatszerű leírásnak.

1. Számokból karaktersorozat

Egész számokat tartalmazó adatsorozatból állítunk elő karaktereket tartalmazó adatsorozatot. Ezt eddig így mondtuk volna: egy `int`-eket tartalmazó tömbből készítünk egy `string`-et. Sokszor írtunk már ehhez hasonlót, csak éppen nem egy karaktersorozatot hoztunk létre, hanem rögtön a kimenetre, konzolra küldtük az adatokat. A megoldás függvényében nem csak a számot alakítjuk át karakterekké, hanem hozzátesszük az adatokat elválasztó karaktereket is, a szeparátort. Az általános megoldást eljárásként írjuk meg, amelynek bemeneti paraméterei a szeparátor és az egészeket tartalmazó adatsorozat. A másolást majdnem minden elemre azonos függvénnyel kell végezni, de a szeparátorból eggyel kevesebb kell, mint ahány szám van. Ezért trükközünk: az első adatot szeparátor nélkül másoljuk, a többi elé tesszük a szeparátort.

```
10. static void Ki(string szeparátor, int[] adatsor)
11. {
12.     string ki = adatsor[0].ToString();
13.     for (int i = 1; i < adatsor.Length; i++)
14.         ki += szeparátor + adatsor[i];
15.     Console.Write(ki);
16. }
```

Ha az adatsorozatot tömbben tároljuk, akkor előfordulhat, hogy maradnak szabad helyek a tömbben, amit nem szeretnénk átvenni. Hasonlóan, előfordulhat, hogy nem az elejétől kezdve szeretnénk másolni vagy nem mindegyiket, hanem csak minden másodikat... Ezeket a speciális szempontokat is megadhatjuk paraméterként. Ez a típusalgoritmus lényegén nem változtat.

```
17. static void KiN(string szeparátor, int[] adatsor, int N)
18. {
19.     string ki = adatsor[0].ToString();
20.     for (int i = 1; i < N; i++) /*A ciklus paraméterezése módosult*/
21.         ki += szeparátor + adatsor[i];
22.     Console.Write(ki);
23. }
24. }
```

2. Szöveggént megadott számok értéké alakítása

Begépelésekor a program szöveggént kapja meg az adatokat. Ezeket egyenként, másolás típusalgoritmus használatával tehetjük be a kívánt típusú elemeket tartalmazó tömbbe. A megoldást most függvényként írjuk meg, így még inkább látszik, hogy a bemenete is egy adatsorozat és a kimenet is adatsorozat.

```

25. static int[] Másol1(string[] adatsor)
26. {
27.     int[] kimenet = new int[adatsor.Length];
28.     for (int i = 0; i < adatsor.Length; i++)
29.         kimenet[i] = int.Parse(adatsor[i]);
30.     return kimenet;
31. }
32.

```

3. Karaktersorozatból szöveg tömb

A fenti eljárások és függvények felhasználása mellett még egy „rejtett” másolás típusalgoritmus: az egy sorban bekért adatokból – ami egy `string`, azaz karaktersorozat – `string` tömböt készít a `Split()` függvény. Ez a függvény is a másolás algoritmusát használja. Nagyon sokszor megspóroltunk már vele 3 sor kódot.

```

33. static void MásolástHasznál()
34. {
35.     string[] adatsor = "1 5 2 3 4".Split(' ');
36.     int[] szamok;
37.     szamok = Másol1(adatsor);
38.     KiN(" ", szamok, szamok.Length);
39. }

```

34. példa: Kiegészítés: Másolás haladó nyelvi eszközökkel

Ahogy a korábbi típusalgoritmusoknál is láthattuk, a C# nyelvben az egyes gyakran használt algoritmusok kódolása helyett a lista – `List<>` – típusú adatsorozatokhoz „előregyártott” függvényeket is használhatunk. Ahogy eddig is, ezek ismerete nem tananyag, de van, aki nagyon szereti használni, mert rövid, olvasható kódot eredményez. Talán már ismerős a listához szükséges névtér:

```

2. using System.Collections.Generic;

```

Az általunk írt függvényekben a paraméterek valamilyen adattípusok, de most szükség lesz arra, hogy paraméterként adjuk meg a hozzárendelő_függvény()-t vagy az elvégzendő műveletet. Erre ad lehetőséget egy speciális jelölés, **lambda-kifejezés**. A lambda-kifejezés lelke a lambda operátor, jelölése `=>`. Leegyszerűsítve a működését, úgy kell írni, ahogy matematikaórán írjuk a hozzárendelést: „a sorozat minden `x` eleméhez hozzárendeljük az `f(x)` értéket” lambda operátorral leírva: `x => f(x)`.

```

10. static void Másolások(string[] adatsor)
11. {
12.     List<string> adatlista = new List<string>(adatsor);
13.     List<int> szamlista = adatlista.ConvertAll<int>(a => int.Parse(a));
14.     int[] szamok = szamlista.ToArray();
15.     Console.WriteLine(String.Join(" ", szamok));
16. }

```

Figyeljünk arra, hogy a függvények kombinációjának kétféle változata van. A kiírás eljárásában bemeneti paraméter lehet egy szöveg, a 15. sorban ezt a szöveget a szövegösszefűzés függvénnyel hoztuk létre. Ilyenkor az először elvégzendő függvényt a zárójelek közé írjuk. A többi

esetben egy adatsorozatnak a tagfüggvény. ét hívtuk meg, ekkor az adatsorozatot követi a tagot kiválasztó pont, majd a függvény neve. Mindkét írásmódot lehet kombinálva is használni. A 13. sorban az `adatlista` tagfüggvénye a `ConvertAll<int>()`, ennek paramétere a lambda-kifejezés, amiben egy függvényt is használunk, ami az `int` típusnak a (statikus) `Parse()` tagfüggvénye, aminek a paramétere az `a`.

Virtuózok (és akik a funkcionális programozást részesítik előnyben) szeretnek egész programokat függvénykombinációval, „egysorban” megadni.

```
17.     szamok = new List<string>(adatsor)
        .ConvertAll<int>(a => int.Parse(a)).ToArray();
        /*A 12-14. sorok egyben*/
18.     Console.WriteLine(String.Join(" ", szamok));
19.
```

Nem csak adattípusok közötti átalakítás lehetséges lambda-kifejezéssel, nézzünk erre is egy példát, a `Math.Sqrt()` függvénnyel adjuk meg a számsorozat minden elemének a négyzetgyökét:

```
20.     List<double> gyoklista =
        new List<double>(szamlista.ConvertAll<double>(a => Math.Sqrt(a)));
        /*A szamlista minden számának négyzetgyöke*/
21.     Console.WriteLine(String.Join(" ", gyoklista));
        /*double típusú adatok karaktertömbbe másolása*/
22. }
```

35. példa: Taxis adózott bevételei

Adott a tizedik évfolyamos tankönyvből ismerős taxisunk piculában kifejezett bevételeinek listája. A taxis adóterhei jelentősek: mire kifizeti a mindenféle adókat, a bevétel egésze kerekített 49 százalékról lemondhat. Adjuk meg a taxis adózott bevételeinek a listáját!

```
10. static void Taxis_bevételek()
11. {
12.     int[] bevételek = new int[5] { 1, 5, 2, 3, 4 };
13.     int[] adózott = new int[bevételek.Length];
14.     for (int i = 0; i < bevételek.Length; i++)
15.     {
16.         int adó = (int)Math.Round((double)bevételek[i] * 0.49, 0);
17.         adózott[i] = bevételek[i] - adó;
18.     }
19.     Console.WriteLine(String.Join(" ", bevételek));
20.     /*kiírhatjuk ciklussal, vagy a korábban megírt Ki()-vel is.*/
21. }
22.
```

Kódunk 14–18. sorában alkalmazzuk a másolás típusalgoritmust, a 19. sorban pedig a `Join()` függvény végzi a másolást. Ehelyett akár meg is írhatnánk a kiíró eljárást vagy használhatjuk a korábban megírt `Ki()` függvényt.

36. példa: Libatömegek farkas előtt és farkas után

Az a szituáció is ismerős lehet a tizedik évfolyamos kötetből, amikor a róka libát lop a faluból. A libák súlyát – pontosabban tömegét – listában adjuk meg. A farkas a dűlőútnál várja a rókát, és a három kilónál nagyobb libákat elveszi – a piciket nagylelkűen otthagyja a rókának. Adjuk meg azt a listát, amelyik a farkassal való találkozást követő libatömegeket tartalmazza! Amelyik libát a farkas elvette, annak helyére írjunk nullát!


```

10. static void RókaLibái()
11. {
12.     int[] libák = new int[5] { 1, 5, 2, 3, 4 };
13.     int[] saját = new int[libák.Length];
14.     for (int i = 0; i < libák.Length; i++)
15.         saját[i] = libák[i] > 3 ? 0 : libák[i];
16.     Console.WriteLine(String.Join(" ", saját));
17. }
18.

```

if(libák[i] > 3)
saját[i] = 0;
else
saját[i] = libák[i];

A hozzárendelő függvény most egy feltételes értékadás, amit egy elágazásban tudunk megadni, de itt az elágazás két ágában ugyanannak a változónak adunk értéket, ezért helyette használhatjuk a háromoperandusú operátort.

37. példa: A tanya állathangjai

Az előző leckéből már ismert allatok.txt fájl tartalmazza tanyánk állatait. Minden állatfajnak ismert a hangja (például macska: nyaú, malac: röf). Állítsuk össze azt a listát, amely megmutatja, hogy milyen hangokkal köszöntenek bennünket állataink, amikor hazaérünk a tanyára! Feltételezzük, hogy az állatok a fájlban található sorrendjüknek megfelelően szólalnak meg.

Az állatok fajtájának és hangjának összerendeléséhez először definiáljunk struktúrát, ennek elemeiből készítsünk tömböt. (Vagányok megoldhatják a feladatot struktúrák tömbje helyett `Dictionary<string, string>` szóttárral.) A feladat megoldásához meg kell adnunk, hogy egy állatfajta mit mond, erre készítjük el az `AllatHangja()` függvényt, állathangokra csak ezen belül lesz szükségünk

```

10. struct Allathang
11. {
12.     public string fajta;
13.     public string hang;
14.     public Allathang(string f, string h)
15.     {
16.         fajta = f;
17.         hang = h;
18.     }
19. }
20. static string AllatHangja(string allat)
21. {
22.     Allathang[] hangok = new Allathang[9] {new Allathang("kacsa", "háp"),
23.     new Allathang("kutya", "vaú"), new Allathang("birka", "bee"),
24.     new Allathang("kecske", "mek"), new Allathang("szarvasmarha", "mú"),
25.     new Allathang("liba", "gá"), new Allathang("malac", "röf"),
26.     new Allathang("kakas", "qqriq"), new Allathang("macska", "nyaú")
27. };
28.

```

Függvényünk a paraméterként megadott fajtát megkeresi a hangok között és eredményül adja a hangot. Megkeresi, tehát a keresés típusalgoritmust tudjuk használni. Óvatosságból nem csak kiválasztjuk, így, ha hibásan adunk meg egy adatot, akkor „néma” lesz az állat: egy szóközt „mond”.

```

27. string hang = ".";
28. int ez = 0;
29. while (ez < hangok.Length && hangok[ez].fajta != allat)
30.     ez++;
31. if (ez < hangok.Length)
32.     hang = hangok[ez].hang;
33. return hang;
34. }
35.

```

Írjuk meg az állatok teljes szövegét is, azt a függvényt, ami az állatok hangját az elhangzás sorrendjében egyetlen szöveggé fűzi össze. Nézőpont kérdése, de ehhez inkább az összegzés típusalgoritmusát használjuk, nem a másolást, mert az eredeti adatsor elemei is karaktersorozatok.

```

36. static string LStr(string szeparátor, List<string> adatsor)
37. {
38.     string ki = adatsor[0].ToString();
39.     for (int i = 1; i < adatsor.Count; i++)
40.         ki += szeparátor + adatsor[i];
41.     return ki;
42. }

```

Végül olvassuk be a fájl megfelelő adatait, azaz másoljuk be a háttértárról egy listába az adatsorból a fajta megnevezéseket. Ezzel együtt, mindjárt tovább is másolhatjuk a fajta hangját és amikor mind megvan, írjuk ki konzolra az eredményt. A beolvasás és az állathang megadása két, egymástól független eljárásként is megírható, de az összevonás kevesebb kódsort, kevesebb memória lefoglalását és kevesebb időt igényel.

```

43. static void Tanyahangok()
44. {
45.     List<string> köszöntések = new List<string>();
46.     StreamReader sr = new StreamReader("allatok.txt");
47.     while (!sr.EndOfStream)
48.     {
49.         string fajta = sr.ReadLine().Split(' ')[1];
50.         köszöntések.Add(AllatHangja(fajta));
51.     }
52.     sr.Close();
53.     Console.WriteLine(LStr(" ", köszöntések));
54. }

```

Ha csak annyi a feladat, hogy a fájlban kapott állatok hangjait írjuk ki a konzolra, akkor írhatunk olyan függvényt is, ami az adatok tárolása nélkül, a fájlból beolvasott és átalakított adatot azonnal a konzolra másolja.

Kiválogatás és szétválogatás

A kiválogatás típusalgoritmusja majdnem olyan, mint a másolásé. Ezúttal nem alakítjuk át az elemeket, hanem az eredetieket másoljuk, de nem mindet, hanem csak azokat, amelyek megfelelnek valamilyen feltételnek. Még nyilvánvalóbb az algoritmikusbeli kapcsolat a megszámlálás típusalgoritmussal – számlálás helyett gyűjt –, ezért kiegészítésként szerepel a 10-es jegyzetben is.

Mondatszerű leírása a kiválogatás típusalgoritmusnak:

```
program
    sorozat = valamilyen adatsorozat
    kiválasztottak = üres adatsorozat
    ciklus a sorozat minden elem-ére
        ha elem adott tulajdonságú, akkor:
            kiválasztottak-hoz hozzáfűz (elem)
    ciklus vége
program vége
```

A szétválogatás típusalgoritmusában különbözik a kiválogatásától, hogy több listába vagy egyéb objektumba válogatjuk szét az elemeket. Az egyikbe kerülnek azok, amelyek megfeleltek a feltételnek, a másikba azok, amelyek nem. Az is elképzelhető, hogy többfelé válogatjuk szét az eredeti elemsorozatot: az egyik gyűjteménybe kerülnek a kicsik, a másikba a közepesek, a harmadikba a nagyok.

38. példa: A farkas és a róka libalakomája

Az 36. példa: szereplő libatolvajlások ismeretében adjuk meg a két ragadozó szárnyasainak listáit! A listák felhasználásával állapítsuk meg, hogy melyik ragadozónak hány darab, illetve hány kilónyi liba jutott! A kiírást végeztessük eljárással, melynek paraméterei a ragadozó neve és a ragadozó libáinak tömegeit tartalmazó lista!

A feladat szétválogatást kér, két adatsorozatba kell tenni a libák tömegeit. Csak a példa kedvéért, a két adatsorozat eltérő típusú lesz, a róka libáit tömbbe, a farkasét listába gyűjtjük. Emiatt kiírásból is kettőt kell elkészítenünk, mert más adattípusú paramétereket használunk benne. Mivel teljesen azonos a céljuk, a kiírás eljárásoknak legyen azonos a neve. A C# nyelvben ez is megtehető, mert a paraméterezésből megérti, hogy melyik eljárást kell végrehajtani.

```
10. static void Kiír(string ragadozó, int[] ltömb, int N)
11. {
12.     Console.Write("A " + ragadozó + "libái:");
13.     int szum = 0;
14.     for (int i = 0; i < N; i++)
15.     {
16.         Console.Write(" " + ltömb[i] + ","); /*előtte szóköz, utána vessző*/
17.         szum += ltömb[i]; /*ha már nézem, akkor összegzés tétele is */
18.     }
19.     Console.WriteLine("\b "); /*utolsó vessző törlése*/
20.     Console.WriteLine($"A libák száma: {N}, össztömege: {szum} kg.");
21. }
```

A kiírás lényegében másolás típusalgoritmus, amit összevontunk az összegzés típusalgoritmussal, mert a feladat az összeget is kérte. A lista kiírására a C# függvényeit használjuk. A listához szükség vagy a System.Collections.Generic névtérre, a Sum() pedig a System.Linq névtérben van.

```

22. static void Kiír(string ragadozó, List<int> llista)
23. {
24.     Console.Write("A " + ragadozó + "libái:");
25.     Console.WriteLine(String.Join(", ", llista));
26.     Console.WriteLine($"A libák száma: {llista.Count},
                                össztömege: {llista.Sum()} kg.");
27. }
28.

```

A szétválogatáskor a róka libáit tömbben tároljuk, de nem tudhatjuk előre, hogy mekkora méretű tömbre van szükségünk. Csak azt tudjuk, hogy legfeljebb annyi, ahány libát összesen begyűjtött. Ezért ennek a méretét adjuk meg a tömb létrehozásakor és egy változóban – külön – tároljuk, hogy a tömbben épp hány elem van. A farkas libáinak tárolásához – mivel ezt listába gyűjtjük – nem szükséges előzetesen a méret.

```

29. static void RókaLibaKetté()
30. {
31.     int[] libák = new int[5] { 1, 5, 2, 3, 4 };
32.     int[] rokaLibái = new int[libák.Length]; /*akár mind beleférjen!*/
33.     int rLDB = 0; /*ennyi libája van a rókának*/
34.     List<int> farkasLibái = new List<int>();
35.     for (int i = 0; i < libák.Length; i++)
36.         if (libák[i] <= 3)
37.         {
38.             rokaLibái[rLDB] = libák[i]; /*pl. rLDB = 0. helyre 1-et*/
39.             rLDB++; /*utána már 1 libája van*/
40.         } /*felülre a rókáét, alulra a farkasét*/
41.     else
42.         farkasLibái.Add(libák[i]);

```

Figyeljük meg a `Kiír()` eljárás használatakor, hogy a Visual Studio mindkét használati formát felajánlja. Sokszor láttunk már ilyet, például a `WriteLine()` eljárásnál is, de csak akkor érdemes az azonos nevet adni két függvénynek, ha a ugyanúgy szeretnénk használni. Ellenkező esetben zavaró a névazonosság.

```

43. Kiír("róka", rokaLibái, rLDB);
44. Kiír("farkas", farkasLibái);
45. }

```

39. példa: Hőségriadós napok

A hőségriasztás legalacsonyabb fokozata arról tájékoztat, hogy egy napon 25 °C-ot meghaladó hőmérséklet várható. A következő heti előrejelzés programban tárolt adatai alapján gyűjtsük listába, és írjuk a képernyőre azokat a napokat, amikor hőségriasztást kell kiadni!

A rövid adatfelvétel érdekében készítsünk `NapiT` néven struktúrát, egy hételemű tömbbe vegyük fel az adatokat, majd a kigyűjtés típusalgoritmus mintájára végezzük el a kigyűjtést egy listába, végül a 37. példa: példához írt `LStr()` függvénnyel írjuk ki az eredményt.

A feladatot a terveink alapján abban a projektben érdemes megoldani, amelyikben az `LStr()` függvény van, vagy átmásolhatjuk a kódot, vagy – ha minden kötél szakad, meg is írhatjuk újra.

```

60. struct NapiT
61. {
62.     public string nap;
63.     public int fok;
64.     public NapiT(string n, int f) { nap = n; fok = f; }
65. }

```

Ha sok ehhez hasonló kis struktúrát kell írni, akkor zavaró a hosszú kód. A sorok tördelése az olvashatóságot segíti (vagy gátolja). Ha a kód könnyen elfér egy sorba, akkor lehet úgy is írni, mint itt a konstruktort.

```

66. static void Hőségriadó()
67. {
68.     NapiT[] elorejelzés = new NapiT[7]{new NapiT("hétfő", 19),
69.                                         new NapiT("kedd", 23), new NapiT("szerda", 26),
70.                                         new NapiT("csütörtök", 27), new NapiT("péntek", 19),
71.                                         new NapiT("szombat", 18), new NapiT("vasárnap", 18)};
72. };
73. List<string> riadósnapok = new List<string>();
74. for (int i = 0; i < elorejelzés.Length; i++)
75.     if (elorejelzés[i].fok > 25)
76.         riadósnapok.Add(elorejelzés[i].nap);
77. Console.WriteLine($"Hőségriadós napok: {LStr(",", " ", riadósnapok)}");
78. }

```

FELADATOK A KILENC TÍPUSALGORITMUSRA ÉS A FÁJLOK HASZNÁLATÁRA

Ebben a fejezetben a tankönyv két Kópiakészítés és digitális Hamupipőke gyakorló feladatait dolgozzuk fel. Az eddig tanult kilenc típusalgoritmus: Eldöntés, Keresés, Kiválasztás, Összegezés, Megszámolás, Szélsőérték-kiválasztás, Másolás, Kigyűjtés, Szétválogatás. Ezek mind egyszerű algoritmusok. Mindegyik egy adatsorozaton végez el valamilyen műveletet. Az első hat eredménye egyszerű adat, az utolsó három eredménye egy vagy több adatsorozat, ezért a tankönyvben összetett algoritmusként szerepelnek. A típusalgoritmus – ez talán érezhető is – nem összetett, csak a kimenete az. Az algoritmusokat azonban nem csak önmagukban használtuk, már többször kombináltuk őket. Többször készítettünk függvényt egy-egy részfeladat megoldására – például eldöntés, kigyűjtés, másolás típusalgoritmusokkal – máskor összevontuk őket és „egy menetben” több kérdésre is válaszoltunk. Így a kilenc típusból építkezve összetett feladatokat is meg tudunk oldani. A következő gyakorlófeladatok is ilyen összetett feladatok lesznek.

A feladatok megoldása előtt praktikus tervet készíteni, amelyben a megoldást típusalgoritmusokra bontjuk. Ezt követően eldönthetjük, hogy melyik részre írunk eljárást vagy függvényt, mely részeket kódoljuk a főprogramban... A tankönyvtől eltérően, az önálló kivitelezés érdekében, először csak a feladatok és a megoldások terve olvasható. A fejezet végén van minta a megoldásra, de természetesen – ahogy azt már kezdettől megszokhattuk – ez csak egy, esetleg néhány lehetséges megoldás a sok millió közül. Ha az önálló megoldás nem megy, először a jegyzet eleje felé és korábbi jegyzetekben érdemes utánajárni a hiba okának, csak ezt követően, ellenőrzésre ajánlott a megoldási minta.

40. példa: Gyalogtúra

A könyv webhelyéről letöltött `tura.txt` állományban egy gyalogtúrán hárompercenként rögzített magassági adatokat találunk. Olvassuk be a fájlt, majd állítsunk elő a felhasználásával

egy olyan listát, amelyik azt mutatja, hogy felfelé „/” vagy lefelé „\” változott-e a túrázónál lévő GPS-készülék által mért magasság az előző mérési pont óta, esetleg megegyezik az előzővel („=”)! Írjuk az új lista elemeit vesszővel elválasztva a képernyőre!

Írjuk meg a fájlbeolvasást követő részt mondatszerű leírással! Minthogy (majdnem) minden adatnak megfeleltetünk egy újat, a másolás típusalgoritmusa kell használnunk.

```
Program szintmérés
magasságok := fájlból beolvasott adatok adatsorozatban
irányok := üres bejárható objektum
ciklus i = 1-től (magasságok elemszáma)-1 -ig
    ha magasságok[i] < magasságok[i-1], akkor
        irányok := irányok + „/”
    különben ha magasságok[i] > magasságok[i-1], akkor
        irányok := irányok + „\”
    különben
        irányok := irányok + „=”
    ciklus vége
Program vége.
```

A kódolás részei: Fájl hozzáadása a programhoz, fájlból beolvasás, szomszédos elemekből eredmény kiszámolása (másolás típusalgoritmus), eredmény kiírása.

Módosítsuk az előző programot úgy, hogy a három métert meg nem haladó változásokat még egyenlőnek tekintse!

41. példa: E terkes mecske leberetette e tejfelt

Lehet egy egyperces rettenet, melyet eszperente nyelven egy ember nyelvel? Az eszperente programunk egy hangyányit primitív lesz, ugyanis a megadott mondatból úgy ír eszperentét, hogy minden magánhangzót e-re cserél, például:

A torkos macska leborította a tejfölt. E terkes mecske leberetette e tejfelt.

Az első verzió elég, ha csupa kisbetűs mondatokat kezel, aztán oldjuk meg, hogy a nagybetűket is tudja kezelni!

Minthogy minden karakternek megfeleltetünk egy másikat, ismét a másolás típusalgoritmusa fog segíteni. A terv: A mondat bekérése; másolás típusalgoritmussal minden betűről el kell dönten, hogy marad-e az, ami volt, vagy e-t kell helyette írni; eredmény kiírása. Annak eldöntése, hogy egy karakter magánhangzó-e, eldöntés típusalgoritmussal lehetséges: felvesszük az összes magánhangzót egy **string** típusú változóba és eldöntjük, hogy a kérdéses karakter benne van-e. Ehhez írhatunk külön függvényt vagy használhatjuk a magánhangzókra a **Contains()** függvényt.

A nagybetűket is kezelő változatban a bekért mondatból a **ToLower()** függvénnyel lehet kisbetűsre alakítani, de lehet, hogy egyszerűbb, ha az eldöntést paraméterezzük és megnézzük, hogy a karakter kisbetűs magánhangzó-e, nagybetűs magánhangzó-e vagy más. (A megoldási minta az itt felvetett megoldásoknál bonyolultabb lehetőséget mutat be.)

42. példa: Kutya és macskaoltások

Kutyáinkat és macskáinkat be szeretnénk oltatni. Mindegyiknek adatnánk veszettség elleni oltást, és a kutyáknak parvovírus ellenit is. A már használt **allatok.txt** fájlból gyűjtjük az

oltás nevének megfelelő listába azoknak az állatoknak a nevét, amelyek az adott oltást kapni fogják! Jelenítsük meg a listák tartalmát!

Természetesen a projekthez hozzá kell adni a fájlt és be kell olvasni. Lépésenkénti megoldáshoz el kell tárolni az összes adatot, ehhez esetleg el kell készíteni az **Allat** struktúrát. De ezt a lépést kihagyhatjuk, ha minden adatsort rögtön beolvasáskor feldolgozunk és ha a feltételeknek megfelelő listába a nevet betesszük. Azaz ez egy keveréke a másolás és kiválogatás típusalgoritmusoknak. Másolásnak indul, a megfelelő adatot (név és fajta) elő kell állítani. Ezt követően kiválogatás a kutyára és macskára, de egy másik eredmény is lesz, a kutyákkal.

Írjuk meg mondatszerű leírással a két oltási listát kialakító részt! Az állatok adatai az **Allat** struktúrában: név, faj, kor, az állatok adatsorozatban **Allat** típusú állatokat tárolunk.

```
Program kutyamacska
    veszettség := üres gyűjteményes adatszerkezet
    parvo := üres gyűjteményes adatszerkezet
    ciklus állatok minden állat-jára
        ha állat.faj = „kutya” vagy állat.faj = „macska” akkor
            veszettség-et bővítjük állat.név-vel
        ha állat.faj = „kutya” akkor
            parvo-t bővítjük állat.név-vel
    ciklus vége
Program vége.
```

43. példa: Tojásrakók

Van egy listánk arról, hogy a háziállataink közül melyik faj egyedei raknak tojást. A már használt `allatok.txt` fájlból⁵ gyűjtsük ki azoknak a nevét, amelyeknél elképzelhető ilyen esemény! Az állatok nevéből ezúttal ne következtessünk a nemükre!

A kód eleje megegyezhet az előző feladat megoldásával, vagy – ha eljárást írtunk a beolvasásra, azt könnyen újra használatba vehetjük. Ezután a kigyűjtéshez – hasonlóan az eszperente nyelvi feladathoz – egy eldöntés típusalgoritmussal lehet a feltételt megadni.

44. példa: Jók és rosszak

George Orwell Állatfarm című regényében egy tanya állatai fellázadnak gonosz gazdájuk, illetve általában az emberek ellen. A legegyszerűbb gondolkodásúak számára is világossá óhajtották tenni az új világrendet, így a lázadás vezetői kiadták a „Négy láb jó, két láb rossz!” jelszót. A szárnyasok rámutattak, hogy ez a szlogen számukra kirekesztő, mire a lázadás ideológusai elmagyarázták, hogy ebben a kontextusban a madarak szárnya is lábnak minősül. Nincs hát szó a szárnyasok megbélyegzéséről, a jelszó az emberi lényeket minősíti.

Írjunk programot vagy eljárást az előző példában megírt program átalakításával `orwell` néven, amely a jelszónak még az említett utólagos korrekciója előtt, a jelszó szigorúan vett jelentése szerint kategorizálja állatainkat! Ha elkészültünk, bővítsük a programot úgy, hogy az egyes sorok beolvasását követően adjon szövegesen megfogalmazott véleményt is, például:

Totyak kacsá kétlábú, azaz rossz.

⁵ Tarajos falként a fájlban „kakas” szerepel, aminek az az oka, hogy a „házi tyúk” szóban szerepel ékezetes karakter, és ez egyes programozási nyelvekben bonyolíthatja a programot.

Első lépésként vagy használjuk az előző feladatban megírt beolvasást vagy olvassuk be soronként az `allatok.txt` fájlt. Az adatokat tároljuk el valamilyen kezelhető, struktúrát tartalmazó adatsorozatban. Minden egyes állat nevét helyezzük a jók, illetve a rosszak listába! (Azaz: esetleg fájlbeolvasás, utána szétválogatás és hozzá eldöntés típusalgoritmus.) Jelenítsük meg vesszővel elválasztott felsorolásként az egyes listák tagjait! (Azaz másolás típusalgoritmus.) Ha ezzel készen vagyunk, akkor a véleményalkotást és a vélemény megjelenítését már tényleg megéri függvénybe kiszervezni. Ha eddig nem, akkor most írjunk olyan függvényt,

- mely létrehoz egy `Allat` típusú adatot (konstruktort);
- amely eldönti, hogy egy fajta nevet tartalmaz-e egy lista;
- amelyik a fenti mintának megfelelő véleményezést ír a képernyőre!

A jók és a rosszak listát a programunk új változata az előbb elkészített függvények használatával töltse fel!

45. példa: Hajónapló

Adott egy fájl `hajonaplo.txt` néven, melyet a tankönyv weblapjáról letöltött fájlok között találunk. A fájl soronként két, kötőjellel elválasztott számot tartalmaz. Az első szám minden sorban azt mutatja, hogy hány fokos irányba haladt a hajó, a második azt, hogy hány tengeri mérföldet haladt abba az irányba. A `kormanyos` programban (vagy eljárásban) az a feladatunk, hogy megadjunk egy olyan listát, amely azt sorolja fel, hogy a hajót az egyes irányváltások – az egyes sorok – között jobbra vagy balra kormányozták-e. Mindig arra kormányozzák a hajót, amerre kisebbet kell fordulnia. Ha pontosan száznyolcvan fokot kellene fordulni, akkor a kormányos véletlenszerűen dönti el, hogy jobbra vagy balra fordul-e.

Majdnem minden adatnak megfeleltetünk egy újat, azaz a másolás típusalgoritmusát kell használnunk. A fordulás irányát meghatározó algoritmus lehet például a következő:

```
Függvény Irány(régi_irány, új_irány)
    ha (új_irány - régi_irány + 360) mod 360 < 180
        akkor Irány := „J”
    különben ha (új_irány - régi_irány + 360) mod 360 > 180,
        akkor Irány := „B”
    különben:
        Irány := „J” és „B” közül valamelyik véletlenszerűen
Függvény vége.
```

Írjunk a fenti mondatszerű leírásból függvényt, amelynek két paramétere a régi és az új irány, a visszatérési értéke pedig egy J vagy egy B karakter! Írjuk meg a főprogramot, amely beolvassa a `hajonaplo.txt` fájlt, és a kormányzásokat a `kormanyzasok.txt` fájlba írja! A kimeneti fájl egy-egy sora tartalmazza a kormánymozdulat előtti útirányt, a kormánymozdulat irányának betűjét és a kormánymozdulatot követő irányt, szóközzel elválasztva.

Ha a bemeneti fájl adatai:

107–12
109–34
210–87
202–3
349–198
17–251
197–37

akkor a kimeneti fájl tartalma:

107 J 109
109 J 210
210 B 202
202 J 349
349 J 17
17 J 197

46. példa: Távirat

2021. április 29-én lehetett utoljára táviratot feladni a magyar postahivatalokban. Ekkor már régen nem morzekóddal továbbították a jeleket.

Volt idő, amikor a morzekódokkal csak az angol ábécé (nagy)betűit és a számjegyeket lehetett kódolni, így a magyar ékezetes betűk helyén több betűből álló összetételeket küldtek. Az 'Á'-ból AA, az 'É'-ből EE lett, és volt még II, OO, UU. Az 'Ö' helyett OE, az 'Ő' helyett OOE, az 'Ü' helyett UE, az 'Ú' helyett UUE került a szövegbe. A mondatvégi írásjelek helyett a STOP karakter sorozat morzekódját küldték át az éteren vagy a kábelen, a mondat belsejében lévőkről pedig lemondtak.

A „Förgeteg közeledik, elűz bennünket!” mondat távirati alakja, ahogy a postás a távíróval elküldte, és ahogy a címzett olvashatta, a következő volt: „FOERGETEG KOEZELEDIK ELUUEZ BENNUENKET STOP”. Az „Áá, dehogy!” pedig ezt a formát öltötte: „AAAA DEHOGY STOP”.

Írjunk programot (vagy eljárást) `tavirat` néven, amely a felhasználótól bekért szöveget a fent jelzett formára alakítva írja vissza a képernyőre!

- Programunkban nem minden karakternek feleltetünk meg valamit, azaz lényegében kiválogatjuk azokat, amelyeknek lesz megfelelőjük (nem mondatközi írásjel). Ezt egy másolás követi: az ékezetmentes karakterek és a szóközök helyett saját magukat másoljuk vissza, az ékezetesek helyett a nekik megfelelő összetételt, a mondatvégi írásjelek helyett pedig szóközzel kezdve a „STOP” szöveget.
- A feladat ugyanakkor felfogható egyetlen másolásnak is: a mondat belsejében lévő írásjelek helyére üres karakterláncot, üres karaktert másolunk.
- Ha az általunk használt programozási nyelv kínál egyszerű módot szövegek nagybetűssé alakítására, akkor használjuk azt! Ha nem, akkor ezt egy újabb másolással kell megoldanunk. A nagybetűssé alakításnak célszerű megelőznie a távirati szöveggé való alakítást, így az ékezetes kisbetűk átírása nem jelent külön feladatot.
- A program lényegi és jól elkülöníthető része az átírt szövegváltozat előállítás. Szervezzük ezt ki függvénybe, amelynek paramétere a nagybetűs karakterlánc, visszatérési értéke pedig az átalakított karakterlánc! Az átalakítási szabályokat csak a függvénynek kell ismernie, a megfigyeléseket tároló adatszerkezetet helyezzük el a függvény belsejében!

Mielőtt elkezdnénk kódolni, rendezzük sorba az igényeket, tervezzük meg a megoldás sorrendjét!

- A feladatot először típusalgoritmusok használatával oldjuk meg.

- A második, továbbfejlesztett megoldásunkban törekedjünk a használt programozási nyelv speciális eszközkészletének mind tökéletesebb kihasználására. Legyen a megoldás rövid, lényegre törő.
- A harmadik megoldásban bontuk részekre – eljárásokra, függvényekre – a megoldást, amire nincs előregyártott függvény, azt írjuk meg.

Mintamegoldások

A példák megoldása (a saját megoldás ellenőrzéséhez) itt egy projektbe szervezve láthatók. Minden feladatra az adott eljárás kódja bemásolható egy-egy program `Main()` eljárásának törzsébe, így a részletek külön is futtathatók. Az egybeszerkesztett megoldás előnye, hogy elegendő egyetlen fájlban dolgozni, jegyzetelni. A futtatásnál viszont eléggé zavaró, ha minden esetben az összes, már megírt rész lefut, ezért praktikus a kész feladatok meghívását idővel kommentbe tenni.

Nem felejtjük el, hogy a fájlból olvasáshoz a fájlokat elérhető helyre be kell másolni!

A névterekből elég azt beírni, amit éppen használunk:

```
1. using System;
2. using System.Collections.Generic;      /*Pl. List<> és tagfüggvényei*/
3. using System.Linq;                    /*Pl. Sum() függvényhez*/
4. using System.IO;                      /*Fájlkezeléshez*/
5. namespace Digi11_Cisz_minta
6. {
7.     class Program
8.     {
```

Egy projektben megoldva az összes feladatot, a végére ehhez hasonló lesz a főprogram.

```
9.     static void Main()
10.    {
11.        //Szintmeres();      //40. Gyalogtúra
12.        //Eszperente();      //41 E terkes mecske leberetette a tejfelt
13.        //Eszperente2();     //41 nagybetűkre is
14.        //Eszperente3();     //41 fordított logikával
15.        //KutyaMacska();     //42 Kutya- és macskaoltások
16.        //Tojásrakók();      //43 Tojásrakók
17.        //Orvell();          //44 Jók és rosszak
18.        //Hajónapló();       //45 Hajónapló
19.        //Távirat_1();       //46 Távirat
20.        //Távirat_2();       //46 listával
21.        Távirat_3();         //46 haladó
22.    }
```

40. példa: Gyalogtúra – mintamegoldás

```

23. static void Szintmeres()
24. {
25.     StreamReader sr = new StreamReader("tura.txt");
26.     string[] adatok = sr.ReadLine().Split(", ");
27.     sr.Close(); /*fájlból beolvasás vége*/
28.     int[] magasságok = new int[adatok.Length];
29.     magasságok[0] = int.Parse(adatok[0]);
30.     string[] irányok = new string[adatok.Length - 1];
31.     /*7 mérés között 6 szakasz!*/
32.     for (int i = 0; i < adatok.Length - 1; i++) /*Másolás t.*/
33.     {
34.         magasságok[i + 1] = int.Parse(adatok[i]);
35.         if (magasságok[i + 1] > magasságok[i]) /*rákövetkező feljebb*/
36.             irányok[i] = "/";
37.         else if (magasságok[i + 1] < magasságok[i])
38.             irányok[i] = "\\";
39.         else
40.             irányok[i] = "=";
41.     }
42.     Console.WriteLine($"A magasság változása:
43.                             {String.Join(" ", irányok)}");

```

A feladatot előre eltárolt magasságokkal is megoldhatjuk, ez is egy másolás. Ha a 3 méteres szintkülönbségtől eltekintünk, akkor a kód két helyen módosul egy kicsit:

```

42. for (int i = 0; i < adatok.Length - 1; i++)
43. {
44.     if (magasságok[i + 1] > magasságok[i] + 3) /*itt +3*/
45.         irányok[i] = "/";
46.     else if (magasságok[i + 1] < magasságok[i] - 3) /*itt -3*/
47.         irányok[i] = "\\";
48.     else
49.         irányok[i] = "=";
50. }
51. Console.WriteLine($"A magasság változása:
52.                             {String.Join(" ", irányok)}");
53. }

```

41. példa: E terkes mecske leberetette e tejfelt – mintamegoldás

Ha nem akarunk cikluson belül másik ciklust írni, akkor a magánhangzóságra írhatunk külön függvényt. Eldöntés típusalgoritmus:

```

54. static bool BenneVan(char c, string magánhangzók)
55. {
56.     int ez = 0;
57.     while (ez < magánhangzók.Length && magánhangzók[ez] != c)
58.         ez++;
59.     return ez < magánhangzók.Length;
60. }

```

A „fordítás eszperente nyelvre másolás típusalgoritmussal:

```
61. static void Eszperente()
62. {
63.     Console.Write("Kérek egy mondatot: ");
64.     string magyar_mondat = Console.ReadLine();
65.     string eszperente_mondat = "";
66.     string magánhangzók = "aáééííoóőöuúüű";
67.     foreach (char c in magyar_mondat)
68.         if (BenneVan(c, magánhangzók))
69.             eszperente_mondat += 'e';
70.         else eszperente_mondat += c;
71.     Console.WriteLine("Gyenge eszperente: " + eszperente_mondat);
72. }
```

Nagybetűt is tartalmazó szövegek fordításakor figyelni kell arra is, hogy nagybetűs magánhangzó helyére 'E' kerüljön. Programozási nyelvtől függ, hogy a kisbetűssé alakítás függvénye karaktert vagy szöveget alakít át. C#-ban szöveget. Ezért a karakter kisbetűs alakja – kerülőúton – a karaktert tartalmazó szöveg kisbetűssé alakításának 0. karaktere.

A `Benne()` függvény helyett használható C# nyelven a `Contains()` tagfüggvény.

```
73. static void Eszperente2()
74. {
75.     Console.Write("Kérek egy mondatot: ");
76.     string magyar_mondat = Console.ReadLine();
77.     string eszperente_mondat = "";
78.     string magánhangzók = "aáééííoóőöuúüű";
79.     foreach (char c in magyar_mondat)
80.         if (magánhangzók.Contains(c.ToString().ToLower()))
81.             if (c == c.ToString().ToLower()[0])
82.                 eszperente_mondat += 'e';
83.             else
84.                 eszperente_mondat += 'E';
85.         else eszperente_mondat += c;
86.     Console.WriteLine(Eszperente: " + eszperente_mondat);
87. }
```

Sokkal egyszerűbb az oda-vissza karakter és szöveg átalakításnál, ha a nagybetűs magánhangzókat is változóba tesszük.

A feladatot másképp is megoldhatjuk, kiírt másolás algoritmus nélkül. Nem a mondat betűiből állítjuk össze az új mondatot, hanem az eredeti mondatban cserélünk ki karaktereket. Erre használható a `Replace()` tagfüggvény. Sokkal egyszerűbbnek látszik, de valójában az alábbi megoldásban a `Replace()` függvény minden futtatásakor két másolás történik.

```

88. static void Eszperente3()
89. {
90.     Console.Write("Kérek egy mondatot: ");
91.     string magyar_mondat = Console.ReadLine();
92.     string magánhangzók = "aáééíioóőőuúüü";
93.     string Magánhangzók = "AÁÉÉÍÍOÓŐŐUÚÜÜ";
94.     string eszperente_mondat = magyar_mondat;
95.     foreach (char c in magánhangzók)
96.         eszperente_mondat = eszperente_mondat.Replace(c, 'e');
97.     foreach (char c in Magánhangzók)
98.         eszperente_mondat = eszperente_mondat.Replace(c, 'E');
99.     Console.WriteLine("Replace-elt eszperente: " + eszperente_mondat);
100. }
101.

```

42. példa: Kutya és macskaoltások – mintamegoldás

Mivel a megoldáshoz nem kell eltárolni az állatok minden adatát, az `Allat` struktúra felesleges, de később könnyebben értelmezhető az `allat.Nev`, mint az `adatok[0]`.

```

102. struct Allat
103. {
104.     public string Nev;
105.     public string Faj;
106.     public int Kor;
107.     public Allat(string sor)
108.     {
109.         string[] adatok = sor.Split(' ');
110.         Nev = adatok[0];
111.         Faj = adatok[1];
112.         Kor = int.Parse(adatok[2]);
113.     }
114. }

```

A mondatyszerű leírástól eltérően, az alábbi megoldásban a `parvo` listába gyűjtés nem külön feltételben van, hanem a veszettségre gyűjtésen belül. Ez a megoldás hatékonyabb, mert a kutyákat nem az összes állat közül, hanem csak a macskák vagy kutyák közül választjuk ki.

```

115. static void KutyaMacska()
116. {
117.     StreamReader sr = new StreamReader("allatok.txt");
118.     List<string> veszett = new List<string>();
119.     List<string> parvo = new List<string>();
120.     while (!sr.EndOfStream)
121.     {
122.         Allat a = new Allat(sr.ReadLine());
123.         if (a.Faj == "kutya" || a.Faj == "macska")
124.         {
125.             veszett.Add(a.Nev);
126.             if (a.Faj == "kutya")
127.                 parvo.Add(a.Nev);
128.         }
129.     }
130.     sr.Close();
131.     Console.WriteLine($"Veszettség ellen kap: {LStr(", ", veszett)}");
132.     Console.WriteLine($"Parvovírus ellen kap: {LStr(", ", parvo)}");
133. }

```

43. példa: Tojásrakók – mintamegoldás

Itt is érdemes az eldöntést külön függvényben megírni. Az alábbi megoldás akkor jó, ha listában, vagy teljesen feltöltött tömbben keresgélünk.

```
135. static bool Egyike(string ez, List<string> lista)
136. {
137.     foreach (string l in lista)
138.         if (l == ez)
139.             return true;
140.     return false;
141. }
```

Kigyűjtés típusalgoritmusának egy változata a praktikus megoldás. Az eredeti adatsor hosszát nem ismerjük, az adatokat fájlból olvassuk be és „röptében” válogatunk. Ami nem kell, azt nem tároljuk el.

```
140. static void Tojásrakók()
141. {
142.     StreamReader sr = new StreamReader("allatok.txt");
143.     List<string> tojók = new List<string>(){ "kacsa", "liba", "kakas" };
144.     List<string> tojtNevek = new List<string>();
145.     while (!sr.EndOfStream)
146.     {
147.         Allat a = new Allat(sr.ReadLine());
148.         if (Egyike(a.Faj, tojók))
149.             tojtNevek.Add(a.Nev);
150.     }
151.     sr.Close();
152.     Console.WriteLine($"Tojásrakó fajhoz tartozik:
153.                         {String.Join(", ", tojtNevek)}");
154. }
```

44. példa: Jók és rosszak – mintamegoldás

Ebben a megoldásban is felhasználható az előző feladat `Egyike()` függvénye. A `LStr()` függvényt a jegyzetben előrébb lehet megtalálni. Ami azon túl marad az fájlból olvasás és a szétválogatás típusalgoritmus alkalmazása.

```
155. static void Orvell()
156. {
157.     StreamReader sr = new StreamReader("allatok.txt");
158.     List<string> tojók = new List<string>(){ "kacsa", "liba", "kakas" };
159.     List<string> jók = new List<string>();
160.     List<string> rosszak = new List<string>();
161.     while (!sr.EndOfStream)
162.     {
163.         Allat a = new Allat(sr.ReadLine());
164.         if (Egyike(a.Faj, tojók))
165.             rosszak.Add(a.Nev);
166.         else
167.             jók.Add(a.Nev);
168.     }
169.     sr.Close();
170.     Console.WriteLine($"Rosszak: {LStr(", ", rosszak)}");
171.     Console.WriteLine($"Jók: {LStr(", ", jók)}");
172. }
```

45. példa: Hajónapló – mintamegoldás

Ha egy sorban egy valamiről szerepelnek adatok, akkor célszerű a valamire struktúrát (vagy osztályt) létrehozni és a konstruktornak átadni a teljes sort. Így egy helyen kezeljük a felbontást és egyes részek értelmezését.

```

174. struct Haladás
175. {
176.     public int Irány;
177.     public int Táv;
178.     public Haladás(string sor)
179.     {
180.         string[] adatok = sor.Split('-');
181.         Irány = int.Parse(adatok[0]);
182.         Táv = int.Parse(adatok[1]);
183.     }
184. }

```

A mondatyszerű leírásban megadott részletből függvény lesz.

```

185. static string Irany(int regi, int uj)
186. {
187.     if ((uj - regi + 360) % 360 < 180)
188.         return "J";
189.     else if ((uj - regi + 360) % 360 > 180)
190.         return "B";
191.     Random r = new Random();
192.     return r.NextDouble() < 0.5 ? "B" : "J";
193. }

```

A feladat megoldásában az ismeretlen mennyiségű adatsor beolvasását egyszerűsíti a `File.ReadAllLines()`. Ez a fájlt megnyitja, végigolvassa és be is zárja. Az eredményből az adatsorok számát is megkapjuk. A megoldás kiírásakor figyelni kell az indexelésre: az egymás utáni elemekkel számolás során ne akarjunk a 0. előtti vagy az utolsó utáni adattal dolgozni.

```

194. static void Hajónapló()
195. {
196.     string[] fajl = File.ReadAllLines("hajonaplo.txt");
197.     Haladás[] naplo = new Haladás[fajl.Length];
198.     for (int i = 0; i < fajl.Length; i++)
199.         naplo[i] = new Haladás(fajl[i]);
200.     StreamWriter sw = new StreamWriter("kormanyzasok.txt");
201.     for (int i = 1; i < naplo.Length; i++)
202.         sw.WriteLine("{0} {1} {2}", naplo[i - 1].Irány,
203.             Irany(naplo[i - 1].Irány, naplo[i].Irány), naplo[i].Irány);
204.     sw.Close();
205. }

```

46. példa: Távirat – mintamegoldás

Annak is lenne értelme, ha minden karakterre megadnánk, hogy mi legyen helyette a telexben. Ezt fájlban kellene tárolni és ez lenne a telex kódtáblánk. Most csak a speciális karakterekhez adjuk meg a helyettesítést, ami szintén párokat jelent, de van egy trükkös,

egyszerűbb megoldás is: tegyük egy – szövegeket tartalmazó – listába felváltva az adatokat. A páros helyekre a speciális karaktert, utána, a páratlan indexű helyre a nekik megfelelő telex szöveget.

```
206. static void Távirat_1()
207. {
208.     List<string> helyett = new List<string>() { "Á", "AA", "É", "EE",
        "Í", "II", "Ó", "OO", "Ú", "UU", "Ö", "OE", "Ő", "OOE", "Ü", "UE",
        "Ű", "UUE", ".", " STOP", "?", " STOP", "!", " STOP",
        ",", " ", ":", " ", "\"", " ", "\u2026", " " }; /*az utolsó a ...*/
209.     Console.WriteLine("Mi lesz a távirat szövege?");
210.     string eredeti = Console.ReadLine().ToUpper();
```

Ki lehet emelni a megoldásból egy függvénybe az egyes karakterek megkeresését a helyettesítő tömbben. Ez a megadott egy hosszúságú szöveghez visszaadhatja a telexes helyettesítőt, vagy a bemeneten kapott szöveget. De most – szinte kivételként – legyen itt egy olyan megoldás, amelyben az egyik típusalgoritmusba bele van írva egy másik típusalgoritmus.

```
211. /*Másolás típusalgoritmus*/
212. string tavirat = "";
213. for (int i = 0; i < eredeti.Length; i++)
214. { /*Keresés típusalgoritmus: karakter keresése a helyett-ben */
215.     int ez = 0;
216.     while (ez < helyett.Count && eredeti[i].ToString() != helyett[ez])
217.         ez += 2; /*<= a trükkös tárolás miatt*/
218.     if (ez < helyett.Count)
219.         tavirat += helyett[ez + 1];
220.     else
221.         tavirat += eredeti[i];
222. }
```

Haladó programozóknak érdekes lehet, hogy ha a keresésre függvényünk van, akkor az hatékonyabbá tehető, ha átírjuk kiválogatásra. Ehhez az kell, hogy biztosan megtalálható legyen a karakter a helyettesíthetők között. A kódtábla, fájlból olvasás... nem hatékony, de nem is kell minden karakternek ott lennie, csak a keresettnek. Ez megoldható úgy, hogy a kiválasztás előtt a keresett karaktert kétszer hozzáadjuk a listához. Ha eredetileg benne volt, akkor azt fogja megtalálni, ha nem volt benne, akkor az elsőnek hozzáadottat találja meg és a másodiknak hozzáadottat – ami ugyanaz – adja vissza. Ha a függvényen belül történik mindez (ott a helyettesítések listája is), akkor a lista módosítása a végén elvész, mindig ugyanazzal a listával indul a függvényünk. Az eredmény: két elem hozzáadásával több, de egy feltétellel és egy elágazással kevesebb lesz a kód.

```
223. Console.WriteLine("Táviratként: " + tavirat);
224. }
225.
```

A C# függvényeinek használatával is rövidebb kódot kapunk. A megoldás áttekinthetőbb lesz, ha az eredeti szövegben cseréljük ki a karaktereket, amihez a helyettesítés listát veszünk végig. Olvashatóbb, de nem hatékonyabb ez a megoldás.


```

226. static void Távirat_2()
227. {
228.     List<string> helyett = new List<string>() { "Á", "AA", "É", "EE",
        "Í", "II", "Ó", "OO", "Ú", "UU", "Ö", "OE", "Ő", "OOE", "Ü", "UE",
        "Ű", "UUE", ".", " STOP", "?", " STOP", "!", " STOP",
        ",", " ", ":", " ", "\"", " ", "\\", " ", "\u2026", " " };
229.
230.     Console.WriteLine("Mi lesz a távirat szövege?");
231.     string szoveg = Console.ReadLine().ToUpper();
232.     for (int i = 0; i < helyett.Count; i += 2)
233.         szoveg = szoveg.Replace(helyett[i], helyett[i + 1]);
234.     Console.WriteLine("Táviratként: " + szoveg);
235. }
236.

```

Trükkös listánkat „normálisan” szótárban vagy karakter struktúra listában illene előállítani.

```
237. struct Karakter
238. {
239.     public char UTF;
240.     public string Telex;
241.     public Karakter(char c, string s) { UTF = c; Telex = s; }
242. }
```

A szótárbejegyzésként is megoldható struktúra konstruktora olyan rövid, hogy akár egy sorban is elfér. Nem a tördelés számít, hanem a tartalom.

A kifelé tekintők kedvéért az utolsó megoldásban C# szótárban vannak a helyettesítések:

```
243. static string Átalakít(string eredeti)
244. {
245.     Dictionary<char, string> UtfTelex = new Dictionary<char, string>() {
246.         {'A', "AA"}, {'E', "EE"}, {'I', "II"}, {'O', "OO"}, {'U', "UU"},
247.         {'ö', "OE"}, {'ő', "OOE"}, {'Ü', "UE"}, {'ű', "UUE"},
248.         {'.', " STOP"}, {'?', " STOP"}, {'!', " STOP"} };
249.
250.     List<char> mondatkozi = new List<char>()
251.     {
252.         ',', ';', ':', '\'', '-', '\u2026'
253.     };
254.
255.     string[] szavak = eredeti.Split(mondatkozi.ToArray(), StringSplitOptions.RemoveEmptyEntries);
256.
257.     string[] ujSzavak = new string[szavak.Length];
258.     for (int i = 0; i < szavak.Length; i++)
259.     {
260.         ujSzavak[i] = Átalakít(szavak[i]);
261.     }
262.
263.     string ujEredeti = string.Join(" ", ujSzavak);
264.
265.     return ujEredeti;
266. }
```

A megoldásban elenyészőnek látszik a különbség, de a 254–255. sor helyett a 214 – 219. sorokhoz hasonló megoldás, vagy annak megfelelő saját készítésű függvény kellene.

```
248. string szurt = "";
249. for (int i = 0; i < eredeti.Length; i++)
250.     if (!mondatkozi.Contains(eredeti[i])) /*először az eltűnők*/
251.         szurt += eredeti[i];
252. string tavisat = "";
253. for (int i = 0; i < szurt.Length; i++)
254.     if (UtfTelex.ContainsKey(szurt[i])) /*string-ben karakter keresése*/
255.         tavisat += UtfTelex[szurt[i]];
256.     else
257.         tavisat += szurt[i];
258. return tavisat;
259. }
```

Az **Átalakít()** függvény kódja viszonylag hosszú, cserébe a program maradék része szinte csak egy adatbekérés és eredménykiírás. Még ez is rövidíthető a 263. és 264. sor összevonásával.

```

260. static void Távirat_3()
261. {
262.     Console.WriteLine("Mi lesz a távirat szövege?");
263.     string eredeti = Console.ReadLine().ToUpper();
264.     Console.WriteLine("Táviratként: " + Átalakít(eredeti));
265. }

```

MINDENT BELE! – ÖSSZEFÜGGŐ FELADATSOR MEGOLDÁSA

Feladatok

Ahogy egy szép, harmatos reggelen szertenézünk nagy tölgyek alatt megbúvó, fehérre meszelt tanyákon, kérdések és feladatok özöne kezd bennünket nyugtalanítani:

1. Hány állatunk van összesen?
2. Melyik állatunk a legöregebb?
3. Van-e olyan fajú állatunk, amelyet a felénk járó postás (a felhasználó) kérdezett?
4. Írjuk ki az állatfajok listáját! Kérjünk be a felhasználótól ezek közül egy fajnevet! Írjuk ki, hogy az egyéves állataink között van-e ilyen fajú!
5. Melyik állatfajhoz tartozó állatból van a legtöbb?
6. Mennyi az állataink átlagéletkora fajonként?

És még egy feladatunk van:

7. Írjuk ki az egyes állatok neveit a fajuknak megfelelő nevű szövegfájlba!

Írjuk meg a fenti feladatokat megoldó programot `MindentBele` néven! A kipróbáláshoz rendelkezésünkre áll a már ismert `allatok.txt` fájl.

Megoldás praktikus, minimális nyelvi eszközzel

Az előző fejezetben többféle feladatot kellett megoldanunk, akár minden feladatot másik projektben megoldhattunk, ekkor az egyes feladatokra írt eljárások lettek a főprogramok. Ebben a feladatban ugyanazokról az adatokról szólnak a feladatok, ezeket hiba lenne különböző projektekben megoldani. De ezek túlmenően is vannak választási lehetőségeink:

- Az egyes részfeladatokat egymás után megoldhatjuk a főprogramban, vagy feladatonként készíthetünk eljárást, függvényt és a főprogramban csak a válaszokat adjuk meg. Van választásunk, de ha át szeretnénk látni a második órán is, hogy hol tartunk, akkor mindenképp a második módszert érdemes választani.
- Ha úgy döntünk, hogy programunk több eljárásból és függvényből fog állni, akkor el kell gondolkodni azon is, hogy az adatokat hogyan tároljuk. A forrásfájlban egy sorban egy állatról három adat szerepel. Kettő szöveges, a harmadik szám. Mivel már a 2. feladatban szükség van a szám értékére is, a sor nem lehet szövegtömb, struktúrába át kell áttenni.
- A tanya minden állatát figyelembe kell venni, ezért kell egy tömb vagy lista a tároláshoz. Mivel a tömbhöz nem kell kiegészítés, ezért ezt választjuk. Ezeket az adatokat minden részfeladatban fel kell használni, ezért az eljárásokon kívül, mindegyik számára láthatóan (statikusan) deklaráljuk. Így megspóroljuk a függvények paraméterezését.
- Végül még egy fontos elhatározás: A feladatokat nem varázslással oldjuk meg, hanem a típusalgoritmusok használatával.

A legegyszerűbb megoldást választottuk, ehhez csak két névteret kell bevennünk.

```
1. using System;
2. using System.IO;
3. namespace MindentBele
4. {
5.     class Program
6.     {
```

Döntéseinkből következően a megoldást a struktúra elkészítésével kezdjük. A konstruktorba egy beolvasott sort feldarabolva várunk.

```
7.     struct Allat
8.     {
9.         public string Nev;
10.        public string Faj;
11.        public int Kor;
12.        public Allat(string[] adatok)
13.        {
14.            Nev = adatok[0];
15.            Faj = adatok[1];
16.            Kor = int.Parse(adatok[2]);
17.        }
18.    }
```

A tömböt csak deklaráljuk az elején, mert még nem tudjuk, hogy mekkora lesz. Rögtön az első kérdés az állatok száma, és erre a többi részfeladatban is szükség lesz, ezért ezt is statikusan deklaráljuk.

```
19. static Allat[] allatok;
20. static int N;
21.
```

Az **1. feladat** megoldását az adatok beolvasásával együtt végezzük el. Nem a legszebb, de a legkevesebb megtanulandót jelentő módja a fájlból beolvasásnak, ha egyszerre betöltjük az összes sort. Ebből már tudjuk, mekkora tömb kell az állatoknak, ezért létrehozuk, majd másolás típusalgoritmussal feltöltjük. Mivel az `Allat` konstruktor a tömböt vár, a sort felbontva adjuk át. Csak azért, hogy a tömb mérete külön változóba kerüljön, a beolvasás nem eljárás, hanem függvény lesz, aminek a visszatérési értéke a tömb mérete.

```
22. static int Beolvas()
23. { /*Másolás*/
24.     string[] sorok = File.ReadAllLines("allatok.txt");
25.     allatok = new Allat[sorok.Length];
26.     for (int i = 0; i < sorok.Length; i++)
27.         allatok[i] = new Allat(sorok[i].Split(' '));
28.     return allatok.Length;
29. }
30.
```

Függvényünket a `Main()` eljárásban hívjuk meg, egyúttal értéket adunk az `N`-nek. Kiírjuk a választ és ezután jöhet a 2. feladat:

```
N = Beolvas();
Console.WriteLine($"1. Összesen {N} állatunk van.");
Console.WriteLine("2. A legöregebb neve: " + Legoregebb());
```

A 2. feladat egy maximumkiválasztás.

```
31. static string Legoregebb()
32. { /*Maximumkiválasztás*/
33.     int maxi = 0;
34.     for (int i = 0; i < N; i++)
35.     {
36.         if (allatok[maxi].Kor < allatok[i].Kor)
37.             maxi = i;
38.     }
39.     return allatok[maxi].Nev;
40. }
41.
```

A 3. feladatban a Main() eljárásban kérjük be a faj nevét, és a szöveges választ is ide írjuk.

```
Console.WriteLine("3. Adjon meg egy fajnevet: ");
string faj = Console.ReadLine();
if (Van(faj))
    Console.WriteLine($"Van {faj} a tanyán");
else
    Console.WriteLine($"Nincs {faj} a tanyán");
```

A kérdésből – Van-e ... – könnyen kitalálhatjuk, hogy eldöntés típusalgoritmusra lesz szükségünk. A fenti elágazás helyett egy sorban is lehet válaszolni, a háromoperandusú feltételes értékadás művelet segítségével (?:), de ez már nem tartozik a minimumhoz.

```
42. static bool Van(string faj)
43. { /*Eldöntés típusalgoritmus*/
44.     int ez = 0;
45.     while (ez < N && allatok[ez].Faj != faj)
46.         ez++;
47.     return ez < N;
48. }
49.
```

Eddig egyszerű volt a kérdésekre válaszolni, de a 4. feladatot már a kérdés hossza miatt is ijesztő. Nem is egy feladat...

A feladat vége nem lenne probléma, akár az előzőleg megadott fajt újra megadhatja a felhasználó, csak az eldöntésnél még arra is kell figyelni, hogy egy éves-e az állat. De minek ehhez az állatfajok listája? Abból kellene választania a felhasználónak. Persze kiírhatnánk az összes állat fajtaját, de hogy nézne ki, hogy válassz a macska és a macska közül? Ebben a feladatban nem a fajok kiírása a nehéz, hanem az, hogy mindegyik fajt csak egyszer szabad kiírni. Rádásul innentől kezdve minden feladatban a fajra vonatkozik a kérdés, nem az állatra.

Mivel a következő feladatban az egyes fajokhoz az egyedek száma is kell, olyan tömböt kell létrehozni, amiben egy elem megnevez egy fajt és egy hozzá tartozó darabszámot. Szöveg és szám együtt, tehát `struct`, és ilyenekből tömb. (Mintha a feladat elejét írnánk, csak nem fájlból, hanem egy másik adathalmazból kell megalkotni a tömböt.)

```

50. struct Fajta
51. {
52.     public string Faj;
53.     public int Db;
54.     public Fajta(string fajta) {Faj = fajta; Db = 1;}
55. }
56. static Fajta[] fajhalmaz;

```

A következő függvény ezt fajhalmazt fogja feltölteni adatokkal. Talán furcsa, de a visszatérési érték most nem a tömb hossza, hanem maga a tömb lesz. No, persze most sem tudjuk, hogy hány eleme lesz. (Persze, ha `List<>`-et használnánk, akkor ez nem lenne gond, de – ahogy mondani szokták, – szegény ember vízzel főz.) Az biztos, hogy legfeljebb annyi fajta van, ahány állat. Ezért ekkora méretű tömböt hozunk létre, valamint egy egész típusú változót, a mennyiség jegyzésére.

```

57. static Fajta[] Fajtak()
58. {
59.     Fajta[] temp = new Fajta[allatok.Length];
60.     int n = 0;

```

Most jön a neheze:

- végig kell nézni az összes állatot;
- amelyiknek a fajtája még nincs benne a `temp`-ben, azt a fajtát be kell tenni;
- amelyik benne van a `temp`-ben, annak a darabszámát meg kell növelni.

Honnan tudjuk, hogy benne van-e a fajta a `temp`-ben? Megkeressük... benne van-e vagy nincs... ehhez eldöntés típusalgoritmus kell, az eredménytől függ, hogy kigyűjtjük-e az új fajt.

```

61. for (int i = 0; i < allatok.Length; i++) /*minden állatot nézni*/
62. {
63.     int ez = 0; /*eldöntés: az i-edik állatnak temp-ben van a fajta?*/
64.     while (ez < n && temp[ez].Faj != allatok[i].Faj)
65.         ez++;
66.     if (ez < n) /*ha benne van*/
67.         temp[ez].Db++; /*növeljük a mennyiséget*/
68.     else /*ha nincs benne*/
69.     { /*a kigyűjtés típusalgoritmus alapján*/
70.         temp[n] = new Fajta(allatok[i].Faj); /*betesszük*/
71.         n++; /*1-gyel több fajta lett*/
72.     }
73. }

```

A `Fajta` konstruktor nem kér mennyiséget, de alaphoz 1-re állítja a `Db`-t. Ha nem lenne konstruktor, akkor 0 lenne a kezdőérték.

Levezetésként másoljuk át az adatokat egy megfelelő méretű tömbbe, és ezt adjuk vissza.

```

74. Fajta[] ki = new Fajta[n]; /*Másolás típusalgoritmus*/
75. for (int i = 0; i < n; i++)
76.     ki[i] = temp[i];
77. return ki;
78. }
79.

```

Valamint gondoskodjunk a főprogramban arról, hogy a függvényt a program végrehajtsa, és írassuk is ki őket, hogy lehessen választani.

```
fajhalmaz = Fajtak();
Console.WriteLine($"4. Az állatok listája: {Fajnevek(", ")}.");
```

Az eredmény egy lista, amit bejárás nélkül ír ki a főprogram, tehát a függvényben egyetlen szöveggé fűzzük össze az adatokat.

```
80. static string Fajnevek(string sep)
81. {
82.     string nevek = fajhalmaz[0].Faj; /*trükk: első sep nélkül*/
83.     for (int i = 1; i < fajhalmaz.Length; i++)
84.         nevek += sep + fajhalmaz[i].Faj;
85.     return nevek;
86. }
87.
```

Ezután jöhet a feladat második része: van-e adott fajú egyéves állat. A változatosság kedvéért, most a válasz háromoperandusú feltételes értékadással adjuk meg.

```
Console.Write("Válasszon egy fajnevet: ");
faj = Console.ReadLine();
Console.WriteLine("{1} egyéves {0} a tanyán", faj,
    Egyeves(faj) ? "Van" : "Nincs"); /*{1} helyén Van/Nincs*/
```

Ha az `Egyeves()` függvényt ebben a formában írjuk meg, akkor ragaszkodjunk a típusalgoritmushoz: „`ez < N` és nem jó”. A „jó” – az „ilyen fajú és egyéves” – feltételt tegyük zárójelbe és úgy tagadjuk.

```
88. static bool Egyeves(string faj)
89. { /*Eldöntés típusalgoritmus*/
90.     int ez = 0;
91.     while (ez < N && !(allatok[ez].Faj == faj && allatok[ez].Kor == 1))
92.         ez++;
93.     return ez < N;
94. }
95.
```

Ezzel az írásmóddal két hibától is megóvhatjuk magunkat. Egyrészt „nem jó” az az állat, amelyik „nem ilyen fajú vagy nem egyéves”, nagyon oda kell figyelni arra, hogy a tagadás zárójelen belülre írásakor az ‘és’ helyett ‘vagy’ kell.

```
91. while (ez < N && (allatok[ez].Faj != faj || allatok[ez].Kor != 1))
```

Másik, nehezebben átlátható hiba a zárójel elhagyása, mert az ‘és’ erősebb, mint a ‘vagy’:

```
ez < N && allatok[ez].Faj != faj || allatok[ez].Kor != 1 értelmezése:
(ez < N && allatok[ez].Faj != faj) || allatok[ez].Kor != 1.
```

Mi ezzel a gond? Konkrétan az, hogy ha a faj csiga vagy szarvasmarha, akkor túlindexelés miatt lefagy a program. Ezt pedig azért teszi, mert az informatikában az ‘és’ kiértékelése az első hamis eredményig tart, utána már úgysem lehet igaz a végeredmény sem. Hasonlóan, a ‘vagy’ kiértékelése az első igaz értékig tart. Egy helyes feltételben az `ez < N &&` védi a programunkat attól, hogy `ez >= N` esetén a másik oldalt kezdje vizsgálni a program. Ez a szabály zárójel nélkül is megakadályozza, hogy a faj-t megnézzék, az eredmény hamis lesz. De, ha a ‘vagy’ bal oldala hamis, akkor megnézi, mi van a jobb oldalon: egy nemlétező (mert `ez >= N`) állat kora. de, ha ez értelmezhető és igaz, akkor a legelső feltételtől függetlenül, igaz lenne a végeredmény is.

Természetesen van más megoldás is: az eldöntésre egy másik (jó) algoritmust is írhatunk.

Az **5. feladat** – mivel előkészítettük az előző feladatban – csak annyiban tér el a 2. feladattól, hogy teljes adatot (név és darab) érdemes visszaadni.

```

96. static Fajta MaxFajta()
97. {
98.     int maxi = 0;
99.     for (int i = 1; i < fajhalmaz.Length; i++)
100.         if (fajhalmaz[maxi].Db < fajhalmaz[i].Db)
101.             maxi = i;
102.     return fajhalmaz[maxi];
103. }
104.

```

A függvény meghívásakor az eredményt változóban tároljuk, hogy mindkét adatát újraszámolás nélkül ki tudjuk írni.

```

Fajta legtobb = MaxFajta();
Console.WriteLine("5. A {0} populáció a legnagyobb: {1} példány.",
    legtobb.Faj, legtobb.Db);

```

A **6. feladat** ismét összetett és a megoldása közben célszerű a következő feladat megoldását is előkészíteni. Mindkettőhöz fajonként kell a tanyán élő állatokat csoportosítani. Ezért a kérdésre válaszolás előtt átcsoportosítjuk az adatokat: fajonként egy-egy tömböt hozunk számukra létre. Az állatok nevének és korának a sora fajonként más-más elemszámú, de együtt kellene kezelni őket. Mivel a fajokat és ehhez az egyedek számát is kigyűjtöttük, tudjuk, hogy hány fajta állat van a tanyán és minden fajtáról tudjuk, hogy mennyi egyed van belőle. Az egyedeket így egy kétdimenziós táblázatban el tudnánk helyezni, csak hogy minden sor hossza más lesz. Ezért egy olyan tömböt hozunk létre, amelyiknek minden eleme egy tömb. (Kétdimenziós táblázattal is megoldható a feladat, ekkor a felső korlátot kell megadnunk az oszlopok számára, de ezt már kiszámoltuk: `MaxFajta().Db`.)

```

105. static Allat[][] fajtankent; /*a kezdetben nem ismerjük a méretét*/
106. static void Szetvalogat()
107. { /*szétválogatás = sok kiválogatás*/
108.     int fajdb = fajhalmaz.Length; /*ennyi fajta állat van*/
109.     fajtankent = new Allat[fajdb][]; /*csak a sorok számát adjuk meg*/

```

Minden fajtához létrehozunk annyi helyet, amennyi a `fajhalmaz` tömbben lévő adatok szerint kell. Ezt követően, a szétválogatásra több módszer is létezik, itt két stratégia jöhet szóba:

- Végigvesszük az egyes fajtákat, mindegyiket külön-külön kigyűjtjük az állatok közül.
- Végigvesszük az állatokat, mindegyiknek kikeressük, hogy hányadik sorban van a fajtája és ebbe a sorba betesszük.

Ez a két módszer továbbgondolásra érdemes, mert a mindennapi életben is szükségesek ilyen típusú választások. Ilyen például a kimosott ruhák leszedése és családtagok részére szétválogatása, vagy közös bevásárlást követően a termékek szétosztása, levelek szortírozása, rendrakás a LEGO elemek között. Matematikafeladat belátni, hogy egy személynek (egy proceszszornak) a második megoldás átlagosan fele annyi időt vesz igénybe, de vannak „egyéb körülmények”, amiktől az első lesz jobb.

Most az első módszert választjuk, mert a kiválogatás kódolása könnyebb, mint keresés és elhelyezés.

```

110. for (int f = 0; f < fajdb; f++) /*minden fajtára*/
111. { /* az aktuális fajta kiválogatása*/
112.     fajtankent[f] = new Allat[fajhalmaz[f].Db] /*pont annyi hely*/
113.     int egyed = 0; /*ennyi van meg már ebből a fajtából*/
114.     for (int i = 0; i < N; i++)
115.     {
116.         if (allatok[i].Faj == fajhalmaz[f].Faj)
117.         {
118.             fajtankent[f, egyed] = allatok[i];
119.             egyed++;
120.         } /*elágazás vége*/
121.     } /*kigyűjtés vége (i)*/
122. } /*fajták kiválasztásának vége (f)*/
123. } /*eljárás vége*/
124.

```

Ezt követően az átlagszámítás nem gond, egy összegzés típusalgoritmus kell hozzá. Mindegyik állatfajra kell, tehát az előző feladathoz hasonlóan a számítás egy for-cikluson belül lehet, az eredményt praktikusán egy tömbben vagy esetleg a fajta tulajdonságaként el lehet tárolni. Lehet, de a gyakorlati életben nem praktikus. A fajták száma esetleg tekinthető állandónak, egy változást észreveszünk a tanyánkon, de az átlagéletkor az idővel, szinte észrevétlenül változik. Erre inkább függvényt kellene írni, ami egy-egy fajra az aktuális értéket kiszámolja.

Kihasználva, hogy az állatokat úgy válogattuk szét fajtákra, hogy minden fajta külön tömbben van. Ugyanazt a függvényt tudjuk használni akár az összes állatra is és egy-egy fajtára is: paraméternek megadjuk az állatokat tartalmazó (egydimenziós) tömböt.

```

125. static double Atlag(Allat[] allatok)
126. { /*Összegzés*/
127.     double szum = 0;
128.     for (int i = 0; i < allatok.Length; i++)
129.         szum += allatok[i].Kor;
130.     return szum / allatok.Length;
131. }

```

A feladat megoldása a függvény meghívása minden fajtára:

```

132. static void Atlageletkorok()
133. { /*Másolás konzolra*/
134.     for (int i = 0; i < fajtankent.Length; i++)
135.         Console.WriteLine("\t{0,-14} {1,4:F2}", fajtankent[i][0].Faj,
136.                             Atlag(fajtankent[i]));
137. }

```

A megoldásban a kiírás igazítása nem része a minimálisan szükséges ismereteknek, de néha praktikus. A számok formátumának megadása viszont fontos, mert enélkül mindenféle matematikai varázslás kellene a megfelelő formátumú, helyes értékű kimenet előállításához. A megoldás kiírásait ezzel együtt érdemes végignézni. A különböző kiírási módok ismerete segíti az elvárt formátumú válasz gyors előállítását.

Az 5. feladatban a kigyűjtést (112–121. sorok) ugyanilyen módon megírhattuk volna paraméteres függvényként is. Egyéni ízléstől függ, hogy ha egy eljárást egy cikluson belül kell használni, akkor azt – merthogy sokszor használjuk – külön eljárásként vagy függvényként megírjuk, vagy inkább beleírjuk a ciklusmagba.

A főprogramunk vége:

```
Console.WriteLine("6. Az egyes fajták átlagéletkora:");
Szetvalogat();
Atlageletkorok();
Console.WriteLine("7. Fajtánként fájlokba írás");
Kiír();
```

A **7. feladat**, az adatok fájlba írása. Mivel a kiírandó adatokat már az előző feladatban előkészítettük, az egyetlen kihívás, hogy több, különböző nevű fájlt kell készíteni. Mivel a fájlnev egy szöveg, bármilyen eljárás, aminek az eredménye az a szöveg, megteszi. Mivel a szétválogatás elemei állatok, az egyes sorokban olyan állatok vannak, akiknek a Faj tulajdonsága megegyezik. A sorban első állat faja épp megfelel a fájlnev elejének, ehhez kell a kiterjesztést hozzáfűzni.

Másik kérdés, hogy hány `StreamWriter` változó kell a fájlba íráshoz. Az biztos, hogy változónévből elég egy (`sw`). Az is biztos, hogy ehhez a cikluson belül rendeljük hozzá a fájl nevét és ugyanazon a cikluson belül le is zárjuk a fájlt. Ha az `sw` deklarációja így (140. sor), a cikluson kívül van, akkor ugyanazt a változót használjuk újra és újra, reinkarnálódik az `sw`. Ha a 143. sor elején deklaráljuk az `sw` típusát, akkor minden cikluslépésben újra létrehozza a program. De a fordítóprogram készítői sem estek a fejükre, nagyon jó eséllyel ugyanazt a memóriát fogja a program felhasználni az újbóli létrehozáshoz. Így a válasz: mindegy.

```
138. static void Kiír()
139. { /*Másolás, fájlbaírás*/
140.     StreamWriter sw;
141.     for (int i = 0; i < fajtankent.Length; i++)
142.     {
143.         sw = new StreamWriter(fajtankent[i][0].Faj + ".txt");
144.         for (int a = 0; a < fajtankent[i].Length; a++)
145.             sw.WriteLine(fajtankent[i][a].Nev);
146.         sw.Close();
147.     }
148. }
```

Megoldás Lambdával és egyéb virtuóz eszközökkel

A kilencedikes, a tizedikes és ez a jegyzet is számos kiegészítést tartalmaz. Használtunk listát és osztályt is. Ha ezeket a kiegészítéseket helyezzük a fókuszba, akkor egy egész más jellegű nyelvet tudunk használni. Napjaink ismertebb programozási nyelvei általános célúak, ez azt jelenti, hogy a strukturált, a procedurális, illetve a funkcionális és az objektum orientált programozást is valamilyen szinten támogatják. Az egyes nyelveket jellemzi, hogy melyik paradigmát milyen mértékben helyezi előtérbe, melyikre optimális. A C# nyelv alkotói a programozók igényeihez igazítják a nyelv szabályait. A fejlődésével egyre több funkció kerül bele, amelyek egyre több platformon használhatók. Ilyen a Lambda kifejezés is, ami a tagfüggvényeket kiegészítve helyettesíti a típusalgoritmusok procedurális kódját. Az alábbi megoldás a minimális eszközökkel történő megoldásunk átirata. Néhány módosítás

- A `struct` helyett `class` kell, ezért ez a `class Program` elé került, de csak az első szó módosult.
- A tömb helyett `List<>` típus szerepel.
- Az eljárásokat és függvényeket C# függvények helyettesítik, így gyakorlatilag értelmetlenné vált a feladat részekre bontása. A teljes kód a `Main()` eljárásan belül található.

- Mivel nincsenek függvények, felesleges a statikus adat, ezek a `Main()`-en belülre kerültek.
- A 7. feladat nem változna érdemben, az 1–6. feladatok megoldása készült el.

Az eredményt érdemes összevetni az előző megoldás egyes részleteivel:

```

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.IO;
5. namespace MindentBeleLambda
6. {
7.     class Allat
8.     {
9.         public string Nev;
10.        public string Faj;
11.        public int Kor;
12.        public Allat(string[] adatok)
13.        {
14.            Nev = adatok[0];
15.            Faj = adatok[1];
16.            Kor = int.Parse(adatok[2]);
17.        }
18.    }
19.    class Fajta
20.    {
21.        public string Faj;
22.        public int Db;
23.        public Fajta(string fajta) { Faj = fajta; Db = 1; }
24.    }
25.    class Program
26.    {
27.        static void Main()
28.        {
29.            List<Allat> allatok = new List<Allat>();
30.            string[] sorok = File.ReadAllLines("allatok.txt");
31.            foreach (string s in sorok)
32.                allatok.Add(new Allat(s.Split(' ')));
33.            Console.WriteLine($"1. Összesen {allatok.Count} állatunk van.");
34.
35.            Console.WriteLine("2. A legöregebb neve: " +
36.                allatok.Find(a => a.Kor == allatok.Max(a => a.Kor)).Nev);

```

Itt érdemes megállni egy pillanatra. Bár a megoldás csak 1 sor, de a futási ideje ennek a kódnak átlagosan 1,5-szerese a típusalgoritmussal megírt kódnak. A `Max()` egyszer végigmegy az adat-sorozaton, ezt követően a `Find()` amíg meg nem találja – átlagosan az adatok felét – újra előveszi. Nagyon sok alkalmazás, ami kicsiben jól működik, nagyméretű adathalmazmal az ilyen típusú megoldások miatt válik használhatatlanná.

```

37. Console.Write("3. Adjon meg egy fajnevet: ");
38. string faj = Console.ReadLine();
39. Console.WriteLine("{0} {1} a tanyán",
40.     allatok.Exists(a => a.Faj == faj) ? "Van" : "Nincs", faj);

```

Fajtak() helyett:

```

41. List<Fajta> fajhalmaz = new List<Fajta>();
42. foreach (Allat allat in allatok)
43.     if (!fajhalmaz.Exists(k => k.Faj == allat.Faj))
44.         fajhalmaz.Add(new Fajta(allat.Faj));
45.     else
46.         fajhalmaz.Find(k => k.Faj == allat.Faj).Db++;
47. Console.WriteLine($"4. Az állatok listája:
    {String.Join(" ", fajhalmaz.Select(f => f.Faj))}.");
48.
49. Console.Write("Válasszon egy fajnevet: ");
50. faj = Console.ReadLine();
51. Console.WriteLine("{0} egyéves {1} a tanyán",
    allatok.Exists(a => a.Faj == faj && a.Kor == 1) ? "Van": "Nincs", faj);
52.
53. Fajta legtobb =
    fajhalmaz.Find(f => f.Db == fajhalmaz.Max(f => f.Db));
54. Console.WriteLine("5. A {0} populáció a legnagyobb: {1} példány.",
    legtobb.Faj, legtobb.Db);
55.

```

Szetvalogat() helyett:

```

56. List<List<Allat>> fajtankent = new List<List<Allat>>();
57. foreach (Fajta f in fajhalmaz)
58.     fajtankent.Add(
        new List<Allat>(allatok.FindAll(a => a.Faj == f.Faj)));
59. Console.WriteLine("6. Az egyes fajták átlagéletkora:");
60. foreach (List<Allat> allat in fajtankent)
61.     Console.WriteLine("\t" + allat[0].Faj.ToString().PadRight(14) +
        " " + allat.Average(a => a.Kor).ToString("F2").PadLeft(4));
62. }
63. }

```

GYAKRAN HASZNÁLT ÖSSZETETT ALGORITMUSOK

A 11-es tankönyvben szó esik a rendezésről, a metszetképzésről és az unió képzéséről. Ezeket fogjuk most megnézni több más, gyakran használt algoritmussal kiegészítve.

Egyesítés

Típusalgoritmusokkal foglalkozó leckéink közül most két olyan módszerrel ismerkedünk meg, amelyek nem is egy, hanem két vagy több kiindulási adatszerkezethez rendelnek eredményt. Ezek sok szempontból hasonlítanak a kigyűjtéshez, de a kiválasztás feltétele két (vagy több) adatsorozathoz kapcsolódik.

A metszetképzés és az unió valójában egyaránt halmazművelet. Két halmaz metszete egy olyan halmaz, amelyik a két halmaz közös elemeit tartalmazza. Két halmaz uniója egy olyan halmaz, amelyik a két halmaz minden elemét tartalmazza – a közöset is, meg azokat is, amelyek csak az egyik kiindulási halmazban vannak benne.

Metszetképzéskor a két lista közös elemeit írjuk újabb listába, egyesítéskor pedig az egyik lista elemeit bővítjük a második listában lévő, az elsőben eddig nem szereplő elemekkel. A listák és a tömbök azonban két fontos dologban különböznek a halmazoktól:

- a listákban és a tömbökben előfordulhat egy-egy elem többször is, a halmazokban nem;
- a listákban és a tömbökben kötött sorrendjük van az elemeknek, a halmazokban nincs.

Ebben a fejezetben feltesszük, hogy nincs ismétlődés egy-egy adatsorozat elemei között, de a sorrendet kötöttnek tekintjük. Gyakorlatilag ez azt jelenti, hogy nincsen sorbarendezebe, hanem a helyzetből adódó vagy a feltöltés sorrendjében tudjuk elérni az egyes elemeit a sorozatnak.

Ha az eredmény adatsorozat, mindig kérdés, hogy milyen típusú. Láthattuk, hogy a tömb kevésbé felel meg a célnak, amikor nem ismerjük előre az eredmény elemszámát. Metszet esetén azt mondhatjuk, hogy az eredmény el fog férni egy akkora tömbben, mint a kevesebb elemszámú adatsorozat mérete. Unió esetén a két adatsorozat elemszámának az összegével kell számolnunk. De most ezzel nem foglalkozunk, inkább listába gyűjtjük a megfelelő (kigyűjtendő) elemeket.

Metszetképzés

47. példa: Tömegközlekedés Százzorszépvárosban

Százzorszépvárosban szeretnénk közösségi közlekedéssel eljutni a 92-es villamos végállomásáraól, a Fapapucs sugárútról a 19-es busz Balambér tér nevű megállójába. A két járat megállói listaként állnak rendelkezésünkre. Melyik megállóban tudunk átszállni a villamosról a buszra?

A közös megálló nevére van szükségünk, azaz metszetet kell képeznünk. A feladat szövege alapján feltételezhetők, hogy mindkét járaton a megállók nevei egymástól eltérnek (nincs ismétlődés).

A következő műveleteket kell kódolnunk az `atszallas` nevű programban:

- Felveszünk egy üres listát, például átszállások néven.
- Ciklussal bejárjuk a villamos valamennyi megállóját.
- Ha az aktuális villamosmegálló a buszjáratnak is megállója, és még nem szerepel az átszállások listában, akkor hozzáfűzzük.

Egy pillanatra gondoljunk vissza a tanya állatai programra. Ott elő kellett állítani a fajtákat tartalmazó tömböt, aminek a megoldási terve a fenti leíráshoz nagyon hasonló. Miben más?

Kódoljuk a programot!

Programunkban a `List<>` típus miatt szükség lesz a `System.Collections.Generic` névtérre. A bővíthetőség miatt az adatokat statikusan vesszük fel, a megoldásra eljárást készítünk, amit

a `Main()` eljárásban hívunk meg, de ezt a jegyzetben most már nem írjuk ki, de a kódot a 20. sortól számozzuk.

```
20. static string[] vill92 = new string[7] {"Kiss u.", "Zöld fasor",
21.                                     "Piros tér", "Malom-patak", "Puccos u.",
22.                                     "Remegő-erdő", "Fapapucs sgt."};
23. static string[] busz19 = new string[8] {"Nagy u.", "Balambér tér",
24.                                     "Szent Lajos hídja", "Által-ér", "Reghős Bendegúz sz.k.",
25.                                     "Puccos u.", "Remegő-erdő", "Pöszke-liget"};
26.
```

A közös megálló listájában a vill92 megállói közül azokat gyűjtjük ki, amelyek benne vannak a busz19 megállói között is.

```
27. static void Atszallas()
28. {
29.     List<string> atszall = new List<string>();
30.     for (int ez = 0; ez < vill92.Length; ez++)
31.     {
32.         if (BenneVan(vill92[ez], busz19))
33.             atszall.Add(vill92[ez]);
34.     }
35.     Console.WriteLine($"Átszállási lehetőségek:
36.                                     {string.Join(", ", atszall)}.");
37. }
```

A kigyűjtés feltétele, hogy a másikban benne van-e, ezt el kell dönteni.

```
38. static bool BenneVan(string elem, string[] tomb)
39. { /*Eldöntés típusalgoritmus*/
40.     int itt = 0;
41.     while (itt < tomb.Length && elem != tomb[itt])
42.         itt++;
43.     return itt < tomb.Length;
44. }
45.
```

A feladat megoldható az adatsorozatok `Intersect()` függvényével is

```
46. static void AtszallasL()
47. {
48.     List<string> atszall = vill92.Intersect(busz19).ToList();
49.     Console.WriteLine($"Átszállási lehetőségek:
50.                                     {string.Join(", ", atszall)}.");
51. }
```

Unió

48. példa: Fricska és Kökény első szavainak jegyzése

A szomszédban lakó két bölcsis – Nagy Fricška és Zsémber Kökény – első hét, illetve nyolc szavát feljegyezték anyukáik. Közösben tartott névnapi zsűrjukon olyan dalt akarnak nekik énekelni, amely emléket állít első szavaiknak. Állítsuk össze azt a szójegyzéket, amelyben a két tipegő minden feljegyzett szava szerepel, ismétlődés nélkül!

A két jegyzék minden szavára szükségünk van, tehát az egyesítés típusalgoritmusát használjuk. A következő műveleteket kell kódolnunk, ha nem használunk halmaz adattípust:

- Felveszünk egy üres listát.
- Az üres listába átmásoljuk az első lista elemeit.
- A – most már nem üres – listába átmásoljuk a második listából azokat az elemeket, ami még nincs benne.

Végezzük el a kódolást a `kozos_szavak` programban!

Mielőtt nekifogunk a kódolásnak, kitérünk a listák másolásával kapcsolatos sajátosságára. Ha egy listának egy másik listát értékadással (= jellel) próbálunk átadni, akkor az új listánk pontosan a régi lesz, csak másik néven. Ilyenkor az új változó megkapja a régi változótól a lista helyének a címét, de az adatokat nem fogja duplikálni. Ez nagyon jó például akkor, ha egy függvény értékét adjuk így át, de a duplikáláshoz vagy másolás algoritmus kell vagy egy ezt megvalósító függvény.

A program részeit az átszállás programhoz hasonlóan szervezve és a `BenneVan()` függvényt átvéve, a közös szavakat így kaphatjuk meg:

```
20. static string[] fricska = new string[7] {"anya", "maci", "baba",  
21.     "apa", "kaja", "ló", "erősáramú feszültségszabályzó"};  
22. static string[] kökény = new string[8] {"apa", "autó", "anya",  
23.     "víz", "maci", "könyv", "nagyi", "pörgettyűs tájoló"};  
24.  
25. static void Szojegyzek()  
26. {  
27.     List<string> szojegyzek = new List<string>();  
28.     for (int i = 0; i < fricska.Length; i++)  
29.         szojegyzek.Add(fricska[i]);
```

Fricska szavainak átmásolása után Kökény szavairól eldöntjük, hogy benne vannak-e Fricska szavai között. Ha nincsenek benne, akkor adjuk hozzá a közös jegyzékhez.

```
30. for (int i = 0; i < kökény.Length; i++)  
31.     if (!BenneVan(kökény[i], fricska))  
32.         szojegyzek.Add(kökény[i]);  
33. Console.WriteLine($"A kész szójegyzék:  
34.     {string.Join(", ", szojegyzek)}.");  
35. }
```

Listafüggvényekkel megoldva, kicsit rövidebb a kód. Először létrehozuk a `szolista`-t a `fricska` listából. Ez átmásolja az adatokat. Ezt követően, (rossz angolsággal, de) szinte szó szerint azt írjuk, amit hozzáadási szabályként elmondunk.

```
36. static void SzojegyzekL()  
37. {  
38.     List<string> szolista = new List<string>(fricska);  
39.     foreach (string szo in kökény)  
40.         if (!szolista.Contains(szo))  
41.             szolista.Add(szo);  
42.     Console.WriteLine($"A szó-lista: {string.Join(", ", szolista)}.");  
43. }  
44.
```

Ahogy a metszetnek, úgy az unióképzésnek is van kész függvénye, az `Union()`. Ezzel egy sor a megoldás:

```
45. static void SzोजgyzekU()
46. {
47.     Console.WriteLine($"Unió:
                           {string.Join(", ", fricska.Union(kökény))}");
48. }
49.
```

És a többi halmazművelet

Lényegében minden adatsorozat műveletre van függvény C# nyelven, a Linq készletben. A LINQ a Language Integrated Query kifejezés rövidítése, azaz egy olyan nyelvi eszközkészlet, amivel mindenféle – relációs – adatbázis-művelet megvalósítható. Mivel a relációs adatbázisban a tábla rekordokból álló lista, de egyúttal rekordhalmaz is, ezért amit adatbázis-kezelésben meg tudunk csinálni, arra van C# LINQ kifejezés is. Például a listák egybefűzésére a `Concat()`, az ismétlődések eltávolítására a `Distinct()`, amelyekkel két lépésben szintén előállítható az két szöveglista uniója.

```
50. static void SzोजgyzekD()
51. {
52.     List<string> szavak = fricska.Concat(kökény).ToList();
53.     Console.WriteLine($"Fűzés: {string.Join(", ", szavak)}");
54.     szavak = szavak.Distinct().ToList();
55.     Console.WriteLine($"Egyedi: {string.Join(", ", szavak)}");
56. }
57.
```

A metszeten és unión túl, igény lehet két halmaz különbségére, ez az `Except()` (azaz „kivéve”) függvény. Ezeket felhasználva a szimmetrikus különbség is előállítható: az unióból kivonjuk a metszetet.

```
58. static void Extrak()
59. {
60.     List<string> frics = fricska.Except(kökény).ToList();
61.     Console.WriteLine($"Csak fricska mondja:
                           {string.Join(", ", frics)}");
62.     List<string> metszet = fricska.Intersect(kökény).ToList();
63.     Console.WriteLine($"Mindkettő mondja:
                           {string.Join(", ", metszet)}");
64.     List<string> spec = fricska.Union(kökény).Except(metszet).ToList();
65.     Console.WriteLine($"Csak egyik mondja: {string.Join(", ", spec)}");
66. }
```

Rend a lelkünk – A rendezés algoritmusai

Adatokat számítógéppel rendezni – mind a mai napig – az informatika egyik legérdekesebb problémája. Mindennapos élményünk, hogy egy weboldal – kereső, webshop, internetes adatbázis – sok ezer találatot, terméket, szócikket jelenít meg egyik-másik szempont szerint rendezve. A rendezéseknek ilyenkor nagyon gyorsan kell megtörténniük, különben a weboldal látogatója elunja a várakozást és továbbáll. Az alkalmazások tervezői praktikák sokaságát vetik be, hogy a rendezett eredmények minél gyorsabban megjelenjenek, de a gyors eredmény alapja mindig a megfelelően megválasztott, hatékony rendezési algoritmus.

Eddigi típusalgoritmusainkról azt mondhatjuk, hogy az elemszámtól lineárisan függ a tároláshoz szükséges méret, azaz kétszer akkora adatsorozattal dolgozva durván kétszer akkora helyre van szükség a memóriában. Ha ez probléma, akkor egyes feladatokat „röptében” – az adatsorozat tárolása nélkül, elemenként feldolgozva a sorozatot – is megoldhatunk, így memória foglалásban hatékonyabb lehet. Hasonlóan a futtatási időről is érezhető, hogy ha ezerszeresre növeljük az adatok számát, akkor körülbelül ezerszeres lesz a futtatási idő is. Egy típusalgoritmus a kevésbé hatékony megoldástól abban különbözik, hogy adott mennyiségű adatra a futási idő esetleg csak fele annyi.

A rendezésre egy egyszerű megoldás az elemszám növelésével nem lineárisan, hanem négyzetesen növeli meg a futási időt. Ha ezerszer több adatot rendezünk, az milliószor több időt vesz igénybe. Képzeljük el, hogy van egy program, ami egy osztályra 1 ms alatt lefut, de egy iskolára már 0,5 s, a végzős középiskolásokra 3 óra lenne szükséges. Ezért fizetik meg nagyon azokat, akik jó praktikákat találnak ki, akik javítani tudnak a meglévő algoritmusok hatékonyságán.

Ebben a fejezetben több „közismert” rendezési algoritmussal megismerkedhetünk, de ez csak töredéke annak, ami az interneten fellelhető és az is csak minta a profi alkalmazásokban használt titkos módszerekhez képest. Egyiket sem kell megtanulni.

Mezítlábas rendezés

Mezítlábas (barefoot) programozásnak hívják azt, amikor valójában nem programozunk, hanem eljártsszuk az algoritmust. A rengeteg rendezési algoritmusból annak az egynek a kódolását érdemes (először) megtanulni, amelyiket mi magunk is kitalálunk, aminek legalább a vázlatát, mondatszerű leírását el tudjuk készíteni. Lehet, hogy egy csoportban mindenki más-más stratégiát választ, de a lényeg, hogy mindenkinek a saját módszere a legkönnyebben megérthető – hiszen azt ő találja ki.

Majd, ha már megvan a rendezés stratégiája, akkor érdemes körülnézni a rendezési algoritmusok között, hogy melyik az, amelyik eszerint végzi a rendezést, mert ezt lesz a legkönnyebb a számítógépen is megvalósítani. ...és ennél nem is kell több.

A megfelelő stratégia kialakításához szükség lesz néhány szabályra:

- A rendezést egy adatsorozaton úgy kell végrehajtani, hogy a végeredmény az eredeti adatsorozat helyén legyen. Ugyanott, csak más sorrendben lesznek az adatok.
- A rendezéshez nem használhatjuk ki, hogy ismerjük az adatokat, hogy emlékszünk, látjuk, hogy mit hova tettünk korábban, mert erre a számítógép nem képes, a keresés pedig azt jelenti, hogy egyenként kell megnézni az adatokat. Például különböző méretű kockákat csukott szemmel kell rendezni.
- A rendezés során a rendezendő elemek csak adott helyeken lehetnek, ami a sorozaton kívül van, azt is figyelembe kell venni. Például, ha különböző mennyiségű folyadékokat kellene áttöltéssel térfogat szerint rendezni, nem tölthetjük átmenetileg sem a folyadékot az asztalra és nem tudunk két tartályban „feldobással” folyadékot cserélni.

A csere algoritmus

Az utolsó pont megvalósításához szükséges a csere algoritmusának az ismerete. Ennek lényege: ahhoz, hogy az egyik adatot a másik helyére tegyük, előbb a másikat „félre” kell tenni. Másképp: mielőtt az adatot felülírnánk, készítsünk róla másolatot valahol. Ez a „valahol” egy

konkrét hely, hiszen később szükség lesz rá. Itt átmenetileg (temporally) tároljuk az adatot, ezért a neve gyakran `temp` vagy `tmp`.

Csere algoritmus:

```
temp := egyik
egyk := másik
másik := temp
```

Csere algoritmus C# kód részlete:

```
Adattípus temp = egyik;
egyk = másik;
másik = temp;
```

Figyeljük meg: első a `temp`; a következő sor mindig azzal kezdődik, ami az előző végén van; a legvége ismét a `temp` – így zárul be az átadási kör.

A csere a rendezés alpművelete, ezért sok programozási nyelvben van rá eljárás, általában `swap` néven, de ennek a használata mindenképp nagy körülményt igényel, mert a csere lényegében három másolás, de egy listát más módszerrel kell másolni, mint egy egész számot. Már akkor is észnél kell lenni, ha egy tömb két elemének cseréjére paraméteres eljárást készítünk. Nem mindegy, hogy a tömb elemet vagy a cserélendő elemek helyét adjuk meg.

Két érték (pl. tömbelem) cseréje:

```
static void Csere
    (ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Egy tömb két helyén lévő értékek cseréje:

```
static void Csere
    (int[] Tömb, int i1, int i2)
{
    int temp = Tömb[i1];
    Tömb[i1] = Tömb[i2];
    Tömb[i2] = temp;
}
```

Ha két egész számot adunk felcserélésre, akkor `ref` nélkül csak a másolatot cseréli az eljárásunk, az eredeti számokkal nem történik semmi. Ha a tömböt és a felcserélendő adatok indexét adjuk meg, akkor a két megadott index másolata teljesen jó, azokat nem módosítjuk, csak azt a két értéket, amelyekre mutatnak. A sikerhez azonban az is hozzá tartozik, hogy a tömb másolata valójában a tömb elejére mutató hivatkozás másolata, így a tömbelemek `ref` nélkül is az eredeti adatok, nem a másolatuk.

Helyben szétválogatás

A sorbarendezés nagyon gyakori feladat, de sokszor nincs szükség teljes rendezettségre, elegendő az adatok részleges rendezése. Az egyik ilyen részművelet lehet növekvő rendezéshez a legkisebb elem kiválasztása – ezt már tanultuk. Egy másik részművelet lehet, hogy az első két elemet megfelelő sorrendbe tesszük, ha kell felcseréljük – ehhez van már csere algoritmusunk. További lehetőség az szétválogatás: előre a kicsik, hátulra a nagyok.

Szétválogatásra van típusalgoritmusunk, ami egy adatsorozatból két (több) adatsorozatba másolja az adatokat. Az adatok átrendezésének az is lehet egy lépése, hogy kicsikre és nagyokra szétválogatjuk az adatokat, majd a két listát visszamásoljuk az eredeti helyére.

Egy kis ügyeskedéssel a két listát és a visszamásolást elkerülhetjük. Erről szól a helyben szétválogatás típusalgoritmus. A szétválogatás stratégiája: az adatsorozatot mindkét végéről egymásfelé haladva járjuk be és a nem odaváló elemeket cseréljük ki.

Például: egy osztály tanulói névsorát kell átrendeznünk úgy, hogy elől legyenek a lányok, hátul a fiúk. A megoldás: az előlről az első fiút kicseréljük hátulról az első lánnyal. Ezt addig ismétel-

jük kétirányból egymásfelé, amíg össze nem ér a két válogatott sorozat. Figyeljünk: nem középen fejezzük be a szétválogatást; legfeljebb annyiszor kell cserélni, amennyi a kisebb rész létszáma.

49. példa: Rosszak előre, jók hátra

Szétválogatással adtuk meg korábban (44. példa:), hogy melyek a jó (négylábú) és rossz (kétlábú) állatok a tanyánkon. Helyben szétválogatás célja lehet az, hogy előre kerüljenek a rosszak (kétlábúak), a végére a jók (négylábúak). A válogatás során előlről az első négylábút kicseréljük hátulról az első kétlábúval. Így mindkét végén a sorozatnak több elem lesz a helyén. ugyanezt folytatva, idővel eljutunk addig, hogy kétirányból összeérnek a rendben lévő sorozatok.

Így, elmondva, nem bonyolult a stratégia, de a kódolása során sok apróságra kell figyelni. Az `orwell` program szétválogatás feladata helyben például így kódolható:

```
1. static void Main_OrvellHelyben()
2. {
3.     List<string> tojók = new List<string>(){ "kacsa", "liba", "kakas" };
4.     int N = 15;                               /*az allatok.txt-ben*/
5.     Allat[] allatok = new Allat[N];
6.     Beolvasas()                               /*megírt eljárás*/
7.     /*szétválogatás mutatóinak a kezdőértéke:*/
8.     int eleje = 0;
9.     int vége = N - 1;
10.    int rosszdb;                               /*az első rész hossza, itt a kétlábúak száma*/
11.    while (eleje < vége)
12.    {
```

Az algoritmus több ponton módosul, ha innentől egy kicsit másképp gondolkodunk. (Később azt is megnézzük.) A sorozat elején kell lennie a kétlábúaknak, ezért az `eleje` változónk addig továbbléphet, amíg kétlábúnak az adatára mutat.

```
13.        /*négylábút keres előlről*/
14.        while (eleje < vége && Egyike(allatok[eleje].Faj, tojók))
15.            eleje++;
16.        if (eleje < vége) /*ha nem értek össze*/
17.        {
```

Ha találtunk egy négylábút... De ha nem, akkor: vagy csak kétlábú állatunk van (`eleje == N - 1`), vagy a végétől hátrafelé mindegyik négylábú, előre felé mindegyik kétlábú, tehát nincs teendőnk. Ha `eleje < vége`, akkor a kettő közötti részt még nem válogattuk szét.

```
18.        /*kétlábút keres a végétől visszafelé*/
19.        while (eleje < vége && !Egyike(allatok[vége].Faj, tojók))
20.            vége--;
21.        }/*elágazás vége; lehet, hogy közben összeértek*/
```

Ezen a ponton, ha csak négylábú állataink vannak, akkor az `eleje` értéke `0`, a `vége` értéke is `0`. Ha a két érték egyenlő, akkor lehet, hogy csak az egyik keresés futott le, lehet, hogy mindkettő, de biztos, hogy véget ért az átrendezés. Ha `eleje < vége`, akkor kicseréljük a mutatott két elemet.

```

22.     if (eleje < vége) /*ha mindkét irányból talált nem odaválót*/
23.     { /*felcserélés*/
24.         Allat temp = allatok[eleje];
25.         allatok[eleje] = allatok[vége];
26.         allatok[vége] = temp;
27.     }

```

Megfontolandó, hogy a csere után 1-1 lépéssel közelebb léptethetjük-e a mutatókat, és ha igen, akkor ennek milyen hatása lesz a befejezésre. Most azt tudjuk mondani, hogy ha mindegyik kétlábú ($\text{eleje} == N - 1$), akkor ennél eggyel több kétlábú van. Egyébként, amikor az elejét léptetjük, akkor a vége egy négy lábúra mutat: vagy a legelsőre, vagy az éppen kicserélt elemre, ami emiatt négy lábú. Mindkét esetben az eleje éppen a kétlábúak számát fogja megadni.

Az elejére válogatott elemek számának ismeretében a két lista külön is kezelhető:

```

28.     } /*ciklus vége: eleje és vége összeértek | átlépték egymást*/
29.     /*ha eleje == vége, akkor milyen állat van ott?*/
30.     if (Egyike(allatok[eleje].Faj, tojók)) /*még ez is...*/
31.         rosszdb = eleje + 1; /*0..eleje 2lábú, eleje+1..N-1 4lábú*/
32.     else
33.         rosszdb = eleje; /*0..eleje-1 2 lábú, eleje..N-1 4lábú*/
34.     Console.WriteLine("Rosszak: ");
35.     for (int i = 0; i < rosszdb; i++)
36.         Console.WriteLine($"{allatok[i].Nev}, {allatok[i].Faj}");
37.     Console.WriteLine("\nJók: ");
38.     for (int i = rosszdb; i < N; i++)
39.         Console.WriteLine($"{allatok[i].Nev}, {allatok[i].Faj}");

```

A megoldás stratégiája és a kód között a részletek megfogalmazásának pontosságában van nagy különbség. Az ördög, amitől egy program rosszul működik, általában a részletekben bújkik meg. Ezért fontos lesz, hogy a rendezés stratégiájának kitalálása után a kódolásnál minden apróságra figyeljünk.

A megoldás 7–33. sorát okosabban, rövidebben is megírhatjuk.

```

7.     /******szétválogatás okosabban******/
8.     int eleje = 0;
9.     int vége = N - 1;
10.    int rosszdb; /*az első rész hossza, itt a kétlábúak száma*/
11.    while (eleje < vége)
12.    {

```

A `rosszdb` változóra nincs szükség, csak az előző megoldással való összehasonlítás miatt maradt a kódban. Az egyenlőségjel miatt a ciklus után biztosan kisebb lesz a vége, mint az eleje.

```

13.        /*négy lábút keres előlről*/
14.        while (eleje < N && Egyike(allatok[eleje].Faj, tojók))
15.            eleje++;
16.        /*kétlábút keres a végétől visszafelé*/
17.        while (vége >= 0 && !Egyike(allatok[vége].Faj, tojók))
18.            vége--;

```

A két keresés egymástól független, akár a sorrendjük is felcserélhető. De ebben az esetben arra kell figyelni, hogy az adatsorozat határain túl ne keressünk semmit.

Ha csak egyik típusú állat van, akkor a határ állítja meg az egyik keresést és nem lép sehova a másik. Ha a **vége -1** lett, akkor az **eleje 0** maradt, ha az **eleje N** lett akkor a **vége N - 1**. Mindkét esetben igaz, hogy **vége < eleje**. A feldolgozandó elemek az [**eleje**; **vége**] zárt intervallumban vannak. Ha ez létezik, azaz, ha **eleje < vége**, akkor a két helyen lévő adat jó helyre kerül, ezért mindkét irányból szűkíthetjük a vizsgálandó tartományt.

```

19.     if (eleje < vége) /*ha mindkét irányból talált átrakandót*/
20.     {
21.         /*felcserélés*/
22.         Allat temp = allatok[eleje];
23.         allatok[eleje] = allatok[vége];
24.         allatok[vége] = temp;
25.         /*tovább lép, mert az eleje és a vége helyen már kész*/
26.         eleje++; vége--;
    }

```

Általános esetben a külső ciklus végére az **eleje** az első nem kétlábúra mutat, míg a **vége** az első nem négylábúra. Akkor lesz az **eleje == vége**, ha van egy „harmadik érték”. Például, ha az **Egyike()** helyett a **< határ**, a **!Egyike()** helyett **> határ** a keresés feltétele, akkor az az érték, ahol az **eleje** és a **vége** találkozik, egyenlő a **határ**-ral.

Most ilyen határértékünk nincs, ezért az **eleje = vége + 1** értéken lesz vége a ciklusnak. Az **eleje** az első négylábú helyére mutat, ami éppen – a 0-tól indexelt – kétlábúak száma lesz.

```

27.     }
28.     rosszdb = eleje;

```

Egyszerű cserés rendezés

Sokféle rendezési típusalgoritmus létezik, először azzal ismerkedünk, amelyiknek a magyar nevében is benne van, hogy egyszerű. Lehet, hogy a világ más táján másképp gondolják, mert a „simple sort algorithm” kifejezés keresésére több más algoritmust találunk, de ezt nem. Ez a rendezés általában nem hatékony, de számunkra ez most érdektelen, mert a mi adataink elemszáma elég kicsi.

Az egyszerű cserés rendezés lényege, hogy az adatsor első elemét összehasonlítjuk az összes mögötte lévővel, és amelyik a későbbiek közül kisebb, azzal kicseréljük. Így a nagyobb elem az adatsor vége felé mozdul, a kisebb pedig az eleje felé. Ugyanezt a műveletet megismételjük az adatsor második, harmadik és az összes többi elemével, így a végén rendezett adatsort kapunk.

Futtassuk az alábbi programot, és nézzük meg a lista alakulását!

```

8.     static void CserésRendezés_Lépésenként()
9.     {
10.        int[] A = new int[5] { 5, 3, 9, 1, 7 };
11.        Console.WriteLine("Egyszerű cserés rendezés");
12.        for (int itt = 0; itt < A.Length - 1; itt++)
13.        {
14.            for (int utána = itt + 1; utána < A.Length; utána++)
15.            {
16.                Console.Write($"{String.Join(" ", A)},
                                itt: {itt + 1}, utána: {utána + 1}.");

```

```

17.     if (A[itt] > A[utána])
18.     {
19.         Console.WriteLine(" => csere");
20.         int temp = A[itt];
21.         A[itt] = A[utána];
22.         A[utána] = temp;
23.     }
24.     else
25.     {
26.         Console.WriteLine(" => nem cserélünk");
27.     }
28. }
29. }
30. Console.WriteLine($"Vége: {String.Join(" ", A)}");
31. }
32.

```

A belső ciklus mindig a soron következő indexű helyre keresi meg az oda való elemet. Amikor először fut végig, a 0. indexű helyre kerül a legkisebb elem. Amikor másodszor fut végig, az 1. indexű helyre kerül a második legkisebb elem. És így tovább.

A külső ciklus az adatsorozat vége előtt egygel megáll, mert nincs értelme az utolsó elemet az őt követővel (ami nem létezik) összehasonlítani. A belső ciklus végig megy, de mindig a külső ciklus aktuális indexe után egygel indul. Ha a relációs jelet fordítva tesszük ki, akkor csökkenő lesz a rendezés.

Az algoritmus jobban áttekinthető, ha a kiírásokat elhagyjuk és a cserére külön eljárást készítünk:

```

8.  static void CserésRendAlgoritmus(int[] A)
9.  {
10.     for (int itt = 0; itt < A.Length - 1; itt++)
11.         for (int utána = itt + 1; utána < A.Length; utána++)
12.             if (A[itt] > A[utána])
13.                 Csere(A, itt, utána);
14. }
15.

```

Szélsőérték-kiválasztásos rendezések

A következő rendezésnek magyar neve sincs, hatékonysága rosszabb, mint a cserés rendezésé, de sokan így csinálnák és kis módosítással sokat lehet rajta javítani.

A rendezéshez az első gondolat, hogy mindig kiválasztjuk a legkisebbet – ennek ismerjük a típusalgoritmusát – és ezt ki is vesszük, áttesszük sorba egy másik tömbbe, amit a végén visszaszámolunk az eredetire. Problémát jelent a „kivesszük”. Ha kivesszük, mi marad a helyén? Lyuk? Ha nem teszünk semmit a helyére, akkor marad az értéke annyi, amennyi volt, és mindig ez lesz a legkisebb. Ötlet: ne a legkisebbet keressük, hanem a már megtaláltnál nagyobbak közül a legkisebbet. Ezzel a kisebbik probléma az, hogy először tényleg a legkisebb kell, csak utána lehet a feltételes minimumot kiválasztani, és még az is leküzdhető, hogy az algoritmus „kissé” bonyolultabb lesz az összetett feltétel miatt. Az ötlet azon bukik el, hogy ha az adatsorozatunkban több azonos érték van, akkor ezekből csak egy kerül be az eredménybe. Ha pedig az utoljára találnál nagyobb vagy egyenlőket keressük, akkor végtelenségig ugyanazt az egy elemet találja meg. Ha tovább ragaszkodunk az ötletünkhöz, akkor minden megtalált új minimális érték után ki kell gyűjteni a vele azonosakat is. Tehát a rendezési stratégia az lenne, hogy

kikeressük a legkisebb értéket és kigyűjtjük az ilyen értékűeket egy másik tömbbe; ezután amíg a kigyűjtött elemszám kisebb a sorozat elemszámánál, keressük a legutoljára kigyűjtött értékünél nagyobb minimális értéket és az ilyen értékűeket kigyűjtjük – kiváló (jó nehéz) feladat a már tanult típusalgoritmusok gyakorlására, de talán nem így kellene megoldani a feladatot.

Megoldást jelenthet, ha a kivett (átmásolt) minimum helyére egy olyan nagy értéket írunk, aminél nagyobb nem lehet a sorozatban. A stratégia: ahány adata a sorozatnak van, annyiszor megkeressük a minimumot, a talált értéke egy másik sorozatba átmásoljuk és ezt az értéket nagyobbra (esetleg egyenlőre) állítjuk a sorozat legnagyobb értékénél. Ez sokkal egyszerűbbnek hangzik, próbáljuk meg kódolni.

```
8. static void CsakMostRend()  
9. {  
10. int[] A = new int[5] { 5, 3, 9, 1, 7 };  
11. Console.WriteLine("Másik adatsorozatba átrakással:");  
12. int[] temp = new int[A.Length];  
13. for (int db = 0; db < A.Length; db++)  
14. {  
15.     int mini = 0;  
16.     for (int i = 1; i < A.Length; i++)  
17.         if (A[i] < A[mini])  
18.             mini = i;  
19.     temp[db] = A[mini];  
20.     A[mini] = int.MaxValue;  
21. }  
22. /*az eredmény visszamásolása!*/  
23. for (int i = 0; i < A.Length; i++)  
24.     A[i] = temp[i];  
25. Console.WriteLine($"Vége: {String.Join(" ", A)}");  
26. }  
27.
```

Az eljárás neve azért CsakMost... mert ezt az életben csak egyszer, tanuláskor szabad kipróbálni. Elkeserítő, hogy az algoritmus lényege a szemétttermelés. Szintén névtelen, de jó javítása ennek az algoritmusnak, ha az üres helyre az utolsó elemet másoljuk át. Persze így minden lépésben eggyel rövidebb lesz a sorozatunk és erre illik figyelni.

```
8. static void LyukTömőRend()  
9. {  
10. int[] A = new int[5] { 5, 3, 9, 1, 7 };  
11. Console.WriteLine("Legkisebbet ki, lyukat a végéről betömni.");  
12. int[] temp = new int[A.Length];  
13. for (int db = 0; db < A.Length; db++)  
14. {  
15.     int utolsó = A.Length - 1 - db; /*hasznos elnevezni*/  
16.     int mini = 0;  
17.     for (int i = 1; i <= utolsó; i++) /*<=-re figyelni*/  
18.         if (A[i] < A[mini])  
19.             mini = i;  
20.     temp[db] = A[mini];  
21.     A[mini] = A[utolsó]; /*utolsóval felülírni mini-t */  
22. }
```

Így már nem vizsgál meg feleslegesen elemeket az algoritmusunk, de sok a másolás.

```

23.  /*az eredmény visszamásolása!*/
24.  for (int i = 0; i < A.Length; i++)
25.      A[i] = temp[i];
26.  Console.WriteLine($"Vége: {String.Join(" ", A)}");
27.  }
28.

```

Az egyszerű cserés rendezés javítható úgy, hogy sok csere helyett, kiválasztjuk a minimumot. A LyukTömő rendezésünk pedig úgy is megírható, hogy nem a végéről, hanem az elejéről tömjük be a lyukat, így az elején szabadulnak fel a sorozat helyei, ahova éppen betehető a kiválasztott minimum. Így nem kell másolni sem. E két javítás ugyanarra az algoritmusra vezet, a szélsőérték-kiválasztásos rendezésre. Magyarul a növekvő rendezést szokták minimumkiválasztásos rendezésnek hívni, a csökkenőt pedig maximumkiválasztásosnak, de a nemzetközi szakirodalomban ennek a neve: **Selection sort**. Futtassuk az alábbi programot, nézzük meg, hogyan alakul a rendezés!

```

8.  static void MinKivRendezés_Lépésenként()
9.  {
10.     int[] A = new int[5] { 5, 3, 9, 1, 7 };
11.     Console.WriteLine("Minimumkiválasztásos rendezés");
12.     for (int itt = 0; itt < A.Length - 1; itt++)
13.     {
14.         int mini = itt; /*innenről min.-kiválasztás*/
15.         for (int utána = itt + 1; utána < A.Length; utána++)
16.             if (A[mini] > A[utána])
17.                 mini = utána;
18.         Console.Write($"{String.Join(" ", A)},
19.                               itt: {itt + 1}., mini: {mini + 1}.");
20.         if (mini != itt)
21.         { /*csere a belső ciklus után, csak akkor, ha kell*/
22.             Console.WriteLine(" => csere");
23.             int temp = A[itt];
24.             A[itt] = A[mini];
25.             A[mini] = temp;
26.         }
27.         else
28.         {
29.             Console.WriteLine(" => nem cserélünk");
30.         }
31.         Console.WriteLine($"Vége: {String.Join(" ", A)}");
32.     }
33.

```

Ugyanez a rendezés a kiírásoktól megszabadítva:

```

8.  static void MinKivRendezés(int[] A)
9.  {
10.     for (int itt = 0; itt < A.Length - 1; itt++)
11.     {
12.         int mini = itt;
13.         for (int utána = itt + 1; utána < A.Length; utána++)
14.             if (A[mini] > A[utána])
15.                 mini = utána;

```

```

16.     if (mini != itt)
17.     {
18.         int temp = A[itt];
19.         A[itt] = A[mini];
20.         A[mini] = temp;
21.     } /*(mini != itt) esetén csere vége*/
22. } /*külső ciklus vége*/
23. }
24.

```

Ha elkészítjük a minimumkiválasztás függvényét, ami a tömb egy adott eleme utániak közül választ, és a cserére is függvényt írunk, akkor még átláthatóbb lesz az algoritmusunk.

```

16. static int Min(int[] A, int ezután) /*8-14. sorban Csere()*/
17. {
18.     int mini = ezután;
19.     for (int utána = ezután + 1; utána < A.Length; utána++)
20.         if (A[mini] > A[utána])
21.             mini = utána;
22.     return mini;
23. }
24.
25. static void MinKivRend(int[] A)
26. {
27.     for (int itt = 0; itt < A.Length - 1; itt++)
28.     {
29.         int mini = Min(A, itt);
30.         if (mini != itt)
31.             Csere(A, itt, mini);
32.     }
33. }
34.

```

Megfigyelhető, hogy ez a rendezési algoritmus az egyszerű cserés rendezésunktől örökölte a ciklushatárokat, a másiktól a minimumkiválasztást.

Buborék rendezés

Nemzetközileg ismert, sokak szerint legegyszerűbb rendezési mód a buborék rendezés, angolul **Bubble sort**. Az egyszerű cserés rendezéshez hasonlóan, itt is, ha hibás két elem között a sorrend, azt azonnal kicseréljük. Azonban itt nem egy adott helyre keressük a megfelelő elemet, hanem mindig egymás utáni elemeket hasonlítunk össze. Elsőre, amikor végigvesszük a tömb elemeit, az első elemet hasonlítjuk a másodikhoz, ha nagyobb nála, akkor kicseréljük őket. Így következő lépésben ugyanazt hasonlíthatjuk a harmadik elemhez, ha nagyobb, cserével visszük tovább, a sorozat vége felé. Ha a vizsgált két elem közül a második a nagyobb, akkor nem cseréljük ki őket, így a következő lépésben ezzel (a nagyobb) elemmel folytatjuk a vizsgálatot. A sorozat vége felé, egyre nagyobb elemet „vizünk”, így a legnagyobb elem kerül a sorozat végére. A következő menetben ugyanígy, a következő legnagyobb kerül hátulról a második helyre... a végén már csak az első és második elemet kell összehasonlítani és szükség esetén felcserélni. A stratégiánk kódolható:


```

8. static void BubiRendezés_Lépésenként()
9. {
10.     int[] A = new int[5] { 5, 3, 9, 1, 7 };
11.     Console.WriteLine("Buborék rendezés - nagy a végére");
12.     for (int teteje = A.Length; teteje >= 1; teteje--)
13.     {
14.         for (int bubi = 0; bubi < teteje - 1; bubi++)
15.         {
16.             Console.WriteLine($"Most: {String.Join(" ", A)}");
17.             if (A[bubi] > A[bubi + 1])
18.             {
19.                 Console.WriteLine($"=> {A[bubi]} tovább");
20.                 int temp = A[bubi];
21.                 A[bubi] = A[bubi + 1];
22.                 A[bubi + 1] = temp;
23.             }
24.             else
25.             {
26.                 Console.WriteLine($"=> {A[bubi + 1]} fog menni");
27.             }
28.         }
29.     }
30.     Console.WriteLine($"Vége: {String.Join(" ", A)}");
31. }
32.

```

Elhagyva a kiírásokat és a `Csere()` eljárást behelyettesítve, a cserés rendezéshez nagyon hasonló algoritmust kapunk. De ahogy ott is, ebben az esetben is a ciklusok szerevezésének módjától, a vezérlési struktúrák paraméterezésétől függ, hogy jó lesz-e a rendezésünk.

```

16. static void BubiRend(int[] A) /*8-14. sorban Csere()*/
17. {
18.     for (int teteje = A.Length; teteje >= 1; teteje--)
19.         for (int bubi = 0; bubi < teteje - 1; bubi++)
20.             if (A[bubi] > A[bubi + 1])
21.                 Csere(A, bubi, bubi + 1);
22. }
23.

```

A buborék rendezésnek több javítása ismert. Az egyik javítási lehetőség, hogy egy változóban megjegyezzük, hogy hol volt az utolsó csere, mert az algoritmusból következik, hogy onnantól kezdve már rendezett a sorozat és a többi elem mind kisebb ezeknél. Ebből következően a belső ciklus határa nem egyesével csökken, hanem az előző menet utolsó cseréjéig tart.

Egy másik hatékonyságnövelő lehetőség, hogy nem csak felfelé visszük a legnagyobbat, hanem elérve a végét, lefelé is visszük a legkisebbet. Ennek a rendezési algoritmusnak a neve koktélt rendezés, angolul **Cocktail sort**.

Beszűrő és Törpe rendezés

A beszűrő rendezés, angol nevén az **Insertion sort**, a kártyások rendezési módja. Egyik kezünkben a szétnyitott pakli, másik kezünkkel a nem megfelelő sorrendben lévő lapokat cserélgetjük, az előre tett lapnak a többi hátrafelé igazításával csinálunk helyet. Ezzel a módszerrel a rendezési stratégia: elsőként a második elemet hasonlítjuk az elsővel, ha rossz sorrendben vannak, akkor kicseréljük. a továbbiakban egymás után vesszük az adatsorozat elemeit és mindegyiket addig visszük előrele cserélgetéssel amíg az előtte lévő nagyobb nála. Miután

az első kettő növekvő, a harmadik vagy a helyén marad, vagy felcserélődik a másodikkal, vagy, ha mindkettőnél kisebb, akkor az elsővel is, így az első helyre kerül, az elsőből második, a másodikból harmadik lesz. A következő eleme a sorozatnak ehhez hasonlóan találja meg a már rendezett részben a helyét, miközben a csere során, amivel kicserélődik, az eggyel hátrébb kerül.

Ennek az eljárásnak van egy egyszerűsített verziója is, a rotálás, azaz forgatás. A csere lényegében két elem rotálása. Ugyanezt több elemre úgy lehet megvalósítani, hogy a végén lévő elemről készítünk egy másolatot (ez a cserénél a temp), majd hátulról előre felé minden elemet hátrébb teszünk: hátulról a másodikat a hátulsóba, hátulról a harmadikat hátulról a másodikba... Végül a kimásolt elemet betesszük az elejére (ez a cserénél a harmadik értékadás). Ezzel megspóroljuk azt, hogy az előre vitt elemet (kártyát) minden helyre betegyünk csak azért, hogy a következő lépésben onnan kivegyük.

Az alábbi beszűrő rendezés stratégiája tehát: a második elemtől az utolsóig minden elemet kiveszünk és az előtte lévőket – amíg a kiválasztott elemnél nagyobbak – eggyel hátrébb teszünk.

```
8. static void BeszűrőRendezés()
9. {
10.     int[] A = new int[5] { 5, 3, 9, 1, 7 };
11.     Console.WriteLine("Beszűrő rendezés");
12.     for (int i = 1; i < A.Length; i++)
13.     {
14.         int ez = A[i];           /*kitessük a vizsgáltat*/
15.         int ide = i;
16.         while (ide > 0 && A[ide - 1] > ez)
17.         {
18.             A[ide] = A[ide - 1]; /*hátrébb másoljuk a nagyobbakat*/
19.             ide--;
20.         }
21.         A[ide] = ez;           /*az így biztosított helyre betesszük*/
22.     }
23.     Console.WriteLine($"Vége: {String.Join(" ", A)}");
24. }
```

Az eddigi rendezési algoritmusokra jellemző volt, hogy két for-ciklusból, egy elágazásból és egy cseréből épültek fel. A beszűrő rendezésben a belső ciklus feltételes, while-ciklus. Egy kiválasztás, de nem minimumkiválasztás. A külső ciklusnak nincs más szerepe, minthogy megjegyezze, hol tart, meddig készült el a rendezéssel.

De miért kell megjegyezni, hogy hol tart, ha ez pont az a hely, ameddig biztosan jó a rendezés? Mi van, ha a következő ennél nagyobb? Az, hogy onnan nem kell előre mozgatni, tehát addig van kész. Ha pedig kisebb, akkor ez eleje felől nézve az első hibásan rendezett elempár, innen kell előre vinni az elemet. Szóval, minek azt tudni, hogy hol tartottunk? Nem kell tudni, csak addig menni, amíg jó. Ha rossz elemhez érünk, akkor azt addig kell előre felé vinni, amíg el nem érjük a jó helyét. Ezt a hülye is meg tudja jegyezni... Így ennek a rendezésnek kezdetben Stupid sort volt a neve, később kapta a **Gnome sort**, azaz törpe rendezés nevet.

```

8.     static void TörpeRendezés()
9.     {
10.    int[] A = new int[5] { 5, 3, 9, 1, 7 };
11.    Console.WriteLine("Törpe rendezés");
12.    int i = 0;
13.    while (i < A.Length - 1)
14.    {
15.        if (A[i] <= A[i + 1])
16.            i++;           /*ha jó a sorrend: index növelése*/
17.        else
18.        {
19.            int tmp = A[i]; /*i. és i+1. elem cseréje*/
20.            A[i] = A[i + 1];
21.            A[i + 1] = tmp;
22.            if( i > 0) i--; /*vissza, mert az új i. kisebb lehet az
                             előzőnél is, kivéve, ha már a legelején van.*/
23.        }
24.    }
25.    Console.WriteLine($"Vége: {String.Join(" ", A)}");
26.    }
27.

```

A megoldás különlegessége, hogy egyetlen while-ciklust használ. Az átnevezés egy kicsit utal arra is, hogy hol lehet jól használni: a szorgos automatáknál, robotoknál. Például két fal között a ciklusfeltétel lehet ütközés az egyik fallal, a második elágazás az ütközés a másik fallal, közben pedig csak méretet vesz, megy fel-alá és cserél a robot.

A Sort(), a Reverse(), az OrderBy() és az OrderByDescending()

A rendezési algoritmusok ismerete olyan, mint a főzés. Van, aki szeret kitalálni megoldásokat, másképp is megoldani problémákat. Másoknak elég az, ha egy megoldást tud, de azt bármikor meg tudja csinálni és van, aki a konzervet szereti, mert ahhoz semmi sem kell, azonnal használható. Most ilyen rendezési konzerveket próbálunk ki úgy, hogy valójában nem tudjuk, mi van benne.

A fejlett programozási nyelveken a rendezésre is van eljárás vagy függvény. C#-ban mindkettő. Ezek az eszközök nagyon hatékonyak, gyorsan tudnak rendezni nagy adathalmazokat, amit úgy érnek el, hogy titkos receptjükben több rendezési algoritmus van, amelyekből egy gyors mintavétel és elemzés után választják ki, hogy melyiket alkalmazzák. Egyszerű adatokból álló sorozatra a rendezés eléggé egyértelmű, legfeljebb a növekvő vagy csökkenő irányt kell megadni, de egy összetett adat – például a tanyán élő állataink – rendezése nem egyértelmű, nem tudjuk, melyiket vegyük figyelembe, a kort, a nevet vagy a fajtát, esetleg ezek közül többet... Ezért, ha előregyártott rendezési eljárást használunk, meg kell tanulni, hogyan adhatjuk meg a rendezési szempontokat. Az alábbi példákban erre láthatunk mintákat.

Először ezt a tömböt rendezzük:

```
int[] A = new int[5] { 10, 6, 18, 2, 14 };
```

A tömbök – angolul `Array` – növekvő rendezésére a `Sort()`, csökkenő rendezésére a `Reverse()` eljárás használható:

```
Array.Sort(A);
Console.WriteLine($"Növekvő: {String.Join(" ", A)}");
Array.Reverse(A);
Console.WriteLine($"Visszafelé: {String.Join(" ", A)}");
```

Ebben az egyszerű esetben a rendezés alapja az `A` adattípusára (`int`) értelmezett `<` relációs operátor. Ha a számokat szöveggént értelmezve szeretnénk rendezni ("`10`" `<` "`9`"), akkor a rendezés paramétereként meg kell adni az összehasonlítás szabályát, például egy Lambda-kifejezésbe írt feltétellel:

```
Array.Sort(A, (x, y) => x.ToString().CompareTo(y.ToString()));
Console.WriteLine($"Szövegesen: {String.Join(" ", A)}");
```

Szövegek között a `<`, `==` és `>` relációk helyett a `CompareTo()` függvényt használjuk, hiszen két szöveg nem két egyszerű adat, hanem két adatsorozat. A `CompareTo()` a „komparátor”, azaz összehasonlító függvény, aminek három értéke van, 1, 0 és -1. Az 1 olyan, mintha a függvény helyére `>` kerülne, jelentése: követi. A -1 jelenti a `<`, megelőzi relációt és 0 az `==`, a két szöveg azonos.

Ha az összehasonlítás szabályára nincs kész komparátor, akkor írhatunk. Ez főleg a többszem pontú rendezéseknél hasznos. A komparátor – a `CompareTo()` mintájára egy egész értéket visszaadó függvény, ami háromféle értéket adhat eredményül: 1 vagy 0 vagy -1. Ezt a függvényt külön megírhatjuk, vagy név nélkül, úgynevezett **delegate** függvényként a lambda kifejezésben adjuk meg.

Az alábbi rendezés első rendezési szempontja a számok 3-mal vett osztási maradéka, amennyiben ez megegyezik, akkor a szám értéke.

```
Array.Sort(A, (x, y) => /*ez itt egy lambda-kifejezés jobb oldala*/
    /*delegate függvény törzse, nincs neve*/
    if (x % 3 > y % 3) return 1;
    else if (x % 3 < y % 3) return -1;
    else if (x > y) return 1;
    else if (x < y) return -1;
    else return 0;
    } /*a delegate fgv és a lambda-kifejezés vége*/
);
Console.WriteLine($"tx mod 3, ezen belül érték szerint növekvő:
    {String.Join(" ", A)}");
```

Ugyanaz a komparátor a `Reverse()` eljárásban fordított rendezést jelent, de `Sort()` függvény-nyel is lehet visszafelé rendezni, ehhez a komparátorban a -1 és 1 értékeket fel kell cserélni.

A System.Linq névtér programunkhoz adásával lekérdezést (query) is írhatunk. A választó lekérdezés az adatokból előállít egy eredményhalmazt, az eredeti adatsorozatot nem módosítja. Az SQL nyelv Order By utasítására a LINQ-ban ennek megfelelő függvények vannak, amelyek a lekérdezés paramétereinek megfelelő eredményt hoznak létre a memóriában. Ha ezt egy változóban el akarjuk tárolni, akkor arra a típusra az eredményt át is kell alakítani. Egy adatbázis-lekérdezésben a rendezési szempontot a mező névvel és a rendezés irányával adjuk meg. LINQ nyelven az `OrderBy()` függvény növekvő, az `OrderByDescending()` csökkenően

rendezi a paraméterként lambda-kifejezésben megadott adatsorozatot (nem komparátor, hanem adat értékkel definiált „mező”).

```
Console.WriteLine("Tömb System.Linq és A.OrderBy eljárással:");
A = A.OrderBy(x => x % 5).ToArray();
Console.WriteLine($"\\tx mod 5 szerint: {String.Join(" ", A)}");
```

Az példában a tömb minden x elemének 5-tel vett osztási maradéka alapján rendezzük sorba a rekordokat – itt az egész számokat. Ahhoz, hogy ezt egy változóban el tudjuk tárolni, a rendezett rekordhalmazból megfelelő típusú tömböt készítünk (`ToArray()`). Az eredmény csak azért „rendezi át” az A tömböt, mert a végén felülírtuk az értékadással az eredeti adatokat.

Ugyanezeket a rendezéseket majdnem ugyanígy végezhetjük el `List<>` típusú adatokon, de a megvalósításon látszik, hogy a tömb egy egyszerű adatsorozat, amihez egy `Array` nevű osztályban írtak rendezési – és más – függvényeket, míg a `List<>` egy collection, egy gyűjtemény osztály, aminek tagfüggvényei vannak. A szakmai háttérből a gyakorlati tudnivaló, hogy az `Array.Sort(tömb, hogyan)` az első paramétere a tömb, a listáknak viszont a neve után írt ponttal választható ki a tagfüggvény: `lista.Sort(hogyan)`.

```
List<int> L = new List<int>() { 10, 6, 18, 2, 14 };
Console.WriteLine(".Sort eljárás és .OrderBy() függvény");
L.Sort();
Console.WriteLine($"\\tNövekvő: {String.Join(" ", L)}");
L.Reverse();
Console.WriteLine($"\\tVisszafelé: {String.Join(" ", L)}");
L = L.OrderBy(x => x).ToList();
Console.WriteLine($"\\tOrderBy: {String.Join(" ", L)}");
L.Sort((x, y) => y.CompareTo(x));
Console.WriteLine($"\\tSort-tal visszafelé: {String.Join(" ", L)}");
```

És a többi...

A bemutatott algoritmusok közös jellemzője, hogy az elemeket – általában helyben – cserélgeti. Ezeken kívül, sok más elv is lehetséges a rendezésre. Ilyen például a rekurzív algoritmus, amire később láthatunk példát, de érdekes megoldásokat ad az adatok láncolása, fában elhelyezése, a különböző részleges rendezések – például a kupac rendezés. A különböző algoritmusok bemutatása gyakran nehézkes elmesélve vagy leírva. A képi, animációs bemutatás sokkal hatékonyabb. A rendezési és sok más algoritmust is eltáncol a Maros Művészegyüttes a Sapiientia Erdélyi Magyar Tudományegyetem munkatársai rendezésében készült videókon, ami méltán világhírű (keresőszavak: sorting algorithms dance). Ezenkívül további érdekes animációkat is érdemes megnézni az interneten (például http://anim.ide.sk/rendezesi_algoritmusok_1.php) vagy akár meg is hallgatni (keresőszavak: sorting algorithms sound)

Rendezés a gyakorlatban

50. példa: A tanyán élő állataink neve és kora

Újabb példánkban tanyánk állatai közül a kutyák és a macskák szerepelnek, no meg a kakasok – rájuk mindig számíthatunk, ha nem akaródzik hajnalban kelni. Írjuk programot `rendezett_allatok` néven, amelyben állatainkról, illetve közülük a kedvenceinkről készítünk különböző szempontokkal listákat!

1. Gyűjtsük ki a kutyák, a macskák és a kakasok nevét!
2. Rendezzük a neveket névsorba és írjuk ki!
3. Gyűjtsük ki a kutyák, a macskák és a kakasok minden adatát!
4. Rendezzük kedvenc állatainkat kor szerint csökkenő sorrendbe!
5. Írjuk ki kedvenc állataink nevét és – nevük után zárójelben – a korukat!
6. Rendezzük kedvenceinket fajuk szerint, azon belül név szerint! Írjuk ki soronként minden adatukat!
7. Rendezzük a tanya összes állatát névsorba Sort() eljárással és írjuk ki a névsort!
8. Rendezzük át a tanya állatait fajonként, azon belül névsorba a Sort() eljárással! Írjuk ki az állatok összes adatát egymás alá!
9. Rendezzük a tanya állatait kor szerint csökkenő sorrendbe, ezen belül nevük hossza szerinti sorrendbe! Írjuk ki soronként az adatokat!

Kódoljuk a feladat megoldását rendezési algoritmus használatával és rendezési eljárás vagy függvény használatával is!

Előzetes: Rendezési algoritmusból elég egyet tudni, mindegy melyiket. A kedvenceink rendezését itt három különböző algoritmussal lehet összehasonlítani. A kigyűjtés eredményét listában érdemes tárolni, de ezt nem kötelező ismerni, ezért a nevek tömbbe, az állatok listában lesznek tárolva. Az utolsó feladatokban többször ki kell írni az állatok minden adatát, ezért erre már az elején külön függvény készül.

Az első feladatunk a fájl beolvasása és az adatok tárolása egy `Allat` struktúrát tartalmazó tömbben. Ezt már többször megoldottuk, csak le kell másolnunk valamelyik régebbi programunk elejét. Vagy, újraírjuk, ha sokkal gyorsabban elkészülhetünk vele, mint a kódjaink között keresgéléssel.

A feladatból és az előzetes döntésekből következően több névteret kell használni és a forrásfájlt is elérhetővé kell tenni.

```
1. using System;
2. using System.Linq;
3. using System.Collections.Generic;
4. using System.IO;
5. namespace Rendezett_allatok
6. {
7.     class Program
8.     {
```

Az `Allat` struktúrát a rendezés miatt feltétlenül érdemes létrehozni, nehogy összekeverjük az adatokat.

```
9. struct Allat
10. {
11.     public string Nev;
12.     public string Faj;
13.     public int Kor;
```

```

14. public Allat(string sor)
15. {
16.     string[] sors = sor.Split(' ');
17.     Nev = sors[0];
18.     Faj = sors[1];
19.     Kor = int.Parse(sors[2]);
20. }
21. }

```

Egy struktúrát nem lehet ciklussal kiírni, ha többször van szükség ugyanarra a megjelenési formára, akkor erre érdemes függvényt írni. Az alábbi megoldás egyben minta arra, hogy hogyan lehet adott szélességben balra, illetve középre igazítani a szöveget.

```

23. static void Ki(Allat a)
24. {
25.     Console.WriteLine(
26.         $"{a.Nev,-13}{a.Faj.PadLeft((14 + a.Faj.Length) / 2),-14}{a.Kor,2}");

```

A feladatok megoldását az adatok beolvasása is megelőzi. A változatosság kedvéért, a kód jelentős része egy eljárásban lesz, a tagolást a #region–#endregion jelenti. Ennek csak annyi a hatása, hogy a köztük lévő kódrészlet összecsukható, így átláthatóbb a kód.

```

28. static void Main()
29. {
30.     StreamReader sr = new StreamReader("allatok.txt");
31.     Allat[] mind = new Allat[15];
32.     for (int i = 0; i < 15; i++)
33.         mind[i] = new Allat(sr.ReadLine());
34.     sr.Close();

```

1. Gyűjtsük ki a kutya, a macskák és a kakasok nevét!

Ha tömbbe gyűjtjük ki a neveket, akkor a „Legrosszabb” esetre kell felkészítenünk a programunkat: lehet, hogy minden állat kedvenc is. Emellett egy változóban tárolni kell, hogy valóban hány kedvencünk van.

```

35. #region kedvenceink neveinek kigyűjtése tömbbe
36. string[] nevek = new string[mind.Length]; /*mindegyik kedvenc?*/
37. int N = 0; /*kedvencek száma*/
38. for (int i = 0; i < mind.Length; i++)
39. {
40.     if (mind[i].Faj == "kutya" || mind[i].Faj == "macska" ||
41.         mind[i].Faj == "kakas")
42.     {
43.         nevek[N] = mind[i].Nev;
44.         N++;
45.     }
46. }

```

2. Rendezzük a neveket névsorba és írjuk ki!

Ha az adataink tömbben vannak, akkor figyelniük kell arra is, hogy csak a valódi adatokat rendezzük. Egy teljesen feltöltött tömb esetén, ha túl lépünk az adatokon, akkor a tömb határán is túllépünk, ezt észreveszi a fordító program, IndexOutOfRangeException hibával leáll.

De ha – például – egész számokat tartalmazó tömbünket félig töltjük fel, akkor a hiányzó helyeken 0 érték lesz. Ez a fordító program számára nem okoz problémát, azonban az eredmény hibás lesz. A `string[]` hiányzó helyein null érték van. Rendezésekor az összehasonlításhoz használt `CompareTo()` függvényt nem lehet null értékre alkalmazni, de lehet paramétere a null. Így előfordulhat, hogy az `a.CompareTo(b) == 1` jó, de, ha b-nek nincs értéke, akkor a normális esetben ugyanezt jelentő `b.CompareTo(a) == -1` `NullReferenceException` hibával leállítja a futtatást.

```
47. #region nevek rendezése
48. Console.WriteLine("Kedvenceink ábécében:");
49. /*minimumkiválasztásos rendezés, de csak N-ig*/
50. for (int ez = 0; ez < N - 1; ez++)
51. {
52.     int mini = ez;
53.     for (int i = ez + 1; i < N; i++)
54.         if (nevek[i].CompareTo(nevek[mini]) == -1)
55.             mini = i;
56.     if (mini != ez)
57.     {
58.         string temp = nevek[ez];
59.         nevek[ez] = nevek[mini];
60.         nevek[mini] = temp;
61.     }
62. }
63. Console.WriteLine($"{string.Join(", ", nevek, 0, N)}");
64. #endregion
```

3. Gyűjtsük ki a kutya, a macskák és a kakasok minden adatát!

Most listába gyűjtjük az adatokat, mert így pont akkor lesz az adatsorunk, amennyi a figyelembe vett elemszám.

```
65. #region kedvenceink kigyűjtése listába
66. List<Allat> kedvencek = new List<Allat>();
67. for (int i = 0; i < mind.Length; i++)
68.     if (mind[i].Faj == "kutya" || mind[i].Faj == "macska" ||
69.         mind[i].Faj == "kakas")
70.         kedvencek.Add(mind[i]);
71. #endregion
```

4. Rendezzük kedvenc állatainkat kor szerint csökkenő sorrendbe!

Ez egy másik rendezési algoritmus lesz, de a csökkenő rendezés mindegyik algoritmusban ugyanúgy érvényesíthető: A sorrend helyességét vizsgáló kifejezésben (ez a komparátor) a reláció jelet meg kell fordítani. Bár a `<` ellentettje a `>=`, az egyenlőséget nem írjuk ki, mert nincs értelme két egyenlő elem felcserélésének. Lehet, de minek...

A rendezendő adatok struktúrák, ezért a vizsgálat során ki kell választanunk, hogy melyik adat-tag legyen az összehasonlítás alapja. Az `Allat` típusú változók az `=` jellel másolhatók, ezért könnyű a tömelemek cseréje. Sokkal bonyolultabb lenne, ha listákat kellene felcserélni.


```

71. #region Allat egyik tulajdonsága szerint csökkenő rendezés
72. Console.WriteLine("Kedvenceink kor szerint:");
73. /*buborék rendezés csökkenőre - kicsi megy a végére*/
74. for (int eddig = kedvencek.Count; eddig >= 1; eddig--)
75. {
76.     for (int i = 0; i < eddig - 1; i++)
77.     {
78.         if (kedvencek[i].Kor < kedvencek[i + 1].Kor)
79.         {
80.             Allat temp = kedvencek[i];
81.             kedvencek[i] = kedvencek[i + 1];
82.             kedvencek[i + 1] = temp;
83.         }
84.     }
85. }

```

5. Írjuk ki kedvenc állataink nevét és – nevük után zárójelben – a korukat!

```

86. for (int i = 0; i < kedvencek.Count; i++)
87.     Console.WriteLine($"{kedvencek[i].Nev} ({kedvencek[i].Kor})");
88. Console.WriteLine();
89. #endregion

```

6. Rendezzük kedvenceinket fajuk szerint, azon belül név szerint! Írjuk ki soronként minden adatukat!

Ilyen még nem volt... Több szempont szerint kell rendeznünk az adatokat. A cserélés feltételét jól át kell gondolni és helyes zárójelezéssel, összetett logikai kifejezésként adhatjuk meg:

```

90. #region két-kulcsos rendezés
91. Console.WriteLine("Kedvenceink faj, azon belül név szerint");
92. /*beszűrő rendezéssel*/
93.
94. for (int i = 1; i < kedvencek.Count; i++)
95. {
96.     Allat ez = kedvencek[i];
97.     int ide = i;
98.     while (ide > 0 && (kedvencek[ide - 1].Faj.CompareTo(ez.Faj) == 1 ||
99.         (kedvencek[ide - 1].Faj.CompareTo(ez.Faj) == 0 &&
100.             kedvencek[ide - 1].Nev.CompareTo(ez.Nev) == 1)))
101.     {
102.         kedvencek[ide] = kedvencek[ide - 1];
103.         ide--;
104.     }
105.     kedvencek[ide] = ez;
106. }
107. foreach (Allat k in kedvencek)
108.     Ki(k);
109. #endregion

```

Van, amikor két szempont sem elég. Az a feltételt még tovább bonyolítja, pedig már ez is eléggé átláthatatlan. A megoldás a rendezési relációra egy függvényt írni, ami akkor igaz, ha a két adatot fel kell cserélni. Az összehasonlítás kedvéért írjuk meg a rendezést eljárás-ként is:

```

94. Maskepp(kedvencek); /*a 106. sorig helyettesít*/

```

A 145. sorban kényelmesen elfér a ciklusfeltétel:

```
139. static void Maskepp(List<Allat> kedvencek)
140. {
141.     for (int i = 1; i < kedvencek.Count; i++)
142.     {
143.         Allat ez = kedvencek[i];
144.         int ide = i;
145.         while (ide > 0 && a7b(kedvencek[ide - 1], ez))
146.         {
147.             kedvencek[ide] = kedvencek[ide - 1];
148.             ide--;
149.         }
150.         kedvencek[ide] = ez;
151.     }
152. }
```

Az a7b – helyett bármilyen más neve is lehet a függvénynek, de a>b nem. A függvénybe írhatnánk ugyanazt, mint a 98–100. sorba, de ettől nem lesz átláthatóbb a szabály. Praktikus elágazásra átfogalmazni a feltételt: Ha az egyik ebben jobb, mint a másik, akkor nagyobb, ha egyenlők, akkor, ha a másik tulajdonságban jobb... Két tulajdonság között háromféle reláció lehet: kisebb, egyenlő és nagyobb. Ha egyenlők, akkor jön a következő tulajdonság, ami szintén háromféle lehet... Így minden lehetséges esetet fel tudunk írni „sormintászerűen”.

```
153. static bool a7b(Allat a, Allat b)
154. {
155.     bool nagyobb;
156.     if (a.Faj.CompareTo(b.Faj) == 1)           /*a.Faj > b.Faj*/
157.         nagyobb = true;
158.     else if (a.Faj.CompareTo(b.Faj) == 0)       /*u. az a faj*/
159.         if (a.Nev.CompareTo(b.Nev) == 1)       /*a.Nev > b.Nev*/
160.             nagyobb = true;
161.     else if (a.Nev.CompareTo(b.Nev) == 0)       /*a.Nev == b.Nev*/
162.         nagyobb = false;
163.     else                                         /*a.Nev < b.Nev*/
164.         nagyobb = false;
165.     else                                         /* a.Faj < b.Faj*/
166.         nagyobb = false;
167.     return nagyobb;
168. }
```

Ennél jóval rövidebb lenne a megoldás, ha a nagyobb kezdőértéke `false` lenne, mert ezzel az összes `false`-ra végződő ágat el lehetne hagyni. (161–166. sorok). Még tovább rövidíthető a kód, ha nem gyűjtjük logikai változóba az eredményt, hanem azonnal visszatérünk vele.

Haladó szintű feladat: Az alábbi kód jó-e és ha igen, ha nem, akkor miért:

```
153. static bool a7b(Allat a, Allat b)
154. {
155.     if (a.Faj.CompareTo(b.Faj) == 1)           /*a.Faj > b.Faj*/
156.         return true;
157.     if (a.Nev.CompareTo(b.Nev) == 1)           /*a.Nev > b.Nev*/
158.         return true;
159.     return false;
```

7. Rendezzük a tanya összes állatát névsorba Sort() eljárással és írjuk ki a névsort!

Ismét tömböt rendezünk (kivéve, ha eredetileg listába olvassuk be az adatokat), de most tudjuk, hogy a tömb minden eleme valid (érvényes, értelmes). Mivel a tömbelemek összetett adatok, meg kell adnunk, hogy mi legyen az összehasonlítás, amit Lambda-kifejezésben fogalmazzunk meg.

```
110. #region minden állat név Lambda
111. Console.WriteLine("Minden állat név szerint: ");
112. Array.Sort(mind, (x, y) => x.Nev.CompareTo(y.Nev));
113. for (int i = 0; i < mind.Length; i++)
114.     Console.Write(mind[i].Nev + ", ");
115. Console.WriteLine("\b.");
116. #endregion
```

8. Rendezzük át a tanya állatait fajonként, azon belül névsorba a Sort() eljárással! Írjuk ki az állatok összes adatát egymás alá!

Több szempontú rendezésnél a Lambda-kifejezés jobb oldala egy olyan függvény, ami 1, 0 vagy -1 értéket ad vissza. Ezt megírhatjuk a lambda kifejezésen belül:

```
117. #region minden állat két-kulcsos Sort, komparátor delegált
118. Console.WriteLine("\nMinden állat faj, azon belül név szerint:");
119. Array.Sort(mind, (x,y) =>
120.     {
121.         if (x.Faj.CompareTo(y.Faj) == 0)
122.             return x.Nev.CompareTo(y.Nev);
123.         else
124.             return x.Faj.CompareTo(y.Faj);
125.     });
126. for (int i = 0; i < mind.Length; i++)
127.     Ki(mind[i]);
128. #endregion
```

Ebben a formában névtelen a függvényünk, azaz delegate függvényt írunk. Ha többször kell ugyanazt az összehasonlító függvényt megadni, akkor érdemes nevet is adni neki

```
119. Array.Sort(mind, (x, y) => Hasonlit(x, y)); /*124. sorig helyettesít*/
```

A függvény csak a fejlécben tér el a delegate-től: van neki neve, és a paramétereit a név utáni zárójelekből veszi, nem a lambda-operátor előttiből:

```
169. static int Hasonlit(Allat x, Allat y)
170. {
171.     if (x.Faj.CompareTo(y.Faj) == 0)
172.         return x.Nev.CompareTo(y.Nev);
173.     else
174.         return x.Faj.CompareTo(y.Faj);
175. }
```

Ez a függvény is lehetne pont olyan, mint az `a7b()` a 153–168. sorokban, csak a két logikai érték helyett háromféle számot kellene kiosztani. Végigírva láthatóvá válik, hogy sokszor a feltételben megadott érték egyben a függvény értéke is, ezt az egyenlőséget vettük itt figyelembe.

9. Rendezzük a tanya állatait kor szerint csökkenő sorrendbe, ezen belül nevük hossza szerinti sorrendbe! Írjuk ki soronként az adatokat!

A változatosság kedvéért, ezt nem a `Sort()` vagy `Reverse()` függvénnyel, hanem az `OrderBy()` és társaival oldjuk meg. Itt adattagot (adatbáziskezelősen: mezőnevet) kell megadni. Ezt nagyon nehéz lenne összevonni, főleg úgy hogy az egyik csökkenő, a másik növekvő. A megoldás, az `OrderBy()` és `OrderByDescending()` után akárhányszor írható `ThenBy()` és `ThenByDescending()`.

```
131. #region mindre két-kulcsos, OrderByDescending, ThenBy
132. Console.WriteLine("\nMind kor csökk., név hossza növ:");
133. Allat[] rend = mind.OrderByDescending(x => x.Kor)
134.                      .ThenBy(x => x.Nev.Length).ToArray();
135. for (int i = 0; i < rend.Length; i++)
136.     Ki(rend[i]);
137. #endregion
```

Típusalgoritmusok rendezett sorozaton

Láthattuk, hogy a rendezés – főleg nagy adathalmazra – erőforrásigényes, mégis sokszor szükséges, mert egy rendezett adathalmazon a korábbi típusalgoritmusok helyett hatékonyabb, gyorsabb algoritmusokkal kaphatunk eredményt. Sok esetben a probléma úgy vetődik fel, hogy hosszabb távon mi éri meg jobban: rendezni az adatokat, majd utána gyorsabban megkaphatjuk a válaszokat vagy rendezés nélkül lassabban kapunk választ. A választ jelentősen befolyásolja, hogy az adathalmaz milyen gyakran változik. Egy adat módosulása, vagy újabb adattal bővülés vagy törlés a rendezett adathalmaz karbantartását, újra rendezését igényli. Ennek „költségét” – idő- és memória-igényét – kell összevetni a felhasználással kapcsolatos algoritmusok „költségeivel”.

Mennyiben változnak az egyes feladattípusokra a megoldások, ha rendezett az adatsorozat? Az összegzés és általában a megszámlálás, valamint a kigyűjtés típusalgoritmust nem lehet hatékonyabbá tenni. Ez utóbbi két feladattípusnál egyszerűsítésre ad lehetőséget, ha a feltétel a rendezési szemponthoz köthető, a számolandó vagy kigyűjtendő adatok egy intervallumot alkotnak. A szélsőérték kiválasztás algoritmusára viszont egyáltalán nincs szükség, hiszen a két szélsőérték az adatsorozat első, illetve utolsó eleme. Az eldöntés, a keresés és a kiválasztás lineáris megoldásánál sokkal hatékonyabb a – csak rendezett adatokon használható – bináris keresés. Az unió és a metszett képzés esetén pedig az összefuttatás jelenti a hatékonyabb megoldást.

Bináris keresés

A bináris – más néven logaritmikus vagy felező módszerrel – kereséshez hasonlót sokszor használunk a mindennapi, applikációmentes életben. Például, amikor egy nyomtatott szótárban keresünk egy szót vagy jelenléti íven keressük a nevünket; esetleg egy távolsági busz tarifa határai között keressük az utazásunk távolságát. A keresett értéket egy mintaadattal összehasonlítjuk és az eredménytől függően – amennyire lehet – egy nagy részt kizárunk a tartományból. Persze, ha a keresett szöveg 'Z'-vel kezdődik, akkor a tapasztalatunk alapján inkább a végén kezdjük a keresést, az 'M'-et inkább a közepén. Az algoritmusban nem számítunk ilyen előzetes elképzeléssel az adatok eloszlásáról, mindig középről vesszük a mintát.

A keresés stratégiája: Nézzük meg, hogy az adatsorozat felénél levő érték a keresett értékkel milyen viszonyban van. Ha egyenlő, akkor megtaláltuk, a felezőben lévő érték nagyobb a keresett értéknél, akkor vegyük a sorozat első felét, ha kisebb akkor a hátsó felét. Ezt folytatva, minden lépésben az előzőhöz képest fele annyi adat között keresünk, míg végül összeér az adatsorozatunk eleje és vége – ami vagy a helyes értéket adja, vagy megállapíthatjuk, hogy nem szerepel az adatsorozatban a keresett érték.

Ha az adatsorozat hosszát binárisan írjuk fel, akkor a megoldáshoz szükséges cikluslépések maximális száma (a feleződés miatt) a bitek számával adható meg. Másképp: ha az adatsorozat hosszát a kettő hatványaként írjuk fel, akkor a lépések számát a kitevő adja, azaz a hossz kettes alapú logaritmusának értékéből becsülhető a cikluslépések száma. 1024 rendezett adatból legfeljebb 11 lépésben eldönthető, hogy egy érték megtalálható-e benne. Lineáris keresésnél átlagosan 512 lépés kell ugyanehhez.

A bináris keresés hatékonysága sokkal jobb, mint a lineáris keresésé, ezért a táblázatkezelőkben a keresőfüggvények ezt a módszert is ismerik, sőt, sokszor alapértelmezettként is ezt a módszert használják (például FKERES(), VKERES() HOL.VAN(), KERES()), amikor „tartományban keresés”-t végeznek. Ezek a példák arra is felhívják a figyelmet, hogy a bináris keresés eredményét többféleképpen lehet értelmezni. Van, amikor csak a pontos érték előfordulását nevezzük találatnak, máskor kiválasztjuk a tartományt, amiben az érték szerepel. Ez a választás történhet a tartomány alsó, és felső korlátjával is. Ilyenkor elég kicsi az esélye annak, hogy nem találunk tartományt. Például alsó korlát esetén akkor, ha a legkisebb értéknél kisebb értéket keresünk.

51. példa: Merre van?

Adott egy számsorozat, növekvő értékekkel. Kérdés, hogy egy adott szám ezekhez képest merre van. Ha benne van a sorozatban, akkor hányadik, ha nincs benne, akkor melyik tartományban van. A szokásoktól eltérően, a tartomány mindkét végét írjuk ki.

A sokféle válasz miatt, az eredmény a konzolra írt szöveg lesz, ezért eljárást írunk. Ahhoz, hogy a `BinMerreVan()` függvényünk minden esetben jó választ adjon, a triviális (magától értetődő) eseteket külön meg kell vizsgálni: az első előtt, az utolsó után vagy ezekkel egyenlő értékekre nem is keressük a többi adat között a választ:

```

9.  static void BinMerreVan(int ez, int[] Lista)
10. {
11.     int eleje = 0;
12.     int vége = Lista.Length - 1;
13.     if (ez < Lista[eleje])
14.         Console.WriteLine("a legkisebb előtti, nincs a Listában.");
15.     else if (ez > Lista[vége])
16.         Console.WriteLine("a legnagyobb utáni, nincs a Listában");
17.     else if (ez == Lista[eleje])
18.         Console.WriteLine($"A {eleje}. helyen van {ez}.");
19.     else if (ez == Lista[vége])
20.         Console.WriteLine($"A {vége}. helyen van {ez}.");
21.     else
22.     {

```

Ezt követően jöhet a felező módszerrel történő keresés. Most óvatoskodó módszerrel nézzük, mert nem csak azt szeretnénk megtudni, hogy benne van-e vagy nincs benne a keresett szám a listában, hanem azt is, hogy melyik két listaelem között lenne a helye.

```

23.     int közepe;
24.     do
25.     {
26.         közepe = (eleje + vége) / 2;
27.         if (ez < Lista[közepe])
28.             vége = közepe; /*a Lista[vége] != ez*/
29.         else
30.             eleje = közepe; /*a Lista[eleje] != ez*/
31.     } while (Lista[közepe] != ez && vége - eleje > 1);
32.     if (Lista[közepe] == ez)
33.         Console.WriteLine($"A {közepe}. helyen van {ez}.");
34.     else
35.         Console.WriteLine($"{{Lista[eleje]}} és {{Lista[vége]}} között,
                                     nincs a Listában. ");
36. }
37. }

```

A vizsgálat előtt kizártuk, ezt követően, amikor az **eleje** vagy a **vége** értéke változik, tudjuk, hogy ott nem lehet a keresett érték, mert a **közepe** értékét csak akkor kapják meg, ha a **Lista[közepe]** nem egyenlő a keresett értékkel. Ez akkor is igaz, amikor már szomszédos értékekre mutat az **eleje** és a **vége**. Ez viszont éppen kijelöli azt az intervallumot, amelyikben a keresett érték van.

Az eljárás működését minden lehetséges esetre teszteljük, ezért a számsorozatunk néhány 1–9 közötti egész lesz, amire 0-tól 10-ig minden egészről megadjuk, hogy merre van. Például így:

```

/*bináris keresés számokon teszt*/
int[] Lista = new int[7] { 1, 3, 5, 6, 7, 8, 9 };
for (int i = 0; i < 11; i++) /*tesztadatok*/
{
    Console.WriteLine($"{i} keresése");
    BinMerreVan(i, Lista);
}

```

52. példa: Felénk járó kíváncsi postás – melyik állatok vannak

Az összefoglaló feladatsor 3. kérdése arról szólt, hogy a postásunk érdeklődve kérdezi, van-e egy bizonyos állatunk. Most is ez a helyzet, de az **állatok.txt** beolvasásakor fajták alapján sorba rendezve tároljuk el az adatokat, ezért bináris kereséssel is meg tudjuk adni a választ. Postásunk azonban az elmúlt időben vérszemet kapott, naponta jött új kérdésekkel, ezeket listába gyűjtöttük. Feladatunk az, hogy mindegyikről megmondjuk, hogy van-e az adott fajú állatból és ha van, akkor adjuk meg egy ilyen fajtájú állatnak a nevét és korát is.

A példában szereplő adatok nem csak adattípusban térnek el – **int** helyett **Allat** –, itt lehetnek ismétlődések is, másrészt nincs értelme a macska és a malac közé helyezni egy állatfajt, ha nincs, akkor mindegy, hogy az első előtt vagy az utolsó mögött nincs.

A feladat megoldása előtt ne felejtjük el, hogy be kell olvasni a fájlt és rendezni is kell. Ezt a két lépést érdemes egyben megoldani. A beolvasás lehet egy olyan függvény, aminek az eredménye az állatok – fajtánként rendezett – tömbje. A bináris keresésnek tudnia kell, hogy van-e az adott állat a tanyán, ami egy logikai érték (bináris eldöntés), ezt adja vissza a függvényünk. Ha az eredmény igaz, akkor egy állatot is meg kell adni. Ezt most a **TryParse()** függvényhez hasonlóan, egy **out**-tal elérhetővé tett változóba tesszük ki.

A keresés függvényét a végén ebben a programban teszteljük:

```
/*bináris keresés teszt: van-e postás által kérdezett állat*/
Allat[] allatok = RendezveBeolvas();
for (int i = 0; i < allatok.Length; i++)
    Console.Write(allatok[i].Faj + " ");
Console.WriteLine();
string[] kerdezett = new string[6]{ "atka", "birka", "egér", "kutya",
                                     "szarvasmarha", "zsizsik" };
for (int i = 0; i < kerdezett.Length; i++)
{
    Allat ez;
    if (BinKeres(kerdezett[i], allatok, out ez))
        Console.WriteLine($"A(z) {ez.Faj} neve {ez.Nev}, {ez.Kor} éves.");
    else
        Console.WriteLine($"Nincs {kerdezett[i]} a farmon.");
}
```

A hozzávalók közül, először az `Allat` struktúra megírása a már rutinfeladat.

```
9.  struct Allat
10. {
11.     public string Nev;
12.     public string Faj;
13.     public int Kor;
14.     public Allat(string[] adatok)
15.     {
16.         Nev = adatok[0];
17.         Faj = adatok[1];
18.         Kor = int.Parse(adatok[2]);
19.     }
20. }
```

Az adatok beolvasása valamilyen átmeneti tárolóba történik, hogy tudjuk, hány állatunk van, mekkora méretű tömbre van szükségünk. Ha listát készítünk vagy lehet bőségesen nagy a tömb mérete, akkor a beolvasás a rendezéssel egybevonható.

A beolvasás során történő rendezés tipikusan beszűrő rendezés, olyan, mint egy állandó karbantartási feladat: sorba vesszük az állatokat és betesszük a tömbünk megfelelő helyére. Eközben az új által megelőzött állatokat 1-gyel a tömb vége felé el kell mozdítani.

```
22. static Allat[] RendezveBeolvas()
23. { /*igazi beszűrő rendezés*/
24.     string[] sorok = File.ReadAllLines("allatok.txt");
25.     Allat[] a = new Allat[sorok.Length];
26.     for (int i = 0; i < sorok.Length; i++)
27.     {
28.         int hely = i;
29.         Allat ez = new Allat(sorok[i].Split(' '));
30.         while (hely > 0 && a[hely - 1].Faj.CompareTo(ez.Faj) == 1)
31.         {
32.             a[hely] = a[hely - 1];
33.             hely--;
34.         }
35.         a[hely] = ez;
36.     }
37.     return a;
38. }
```

Érdekesség: Az új állat helyének megkeresése lineáris kereséssel történik. Ezt le lehetne cserélni bináris keresésre, de semmivel sem jutnánk előbbre, mert az adatokat a sorrendjük megtartásával, hátulról kezdve, egyenként (lineárisan) kell hátrébb mozdítani.

A `BinKeres()` függvényünk megkapja a postás által kért fajta nevét, az állatok összes adatát tartalmazó tömböt és egy `Allat` típusú változót, amibe majd az eredményt tesszük.

Most a bináris keresés bátor módját használjuk: Ha a kiválasztott `kozep` helyen az érték nem egyenlő a keresett értékkel, akkor ezt a határok értékével átlépjük. Ekkor az új `elso` vagy új `utolso` helyen lévő állat lehet a keresett fajtából való, de ezzel nem igazán törődünk, csak szűkítjük a tartományt egész addig, amíg az `elso` és az `utolso` összeér. Ebben az állapotban a `kozep` két egyenlő érték számtani közepe, az itteni érték lehet a keresett érték. Ha nem ez a keresett érték, akkor az `elso` és `utolso` közül az egyik továbblép, az `utolso` kisebb lesz az `elso`-nél, tehát a keresett érték nem található az adatsorozatban.

```
39. static bool BinKeres(string fajta, Allat[] a, out Allat ez)
40. {
41.     int elso = 0;
42.     int utolso = a.Length;
43.     while (elso < utolso)
44.     {
45.         int kozep = (elso + utolso) / 2;
46.         if (a[kozep].Faj == fajta)           /*talált*/
47.         {
48.             ez = a[kozep];                  /*értékkadás*/
49.             return true;                    /*fgv vége*/
50.         }
51.         else if (fajta.CompareTo(a[kozep].Faj) < 0)
52.             utolso = kozep - 1; /*kisebb => nem középen, előtte*/
53.         else /*fajta > a[kozep]*/
54.             elso = kozep + 1; /*nagyobb => az eleje a közép után*/
55.     }
56.     ez = new Allat(); /*a +-1 miatt elso > utolsó => nincs benne*/
57.     return false;
58. }
```

Az 51. sorban – az előtte lévő `return` következtében – az `else` nem szükséges, de talán egy kicsit jobban olvasható így a kód.

Ezzel a megoldási módszerrel is meg tudjuk mondani, hogy hol lehetne a keresett elem, arra kell csak figyelni, hogy helyesen értelmezzük, ha az `utolso` értéke `-1` vagy az `elso` értéke a tömb hosszával egyenlő és figyeljünk arra, hogy az `utolso` jelenti az intervallum kezdetét, az `elso` pedig a végét. Ha ismétlődő értékek vannak a sorozatban, akkor a `kozep` környezetében újabb keresésekkel adhatjuk meg a részsorozat kezdetét és végét.

Összefuttatás

Az metszetet, az uniót – és halmazok különbségét is – sokkal gyorsabban, egyszerűbben állíthatjuk elő, két rendezett adatsorozatból. Míg rendezetlen adatsorozatok lényege az, hogy az egyik halmaz minden eleméhez viszonyítom a másik halmaz minden elemét, addig rendezett sorozatoknál elegendő a két adatsorozat egy-egy elemét figyelembe venni és ezek összehasonlítása alapján eldönteni, hogy melyik lesz a következő két elem. Így keresés nincs benne, a lépések száma legfeljebb a két adatsorozat elemszámának összege lesz.

A következő két példában a 48. példa gyerekeinek, Fricskának és Kökénynek az első szavait fogjuk ismét egyesíteni, de most a szavak nem a megtanulás sorrendjében, hanem kis szótárként, ábécérendben vannak feljegyezve:

```
/*fricska-kökény: összefuttatás: unió, metszet*/
List<string> fricska = new List<string>() { "anya", "apa", "baba",
    "erősáramú feszültség szabályzó", "kaja", "ló", "maci" };
List<string> kökény = new List<string>() { "anya", "apa", "autó",
    "könyv", "maci", "nagyi", "pörgettyűs tájoló", "víz" };
```

53. példa: Fricska és Kökény szótárainak közös része (metszete)

Készítsük el a két gyerek közös szótárát a szótáraik rendezett listáiból:

```
List<string> metszet = RendezetMetszet(fricska, kökény);
Console.WriteLine("fricska és kökény közös szavai: " +
    String.Join(", ", metszet));
```

Mivel mindkét lista növekvő rendezettségű, az első két szó összehasonlításával vagy kapunk egy közös szót, vagy, amelyik megelőzné a másikat, azt elhagyva, a következő szót vehetjük abból a listából. Másképp: amelyik lista épp vizsgált szava le van maradva az ábécében, annak a listának vesszük a következő elemét. Ha utoléri a másik listát – így egyezik a két lista aktuális szava –, akkor találtunk egy közös elemet, ezt kell a metszetbe betenni. Ha átugrik egy lehetséges egyezést, akkor a másik lista lesz lemaradva, azzal kell lépni. Mindez kódolva így néz ki:

```
9. static List<string> RendezetMetszet( List<string> fricska,
10.                                     List<string> kökény)
11. {
12.     List<string> kozos = new List<string>();
13.     int f = 0;
14.     int k = 0;
15.     while (f < fricska.Count && k < kökény.Count)
16.     {
17.         if (fricska[f].CompareTo(kökény[k]) < 0)
18.             f++;
19.         else if (fricska[f].CompareTo(kökény[k]) > 0)
20.             k++;
21.         else /*fricska[f] == kökény[k] */
22.         {
23.             kozos.Add(fricska[f]);
24.             f++; k++;
25.         }
26.     }
27.     return kozos;
}
```

54. példa: Fricska és Kökény szótárainak egyesítése (unió)

A szótárak egyesítése itt most azt jelenti, hogy ha valamelyikük egy szót ismer, akkor az bekerül az egyesített szótárba, ha mindketten ismernek egy szót, akkor azt csak egyszer vesszük figyelembe.

```
List<string> unio = RendezettUnio(fricska, kökény);
Console.WriteLine("Fricska és Kökény szótára: " +
    String.Join(", ", unio));
```

A két gyerek szótáraiban sincs ismétlődés és az egyesítettben sem lesz, de az algoritmus úgy is megvalósítható, hogy minden ismétlődés is bekerüljön. Ekkor kell két lista hosszának összege számú lépés a megoldáshoz.

A megoldás nagyon hasonlít a metszethez, de a „leamaradó” listaelemet nem eldobjuk, hanem betesszük a közös listába, ha pedig egyenlő elemeket találunk, akkor a két elemből csak az egyiket (mindegy, hogy melyiket) tesszük be a listába, de mindkét listában eggyel továbblépünk.

Amikor az egyik lista végére érünk, a másik lista minden elemét be kell tenni az egyesített listába. Ez – bár nem nehéz, – eléggé növeli a kódsorok számát, ezért érdemes gondoskodni arról, hogy mindkét lista ugyanazzal az elemmel fejeződjön be. Ez lehet a két lista utolsó elemei közül az ábécében hátrébb sorolt vagy egy biztosan nagyobb elem.

```
28. static List<string> RendezettUnio(List<string> fricska,
                                     List<string> kökény)
29. {
30.     fricska.Add("zzzzz");           /*végére egy nagy érték*/
31.     kökény.Add("zzzzz");           /*ugyanaz a nagy érték*/
32.     List<string> egyben = new List<string>();
33.     int f = 0;
34.     int k = 0;
35.     while (fricska[f] != "zzzzz" || kökény[k] != "zzzzz")
36.     {
37.         if (fricska[f].CompareTo(kökény[k]) < 0)
38.         {
39.             egyben.Add(fricska[f]);
40.             f++;
41.         }
42.         else if (fricska[f].CompareTo(kökény[k]) > 0)
43.         {
44.             egyben.Add(kökény[k]);
45.             k++;
46.         }
47.         else /*fricska[f] == kökény[k] */
48.         {
49.             egyben.Add(fricska[f]);
50.             f++; k++;
51.         }
52.     }
53.     fricska.Remove("zzzzz");
54.     kökény.Remove("zzzzz");
55.     return egyben;
56. }
```

Figyeljünk arra, hogy ha a listáinkhoz hozzáadunk egy elemet, akkor azt távolítsuk is el a végén, mert a lista paraméterként átadva is a listaelemekre hivatkozást ad, ezért a módosítás a függvény lefutása után is megmaradna.

Az összefuttatás nagy előnye az elemenkénti feldolgozás. Ez lehetővé teszi azt is, hogy két – nagyon sok adatsorból álló – fájlból folyamatosan beolvassa, mindig csak egy adatot tartva az operatív memóriában előállítsuk egy harmadik fájlban akár az uniót, akár a metszetet.

Ó, ió, Rekurzióóó!

Aki tanult Imagine Logot, vagy blokkprogramozást (pl. Scratch), az lehet, hogy már ismeri a rekurziót, a rekurzív eljárásokat. Logóban könnyen készíthetünk mutatós rajzokat, fraktálokat úgy, hogy egy eljárást saját magában hívunk meg. Hasonlóan, Scratchben – és blokkból építkező programozási környezetekben – mód van arra, hogy egy egyedileg készített blokkba behúzzuk önmagát. Bár az elnevezés eltérő (gyerekre optimalizált), de a lényeg ugyanaz. A blokk-programozási környezetben a blokkok az Imagine Logo eljárásainak felelnek meg, ezek a szövegalapú programozási környezetben eljárások, függvények.

Mostanra jelentős rutint szerezhettünk eljárások és függvények készítésében, de eszünkbe sem jutott, hogy eljáráson belül önmagát hívjuk meg vagy, hogy egymást kölcsönösen meghívó eljárásokat írjunk. Ebben a fejezetben – kiegészítve a tankönyvi jegyzetet – a rekurzióról lesz szó, ami, ha úgy nézzük, hogy felsőtagozaton már írtunk ilyen programot, akkor ismétlés, ha pedig a tankönyvhöz hasonlítjuk, akkor kiegészítő tananyag. Mondhatjuk, hogy nem kell tudni... Nem kell tudni, de van, akinek könnyebb a megoldás rekurzív módja, ezért érdemes megismerkedni vele.

A rekurzív eljárások írásához két alapvető dolgot kell szem előtt tartani:

- Akkor beszélünk rekurzióról, ha a programunk futása során egy eljáráson belül ugyanazt az eljárást – általában más paraméterezéssel – meghívjuk.
- Egy rekurzív hívás a programunkban végtelen folyamatot indíthat el, eközben önmagával újabb és újabb memóriát foglal le, ezért a rekurzív függvények és eljárások alapeleme egy programág, ami idővel biztosan bekövetkezik és nem tartalmaz rekurziót.

55. példa: Hello, itt vagyok

Futtassuk az alábbi eljárást, figyeljük meg hogyan kapcsolható a kód a kiíráshoz!

```

9.  static void Rekhivas(int v)
10. {
11.     Console.WriteLine($"Hello, {v} vagyok. ");
12.     if (v == 0)                                /*rekurzió megszakítása*/
13.     {
14.         Console.WriteLine("Már megyek is.");
15.         return;                                /*VÉGE! innen nincs tovább.*/
16.     }
17.     Console.WriteLine($"Hívom {v - 1}-t.");
18.     Rekhivas(v - 1);                            /*Önmagát hívja, más paraméterrel*/
19.     Console.WriteLine($"{v-1} kimúlt, végem van. Pá: {v}");
20. }
21. static void Main()
22. {
23.     int pl = 4;
24.     Rekhivas(pl);
25. }
```

A rekurzív eljárások jellemzője, hogy az eljárás neve szerepel az eljárás törzsében is (18. sor). Jellemzően előtte van egy feltétel, ami az egymást követő hívások sorában valamikor igaz lesz és megszakítja a hívás-sorozatot (12. sor). Ebben a feltételben az eljárás befejeződik. Több programozási nyelvben – a függvényekhez hasonlóan – az eljárások végét is **return** zárja, anynyi eltéréssel, hogy utána nincs megadva visszatérésre semmilyen adat. Ez a C# nyelvben csak akkor szükséges, ha az eljárás futását meg akarjuk szakítani (15. sor).

A 17. sorban nem szükséges az `else-ág`, mivel a program futása a 12. sorban szereplő feltétel esetén a 15. sorban befejeződik, így nem hajtja végre a 14–19. sor utasításait; azonban, ha a 12. sorban szereplő feltétel nem teljesül, akkor a program a 14. soron folytatódik, ezért a végrehajtás olyan, mintha a 14–19. sor az `else-ágban` lenne.

Mondatszerű leírásban (strukturált programozással) nem létezik megszakítás, ezért ott a „különben” ágot ki kell írni:

```
Eljárás Rekhivas(v: Egész)
    ki: v + "vagyok"
    ha v = 0
        ki: "Már megyek is"
    különben:
        ki: "hívom" + v-1 + "-et"
        Rekhivas(v - 1)
        ki: v-1 „kimult, végem van Pá.” + v
    elágazás vége
Eljárás vége
```

56. példa: Faktoriális számítás: ciklus vs. rekurzió

A rekurzív algoritmus egyfajta másképp gondolkodást igényel, de elméletileg az eddig tanult algoritmusok mindegyike megfogalmazható rekurzívan is. Az összegzés tétel egyik matematikai alkalmazása a faktoriálisszámítás ($n!$).

Írjunk függvényt az összegzés típusalgoritmus mintájára is és rekurzívan is egy szám faktoriálisának kiszámítására!

```
9. static void Main()
10. {
11.     int pl = 4;
12.     int f = ForFakt(pl);
13.     Console.WriteLine($"{pl}! = {f}");
14.     int r = RekFakt(pl);
15.     Console.WriteLine($"{pl}! = {r}");
16. }
```

Faktoriális számítás ciklussal:

```
17. static int ForFakt(int n)
18. { /*összegzés típusalgoritmus*/
19.     int fakt = 1;
20.     for (int i = 1; i <= n; i++)
21.         fakt *= i;
22.     return fakt;
23. }
```

Kiértékelés sorrendje: $((((1*1)*2)*3)*4$

Faktoriális számítás rekurzívan:

```
24. static int RekFakt(int n)
25. {
26.     if (n == 0)
27.         return 1;
28.
29.     return n * RekFakt(n - 1);
30. }
```

Kiértékelés sorrendje: $4*(3*(2*(1*(1))))$

Ha nem természetes számok sorozatával kell dolgoznunk, hanem tömb vagy lista – vagy más – adatsorozattal, akkor a rekurzív megoldásban a függvény paramétereiként kell megadnunk vagy (`static` jelzővel ellátott) osztályra nézve globális változóként. Másként a függvény minden futtatásával újra létrehozza az adatokat.

Feladatok

Oldjuk meg rekurzívan az alábbi, korábban már megoldott feladatokat:

1. 15. példa: Hónapok napjaiból az év hossza (Összegzés)
2. 16. példa: Van-e 28 napos hónap az évben? (Eldöntés)
3. 17. példa: Hányadik az első 30 napos hónap? (Kiválasztás)
4. 18. példa: Ha van 28 napos hónap az évben, akkor hányadik hónap ilyen? (Keresés)
5. 19. példa: Hány 30 napos hónap van az évben? (Megszámlálás)
6. 20. példa: Hányadik hónap a legrövidebb? (Szélsőérték-kiválasztás)

Megoldások

A minta megoldásban „ragaszkodunk” az eredeti feladat megoldásához, ugyanazt a környezetet, ugyanazt a `Main()` eljárást fogjuk használni, csak a függvények belső megvalósítását cseréljük le. Az akkori megoldások egyben adják a főprogramot:

```

6. static void Main()
7. {
8.     int[] hónapnapok = new int[12]{31,29,31,30,31,30,31,31,30,31,30,31};
9.     Console.WriteLine("Az év {0} napos", Sorozatszámítás(hónapnapok));
10.    if (Van28_0(hónapnapok))
11.        Console.WriteLine("Van 28 napos hónap.");
12.    else
13.        Console.WriteLine("Nem találtunk 28 napos hónapot.");
14.    Console.WriteLine("A {0}. hónap 30 napos.", Harminc(hónapnapok)+1);
15.    int melyik;
16.    if (Melyik28(hónapnapok, out melyik))
17.        Console.WriteLine("A(z) {0}. hónap 28 napos.", melyik);
18.    else
19.        Console.WriteLine("Nem találtunk 28 napos hónapot.");
20.    Console.WriteLine("{0} 30 napos hónap van.", Darab(hónapnapok, 30));
21.    Console.WriteLine("A legrövidebb hónap {0} napos.",
22.                        hónapnapok[Minhely(hónapnapok)]);

```

Mivel a rekurzió definíciójában csak a rekurzív hívások sorozatának a végét adjuk meg, azt, hogy milyen értékről indulunk, meg kell adni a függvénynek. Ez tipikusan az adatsorozat utolsó adatának indexe (az adatsorozat hosszánál eggyel kisebb érték). Emiatt meg kellene változtatni az összes korábbi felhasználási helyen a függvény paraméterezését, de ezt most egy programtervezői praktikával oldjuk meg: az eredeti függvények a rekurzív megoldásoknak a burkoló függvénye (wrapper function) lesz. Ez a burkoló függvény csak annyit tesz, hogy az új függvényünket a réginek megfelelő paraméterezéssel hívja meg.

```

24. static int Sorozatszámítás(int[] tomb)
25. {
26.     return RekurzivSzum(tomb, tomb.Length - 1);
27. }

```

1. Hónapok napjaiból az év hossza (Összegzés)

```
28. static int RekurzivSzum(int[] tomb, int i)
29. {
30.     if (i == 0)
31.         return tomb[0];
32.     return tomb[i] + RekurzivSzum(tomb, i - 1);
33. }
```

2. Van-e 28 napos hónap az évben? (Eldöntés)

```
34. static bool Van28_0(int[] tomb)
35. {
36.     return RekurzivVane(tomb, tomb.Length - 1);
37. }
38. static bool RekurzivVane(int[] tomb, int i)
39. {
40.     if (i == 0)
41.         return tomb[0] == 28;
42.     return (tomb[i] == 28 || RekurzivVane(tomb, i - 1));
43. }
44. }
```

3. Hányadik az első 30 napos hónap? (Kiválasztás)

```
45. static int Harminc(int[] tomb)
46. {
47.     return RekurzivMelyik(30, tomb, tomb.Length - 1);
48. }
49. static int RekurzivMelyik(int ertek, int[] tomb, int i)
50. {
51.     if (i == -1)
52.         return -1;
53.     return tomb[tomb.Length - 1 - i] == ertek ? (tomb.Length - 1 - i) :
54.         RekurzivMelyik(ertek, tomb, i-1);
55. }
```

Ha a `tomb.Length - 1 - i` helyett `i`-t írunk, akkor a tömbön belül az utolsó találatot adja a rekurzív függvény, mert az `i` csökkenő sorozatban veszi fel az értékeket és rekurzív hívás csak akkor történik, ha még nem talált megoldást.

4. Ha van 28 napos hónap az évben, akkor hányadik hónap ilyen? (Keresés)

A megoldáshoz használhatjuk ugyanazt a rekurzív algoritmust, csak a burkoló függvényben kell módosítani a használatot.

```
55. static bool Melyik28(int[] tomb, out int ez)
56. {
57.     ez = RekurzivMelyik(28, tomb, tomb.Length - 1) + 1;
58.     return ez > 0;
59. }
```

5. Hány 30 napos hónap van az évben? (Megszámlálás)

```
60. static int Darab(int[] tomb, int ertek)
61. {
62.     return RekurzivDarab(tomb, ertek, tomb.Length - 1);
63. }
```

```

64. static int RekurzivDarab(int[] tomb, int ertek, int i)
65. {
66.     if (i == 0)
67.         return tomb[0] == ertek ? 1 : 0;
68.     return (tomb[i] == ertek ? 1 : 0) + RekurzivDarab(tomb, ertek, i-1);
69. }

```

6. Hányadik hónap a legrövidebb? (Szélsőérték-kiválasztás)

```

71. static int Minhely(int[] tomb)
72. {
73.     return RekurzivMinhely(tomb, tomb.Length - 1);
74. }
75. static int RekurzivMinhely(int[] tomb, int i)
76. {
77.     if (i == 0)
78.         return 0;
79.     int eddig = RekurzivMinhely(tomb, i - 1);
80.     return (tomb[eddig] > tomb[i]) ? i : eddig;
81. }

```

Gyorsrendezés

Sokszor nehezebb, máskor természetesebb a rekurzív megoldás. Van, hogy egyszerű a rekurzív megoldás, de a sok függvényhívás miatt nem hatékony. Máskor ügyes használatával egy rövidebb, kifejezőbb kód mellé rövidebb futási idő is társulhat. Az egyik legismertebb – nevében is ígéretes – rendezési algoritmus rekurzív, érdemes megismerni vele.

Nemzetközi neve **Quick sort**. Ha ügyesen írják meg, akkor sokszor győztes a futtatási idők versenyében. Az algoritmus alapeleme a szétválogatás, amit – rekurzívan – ismét a két részre. Mondatszerű leírásban:

```

Eljárás Quicksort(T: tömb, eleje: Egész, vége: Egész)
    határ := Szétválogat(T, eleje, vége)
    ha határ - eleje > 1
        Quicksort(T, eleje, határ - 1)
    elágazás vége
    ha vége - határ > 1
        Quicksort(T, határ, vége)
    elágazás vége
Eljárás vége

```

Ebben a formában a rendezés „pofon egyszerű”, de azért a kivitelezés során rengeteg apróságra, határesetre kell figyelni, amelyek a szétválogatás során okozhatnak nehézségeket. Az egyik ilyen probléma lehet, hogy a szétválogatást alapvetően egy tulajdonság teljesülése vagy nem teljesülése szempontjából végeztük, de egy rendezendő sorozatban előfordulhat értékek ismétlődése. Sőt, az is lehet, hogy az egész (rész) sorozat egyetlen érték ismétlődése. Vajon hogyan rendez a fenti algoritmus egy ilyen sorozatot? A határ esetleg lehet az első, – ha $<$ és \geq a szétválogatás két ága – vagy lehet az utolsó elem. Ezt követően a két elágazás egyike nem teljesül, de a másik teljesül... és a rekurzív hívás paraméterei ugyanazok lesznek, mint a hívó eljárás paraméterei. Ebből következik, hogy „végtelen” soká futó programunk lesz. De valójában még ennél is rosszabb történik, mert minden hívás memóriát foglal le, ezért a programvégrehajtás tárolási területe – a stack, vagy magyarul verem – megtelik, így a program lefagy.

Ilyen „apróságok” miatt a `Quicksort()` megírása nagy odafigyelést igényel. Csak akkor lesz gyors, ha jól és hatékonyan írjuk meg a szétválasztást. Most két esetet fogunk megnézni, elsősorban a jó működés szempontjából, másodsorban a gyorsaságra figyelve.

```
9. static void Main()
10. {
11.     /*adatsorozat nem ismétlődő elemekkel*/
12.     int[] tomb = new int[5] { 5, 3, 9, 1, 7 };
13.     QuickSortH(tomb, 0, tomb.Length - 1);
14.     Console.WriteLine(string.Join(", ", tomb));
15.
16.     /*adatsorozat ismétlődő elemekkel*/
17.     QuickSort_LEG(0, lista.Count - 1);
18.     Console.WriteLine(string.Join(", ", lista));
19. }
20. static List<int> lista = new List<int> { 5, 3, 9, 1, 7, 2, 3, 4, 3 };
```

Az első adatsorozatunk tömb, a második lista, de ez – mivel nem módosítjuk az elemszámot – nem jelent eltérést a megoldásban. Az első esetben paraméterként adjuk át az adatsorozatot, a második esetben globális változóban tároljuk az adatokat. Ez sem lényeges, de a globális tárolásnak – majd látni fogjuk – oka van.

Jelentős különbség lesz a megoldásokban az értékek ismétlődésének előfordulása miatt. A kevésbé bonyolult megoldás választása érdekében az alkalmazott szétválogatás algoritmusok elvileg is különböznek.

Az első megoldásban a szétválogatást külön függvényként adjuk meg, a másodikban a rendezési algoritmuson belül van a szétválogatás kódja. Az első szebb, de a második esetben – az ismétlődéseknél – a hatékonyságot jelentősen növeli, ha nem egy, hanem két határ van, amit nehézkes egyetlen értéként visszaadni.

57. példa: Tömb gyors rendezése (nincs ismétlődő érték)

A rendező algoritmust a mondatszerű leírás alapján írjuk meg:

```
22. static void QuickSortH(int[] tomb, int eleje, int vége)
23. {
24.     int határ = Szetoszt(tomb, eleje, vége);
25.     if (határ - eleje > 1)
26.         QuickSortH(tomb, eleje, határ - 1);
27.     if (határ - vége > 0)
28.         QuickSortH(tomb, határ, vége);
29. }
```

A `Szetoszt()` függvény mellékhatása a tömb átrendezett új állapota, a visszatérési értéke az elejére válogatott adatok száma lesz. Ez egyben a hátra válogatott első adat indexe is. A függvény paraméterezésében a két index zárt intervallumot jelöl ki a tömbben, a vége az utolsó adat indexe. Így a részsorozat hossza: `vége - eleje + 1`.

A szétválogatáshoz ki kell választani egy értéket, ami szerint szétválogatunk. Ennek az értéknek nem kell feltétlenül a sorozatban szerepelnie, de a legkisebb és legnagyobb érték közé kell esnie ahhoz, hogy két típust (nála kisebb, nála nagyobb) szét lehessen választani. Az érték kiválasztásának az egyik módja, hogy véletlenszerűen választunk a lehetséges adatok közül.


```

30. static Random r = new Random();
31. static int Szetoszt(int[] tomb, int eleje, int vége)
32. {
33.     int e = eleje;
34.     int v = vége;
35.     int választ = r.Next(eleje, vége + 1);
36.     int Érték = tomb[választ]; /*egyedi, mert nincs ismétlődés*/

```

A helyben szétválogatás algoritmus szerint rossz helyen lévő adatokat keresünk előlről és hátulról haladva. A talált adatokat egymással felcserélve javítjuk a rendezettséget. A folyamat addig tart, amíg a két index össze nem ér. Valójában kétféle megoldás lehet erre: $e < v$ vagy $e \leq v$. Bármelyiket írjuk, a továbbiakban figyelni kell arra, hogy mi történik egyenlőség esetén.

A két – nem megfelelő oldalon lévő – elem keresésénél könnyű azt mondani, hogy a külső ciklusfeltételt ellenőrizzük itt is, de érdemi problémát a részsorozat határai okoznak. Mivel az e -től lefelé mindig csak azonos típusú adatok lesznek, a v -től felfelé pedig csak a másik típusúak, ha az egyik átlépi a másikat, akkor a keresés úgyis megáll.

A szétválogatás alapja a kiválasztott helyen lévő **Érték**, de ez háromféle állapotot jelent. A megoldásban el kell döntenie, hogy hova tartozzon az éppen kiválasztott elem. A szétválogatásnak van olyan algoritmus, amelyben ez az elem a határ egyik oldalán lesz, de az alábbi algoritmusban nincs ilyen megkülönböztetés. Egyik megoldásnál sem igaz, hogy az érték a részsorozat felénél lesz (akkor sem, ha eredetileg a részsorozat felénél lévő értéket választottuk ki).

```

37. while (e <= v)
38. {
39.     /*nagyobbat keres előlről*/
40.     while (e <= vége && tomb[e] <= Érték)
41.         e++;
42.     /*kisebb vagy egyenlőt keres hátulról*/
43.     while (v >= eleje && tomb[v] > Érték)
44.         v--;

```

Ha mindkét „térfele” találunk nem megfelelő adatot, akkor ezeket felcseréljük. Ilyenkor nem lehet a két adat indexe egyenlő, mert akkor önmagát cserélnénk.

```

45. if (e < v) /*mindkét irányból talált nem odavalót*/
46. { /*felcserélés*/
47.     int temp = tomb[e];
48.     tomb[e] = tomb[v];
49.     tomb[v] = temp;

```

A csere után az e és v helyen a „térfelek” megfelelő adat lesz. Ezzel az elágazást és a ciklusmagot be is fejezhetnénk, de a keresésnél ezeknek az adatoknak a vizsgálata felesleges, ezért mindkét oldalt lehet léptetni. Lehet, de ez a léptetés komoly kihatással van a ciklusfeltételek megfogalmazására is.

```

50.     /*jó helyen vannak, (Érték bárhol) következőre lépés*/
51.     e++; v--;
52. }
53. }
54. /*tomb[i]<=Érték: eleje..e-1; tomb[i]>=Érték: e..vége*/
55. return e;

```

Ha a keresések során az **e** és **v** szomszédosok voltak, akkor a két léptetés hatására helyet cserélnek, többször nem fut le a külső ciklus és **e** a második rész első adatát indexeli, a **v** az első rész utolsó adatára mutat.

Ha a keresések során az **e** és **v** között 2 a különbség, akkor a léptetés után egyenlők lesznek, a külső ciklus még egyszer lefut (mert az egyenlőt is beírtuk a 37. sorban). Az egyik belső ciklus egyszer tud lefutni és a következőre lépni, mert az érték a térfelének megfelel, a másik belső ciklus egyszer sem fut le, mert úgy érzi, hogy nem odavaló az éppen indexelt adat. Ezzel épp átlépi az egyik a másikat és ugyanúgy mutatják a részsorozatok határát, mint az előző esetben.

Ha az **e** és **v** között 2-nél nagyobb a távolság, akkor még lehetnek cserélendő esetek mindkét oldalon, vagy a következő ciklus lefutása után mindkét index eléri a túlsó térfelet, ezért már cserére sem kerül sor és a szétválogatás is befejeződik.

58. példa: Tömb gyors rendezése (ismétlődő értékek is vannak)

Az előző példa átírható úgy, hogy ismétlődő értékek se okozzanak gondot. Arra kell figyelni, hogy azonos értékek sorozatát nem lehet kettéosztani az egyik – minden másikkal egyenlő – érték alapján, ezért egy ilyen részsorozat hossza nem fog csökkenni; ezzel végtelen halmozódó futtatást eredményez.

Az előző megoldás algoritmusa már így is eléggé összetett, ezért nem így írjuk át, hanem a szétválogatás egy másik lehetséges megoldását alkalmazzuk. Ennek része lesz a kiválogatás, illetve további algoritmikus egyszerűsítés miatt a kiválasztott értéknél kisebb, ezzel egyenlő, illetve nagyobb értékek három listába válogatása.

Mivel a rendezés helyben történik, a kiválogatást követően az adatsorozat megfelelő részéből létrejövő eredményt vissza kell másolni az eredeti adatsorozatba. Így az adatok háromfelé másolását még egy másolás követi. Ha a rekurzív eljárásen belül hozzuk létre a három listát, akkor minden rekurzív híváskor új listákat gyártana a programunk, ráadásul a másolás során a fokozatos bővítés miatt egy `List<>` típusú adatsorozat bővítése jelentős időt igényelhet. Olyannyira, hogy a gyorsrendezésből lassú rendezés lenne.

Látni fogjuk, hogy a rendezendő adatsorozat típusa nem módosítja a megoldást, csak annyit várunk el tőle, hogy az adatsorozat értékei módosíthatók legyenek. A kigyűjtések helyét azonban célszerű fixen kijelölni, úgy, hogy ne a rekurzív hívás során jöjjön létre. Erre kézenfekvő megoldás a globálisan (és `static`) elérhető tömb. Másik megoldás, hogy ennek külső helyét is paraméterként adjuk meg. Ebben a megoldásban a rendezendő adatsorozat és az átmeneti tároló is globálisan elérhető lesz. Megadhatunk akár három tömböt is a háromfajta tárolására, de jelentősen egyszerűbb lesz a megoldás, ha egy tömbbe képezzük le a rendezésre váró adatok minél nagyobb részét. Ami ebből kimarad, azt most – a demo kedvéért – az eljárásen belül hozzuk létre.

Egy részsorozat szétválogatásából két kigyűjtést – az adott értéknél kisebbek és nagyobbak – a statikus tömbbe teszünk, még hozzá a részsorozat kezdetének és végének megfelelő indexek szerint. Az adott értékkel egyenlőket az eljárásen belül létrehozott (átmeneti) tömbbe gyűjtjük ki. Végül a három kigyűjtést visszamásoljuk a részsorozatra.

```

30. static int[] Qtemp = new int[lista.Count]; /*hely a kiválogatáshoz*/
31. static void QuickSort_LEG(int eleje, int vége)
32. {
33.     int választ = vége; /*lehetne random.Next(eleje, vége + 1) is*/
34.     int Érték = lista[választ];
35.     int e = eleje;
36.     int v = vége;
37.     int[] kozepe = new int[vége - eleje + 1];
38.     int DbK = 0;

```

Lehetséges, hogy a rendezendő részsorozat minden adata egyenlő, ezért a `kozepe` tömb mérete a részsorozat hossza lesz, külön változóban tároljuk, hogy ebben a tömbben aktuálisan hány érvényes adat van. Ebben a feladatban elegendő lenne a `DbK` számláló, hiszen minden azonos értékű adat teljesen azonos, de ha a rendezést összetett adatra, egy adat-tag szerint végezzük, akkor a teljes adatot el kell tárolni. Ha a `kozepe` vagy más tömb helyett `List<>` lenne, akkor a létrehozás után az 1., 2., 4... adat után a kapacitás növelése érdekében másolás is történne. Nagy, összetett adatok, szövegek esetén ez jelentősen lassíthatja a programot.

```

39. for (int i = eleje; i <= vége; i++)
40. {
41.     if (lista[i] < Érték)
42.     {
43.         Qtemp[e] = lista[i];
44.         e++;
45.     }

```

Az előző módszer egymásba ágyazott while-ciklusai helyett, itt egyszerű számlálós ciklust használhatunk. Ahogy haladunk a részsorozatban, a kiválasztott értéknél kisebb elemeket kigyűjtjük a az átmeneti tároló megfelelő részének elejére: az `e` változó kezdőértéke az `eleje`, és minden új adattal eggyel nő.

```

46.     else if (lista[i] > Érték)
47.     {
48.         Qtemp[v] = lista[i];
49.         v--;
50.     }

```

Az értéknél nagyobb adatokat a tárolón megfelelő részének végére másoljuk: `v` kezdőértéke a `vége` (ami az utolsó adat indexe), minden újabb adat tárolása után eggyel csökken, a következő szabad helyre mutat.

Mivel a szétválogatást egy létező adat értéke alapján végezzük, biztosan lesz ilyen adat is a részsorozatban, ezért a kétirányból történő feltöltés biztosan nem fog összeérni. Csak azt nem tudjuk, hogy mettől, meddig fog tartani. Ezért gyűjtjük új tömbbe.

```

51.     else
52.     {
53.         kozepe[DbK] = lista[i];
54.         DbK++;
55.     }
56. }

```

A tárolás módjából következik, hogy a rendezendő tömbbe visszamásolás az első és harmadik részre – bár nem a megszokott, de – egyszerű. A középső adatoknál szükséges a kigyűjtés indexének eltolását számolni.

```

57.   for (int i = eleje; i < e; i++)
58.       lista[i] = Qtemp[i];
59.   for (int i = 0; i < DbK; i++)
60.       lista[e + i] = kozepe[i];
61.   for (int i = v + 1; i <= vége; i++)
62.       lista[i] = Qtemp[i];

```

Ezzel a szétválogatás elkészült, az egyenlő adatok már a helyükre kerültek, azokat ki lehet hagyni. A két új rendezendő részsorozat belső határait a kiválogatott adatok helyének meghatározása alapján tudjuk. Mindkét rekurzív hívásnál figyelni kell arra, hogy a paraméterek zárt intervallumot jelölnek.

```

63.   if (e - eleje > 1)
64.       QuickSort_LEG(eleje, e - 1);
65.   if (vége - v > 1)
66.       QuickSort_LEG(v + 1, vége);
67. }

```

A gyorsrendezés az egyik legismertebb rekurzív rendező algoritmus. Egy másik, szintén közismert algoritmus a **Merge sort**, – ahogy a neve is mutatja – az összefuttatást használja rekurzívan: Összefuttatja a rendezett két fél-sorozatot miután mindkét félben összefuttatta ezeknek a feleit ... miután összefuttatta a két egyelemű „sorozatot”.

CSOORTNAPLÓ – TAPOGATÓZÁS AZ OBJEKTUMORIENTÁLT PROGRAMOZÁS IRÁNYÁBA

Ebben a leckében egyetlen program elkészítése a feladatunk. A feladatot az eddig tanult eszközeinkkel fogjuk megoldani, több alkalommal meg-megállva. Egy-egy ilyen megtorpanást arra használunk fel, hogy arról elmélkedjünk, milyen problémákkal szembesülünk, mennyire jó a megoldásunk, és hogy mi segíthetne jobbá tenni. Ha eltekintünk az elmélkedésektől, fel-foghatjuk egyszerű gyakorlásnak is.

59. példa: Csoportnapló

Adott egy fájl (naplo.txt) az alábbi vagy hozzá hasonló tartalommal:

```

Nagy Cikornya, 11zs, 5 4 5 3 2 4t 5 4 3 3t 4 3
Kiss Hokedli, 11zs, 2 4 3 5 4 4 4t 4 2
Klassz Piruett, 10zs, 5 5 5t 5 5 5t 5 5 5 5t 5 5 5 5
Papp Pizsama, 11x, 2 5 2 3t 3 4 4 3t 4 4t 2
Falus Bizsu, 11x, 5 5 5 5t 4 5 5 5t 4 4 5t 5 5
Ernyei Florida Paletta, 11x, 2 2 3 3t 2 2 4 4 4t 3 3t 4
Nagy Bukta, 11x, 1 1 1 2t 2t 2t
Majdnem Jeles, 11zs, 4 4 4 4 4 4 4 5t 5t 5t

```

Mint azt látjuk, a fájl lehetne akár egy vegyes (több osztály diákjait is magába foglaló) tanuló-csoport, például egy nyelvórai csoport tanárának feljegyzése a csoportba járó diákok jegyeiről. Az adatok meglehetősen „életszerűek”:

- a nevet és az osztályt vessző és szóköz választja el a következő adattól;
- a név több – nem egy és nem is mindig két – részből áll, szóközzel elválasztva;
- az osztályzatok szóközzel vannak egymástól elválasztva, de néhány jegy mögött egy ‘t’ jelzi, hogy témazáróból születtek.

Feladatok

A fájl beolvasását követően a következő feladatokat kell megoldanunk a csoportnaplos programban:

1. Írjuk ki a 11. zs osztályos diákok nevét!
2. Írjuk ki azoknak a nevét, akik nem írták meg mindhárom idei témazárót, és soroljuk fel a megírt témazáróiknak a jegyeit!
3. Határozzuk meg, ki írta a 11. x osztályban a legkevesebb témazárót!

Az utolsó előtti órán felelésre számítanak a diákok. A diákok szerint, amikor felelés van, az szokott felelni, akinek kevés a jegye. A tanárnő ennél pontosabb: azok közül válogat felelőt, akik a rendes jegyeinek száma kevesebb, mint a legtöbb rendes jeggyel bíró diák rendes jegyeinek 80 százaléka.

4. Írjuk ki azoknak a nevét, akik „veszélyeztetettek”!
5. A tanárnő mégsem feleltet, hanem lezárja a diákok év végi jegyeit. Határozzuk meg és írjuk ki a diákok átlagát és év végi jegyét!

Az átlagszámításkor a témazárók duplán számítanak. Az év végi jegy 1,7-es átlag fölött kettes, 2,5 fölött hármas, 3,5 fölött négyes, és 4,5-től ötös.

Gondoljuk végig, hogy az egyes kérdésekhez melyik típusalgoritmus szükséges!

1. kiválogatás
2. kiválogatások (hol van t betű a szám után) alapján megszámlálások (a témazárók száma); az eredmény alapján kiválogatás (nevek), végül újabb kiválogatások (a jegyek kiírása – elképzelhető, hogy az első kiválogatások eredményét használjuk majd)
3. minimumkiválasztás
4. megszámlálások, maximumkiválasztás, majd kiválogatás
5. sorozatszámítások

Mint ahogy már többször tudatosult bennünk, a jó adatszerkezet megkönnyíti a jó program írását. Határozzunk az adatok tárolásának módjáról!

Sok diák adatait kell tárolnunk. Egy-egy diáknak három tulajdonságát figyeljük meg:

- a nevét,
- az osztályát és
- a jegyeit.

A jegyek tűnnek a legmacerásabbnak, és elég sok művelet van velük kapcsolatosan. A forrásfájlban a témazárókra kapott jegyek egy listát alkotnak a rendes (nem témazáróra kapott) jegyekkel, alighanem azt a sorrendet tükrözve, ahogy a tanár feljegyezte őket. A kérdések között semmi nem vonatkozik a jegyek sorrendjére, az viszont, hogy témazáróra kapta-e a diák a jegyet, vagy sem, több esetben is fontos lesz. A jegyek elkülönítését az előzőek fényében érdemesnek tűnik már a fájl beolvasásakor megtenni. Érdemes-e már a beolvasáskor átalakítani a jegyeket számmá? Számolni fogunk velük, tehát érdemes.

Az osztályokkal kapcsolatos tevékenységeinket vizsgálva észrevesszük, hogy mindössze egyetlen feladatnál érdekes az évfolyam, azaz talán nem szükséges külön tárolnunk az évfolyamot és a betűjelet.

A neveket elég csak kiírni, és soha nem kell külön kezelnünk a vezetékes- vagy keresztnéveket. A nevek tárolhatók egyetlen szöveggént.

Az eddigiek alapján érdemes lesz egy **Diak** struktúrát létrehozni, egy-egy diák adatainak kezelhető tárolására. A **Diak** struktúra adatai a **Név** és az **Osztály**, valamint a jegyeket ketté válogatva a harmadik adatag lehet a **RendesJegyek**, ami a „rendes jegyeket” egész számként tartalmazó adatsorozat és negyedik adatag az előzőhöz hasonló **TZjegyek**.

Eddigi példáinkban még nem kellett egy struktúrán belül adatsorozatot tárolni, most pedig két adatsorozat is lesz a struktúrán belül. A **RendesJegyek** eléggé listaszerű, nem lehet tudni, hogy hány jegyet kap egy diák. A témazárójegyekről azonban tudjuk, hogy legfeljebb három lehet belőle, ezért – és a változatosság kedvéért – ehhez inkább egy háromelemű tömböt használunk. A témazárójegyek száma amúgy is kérdés lesz, de a fix méretű tárolóhely mellett szükséges is lesz a külön, ötödik adatagként a **TZdb** tárolására. (Továbbra is igaz, hogy nem kell tudni mindkét adatsorozat használatát, és itt is bármelyiket helyettesítheti a másik.)

A beolvasás folyamán egy-egy sorból konstruktorral hozzuk létre a **Diak** típusú adatokat és mint már oly sokszor, tömbben (vagy listában) tároljuk a csoport adatait.

Megoldás

A megoldást a névterek felvételével, fájl hozzáadásával érdemes kezdeni ezt követi a **Diak struktúra elkészítése**, körülbelül a 8. sortól kezdődően. A kapott adatokat tekintve, lehet, hogy a konstruktor megírása a teljes feladat legnehezebb része.

```
8. struct Diak
9. {
10.     public string Név;
11.     public string Osztály;
12.     public List<int> RendesJegyek;
13.     public int[] TZjegyek;
14.     public int TZdb;
```

Az adatsorozatokat – és az egyszerű adattípusokat is – először csak deklaráljuk, a létrehozás a konstruktorban történik.

```
15. public Diak(string sor)
16. {
17.     string[] adatok = sor.Split(", "); /*vessző és szóköz is!*/
18.     Név = adatok[0];
19.     Osztály = adatok[1];
20.     string[] jegyek = adatok[2].Split(" "); /*tovább bontjuk*/
21.     RendesJegyek = new List<int>(); /*itt hozzuk létre*/
22.     TZjegyek = new int[3]; /*a feladat szövege szerint 3 TZ van.*/
23.     TZdb = 0;
```

Az adatsorozatokat létrehozása után, a **jegyek**-et feldolgozzuk. A témazáró jegyek specialitása, hogy 2 karakteresek, míg a rendes jegyek 1 karakteresek. Azt is figyelhetjük, hogy a jegy utolsó karaktere 't'-e vagy szám.

```
24. for (int i = 0; i < jegyek.Length; i++)
25. {
26.     if (jegyek[i].Length == 1)
27.         RendesJegyek.Add(int.Parse(jegyek[i]));
28.     else
```

Témazáró jegy értékének megadására többféle módszer létezik. Lehet a `.Substring()` vagy a `.Remove()` függvénnyel "t"-tleníteni a jegyet, majd számmá konvertálni, illetve – tudva, hogy csak az első karakterből kell számot előállítani – ezt a karaktert `.ToString()` függvény szöveggé alakítja, ezt már az `int.Parse()` át tudja konvertálni számmá. Most egy harmadik módszerrel adjuk meg a számértéket: kihasználjuk, hogy a számjegyek ASCII kódjai egymást követik és a karakterek számként is értelmezhetők. A számjegy karakterből kivonva a '0' karaktert, éppen a szám értékét kapjuk meg.

```

29.     {
30.         TZjegyek[TZdb] = jegyek[i][0] - '0';
31.         TZdb++;
32.     }
33. }
34. }
35. }

```

Második lépés, a **0. feladat** megoldása: az adatok beolvasása és eltárolása a programban. Ezt most a `Main()` eljárásban írjuk meg. Az adatok tárolására használhatunk tömböt és listát is. A kicsit több kódolást igénylő tömbös megoldás lesz itt. A beolvasáshoz használhatjuk a `File.ReadAllLines()` függvényt, de itt a tanultak ismétlése szempontjából fontosabb `StreamReader`-t használjuk.

```

36. static void Main()
37. {
38.     const int MaxCsop = 35;
39.     Diak[] csoport = new Diak[MaxCsop];
40.     int N = 0;
41.     StreamReader sr = new StreamReader("naplo.txt");

```

Ha a csoportnak tömbje van, akkor előre ismerni kell a méretét. `StreamReader`-rel beolvasva ezt nem tudjuk előre megadni, ezért a feladat szövege alapján, egy osztály maximális méretét adjuk meg. Ez most 35 fő, de könnyen módosulhat, ezért konstans változóként adjuk meg. A tényleges létszámot az `N` változó fogja tárolni.

```

42. while (!sr.EndOfStream)
43. {
44.     csoport[N] = new Diak(sr.ReadLine());
45.     N++;
46. }
47. sr.Close();
48.

```

A többi feladatra eljárásokat és függvényeket fogunk írni, amelyeket a `Main()`-ben hívunk meg. Akár már munkánk elején elkészíthetjük a válaszok vázlatát.

```

49. Console.WriteLine("1. A 11.zs tanulói:");
50. KiirZs(csoport, N);
51. Console.WriteLine("2. Kevés témazárót írtak:");
52. KevésTZ(csoport, N);
53. Console.WriteLine("3. 11.x-ből legkevesebb témazárót {0} írta.",
                    MinTZ(csoport, N));

```

```

54. Console.WriteLine("4. Felelésre felkészülők:");
55. FelelésVeszély(csoport, N);
56. Console.WriteLine("5. Év végi átlagok és jegyek:");
57. Evvege(csoport, N);
58. }
59.

```

A `csoport` tömböt a `Main()` változójaként hoztuk létre, ezért minden függvénynek, eljárásnak paraméterként át kell adnunk. Mivel várhatóan nem lesz teljesen feltöltve adattal, ezért minden esetben meg kell adnunk a csoportlétszámot is és a feladatoknál ezt kell használni a ciklusokban.

Az adatok beolvasásáról elmondhatjuk, hogy összességében „sikerült” a leghosszabb megoldást megvalósítanunk. Ha lehet, ennél célszerűbb megoldási módokat érdemes összeválogatni.

Az **1. feladat**ot megoldó eljárásban az egyetlen nehézséget az jelenti, hogy nem tudjuk, mire kell figyelnünk a 11. zs értelmezése során. Például, kérdéses, hogy hány és milyen karakterek lehetnek az évfolyam és osztály között, hány számjegyű az évfolyam jelölése. A minta alapján akár a "11zs" keresése is megfelelő lehet, de érdemes egy kicsit körülnézni a `string` függvényei között. (Megjegyzés: a tankönyvben 11.zs-t kér a feladat, de a 11. évfolyamra szűr, aminek a minta minden sora megfelel.)

```

60. static void KiírZs(Diak[] csoport, int n)
61. {
62.     for (int i = 0; i < n; i++)
63.         if (csoport[i].Osztály.StartsWith("11") &&
64.             csoport[i].Osztály.EndsWith("zs"))
65.             Console.WriteLine('\t' + csoport[i].Név);
66. }

```

Rögtön az első feladat megoldása során felmerül a kérdés, hogy mivel sokféleképpen lehet írni azt, hogy egy diák a 11. zs osztályba jár (11.zs, 11zs, 11/zs...), akkor jobb lenne, ha külön tárolnánk az évfolyamot és a tagozatot. A különböző formában megadott adatokat a bevitel során lehet szabványosítani, vagy egyszer, a beolvasáskor értelmezni. Ezt követően a tárolt két értékből az igény szerinti formátum egyszerűen előállítható. A külön adatként tárolásnak további előnye, hogy a sok problémában csak az évfolyam számértéke szükséges, legyen szó akár a következő évfolyamra lépés bejegyzéséről vagy a végzés évének meghatározásáról vagy az érettségizés lehetőségéről. A tagozat külön adatként tárolása pedig a tanterv, a heti óraszámok meghatározásához lehet fontos ismeret. Ezért ígéretet teszünk arra, hogy a feladat továbbfejlesztése során az évfolyam egy egész érték lesz, és a tagozatot külön adatként fogjuk tárolni.

A **2. feladat** megoldásában újdonságnak tűnik a struktúra és adatsorozat többszörös összetétele, bár a 10.-es jegyzetben kiegészítő anyagként volt példa ilyenre. Ráadásul, ha a struktúránk `string` típusú adatot tartalmaz, annak a karaktereit ugyanilyen módon (szintaktikával) érjük el.


```

67. static void KevésTZ(Diak[] csoport, int n)
68. {
69.     for (int i = 0; i < n; i++)
70.         if (csoport[i].TZdb < 3)
71.         {
72.             Console.Write('\t' + csoport[i].Név + ": ");
73.             for (int j = 0; j < csoport[i].TZdb; j++)
74.                 Console.Write(csoport[i].TZjegyek[j] + ", ");
75.             Console.WriteLine("\b\b ");
76.         }
77.     }
78.

```

A 74. sorban a `csoport[i].TZjegyek[j]` jelentése: a `csoport` `i`-edik diájának `j`-edik témazáró jegye. Ha az adatstruktúra létrehozásakor a témazárókat is listában tároljuk, akkor a `TZdb`-re nem lenne szükség, helyette a lista hosszát tudnánk használni, valahogy így: `csoport[i].TZjegyek.Count`.

A **3. feladat** megoldása során ismét felmerül néhány kérdés. Előzetesen: újra probléma az osztály jelölése, elegendő-e a "11x" osztályra szűrni. Ha már az első feladatban adtunk erre a problémára egy megoldást, akkor most is ugyanazt használjuk, de biz'isten legközelebb eleve jobban fogjuk csinálni. További kérdés, hogy mennyire lehetünk biztosak abban, hogy a csoportba jár 11-es diák. Legyünk óvatosak, olyan megoldást írjunk, amelyik nem fagy le akkor sem, ha nem létezik figyelembe vehető diák. A „nem létező diákunk” a csoport `-1`-edik tagja lesz, a maximálisan lehetséges 3 dolgozatból 4-et „írt meg” és a neve – amire a feladat rákérdez – „nincs 11x-es diák” lesz. Ha van olyan diák, akit figyelembe tudunk venni, annak biztosan minden vizsgált tulajdonsága ennél jobb lesz.

```

79. static string MinTZ(Diak[] csoport, int n)
80. {
81.     int mini = -1;
82.     int mindb = 4; /*legfeljebb 3 lehet*/
83.     for (int i = 0; i < n; i++)
84.     {
85.         if (csoport[i].Osztály.StartsWith("11") &&
86.             csoport[i].Osztály.EndsWith("x"))
87.         {
88.             mindb = csoport[i].TZdb;
89.             mini = i;
90.         }
91.     }
92.     return mini == -1 ? "nincs 11x-es diák" : csoport[mini].Név;
93. }

```

A 85. és 86. sor egyetlen feltételként is megfogalmazható, újabb 'és' kapcsolattal. A külön feltételbe írás a kód későbbi olvasásakor jut szerephez: könnyebben értelmezhető, hogy mi a szűrőfeltétel – a lehetséges adatok kigyűjtésének szempontja – és melyik szempont szerint keresünk minimumot.

A függvény visszatérési értékét egy háromoperandusú értékadás adja. Ez egyenértékűen megadható egy elágazással is, ami akár 4–9 plusz sorban is megírható.

A **4. feladat** megoldását a figyelmes olvasással kezdjük és kiszűrjük a megoldandó probléma szempontjából lényegtelen részleteket. Ezt követően is lesz még elég dolgunk, mert először meg kell határozni, hogy mennyi a rendes jegyek számának a maximuma, majd ezt követően tudjuk kigyűjteni azokat, akik felelőként szóba jöhetnek. A két részfeladatot egymás után kell megoldani (nem ismétlődik egyik sem), ezért egyetlen eljárásban írjuk meg a megoldást, de a kód későbbi olvashatósága érdekében régiókra osztással jelezzük az egyes részeket.

```

95. static void FelelésVeszély(Diak[] csoport, int n)
96. {
97.     #region legtöbb rendes jegy
98.     int max = csoport[0].RendesJegyek.Count;
99.     for (int i = 1; i < n; i++)
100.         if (csoport[i].RendesJegyek.Count > max)
101.             max = csoport[i].RendesJegyek.Count;
102.     #endregion
103.     #region kigyűjtés
104.     for (int i = 0; i < n; i++)
105.         if (csoport[i].RendesJegyek.Count < max * 0.8)
106.             Console.WriteLine('\t' + csoport[i].Név);
107.     #endregion
108. }

```

A megoldás két típusalgoritmus tipikus alkalmazása. Talán az egyetlen érdekesség a háromszoros adatösszetétel: a tömb elemének a listarészének a hosszát használjuk. Nüánsznyi hatékonysági kérdés, hogy jó-e a 105. sorban, hogy a ciklusmag minden egyes lefutásakor kiszámoltatjuk a `max` 0,8-szeresét, mikor ezt elegendő lenne egyszer, a ciklus előtt megtennünk. A kód olvashatósága, későbbi értelmezés szempontjából jobb így. Emellett az optimalizálási lehetőség eléggé nyilvánvaló ahhoz, hogy a compiler (`cs.exe`) észrevegye és a bináris kódban a sokszori kiszámolás helyett a kiszámított értékre hivatkozzon.

Az **5. feladat** megoldásához bontsuk a feladatot két részre! Egyrészt meg kell határoznunk egy-egy diák átlagát, másrészt az átlag alapján minden diáknak meg kell határozni az év végi jegyét. Itt a másolás típusalgoritmus belsejében kellene átlagot számolni, ehelyett az átlag megállapítására függvényt írunk. Tudjuk, hogy a témazárók duplán számítanak, de ennek figyelembevételét jelentősen megkönnyíti, hogy már a fájl beolvasásakor elkülönítettük a témazáróra kapott jegyeket a rendes jegyeiktől. Ezt kihasználva a függvényünk viszonylag egyszerű formát ölt:

```

110. static double Atlag(Diak diak)
111. {
112.     double szum = 0;
113.     for (int i = 0; i < diak.RendesJegyek.Count; i++)
114.         szum += diak.RendesJegyek[i];
115.     for (int i = 0; i < diak.TZdb; i++)
116.         szum += 2 * diak.TZjegyek[i];
117.     return szum / (diak.RendesJegyek.Count + 2 * diak.TZdb);
118. }
119.

```

A függvényt felhasználva az értékelés már nem nehéz, csak hosszú.

```

120. static void Evvege(Diak[] csoport, int n)
121. {
122.     for (int i = 0; i < n; i++)
123.     {
124.         double atlag = Atlag(csoport[i]);
125.         int jegy;
126.         if (atlag <= 1.7)
127.             jegy = 1;
128.         else if (atlag <= 2.5)
129.             jegy = 2;
130.         else if (atlag <= 3.5)
131.             jegy = 3;
132.         else if (atlag <= 4.5)
133.             jegy = 4;
134.         else
135.             jegy = 5;
136.         Console.WriteLine($"{csoport[i].Név, -25} átlaga: {atlag,5:F2}
                                jegye: {jegy}.");
137.     }

```

Az elégtelen kivételével, majdnem megoldás lehetne az átlag kerekítése, de a feladat szövege alapján szigorú a tanár, csak a .5 fölött kerekít felfelé. A kerekítéssel egyébként is érdemes óvatosan bánni, mert a `Math.Round(atlag)` a két egész közötti felező értéket a páros szám felé kerekíti, így a 3,5 és a 4,5 is 4-es (jó) lesz. Kellemetlen a jeles osztályzatra vágyóknak, de ez a szakmák körében jellemzőbben elfogadott kerekítési szabvány, ha más szeretnénk, akkor további paramétereket kell megadnunk a függvénynek. A feladat szövegének – az elégtelen kivételével – a `Math.Round(atlag, MidpointRounding.ToZero)` felel meg.

Ezzel elkészültünk a feladat teljes megoldásával, de azért elgondolkodtató, hogy az átlag az egyes diákokhoz tartozó adat, ráadásul diákok értékelésére specifikus a számítási mód. Képzeljük el, hogy nem csak kétféle súly létezik, hanem az évfolyam vizsga háromszoros súllyal számítana, a felszerelés otthonhagyása 1/5 súlyú elégtelen lenne... (nem szabad lebecsülni a tanárokat, ha a jegy súlyozásának a finomításáról van szó). Ezért praktikus lenne, ha a Diák adatai között számolnánk a diák átlagát.

A feladat megoldása során több megállapítást tettünk az adatstruktúra tervezésével kapcsolatban. Minél nagyobb egy program, minél összetettebb a megoldandó feladat, annál hangsúlyosabb lesz az algoritmus kitalálása mellett a könnyen, jól használható adatstruktúra megtervezése és létrehozása. Erre a probléma megoldását támogató adatstruktúra tervezésére és megalkotására eszköz az objektumorientált programozás. Ebben épp úgy, ahogy a strukturált programozásban a vezérlési struktúráknak és típusalgoritmusoknak vannak szabványok, beartandó adatszerkezési elvek és tipikus adatszerkezetek.

A módszer lényegének, fontosabb tulajdonságainak ismerete akkor is segítségünkre lehet, ha olyan rövid programokat írunk, mint amilyeneket eddig írtunk. Ezt már a 9-es jegyzetben is tapasztalhattuk, hiszen már ott ismerkedtünk az objektumok jellegzetes részeivel, a 10-es jegyzetben pedig kiegészítő tananyag volt a `class` használata, ami az objektumorientált programozás alapja. Nemsokára grafikus felhasználói felületre is programokat fogunk írni, mert ha nem is tananyag, de ott lehet „igazi” alkalmazásokat készíteni. Ezek a programok már nem csak egész számokat tartalmaznak, hanem mindenféle objektumokat, ezért hasznos, ha minél pontosabb elképzeléseink vannak az objektumorientált programok mibenlétéről.

ADATSZERKEZETEK TERVEZÉSE ÉS MEGVALÓSÍTÁSA OBJEKTUMOSZTÁLYOKKAL

Már a 10-es jegyzetben volt arról szó – kiegészítésként –, hogy a többféle adattípusból álló (heterogén) összetett adatok kezelésére csak az egyik lehetőség a `struct`. Ez az adatbáziskezelésben is használt rekord C# nyelvű megfelelője. Az adatbázisokban tároljuk az adatokat, a programjainkban azonban ezek „életre kelnek”, az adatok értéke gyakran változik, amitől lehetséges például, hogy a képernyőn mozogjanak, átalakuljanak játékunk elemei. Egy `struct`-tal megadott (definiált) összetett adattípus arra való, hogy egy entitás (létező valami) állapotát leírjunk vele. Ehhez képest az objektumorientált programozásban az objektumnak nem csak állapota van – nem csak létezik –, hanem működik is. Ez a működés nem csak a létrejöttében nyilvánul meg, hanem különböző állapotváltozásokban is és tevékenységekben is.

Az objektumnak – az entitáshoz képest – vannak belső tulajdonságai, belső működése, ami kívülről esetleg csak közvetve érhető el, esetleg csak módosítani tudjuk, vagy módosítani nem tudjuk, de láthatjuk az értékét. Kézzel fogható példa erre a robot, aminek vannak „érzékszervei”, ezek a robot belsejében az adatokat módosíthatják, és vannak „megnyilvánulásai”, amelyeket értelmezni tudunk. Például: néhány gomb lenyomásának hatására valamely belső változónak az értéke 0-ról 1-re változik, ami egy belső eljárás eredményeként másik változót módosítva LED-eket kapcsol be vagy ki. A LED-ek helyére egy másik robotot tehetünk, aminek a gombnyomását helyettesíti az első robot kimenő jele. Az egyes robotok belső működésére írt programot és belső változókat csak az engedélyezett formában – interface-en keresztül – érjük el.

Az objektum adattagjait és működésének módját osztálydefinícióban – `class` – adjuk meg. Az objektum létrehozásakor az ebben megadott konstruktor fogja meghatározni a belső változók kezdőértékeit. Az osztálydefinícióban írjuk le a különböző adatok, eljárások és függvények formájában a belső működést és a kifelé történő kommunikáció módját. A `class`-ban megadunk egy „létező és működő” adattípust, aminek alapján – a konstruktor futtatásával – példányosítunk egy objektumot. Összefoglalva egy definícióba:

`class` : adatszerkezet és az adatszerkezeten végezhető műveletek együttese, ami a belőle példányosított objektum működését írja le.

Objektum inicializálása

Kényelmi okokból a struktúráinknak is volt konstruktora, ezzel megadhattuk az adattagok kezdőértékeit. Ha nem írtunk konstruktort, akkor null, vagy valamilyen, az adattípusra jellemző kezdőértéke volt az adattagnak. Mennyiben lesz más a helyzet az objektumok esetén?

Az első különbség, hogy az objektumosztályok a `Program` osztállyal azonos fajtájú programozási szerkezetek, ezért nem a `Program`-on belül, hanem azon kívül írjuk meg őket. A `Program` – amikor benne példányosítunk egy objektumot – csak azt látja, amit az osztálydefinícióban `public`, azaz publikus jelzővel látunk el.

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

A második különbség, hogy a `struct` nyitott a belepiskálásra, az objektumban ezt szabályozzuk. Ez azt jelenti, hogy a belső változók, adatok `private`-ok. Az, amit kívülről láthatóvá tesszünk, azok **tulajdonságok** (property). A C# nyelvre jellemző, hogy a belső változót nem kell kiírni a kódba, ezért általában csak a tulajdonságot látjuk a kódolás során is. Nyelvi szabály, hogy a tulajdonságokat nagy kezdőbetűvel írjuk, ilyenkor a C# ugyanazon a néven, de kis kezdőbetűvel hozza létre a belső, privát adatot.

Egy belső (rejtett) változót és hozzá tartozó írható-olvasható tulajdonságot hoz létre a `prop` és két TAB billentyűleütés.

60. példa: A tanuló osztálya a Tanulo osztály.

Készítsünk egy objektum osztályt a tanulók vezetéknévének, keresztnévének és évfolyamának tárolására! Próbáljuk ki, hogyan használhatjuk, ha csak a tulajdonságokat adjuk meg, majd írjunk konstruktort is.

```
1. using System;
2. namespace Tanulo_minta
3. {
4.     class Tanulo
5.     {
6.         public string Vez { get; set; }
7.         public string Ker { get; set; }
```

A 'prop' snippetet csak az adattípussal és megnevezéssel kell kiegészíteni. A rejtett kódokat is megadhatjuk. Az `Evf` publikus tulajdonság a privát `evf` (egész) értéket mutatja meg a 11. sorban látható `get` függvény meghívásával. A kívülről láthatatlan `evf` értéket a 12. sorban, az `Evf` tulajdonság `set` függvénye közvetíti.

```
8.     int evf;
9.     public int Evf
10.    {
11.        get { return evf; }
12.        set { evf = value; }
13.    }
```

Ha minden esetben így kellene megírni a kódot, az nagyon unalmas lenne. Ilyen a Java nyelv, ahol a fejlett IDE a teljes kód kiírását azzal könnyíti, hogy egy privát változó megadásakor felajánlja a „getter” és „setter” kódokkal a kiegészítést.

Miután megírtuk az adattagokat, az alapértelmezett (default) konstruktor segítségével már használhatjuk is:

```
27. class Program
28. {
29.     static void Main()
30.     {
31.         Tanulo egyik = new Tanulo();
32.         Tanulo másik = new();
```

A 31. sorban a szabványos létrehozás látható, a 32. sorban látható megoldás a Visual Studio 2019-es verziójánál újabbakban használható. Ismét egy kényelmi szolgáltatásról van szó, mivel a fordítóprogram a típusból – ha csak nem akarunk valami speci dolgot művelni – tudja, hogy melyik osztály konstruktora kell: azé, amelyiket épp példányosítjuk.

Lehetne másképp is? Igen. Például, ha lenne egy Ember osztályunk, aminek neve van, de évfolyama nincs, továbbá a Tanulót úgy írjuk meg, hogy ő egy évfolyammal is bíró Ember. Ezt a lehetőséget hívják objektumosztályok származtatásnak.

A létrejövő két `Tanulo` évfolyama 0, a vezeték és keresztnéve `null` értékű, mielőtt használnánk, minden tulajdonságának – azon keresztül a belső változóinak – egyenként értéket kell adni.

```
33.     egyik.Vez = "Lopakodó";
34.     egyik.Ker = "Kasztanyetta";
35.     egyik.Evf = 6;
36.     másik.Vez = "Surranó";
37.     másik.Ker = "Szalicil";
38.     másik.Evf = 6;
```

Ezt követően már használhatjuk, kiírhatjuk és módosíthatjuk is a – publikus – tulajdonságokat:

```
39.     Console.WriteLine($"{egyik.Ker} {másik.Evf}");
40.     másik.Evf += 1;
41.     egyik.Ker = egyik.Ker + " " + "Harmónia";
42.     Console.WriteLine($"{egyik.Ker} {másik.Evf}");
```

Akárcsak a `struct` esetén, az objektumok létrehozását (példányosítását) is sokkal kellemesebbé lehet tenni egyedi igényeinknek megfelelő konstruktorok írásával.

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

Egy objektumosztálynak többféle konstruktora is lehet, csak annyi az elvárás ezekkel szemben, hogy a konstruktorok paraméterlistája eltérő legyen, mert így lesz egyértelmű, hogy melyik konstruktort kell alkalmazni.

Fontos kiegészítés, hogy a default konstruktor csak addig használható, ameddig nincs másik konstruktor. Ezért gyakori, hogy egyből két konstruktort írnak a programozók, az egyiket a számukra praktikus paraméterezéssel, a másikat paraméter nélkül.

Ha a konstruktor paramétereik közül némelyeket nem szeretnénk minden alkalommal megadni, akkor egy kezdőérték beállításával a paraméter opcionálissá tehető, de ezeket csak a lista végétől visszafelé lehet megadni is és kihasználni is. Így egy paraméterlistán előre kell írni a kötelezően megadandó paramétereket, ezután „fontosság” szerint csökkenően az opcionális paramétereket (az elhagyásuk esetén behelyettesítendő értékkel). A használat során a legkevésbé fontos paramétereket a lista végén nem kell megadni, de közbülső paramétert nem lehet kihagyni.

`Tanulo` osztályunkhoz két konstruktort írunk:

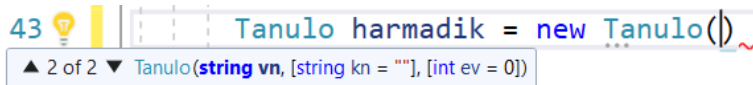
```
14.     public Tanulo(string vn, string kn = "", int ev = 0)
15.     {
16.         Vez = vn;
17.         Ker = kn;
18.         Evf = ev;
19.     }
```

A fenti konstruktor használatkor meg kell adni a vezetéknévét, ezt követően megadhatjuk a keresztnévét és az évfolyamot is. Ha az évfolyamot meg akarjuk adni, akkor a keresztnév is meg kell adnunk.

```
20. public Tanulo()
21. {
22.     Vez = "";
23.     Ker = "";
24.     Evf = 0;
25. }
```

A default konstruktor helyett paraméter nélküli konstruktort írhatunk. Egy másik, kényelmes megoldás lehet, ha az előbbi konstruktorban a `vn`-nek is adunk opcionális értéket, mivel így minden paramétere elhagyható lenne.

A két konstruktor elkészítése után a felhasználáskor választhatunk a konstruktorok közül.



A kiválasztott konstruktornak láthatjuk a szignatúráját (paraméterlistáját is), láthatjuk, hogy a `vn` kötelező, a `kn` és az `ev` opcionális, és az elhagyásuk esetén érvényes értéket. A konstruktorok kipróbálásához a harmadik tanulónak nem adjuk meg az évfolyamát, a negyedik tanuló adatait a felhasználótól kérjük be:

```
27. Tanulo harmadik = new Tanulo("Ó", "Pál");
28. Console.WriteLine("Add meg a diák vezetéknévét, keresztnévét és
    évfolyamát szóközzel elválasztva: ");
29. string[] sor = Console.ReadLine().Split(' ');
30. Tanulo negyedik = new(sor[0], sor[1], int.Parse(sor[2]));
31. Tanulo[] team = new Tanulo[4] {egyik, másik, harmadik, negyedik };
32. for (int i = 0; i < 4; i++)
33.     Console.WriteLine($"{i + 1}. {team[i].Vez} {team[i].Ker}
    {team[i].Evf}");
34. }
```

Ha nincs hiba a programban, akkor a paraméter nélküli konstruktort is teszteltük, mivel az első két tanuló már az alapján jött létre.

Feladat

1. Írjunk programot `allataink` néven! Írjuk meg az `Allat` osztályt és a konstruktorát is, amivel az egyes példányok nevét, fajtát és korát is megadjuk!
2. Állítsunk elő `Allat` osztályú objektumokat az `allatok.txt` fájl alapján! Az objektumokat tároljuk az `allatok` nevű listában!
3. Járjuk be a listát, és írjuk ki a malacok nevét!

Megoldás

```
1. using System;
2. using System.IO;
3. using System.Collections.Generic;
4. namespace Allatok_class
5. {
```

A konzolra íráshoz kell a System, a fájlból olvasáshoz az IO, a listához a Collections.Generic. Az első feladat az `Allat` osztály megírása.

```
6. class Allat
7. {
8.     public string Nev { get; set; }
9.     public string Faj { get; set; }
10.    public int Kor { get; set; }
11.    public Allat(string nev, string faj, int kor)
12.    {
13.        Nev = nev;
14.        Faj = faj;
15.        Kor = kor;
16.    }
17. }
```

A második feladat megoldásához, a fájlból beolvasott sort egyszerűbb konstruktorban bontani, mint a programban, ezért még egy konstruktort írunk (vagy át lehet alakítani az elsőt, mert azt nem fogjuk használni).

```
18. public Allat(string sor)
19. {
20.     string[] bont = sor.Split(' ');
21.     Nev = bont[0];
22.     Faj = bont[1];
23.     Kor = int.Parse(bont[2]);
24. }
25. }
```

A fájlból beolvasást a `Main()` programban végezzük, a feladatban kért listába:

```
26. class Program
27. {
28.     static void Main()
29.     {
30.         List<Allat> allatok = new List<Allat>();
31.         StreamReader sr = new StreamReader("allatok.txt");
32.         while (!sr.EndOfStream)
33.         {
34.             allatok.Add(new Allat(sr.ReadLine()));
35.             sr.Close();
36.         }
37.     }
38. }
```

A harmadik feladatban semmi újdonság nincs.

```
36. foreach (Allat allat in allatok)
37.     if (allat.Faj == "macska")
38.         Console.WriteLine(allat.Nev + " ");
39. }
40. }
41. }
```

Objektumaink működni kezdenek – Függvények az objektumok belsejében

61. példa: A macskák fejlődésének nyomon követése

Egy `macsek` nevű programfájlban hozzunk létre egy `Macska` nevű osztályt! A belőle példányosított objektumok a konstruktorban paraméterként kapják meg a macska nevét, születési évét és grammban kifejezett tömegét, Ezenfelül hozza létre a `tapasztalat` nevű jellemzőt, és adjon

számára értékül egy 10 és 50 közötti véletlen számot! Ezzel a jellemzővel követjük majd nyomon a macskánk fejlődését: Minden sikeres vadászat növeli a tapasztalatot és ezzel a következő vadászat sikerének esélyét.

```
1. using System;
2. namespace Macsek
3. {
4.     class Macska
5.     {
6.         public string Nev { get; set; }
7.         public int SzEv { get; set; }
8.         public int Tomeg { get; set; }
9.         public int Tapasztalat { get; set; }
10.        private Random rnd = new Random();
```

A véletlenszám előállítás érdekében az osztályhoz – osztályon belül globálisan elérhetően – hozzáadjuk a `Random` osztály egy példányát. Korábban is csináltunk már ilyet, csak egy kicsit másképp mondtuk. Mivel akkor a `Program` osztályba tettük, elé kellett írni, hogy **static**, de itt nem írjuk elé. A konstruktorban pont úgy használjuk az `rnd`-t, mintha egy függvényen belül használnánk:

```
11.        public Macska(string nev, int ev, int tomeg)
12.        {
13.            Nev = nev;
14.            SzEv = ev;
15.            Tomeg = tomeg;
16.            Tapasztalat = rnd.Next(10, 50);
17.        }
```

A `Main()` eljárásban hozzunk is létre két macskát:

```
62. class Program
63. {
64.     static void Main()
65.     {
66.         Macska cs = new("Csillus", 2018, 3652);
67.         Macska z = new Macska("Zokni", 2017, 4003);
68.     }
```

Az osztály műveleteit függvényekkel (eljárásokkal) valósítjuk meg. Ez a függvény annyiban tér el az eddig megszokott függvényektől, hogy a `Macska` osztályon belül van és (ezért) a **static** jelzőt nem szabad kiírni. Az osztályok, objektumok belsejében létező publikus függvényeket tagfüggvénynek hívjuk, ez a név kifejezi, hogy ezek is függvények. Talán a legelterjedtebb elnevezésük a metódus, de szokás őket függvénytagnak is hívni. Az első tagfüggvényünk arra való, hogy a macskáink „nyávogni” tudjanak. (Na, jó... csak kiírjuk, de esetleg a `Console.Beep()` is kipróbálható magányos gyakorlás során.)

```
18.        public void Nyavog()
19.        {
20.            Console.Write("Nyaú! ");
21.        }
```

A második tagfüggvényünk azt szemlélteti, hogy a tagfüggvényeknek is lehet visszatérési értékük. A példában szereplő tagfüggvény egy paramétert is felhasznál a visszatérési érték kiszámításához.

```
22. public int Kor(int idén)
23. {
24.     return idén - SzEv;
25. }
```

Próbáljuk ki, a `Main()`-ben a már megírt függvényeink használatát. A példány neve utáni ponttal lehet a tulajdonságot és a tagfüggvényt kiválasztani. A kor kiszámításához a viszonyítás évét meg kell adni:

```
69. cs.Nyavog();
70. Console.WriteLine(cs.Nev + " kora 2022-ben: " + cs.Kor(2022) + " év.");
```

A harmadik tagfüggvény arra mutat példát, hogy miként lehetséges egy jellemző értékének megváltoztatása tagfüggvény hívásával:

```
26. public void Eszik()
27. {
28.     Tomeg += 25;
29. }
```

... és a használata:

```
71. Console.Write($"{cs.Nev} tömege evés előtt: {cs.Tomeg} g");
72. cs.Eszik(); /*nem kell paraméter, mert saját adattal számol*/
73. Console.WriteLine($"... és evés után: {cs.Tomeg} g.");
74.
```

Természetesen egy tagfüggvény többször is hívható:

```
75. Console.Write($"{z.Nev} tömege zabálás előtt: {z.Tomeg} g");
76. for (int i = 0; i < 10; i++)
77.     z.Eszik();
78. Console.WriteLine($"... és zabálás után: {z.Tomeg} g.");
```

Az osztály utolsó előtti tagfüggvénye pedig azt példázza, hogy egy tagfüggvény egészen bonyolult műveleteket is megvalósíthat. Az egerészéshez kell egy egér, valamilyen ésszel, melekülési képességgel. Ezt a macska tapasztalatához viszonyított véletlen értékkel jellemezzük.

```
30. public void Egereszik()
31. {
32.     int eger_esze = rnd.Next(1, 100);
33.     Console.Write($"{Nev} egy {eger_esze} tapasztalatú egérrel
34.                                     próbálkozik. => ");
35.     if (Tapasztalat > eger_esze) /*a macska ügyesebb => nyer*/
36.     {
37.         Console.WriteLine("Megfogta");
38.         if (Tapasztalat < 100)
39.             Tapasztalat++; /*ha lehet, nő a tapasztalata is*/
40.         Eszik(); /*... és megeszi az egeret.*/
    }
```

```

41.     else                                     /*az egér ügyesebb =>*/
42.     {
43.         Console.WriteLine("Elszaladt :( ");
44.         Nyavog();                             /*nem eszik, hanem nyávog.*/
45.         Console.WriteLine();
46.     }
47. }

```

Természetesen ezt is ki kell próbálni: Kövessük végig valamelyik macskánk egerészéseinek egy hosszabb sorozatát:

```

79. Console.WriteLine(z.Nev + " egy hete");
80. for (int i = 0; i < 50; i++)
81. {
82.     Console.WriteLine($"{z.Nev} tömege: {z.Tomeg}, tapasztalata:
83.         {z.Tapasztalat}");
84.     z.Egereszik();
85. }

```

Végül, három nagyon haladó kiegészítés:

- Minden objektumnak van `ToString()` tagfüggvénye, aminek a visszatérési értékét írja ki a `.WriteLine()` és `.Write()` függvény, sőt, bármikor szöveggént is eltárolhatjuk. A C# beépített adattípusai, objektumosztályai mind hasonló módon biztosítanak lehetőséget arra, hogy könnyen értelmezhető formában megismerjük az objektum tartalmát. Ha szeretnénk olyan objektumokat készíteni, amelyek minden szituációban képesek normálisan kiírni az adattartalmukat, ezt viszonylag könnyen megtehetjük, csak meg kell adni, hogy mit szeretnénk látni, azaz az alapértelmezett szöveges formátumot felül kell bíráltni. Ehhez egy varázsszót kell ismerni: `override`.

```

48. public override string ToString()
49. {
50.     return Nev + " születési éve: " + SzEv + ", tömege: " + Tomeg + " g.";
51. }

```

- A műveleti- és relációsjeleket „értelemszerűen” használjuk egyszerű adattípusok közé írva, pedig ezek valójában függvények. Az `a + b` „hivatalos” formája `operator+(a, b)`; Az `a < b` nem más, mint a `operator<(a, b)`. Ezeket az operátorokat objektumok közötti műveletvégzéshez, összehasonlításhoz is használhatjuk, csak meg kell írni az osztályon belül az operátor új értelmezését: mit jelentsen a programnak a `'+'`, a `'<'` stb. A részletekhez az „C# operátor overloading” kifejezésre érdemes rákeresni. A kivitelezéshez lényeges, hogy ezek publikus, statikus függvények legyenek, megfelelő típusú visszatérési értékkel és – operátortól függően – egy vagy két operandussal, azaz paraméterrel. C# specialitás, hogy például a `<` mellé a `>` operátort is meg kell adni.

```

48. public static bool operator <(Macska a, Macska b)
49. {
50.     return a.SzEv > b.SzEv;
51. }
52. /**/
53. public static bool operator >(Macska a, Macska b) => a.SzEv < b.SzEv;

```

Próbáljuk is ki a kiírást és a relációs jel új használati módját:

```
86. Console.WriteLine("Végül: a fiatalabb macskánk:");
87. Console.WriteLine("\t" + (cs < z ? cs : z));
88. }
89. }
90. }
```

- Az objektumorientált programozási paradigmák (elvek) közül a három legfontosabb az **encapsulation**, azaz az összetartozó dolgok egységbe zárása, együtt kezelése; az **inheritance** (öröklés) és a **polymorphism** (többértékűség). Az **override** jelentése a programozási nyelvekben, hogy egy létező függvényt ír felül, az **overloading** egy létező műveletnek (operátornak) ad új szerepet, mindkettő az öröklődéshez kapcsolódó polimorfizmus. Nem csak az operátorokat lehet többféleképpen értelmezni. Láttuk, hogy konstruktorból is lehet több és ugyanez igaz az eljárásokra, függvényekre is. A polimorfizmus miatt tudjuk többféle módon paraméterezni a konstruktort és a függvényeket. Egy szabályt kell betartani: a visszatérési érték, a név és a paraméterlista alapján egyértelműnek kell lennie, hogy melyik értelmezést kell használni.

Feladatok

1. Oldjuk meg, hogy macskáink véletlenszerűen nyávogjanak „Nyaú!”-t vagy „Mijaú!”-t! A `Macska.Eszik()` tagfüggvénye helyett alkossuk meg a `Macska.TápotEszik()` és a `EgeretEszik()` tagfüggvényeit! Az előző vegye át a régi függvény szerepét, az új pedig legyen paraméterezhető a megevett egér grammban kifejezett tömegével! A megevett tipikusan 200 grammos egér tömege véletlenszerűen 5–25 százalékban fordítódjék a macska tömegének növelésére!
2. Írjunk ez egérre is osztályt, amelyben az egérnek esze, véletlenszerű (de a valóságban elképzelhető) tömege is legyen! A `Macska.Egereszik()` tagfüggvényét módosítsuk úgy, hogy a paraméterként kapott egér tapasztalatát és tömegét használjuk fel a `Macska.EgeretEszik()` tagfüggvény hívásakor!
3. *Kihívást jelentő feladat:* Két macska találkozásának gyakran hangos nyávogás és az egyik fél pánikszerű menekülése a vége. De melyik macska fut el? A megfelelő operátorokra írt tagfüggvények megvalósításával tegyük lehetővé, hogy két `Macska` osztályú objektum a szokásos relációs jelekkel összehasonlítható legyen! Az a `Macska` legyen „nagyobb”, amelyiknek a tapasztalata nagyobb! Adjunk az osztályhoz a nemet jelző privát 'Y' adattagot, aminek 50% valószínűséggel lesz kandúrt vagy nőtényt jellemző értéke. Ha különböző nemű macskák találkoznak, akkor az összeadásuk (+) eredménye legyen egy új `Macska`, kiscicához illő paraméterekkel! A nevét a (véletlenszerűen keletkezett) nemének megfelelő szülőtől kapja, elé téve a „kis” kiegészítést, pl. „kisCsilla” vagy „kisZokni”.
4. *Kihívást jelentő kutatási feladat:* A macska és az egér is állat. Elkészíthetjük az `Allat` objektumosztályt, majd ebből kiindulva lehetne fajta jellemzőkkel ellátni a macskákat, illetve egereket, a rájuk specializált objektumosztályokban. Ez esetben az objektumorientált programozás paradigmái közül az öröklődést alkalmazzuk: az `Allat` lesz a szülő (*base*) osztály, a `Macska` és az `Eger` lesznek a gyerek (*derived*) osztályok.

Sok objektum, mindegyikben sok adat

Írjuk át a csoportnaplós programunkat, mégpedig úgy, hogy struktúra helyett osztálydefiníciót írunk és eközben igyekszünk megoldani néhány, az előző megoldás során felvetődő problémát

is: az adatok felbontása során kezelhető tulajdonságok létrehozását és az egyes diákokra jellemző számítások osztályon belül történő elvégzését. (Ezeknek a problémáknak a megoldásához az előző fejezet haladóknak szánt kiegészítése nem szükséges.)

62. példa: Csoportnapló Diak osztályból

Tanulmányozzuk egy kicsit a programunk első kész változatát. Látható, hogy egy-egy sor megfelelő darabolása elég jelentős része a programunknak, ott dől el, hogy később mennyi munkánk lesz a tárolt adatokkal. Megfogadtuk, hogy az évfolyamot és a tagozatot külön eltároljuk, ezért ezt csináljuk meg. Eközben érdemes elgondolkodni azon is, hogy az adatokat milyen módon tesszük elérhetővé a felhasználó számára. Átgondoljuk, hogy melyek azok a függvények, amelyek egy-egy diákkal vagy a diák jegyeivel dolgoznak: azok, amelyek megjelenítik vagy visszatérési értékükben megadják a szóban forgó diák egy-egy jellemzőjét, vagy amelyek műveleteket végeznek a diák valamelyik jellemzőjével. Az ilyen függvényeket tagfüggvényekké alakítjuk, és megvalósítjuk azokat a lehetőségeket, amelyeket töprengéseink során hasznosnak gondoltunk. A **Diak** osztály jelentősen kibővül:

Programunk elején a névterek elfoglalnak néhány sort, így az osztálydefiníció a 6. sorban kezdődik.

```
6. class Diak
7. {
8.     public string Név { get; set; }
9.     private string osztály;
10.    public int Evfolyam { get; set; }
11.    public string Tagozat { get; set; }
12.    public int[] TZJegyek { get; set; }
13.    public int TZdb { get; }
14.    public static int maxTZdb = 3;
15.    public List<int> RendesJegyek { get; set; }
```

Programunk új verziójában a **struct** helyett a **Program** osztály előtt hozzuk létre a **Diak** osztályt. Az adattagok helyett tulajdonságokat definiálunk. A korábbi Osztály adattag helyett az **Evfolyam** és a **Tagozat** tulajdonságokat fogjuk használni, de eltároljuk az osztályt is: az **osztály** privát, belső változó. A **Program** osztályon belül nem tudjuk használni. Fontos jelzés, hogy az **Evfolyam** és a **Tagozat** módosítható később is – a **public** és a **set** miatt –, eközben az osztály megtartja az értékét, azt, ami a létrehozáskor volt. Ez esetleg a következő fejlesztésnél módosítható, de az is elképzelhető, hogy az adatot nem tároljuk.

Összetett adat ugyanolyan módon adható meg tulajdonságként, mint az egyszerű adat. Jelen esetben lehetővé tesszük a listák és az adatok módosítását is.

A témazárók kezelése alapos megfontolásokat igényel. A témazárók maximális száma a feladatból derül ki. Házirendben lehet rögzíteni, hogy egy diákkal egy tárgyból egy év alatt hány témazárót lehet írni, de a házirend módosulhat. Ha ezt az értéket is módosítani kellene, akkor nem egyik vagy másik diákra, hanem mindenkire egyformán kellene érvényesíteni. Erre jó a **static** jelző. A használata emiatt **Diak.maxTZdb** formában lehetséges. Így, hogy **public** jelzőt kapott, egy olyan adattag lesz, amit ki tudunk írni, tudunk módosítani, de valójában privátnak kellene lennie és csak a rendszeradminisztrátor számára elérhető függvénnyel szabadna lehetővé tenni a módosítását.

A másik témazáró adat a **TZdb**. Ez minden diák esetén egyedi, attól függ, hány dolgozatjegy van beírva a **TZJegyek** tömbbe. A program futása során szükséges lehet az adat kiírására,

ezért publikus, de ne akarjuk módosítani. Profi megoldás az lenne, ha egy új témazárójegy beírását egy olyan eljárás végezné, ami a jegy beírása mellett módosítja a tulajdonság mögé rejtett belső változót, de most csak azt jelezzük, hogy ezt a tulajdonságot nem módosíthatja „csak úgy” se a felhasználó, se a programozó.

Amíg a programunk csak néhány száz soros és csak saját használatra készül, a fenti megállapítások irrelevánsok. Minden adathoz rendelkezünk egy publikus, írásra és olvasásra is alkalmas tulajdonságot. Azonban, ha egy grafikus vagy más alkalmazást szeretnénk írni és ehhez mások által elkészített osztályokat szeretnénk használni, akkor érteni kell a sűgő jelzéseit.

Például – a korábban már használt nyelvi elemek közül – az `int.MaxValue` az `int` egy statikus, publikus adata; a diák neve `string` típusú, a név hossza, a `Length` tulajdonság, csak `get` hozzáférésű, nem módosítható. Vegyük észre, hogy a felhasználási helyzetek hasonlóak a témazáró adatainak kezeléséhez és az objektumosztályok megalkotása épp úgy tervezést igényelnek, mint az algoritmusok kitalálása. A szabályok, a „tervezési minták” egy adott felhasználási környezethez adnak javaslatot.

A konstruktorban az adatsort felbontva, elemezve adunk kezdőértéket az egyes tulajdonságoknak.

```
16. public Diak(string sor)
17. {
18.     string[] adatok = sor.Split(", ");
19.     Név = adatok[0];
20.     osztály = adatok[1];
21.     Evfolyam = evfolyam(osztály);
22.     Tagozat = tagozat();
```

Az `Evfolyam` az osztály adatának elejéből lesz, de ez itt nem látszik, mert egy privát, azaz csak az osztályon belül látható függvénnyel adjuk meg. Ugyanígy a `Tagozat` adata is egy privát függvény eredménye. Ezeket a függvényeket pont úgy írjuk és használjuk, mint ahogy a korábbiakat, csak itt nem szabad `static` jelzőt írni eléje, mert nem minden diákra együtt értendő a kiszámításuk, hanem diákonként egyedileg adnak eredményt.

```
23.     string[] jegyek = adatok[2].Split(" ");
24.     TZJegyek = new int[maxTZdb];
25.     TZdb = 0;
26.     RendesJegyek = new List<int>();
27.     for (int i = 0; i < jegyek.Length; i++)
28.     {
29.         if (jegyek[i].Length == 1)
30.             RendesJegyek.Add(int.Parse(jegyek[i]));
31.         else
32.         {
33.             TZJegyek[TZdb] = jegyek[i][0] - '0';
34.             TZdb++;
35.         }
36.     }
37. }
```

A jegyek feldolgozása a konstruktoron belül megírt szétválogatás és másolás típusalgoritmus alkalmazásával történik, a konstruktor belsejében ugyanazok a szabályok érvényesek, mint egy eljárás vagy függvény belsejében.

Az `evfolyam()` függvényben szövegfüggvényekkel és konverziós függvényekkel ügyeskedve is megkaphatjuk az évfolyam számát. Például a `Substring()` függvénnyel leválasztva az első két karaktert, ezt megpróbáljuk átalakítani; ha nem sikerül, akkor csak az első karaktert választjuk le és ezt konvertáljuk. Az alábbi megoldás a karaktersorozatból, a karakterkódok értékéből számolja ki a szám értékét.

```

38. private int evfolyam(string osztály)
39. {
40.     int e = 0;
41.     int poz = 0;
42.     while (osztály[poz] >= '0' && osztály[poz] <= '9')
43.     {
44.         e = e * 10 + (osztály[poz] - '0');
45.         poz++;
46.     }
47.     return e;
48. }

```

A függvény egy kiválasztással kombinált összegzés. Kiválasztjuk az első nem számjegyet, amíg ezt el nem érjük, addig a számjegyek számértéként a korábbi érték 10-szereséhez hozzáadjuk. Hasonló módon lehetne a számjegyeket egy szövegbe összefűzni és utána konvertálni egész számmá.

A `tagozat()` függvényben meg lehetne adni a jelként használható ábécét, ami nagyon hasonlít például a telex üzenet átalakításáról szóló feladathoz. Itt egy másik trükk szerepel: az ábécé betűinek van kis és nagybetűs változata, a számoknak és egyéb jeleknek nincs. Az `osztály` szövegében megkeressük az első betűt, onnantól vesszük a `tagozat` adatát.

```

49. private string tagozat()
50. {
51.     int poz = 0;
52.     while (osztály[poz].ToString().ToUpper() ==
53.           osztály[poz].ToString().ToLower())
54.         poz += 1;
55.     return osztály.Substring(poz);
56. }

```

Az `evfolyam()` és a `tagozat()` függvény is az `osztály` adatból számol (38. és 49. sor). Látható, hogy a paraméter el hagyható, mivel a belső, de az osztályra nézve globális változókat az osztály minden függvénye, konstruktora ismeri. Ha megadjuk, azzal a szöveg későbbi olvashatóságát segítjük.

A konstruktorral elkészültünk. ... és – bár úgy tűnik, még el sem kezdtük a feladat megoldását, már a kód hosszának harmadánál tartunk. Ezt követően az egyes feladatok megoldását támogató függvényeket adunk az osztályunkhoz. Ezeket majd a `Program` osztályban szeretnénk használni, ezért a `Diak` osztályon belül publikusak lesznek, a `Diak` osztály tagfüggvényeiként fognak megjelenni.

A **2. feladathoz** hasznos, ha a „mindhárom témazárót megírta-e” kérdés helyett arra adunk választ, hogy minden előírt témazárót megírta-e. Ebben az esetben a következményt – kinek kell még pótolnia – helyezzük előtérbe, ami a feltételek módosulása után is jó eredményt fog

adni. A függvényünk publikus lesz és a függvény nevéből lehet következtetni arra, hogy mit csinál:

```
56. public bool MindenTzMegvan()
57. {
58.     return TZdb == maxTZdb;
59. }
60.
```

A függvényünk szinte semmit sem csinál, de mégis fontos, mert az objektumorientált programozás elve alapján a `maxTZdb` változónak privátnak kell(ene) lennie, az egyenlőség vizsgálata a `Diak` belső „magán” ügye, csak a függvény eredménye publikus.

A `JegyekSzama()` a **3–5. feladatokhoz** hasznos. Paraméterként kérjük a számlálandó adatok megnevezését, mert a programunk így könnyen bővíthető más típusú jegyekkel, miközben az elemszámhoz egy külső felhasználónak elegendő ezt a függvényt ismerni.

```
61. public int JegyekSzama(string milyen)
62. {
63.     if (milyen.ToLower() == "tz")
64.         return TZdb;
65.     else if (milyen.ToLower() == "rendes")
66.         return RendesJegyek.Count;
67.     else if (milyen.ToLower() == "összes")
68.         return TZdb + RendesJegyek.Count;
69.     else
70.         return -1;
71. }
72.
```

Ha ez a függvény létezik, akkor a `TZdb` tulajdonság helyett lehet privát adat, hiszen ez a függvény azt adja vissza és módosítani kívülről amúgy sem szabad.

Következő függvényünk a **4–5. feladathoz** szükséges `Atlag()` függvény, ami egyértelműen a diák saját adata, csak a jegyeitől függ. Ebben felhasználhatjuk a már megírt `JegyekSzama()` függvényt.

```
73. public double Atlag()
74. {
75.     double szum = 0;
76.     for (int i = 0; i < RendesJegyek.Count; i++)
77.         szum += RendesJegyek[i];
78.     for (int i = 0; i < TZdb; i++)
79.         szum += 2 * TZJegyek[i];
80.     return szum / (JegyekSzama("összes") + JegyekSzama("tz"));
81. }
82.
```


Végül az osztályzatot kiszámító függvény az 5. feladathoz kell – ez is a diákhoz tartozó adat, aminek az eredménye publikus. Most alkalmazzuk a C# kerekítő függvényét, nem törődve az-
zal, hogy az öttizedet merre kerekíti

```
83. public int Osztalyzat() /*év végi*/
84. {
85.     double atlag = Atlag();
86.     if (atlag <= 1.7)
87.         return 1;
88.     return (int) Math.Round(atlag);
89. }
90. }                                     /*ITT a Diak osztály vége!!!*/
91.
```

Befejeztük a **Diak** osztály írását. A programból lehet, hogy semmit sem írtunk meg, de a kód kétharmadánál tartunk. Talán vigasztaló, hogy a jó munka mindig hosszú tervezéssel, előkészülettel ár, hogy a végén gyorsan összeálljon a program, amit ráadásul utána könnyen lehet továbbfejleszteni, módosítani.

A **Main()** eljárást – mivel ugyanazt a feladatot oldjuk meg újra – át lehet másolni az előző megoldásból, az egyes feladatokra adott megoldásokat is többnyire módosítani érdemes, nem újraírni.

```
93. class Program
94. {
95.     static void Main()
96.     {
97.         const int MaxCsop = 35;
98.         Diak[] csoport = new Diak[MaxCsop];
99.         int N = 0;
100.        StreamReader sr = new StreamReader("naplo.txt");
101.        while (!sr.EndOfStream)
102.        {
103.            csoport[N] = new Diak(sr.ReadLine());
104.            N++;
105.        }
106.        sr.Close();
```

Látható, hogy az objektumok példányosítása ugyanúgy történik, ahogy egy struktúrájával megadott adatot létrehoztunk.

```
107. Console.WriteLine("1. A 11.zs tanulói:");
108. KiirZs(csoport, N);
109. Console.WriteLine("2. Kevés témazárót írtak:");
110. KevésTZ(csoport, N);
111. Console.WriteLine("3. 11.x-ből legkevesebb témazárót {0} írta.",
112.                                     MinTZ(csoport, N));
112. Console.WriteLine("4. Felelésre felkészülők:");
113. FelelésVeszély(csoport, N);
114. Console.WriteLine("5. Év végi átlagok és jegyek:");
115. Evvege(csoport, N);
116. }
```

Az egyes feladatok megoldásában felhasználjuk a **Diak** osztály tulajdonságait és függvényeit

```
117. static void KiírZs(Diak[] csoport, int n)
118. {
119.     for (int i = 0; i < n; i++)
120.         if (csoport[i].Evfolyam == 11 && csoport[i].Tagozat == "zs")
121.             Console.WriteLine('\t' + csoport[i].Név);
122. }
123. static void KevésTZ(Diak[] csoport, int n)
124. {
125.     for (int i = 0; i < n; i++)
126.         if (!csoport[i].MindenTZMegvan())
127.         {
128.             Console.WriteLine('\t' + csoport[i].Név + ": ");
129.             for (int j = 0; j < csoport[i].TZdb; j++)
130.                 Console.Write(csoport[i].TZJegyek[j] + ", ");
131.             Console.WriteLine("\b\b ");
132.         }
133. }
134. static string MinTZ(Diak[] csoport, int n)
135. {
136.     int mini = -1;
137.     int mindb = 4;
138.     for (int i = 0; i < n; i++)
139.         if (csoport[i].Evfolyam == 11 && csoport[i].Tagozat == "x")
140.             if (csoport[i].JegyekSzama("rendes") < mindb)
141.             {
142.                 mindb = csoport[i].JegyekSzama("rendes");
143.                 mini = i;
144.             }
145.     return mini == -1 ? "nincs 11x-es diák" : csoport[mini].Név;
146. }
```

Elgondolkodtató, hogy a **JegyekSzama()** függvényt itt minden cikluslépésben kétszer futtatjuk. Ez itt csak egy elágazás és hivatkozás a **TZdb**-re, de akkor is, kétszer egy-egy művelettel több. Ilyen kérdésekben a programozó dönt, a felhasználás során derül ki, hogy a döntése helyes volt-e, vagy – esetleg pont emiatt – a konkurencia programja gyorsabban fut.

```
147. static void FelelésVeszély(Diak[] csoport, int n)
148. {
149.     #region legtöbb rendes jegy
150.     int max = csoport[0].JegyekSzama("rendes");
151.     for (int i = 0; i < n; i++)
152.         if (csoport[i].JegyekSzama("rendes") > max)
153.             max = csoport[i].JegyekSzama("rendes");
154.     #endregion
155.     #region kigyűjtés
156.     for (int i = 0; i < n; i++)
157.         if (csoport[i].JegyekSzama("rendes") < max * 0.8)
158.             Console.WriteLine('\t' + csoport[i].Név);
159.     #endregion
160. }
```

```

161. static void Evvege(Diak[] csoport, int n)
162. {
163.     for (int i = 0; i < n; i++)
164.         Console.WriteLine($"{csoport[i].Név, -25} átlaga:
            {csoport[i].Atlag(), 5:F2} év végi jegye: {csoport[i].Osztalyzat()}.");
165.     }
166. }                                     /*Program osztály vége*/
167. }

```

Kiegészítések a grafikus felhasználói felületen programozás előtt

Modulok, több fájlból álló programok

A C# nyelven programozást úgy kezdtük, hogy a Visual Studio létrehozott egy projektet egy mintakóddal, több, egymáson belüli kapcsolószerűvel. A legbelső kapcsolószerű-páron belül kezdtük a kód írását. Később, eggyel feljebb lépve, írtunk függvényeket, eljárásokat és mostanra még egyet feljebb léptünk, osztályokat írunk. Eközben tapasztalhattuk, hogy a programunk futtatásához nem elég az, amit mi írunk, az éppen írt névtéren kívül más névterek is kellenek. Például, ha csak a konzolra akarunk kiírni valamit, akkor a `Console` osztályhoz használnunk kell (`using`) a `System` névteret. Ugyanakkor, ez a névtér használható a `Math` és a `Random` osztályokhoz is, de a fájlból olvasáshoz már egy alrész, a `System.IO` szükséges.

Egy „igazi” programban rengeteg kódsorból áll. Bár a programozási nyelv nagyon tömör, a megoldás aprólékos leírása óránként 100–200 új sort is eredményezhet. Mi is írtunk több, mint 200 soros programot. Egy hosszú fájl nehéz áttekinteni, ezért a programot projektként, több fájlban szokták megírni. A legáltalánosabb fájlokra bontási lehetőség, ha minden osztályt más fájlban írunk meg. A Visual Stúdió Solution Explorer ablakában a (félkövér) projekt nevének helyi menüjében nem csak a beolvasandó fájl lehet hozzáadni, hanem `Class-t` is. Ekkor egy új `.cs` fájl jön létre a program mappáján belül, ugyanaz a névtér lesz, fordításkor egyetlen `.exe` fájl lesz a végeredmény.

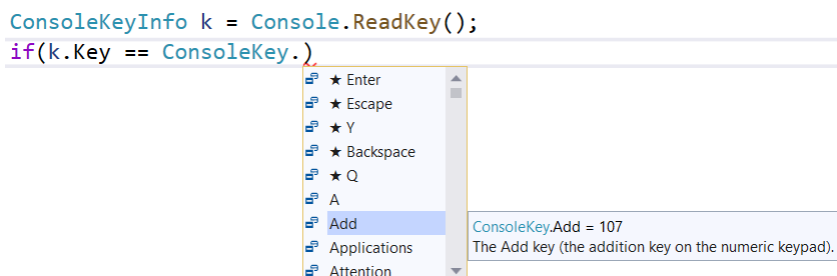
Grafikus alkalmazás készítésekor a látvány tervezését drag&drop technikával, kattintgatva végezhetjük. Eközben a háttérben számos kódsor keletkezik, amit később a programozó kiegészít a működés során elvártaknak megfelelően. Nem szerencsés, ha az IDE által generált kódba a programozó belenyúl és az sem, hogy ha a programozó által írt kódot az IDE módosítja, ezért akár egy `class` is lehet több fájlban. Ekkor azt látjuk, hogy amit éppen írunk az `partial class`.

Egy nagy program sokmillió sorból is állhat (az igazán nagy programok milliárd sorosok). Ennek a fordítása nagyon sok időt venne igénybe, ezért egyes részeit külön lefordítják, így a `using`-gal hivatkozott névterekben nem a kódfájl van, hanem előzetesen fordított modulok. Egy-egy modulban több osztály van, de jellemzően összetartozó dolgokat tesznek bele, amelyeket előzetesen nagyon alaposan tesztelnek. Az ilyen előfordított csomagok kiterjesztése `.dll`, ami a Dynamic Link Library fordítása. A Visual Studio program mappája – a telepített alrendszerektől is függően – több ezer `.dll` fájlt tartalmaz, összességében gigabájt közeli fájl mérettel. A névtereket hivatkozunk a programunkban, ezzel a megfelelő `.dll` fájlból a kívánt osztályt használhatjuk.

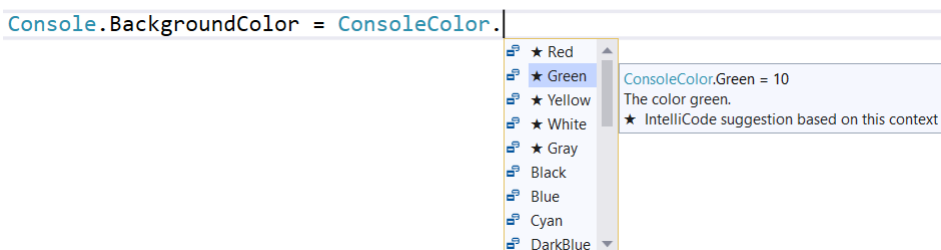
Nem class, nem struct, mégis türkíz színű – mi az?

Amikor egy program több kérdésre válaszol, érdemes menüt készíteni. Jellemző, hogy a menüpontok közül választáshoz csak egy billentyűt kell leütnie a felhasználónak, Ilyet a kódban

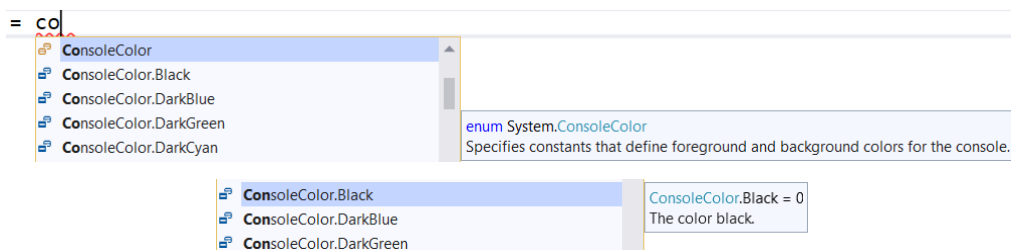
a `Console.ReadKey()` függvénnyel írhatunk elő. Ennek hatására, `ConsoleKeyInfo` típusú objektum keletkezik. Ennek az egyik tulajdonsága a `.KeyChar`, ami `char` típusú, de ez nem alkalmas sem a nyíl billentyű sem a DEL gomb megnyomásának ellenőrzésére. Egy másik tulajdonsága a `ConsoleKeyInfo`-nak a `.Key`, ami nem `char` típusú, hanem `ConsoleKey` típusú. Érdekesége, hogy ennek nincsenek se tulajdonságai se függvényei, csak sok egyforma fajtájú választható billentyűnév, mindegyiknek megfelel egy szám.



Hasonlót láthatunk akkor is, ha színes háttérrel vagy betűszínt állítunk be a konzolra íráshoz. A `Console.BackgroundColor` `ConsoleColor` típusú adat. Ez a típus csak színneveket tartalmaz, mindegyiket megfelelteti egy számnak.



Amikor épp csak elkezdjük beírni a `ConsoleColor`-t, akkor látható, hogy miről lenne szó:



A – sárga kártyás – `ConsoleColor` egy `enum`. A kék kártyás egy konkrét színnév, ami egy számmal egyenlő.

Az `enum` másra nem használható foglalt szó, körülbelül olyan fontos lehet, mint a `struct` vagy a `class`. Mit csinál? Megnevezéseket (szövegeket) számokkal társít.

Kísérletezzünk saját készítésű `enum`-mal:

```

1. using System;
2. namespace EnumMinta
3. {
4.     enum SZINEK { piros, kék, zöld }
5.     class Program
6.     {
7.         static void Main()
8.         {
9.             string[] szinek = { "sárga", "lila", "fekete" };
10.            string t = szinek[0];
11.            SZINEK e = SZINEK.piros;
12.            Console.WriteLine(t + " " + (e + 1) + " " + (int)e);
13.            /*           sárga           kék           0          */
14.            string s = ((SZINEK)2).ToString();
15.            Console.WriteLine(szinek[(int)(e + 1)] + " " + s); /*lila zöld*/
16.        }
17.    }
18. }

```

Az `enum` – teljes nevén enumerátor vagy felsorolás típus – elnevezéseket sorol fel. Ha nem adunk neki máshogyan értéket, akkor 0-tól számozza az elnevezéseket, lehet következő adatot kérni (pl. 12. sorban `(e + 1)`) Ha az adatot írjuk ki, akkor az elnevezést kapjuk, de explicit cast-tal az egész számmá átalakítható (11. sor), ami akár tömb indexelésre is használható (15. sor). Egy számot explicit cast-tal a `enum` megnevezéssé, azt pedig a `ToString()` függvénnyel szöveggé lehet alakítani (14. sor).

63. példa: Mérés és értékelés – (enum és modul)

Az iskolákban rendszeres a mérés és ezek értékelése. Egy-egy mérésre jellemző, hogy mennyire fontos, ami egyrészt az év végi eredményben a beszámítás súlyával fejeződik ki, másrészt a naplóba bejegyzés színével. Az értékelés lehet szöveges, százalékos vagy osztályzat (jegy) formájában. Most csak a jeggyel értékelést vizsgáljuk, ami nem csak egy szám, hanem minősítő szöveg: jeles, jó, ... elégtelen.

1. Készítsük el a `Meres.cs` fájlt, ebben definiáljuk a `Meres` osztályt és a mérést jellemző `Fajta` felsorolást! A `Main()` eljárásban hozzunk létre egy témazáró mérést!
2. A `Program.cs` fájlban definiáljuk a `Jegy` osztályt, amelyet a `Meres` osztályból származtatunk. A jegy elnevezéseit `JegyÉrték` felsorolásban adjuk meg.
3. Írjunk a `Jegy` osztálynak két konstruktort: az egyik egy méréshez adjon jegyet, a másik a mérés fajtája és eredménye alapján hozza létre a `Jegy` típusú objektumot.

A `Meres.cs` fájlt a projekt helyi menüjéből a `Class...` menüre kattintva, vagy a legelső, `New...` menüpont választása után, a `Class` lehetőséget kijelölve hozzuk létre.

```

1. using System;
2. namespace Kiegészites
3. {
4.     enum Fajta { proba, rendes, tz, vizsga }
5.     class Meres
6.     {
7.         public Fajta Suly { get; set; }
8.         public ConsoleColor Szin { get; set; }
9.         public Meres(Fajta fajta)
10.        {
11.            Suly = fajta;
12.            Szin = (ConsoleColor)((int)fajta * 2 + 1);
13.        }
14.    }
15. }

```

A `Meres` osztálynak két tulajdonsága van, mindkettő enumerátor. Lehetnének egész számok is, de az egyiknek csak 4, a másiknak 16 értéke lehet és pont ilyen számtípus nincs. Emellett minden értékhez konkrét elnevezés is társul. A Szin – ha komolyan vesszük – jellemzően kapcsolódik a mérés fajtájához, de nem függvénnnyel, hanem inkább egy megfelelő tömbben tárolva lehetne megadni, hogy melyik méréshez melyik szín illik.

A `Program.cs` fájlban, mivel a `Meres.cs` és a `Program.cs` egy projekten belül vannak korlátok nélkül használhatjuk a másik fájlba írtakat.

```

4. class Program
5. {
6.     static void Main()
7.     {
8.         Meres mérés = new Meres(Fajta.tz);

```

A 8. sorban nem szövegesen adjuk meg a mérés típusát, hanem az egyik `Fajta` értéket adjuk meg.

A Jegyérték felsorolásnál az értékeket is meg kell adni, mert 0-s jegy nincs. Ezt a felsorolást is a fájl elején adjuk meg, a `Program.cs` fájlban, így az előző tesztünk és a `Program` osztály elé írjuk a kódot.

```

1. using System;
2. namespace Kiegészites
3. {
4.     enum JegyÉrték {elégtelen=1, elégséges=2, közepes=3, jó=4, jeles=5}

```

A példától eltérően, nem kell minden értéket megadni, illetve, ha megadnunk minden értéket, akkor nem kell növekvő sorban írni.

A `Jegy` „származtatása” a `Meres` osztályból azt jelenti, hogy a `Jegy` egy `Meres`, ami további speciális tulajdonságokkal bír.

```

5. class Jegy :Meres /*Kettőspont!!!*/
6. {
7.     public JegyÉrték Ertek { get; set; }

```

Bár a `Jegy`nek csak egy saját tulajdonsága van, a `Meres` tulajdonságaival is rendelkezik. Ezért a konstruktorában megadjuk a `Jegyérték`et és a `Fajta` adatot is, és azt is megmondjuk, hogy a fajta az „ősoztály”, a `base` létrehozásához kell.

```

8.     public Jegy(JegyÉrték jegy, Fajta fajta):base(fajta)
9.     {
10.        Ertek = jegy;                                /*meg kell adni, mert új adat*/
11.        Suly = fajta;                                /*elhagyható, a base már adott értéket*/
12.        Szin = (ConsoleColor)((int)fajta);          /*van értéke, de nem ez!*/
13.    }

```

A származtatott osztály nem adattagként tartalmazza az őst, hanem szerves része. Ezért, nem tudunk értékadással átadni a belső (nem létező) adatnak egy paraméterként kapott `Meres`-t, de bármilyen összetett adatnak egy alkalmas része is megfelel értékadáskor, így egy `Meres` típusú is.

```

14.    public Jegy(Meres p, JegyÉrték ertek):base(p.Suly)
15.    {
16.        Ertek = ertek;
17.    }
18. }

```

Tehát: a `p` egy `Meres` típusú adat, ennek a `Suly` tulajdonsága `Fajta` típusú, amivel paraméterezhető az ő, így kap értéket a `Suly` és a `Szin`. Ezután a `JegyÉrték` típusú `Érték` tulajdonságot is megadjuk.

A kipróbálás:

```

23. static void Main()
24. {
25.     Meres mérés = new Meres(Fajta.tz);

```

Ez volt korábban a 8. sor. Most használni fogjuk a jegyek létrehozásához.

```

26.     /*Jegy létrehozása enum értékekkel*/
27.     Jegy a = new Jegy(JegyÉrték.jó, Fajta.tz);
28.     Console.Write("A jó tz: " + a.Ertek + " " + a.Suly + " ");
29.     Console.ForegroundColor = a.Szin;
30.     Console.WriteLine((int)a.Ertek + " " + (int)a.Suly);
31.     Console.ForegroundColor = ConsoleColor.Gray;

```

Futtatáskor figyeljük meg, hogy mikor változik a karakterek színe, valamint a szöveg és szám kapcsolatát.

```

32.     /*Jegy létrehozása explicit konvertált számokkal*/
33.     Jegy b = new Jegy((JegyÉrték)5, (Fajta)1);
34.     Console.Write("B 5, 1 jegy: " + b.Ertek + " " + b.Suly + " ");
35.     Console.ForegroundColor = b.Szin;
36.     Console.WriteLine((int)b.Ertek + " " + (int)b.Suly);
37.     Console.ForegroundColor = (ConsoleColor)8;

```

Az `a` és `b` jegyet ugyanazzal a konstruktorral hoztuk létre, az `enum` értékadásmódja más. A két megoldás mindkét paraméterében tetszőleges módszert alkalmazhatunk.

```

38.     /*Jegy megadása egy mérés felhasználásával*/
39.     Jegy c = new Jegy(mérés, JegyÉrték.közepes);
40.     Console.ForegroundColor = mérés.Szin;
41.     Console.Write("Egy tz mérés 3: " + mérés.Suly);
42.     Console.ForegroundColor = c.Szin;
43.     Console.WriteLine(" " + c.Suly + " " + c.Ertek);

```

Figyeljük meg a kiírás színét! A `mérés.Szin` ugyanaz, mint a `c.Szin`, mert nem adtunk más színt a `Jegy`-nek a 2. konstruktorban.

```
44.      /*Jegy megadása a mérés.Suly Fajtavál*/
45.      Jegy d = new Jegy(JegyÉrték.közepes, mérés.Suly);
46.      Console.ForegroundColor = mérés.Szin;
47.      Console.Write("Másik tz mérés 3: " + mérés.Suly);
48.      Console.ForegroundColor = d.Szin;
49.      Console.WriteLine(" " + d.Suly + " " + d.Erték);
```

Itt a szín megváltozik a kiírás során. A `mérés.Suly` értéke `tz`, azaz 2. A mérés színe $2 * Suly + 1 \Rightarrow 5$, ezt használja a 2. konstruktor a `Meres`-en keresztül a `Jegy`-re is. Az átadott `mérés.Suly` értéke 2, ez lesz a `Jegy` színe, az 1. konstruktor használva. A 2. szín a `DarkGreen`, az 5. szín a `DarkMagenta`.

```
50.      /*Jegy megadása nemlétező enum értékekkel*/
51.      Jegy e = new Jegy((JegyÉrték)6, (Fajta)15); /*15 a szín max értéke*/
52.      Console.ForegroundColor = e.Szin;
53.      Console.Write("Túl nagy értékből: " + e.Erték + " " + e.Suly + " ");
54.      Console.WriteLine((int)e.Erték + " " + (int)e.Suly);
```

Ha a `JegyÉrték`-ként értelmezett számhoz nincs érték, akkor a számot írja ki, de a `ForegroundColor` értéke legfeljebb 15 lehet, e fölött túlindexelés hibát ad.

```
55.      /*Ezt az elején érdemes beírni*/
56.      Console.ForegroundColor = (ConsoleColor)7;
57.      Console.WriteLine("Végén: " + Console.ForegroundColor);
58.  }
```

A feladatokat megoldottuk.

Modul készítése – Csak a mindent tudni akaróknak

Már csak az maradt ki, hogy modult készítsünk, de ez már egyetemeken sem első tananyag. Nem mintha annyira bonyolult lenne, inkább azért, mert a program mérete nem igényli.

- *Előzetes:* Nézzük meg, milyen fájlokat készít a Visual Studio és a `cs.exe` a programunk fordítása során. Lesz közöttük egy `.dll` fájl is, de ez a teljes programot tartalmazza. A másik programban felhasználandó modulban nem lehet benne a `Program` osztály.
- *Modul készítés:* Készítsünk új programot, de ez most ne `Console` alkalmazás legyen, hanem `Library` alkalmazás, és `MeresL` legyen a neve. Az példában elkészített `meres.cs` fájlt másoljuk át ennek a `MeresL.cs` fájljába (ez lesz megnyitva szerkesztésre), majd `F6`-tal fordíttassuk le a programot. (Futtathatjuk is, bár kapunk hibaüzenetet: nincs mit futtatni, de közben elkészül a `MeresL.dll`.)
- *Modul hozzáadás a programhoz:* Az elkészített `.dll` fájlt másolhatjuk. A fájlt a projekthez az `Add... -> Project Reference... -> Browse` panelen keresztül kell kikeresni. Ezt követően a tulajdonságaiban beállíthatjuk, hogy másolja be a projektbe. (Az aktuális állapot a projekt `.csproj` fájlban lesz olvasható.).
- *Modul használatba vétele:* A `Program.cs`-t kell kiegészíteni a „`using MeresL;`” sorral. Ha minden jó, akkor használható lesz a modul.

Valószínű, hogy nem lesz minden jó, mert a hozzáadás módja a Visual Studio verziójától függ. Az aktuális megoldásnak ezért az interneten érdemes utána járni.

GRAFIKUS FELHASZNÁLÓI FELÜLETŰ ALKALMAZÁST FEJLESZTÜNK

Nem tananyag, de ha programot használunk, akkor általában grafikus felhasználói felületet látunk.

... Itt lesz a Windows Form alkalmazás készítésének néhány trükkje. Szinte biztos, hogy nem a tankönyvi alkalmazás lesz, hanem valami más, valamilyen játék alap. Esetleg a 61. példa továbbfejlesztése.

KIEGÉSZÍTÉSEK TÁRGYMUTATÓJA

ASCII	40, 52, 53, 58, 135	Merge sort	132
barefoot	96	metódus	145
base	148, 158	objektumorientált	139, 140, 148, 152
big-endian	52	out	21, 47, 48, 118
breakpoint	6	overloading	147, 148
Bubble sort	104	override	147, 148
burkoló függvény	125, 126	öröklés	148
Carriage Return	40	paradigma	148
cast	157	polymorphism	148
collection	109	property	141
compiler	5, 138	query	108
CR	40, 52	Quick sort	127
delegate	108, 115	ref	21, 97
Dictionary	16, 17, 28, 38, 65	Refactoring	18, 22
encapsulation	148	Selection sort	103
enum	156, 157, 159	setter	141
getter	141	signature	40
háromoperandusú operátor	31, 65	swap	97
IDE	5, 6, 40, 42, 43, 141, 155	szemantikai hiba	6, 7
inheritance	148	szignatúra	47, 143
Insertion sort	105	szteganográfia	60
interface	140	tagfüggvény	18, 64, 76, 109, 148
interpreter	5	temporally	97
Key	16, 17	terminál	42
komparátor	108, 109, 112	többrétegűség	148
lambda-kifejezés	63, 64, 108, 109	Tuple	28
LF	40	UTF-8	40, 57, 58
Line Feed	40	valid	115
Linq	22, 67, 95, 108	Value	16, 17
LINQ	95, 108	wrapper function	125
little-endian	52, 54		