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 megjelenésében az állami tankönyv[1], valamint annak online kiegészítése képezi az alapját. Sokszor a szöveg is azonos, de a tankönyvben bemutatott Python nyelv helyett a C++ nyelvet, illetve a Code::Blocks 20.03 fejlesztőkörnyezetben történő programozást mutatja be.

Először átismételjük a 9-es jegyzetben (Digi09_Cpp_*.pdf[2]) részletezett témákat, a C++ nyelv és a programozás technikájának alapjait. Ezután a 10-es jegyzetben (Digi10_Cpp_*.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 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 vector<> és a class is. Egyes feladatoknál a tankönyvben preferált szótár használatára (map<>) 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 előző két jegyzethez hasonlóan, a bemutatott megoldások, példák, a lehetséges jó megoldások száma sokszorosa egy tanulócsoport létszámának.

Sikerekben gazdag tanulást kívánok:

Budapest, 2023.

Szalayné Tahy Zsuzsanna

---

[1] Digitális 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

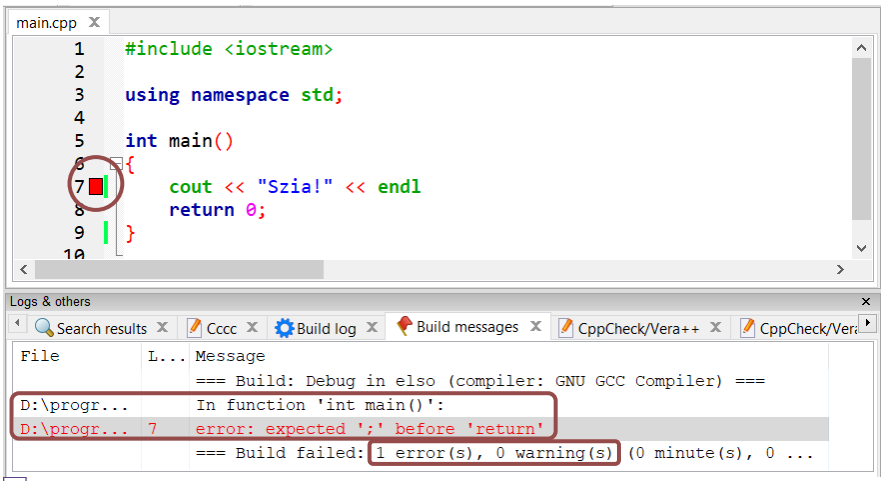
A kilencedik és tizedik évfolyamon jócskán belekóstoltunk a programozásba, és már egészen klassz dolgokra tudjuk rávenni a számítógépet – ebben és a soron következő három leckében először ezeket az ismereteket ismételjük át.

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. A Code::Blocks környezetben a C++ nyelven írt kódból – a **Build** ( ) utasításra vagy az CTRL+F9 billentyű lenyomásakor – a GNU GCC Compiler (g++.exe) készíti el a programot. Ezt követően a **Run** ( ) utasítással vagy CTRL+F10 billentyűkombinációval futtathatjuk a programot. A **Build and run** ( ) vagy F9 billentyű lenyomása a két utasítást egyesíti.

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

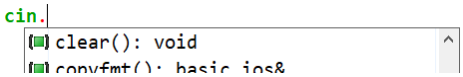


A fordítás során észlelt hibákat az IDE többféle módon jelzi.

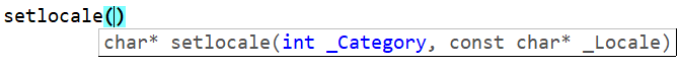
A Code::Blocks – egy integrált fejlesztő környezet (integrated development environment, röviden IDE) – az ilyen típusú hibák elkerüléséhez többféle segítséget, támogatást ad. Ezért a

programkód írásakor a monitoron folyamatosan ellenőrizzük a beírt kódunkat, figyelünk az IDE jelzéseire.

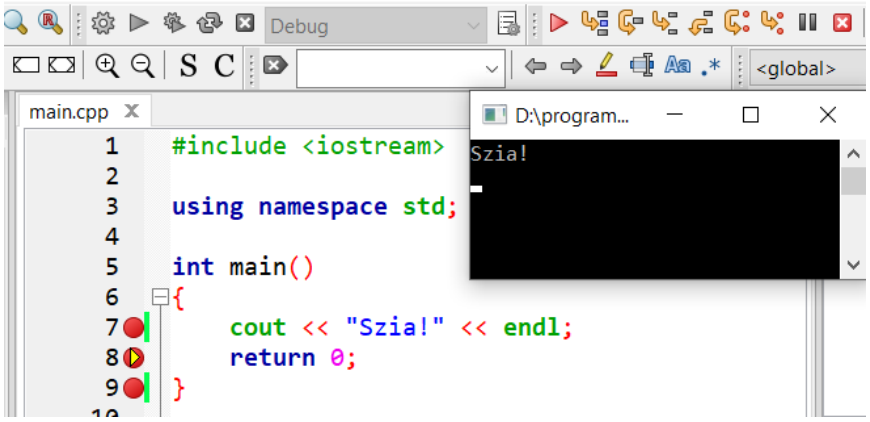
A gyakran hasznos segítség a kódkiegészítés. Egy kódrészlet első néhány karakterének a begépelése után az IDE felajánlja az általa ismert kiegészítéseket.



A C++ függvényeinek paraméterezését nehéz megjegyezni, de nem is szükséges, mert a nyitózárójel beírásakor megjegyezni az elvárt paraméterek.

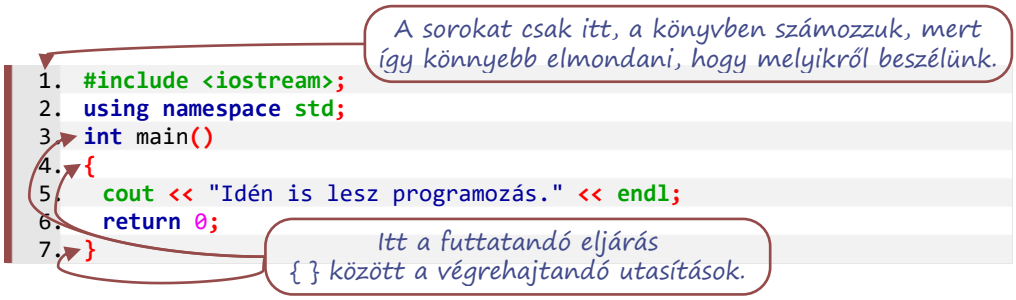


Attól, hogy a programunk futtatható, még nem biztos, hogy azt csinálja, amit szeretnénk. Ilyenkor szemantikai hiba van a programban, a kóddal nem a szándékunknak megfelelő utasítást adtuk meg. A programunk többszöri futtatásával teszteljük a munkánkat, a kapott eredményből következtethetünk a félreértés, a félreértelmezés okára. Ha nem látjuk, hogy mi okozza a félreértést, akkor **Debug** módban, akár lépésenként futtathatjuk a programunkat, lassítva figyelhetjük meg a működését. A ▶ ikonra kattintva vagy F8 billentyű leütésekor a gdb.exe értelmezi a programkódot, a fordítás részeredményét azonnal végrehajtja. Egy hosszú program lépésenkénti futtatása nagyon időigényes, de breakpointtal jelezhetjük, hogy melyik kódrészletnél szeretnénk megállítani a programot, elkezdeni a megfigyelést. A kódsor elé kattintva tehetjük ki a piros megállító pontokat, amelyeknél a futtatás során megáll a programunk.



A hibakeresés (debugolás) talán legnagyobb segítsége, hogy vizsgálhatjuk a program pillanatnyi állapotát, ehhez a **Debug \ Debugging windows \ Watches** menüponttal nyithatunk „betekintőt”, amelyben a megállított program pillanatnyi adatait, állapotát itt láthatjuk.

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 7–10 sor. Ezt teszteljük először.



A nagy programok kódja több fájlban található. A legegyszerűbb programunkban is használjuk az előre elkészített konzollal kapcsolattartó eszközöket, bevonjuk az `iostream` eljárásait, függvényeit. Ehhez hasonlóan, az `#include` utasítással tudunk a programhoz csatolni más, előre megírt eszközöket is. Az elnevezések összekeveredését akadályozza meg, a névtér megadása. A konzolra kimenet teljes neve `std::cout`, de ha a program elején megadjuk, hogy az `std`-t, a standard utasításkészletet használjuk, akkor nem szükséges minden esetben kiírni a névteret. Egy program nagyon sok egységből állhat, ezek kódolása történhet egy vagy több fájlba. A fordítóprogramnak – és a programozónak – tudnia kell, hogy hol van az elsőnek végrehajtandó feladat és hol keresse az ebben hivatkozott programrészleteket.

Ahhoz, hogy a kódunkat a fordítóprogram értelmezni tudja, a végrehajtás kezdőpontjának jól felismerhetőnek kell lennie. A fordítóprogram a `main()` függvénnyel kezdi a végrehajtást, minden adatra, objektumra, eljárásra, vagy függvényre hivatkozás értelmezését az adott hely előtt keresi. Ezért a kódfájl legelején kell feltüntetni (includolni) az összes felhasználandó előre megírt kód fájlt, ezt követi a fájlon belül megírt eljárásoknak, függvényeknek legalább a neve – deklarációja – majd a `main()` függvény. Az eljárások és függvények definíciója – a végrehajtandó kód – lehet a `main()` után, vagy a deklarációval egyben a `main()`, illetve az első használat előtt.

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 kapcsoszárójelekkel `{ }` 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, konzol)**

```
cout << "Idén is lesz programozás." << endl;
cout << "Hát nem remek? ;)" << endl;
system("PAUSE"); /*a program végén billentyűleütésre vár*/
```

Példánk első két utasításában a `<<` (**inserter**) operátor a konzolra kiíró csatornába betölti az idézőjelek közötti karaktereket majd egy sorvég jelet.

Már a legegyszerűbb programok is képesek lehetnek a felhasználóval való kommunikációra. A felhasználó válaszait (is) változókban tároljuk. A változó létrehozásakor megadjuk annak típusát és 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 memóriaterületet 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 változótípusból 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 leütése után egyetlen karaktersorozatként jelenik meg a program konzol bemenetén, a `cin`-en. A beírt adat eltárolásához az adatsorozatból a kívánt adattípust elő kell állítani. Ehhez a C++ speciális eszközét az **extractor**-t, azaz a `>>` operátort használjuk. Ez az operátor – művelet – a konzol (vagy más bemeneti) csatornából „kiveszi” konzol inputból (`cin`) az utána lévő változónak megfeleltethető karaktereket és átalakítja a változónak megfelelő típusra. Az extractor (`>>`) operátor olyan, mint egy kacsa. A „számára érdektelen”, nem nyomtatható karaktereket – például szóközt, tabulátort, bekezdés vége jelet – az értelmezhető karakterek elől, a `cin`-ből kiolvassa és eldobja, az adat utáni nem nyomtatható karaktereket otthagyja. Ezzel a módszerrel a beolvasás során mindegy, hogy az adatok között hány szóköz vagy `\n` `\r` található, mert ezeket mind „eldobja” az adat elől. Kihívást jelent viszont egy többszavas mondat beolvasása, mivel hiába `string`, (szöveg) a feltöltendő változó típusa, csak egy szót fog beletenni a `>>`.

```
string szo; cin >> szo;
int egeszszam; cin >> egeszszam;
double szam; cin >> szam;
```

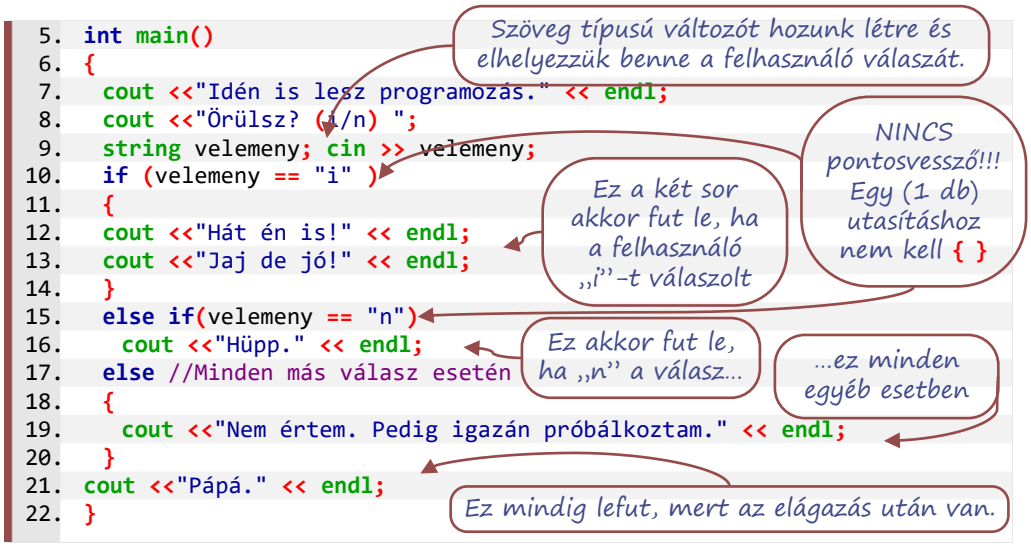
Hosszabb szöveg sorvégjelig olvasásához a `getline(cin, string_adat, határ_karakter)` függvényt használhatjuk. A függvény harmadik paramétere az adatok elválasztó karaktere, amennyiben ez az ENTER (`\n`), akkor elhagyható. Az elválasztó karaktert a `getline()` kiolvassa a `cin`-ből, de a változóban nem tárolja el, helyette a „szöveg vége” jelet, a `'\0'` nullabájtot írja be.

A kétféle adatbeolvasási módszer külön-külön jól használható, de kombinálásuk esetén nagyon figyelni kell arra, hogy a `cin >>` utáni első `getline(cin, …)` az utolsó beolvasott adattól a határolójelig – többnyire az ottmaradt `\n` karakterig olvassa ki a `cin`-ből a karaktereket.

```
6.  string szo; cin >> szo;
7.  string sorveg; getline(cin, sorveg);
8.  string sor; getline(cin, sor, '\n');
```

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.



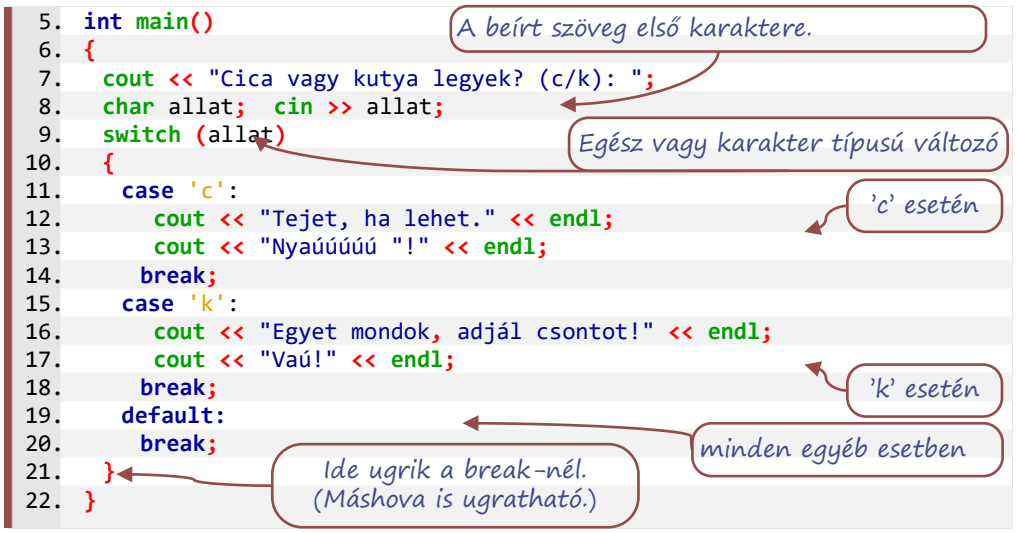

```
 5. int main()
 6. {
 7.   cout <<"Idén is lesz programozás." << endl;
 8.   cout <<"Örülsz? (i/n) ";
 9.   string velemeny; cin >> velemeny;
10.   if (velemeny == "i" )
11.   {
12.   cout <<"Hát én is!" << endl;
13.   cout <<"Jaj de jó!" << endl;
14.   }
15.   else if(velemeny == "n")
16.     cout <<"Hüpp." << endl;
17.   else //Minden más válasz esetén
18.   {
19.     cout <<"Nem értem. Pedig igazán próbálkoztam." << endl;
20.   }
21. cout <<"Pápá." << endl;
22. }
```

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 – 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 hamis-ágon belüli elágazást egyszerűsíti. Ha egy programozási nyelvben létezik **else if**, akkor tetszőleges számú ág létrehozható a használatával.

Ha az elágazás ágait egy változó felsorolható értékeitől függően kell kiválasztani, akkor használhatunk **switch**-et. Ez nemcsak 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.

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 csontra lesz szükség, a megnyilvánulás pedig „Vaú!”.




```
 5. int main()
 6. {
 7.   cout << "Cica vagy kutya legyek? (c/k): ";
 8.   char allat;  cin >> allat;
 9.   switch (allat)
10.   {
11.     case 'c':
12.       cout << "Tejet, ha lehet." << endl;
13.       cout << "Nyaúúúúú "!" << endl;
14.       break;
15.     case 'k':
16.       cout << "Egyet mondok, adjál csontot!" << endl;
17.       cout << "Vaú!" << endl;
18.       break;
19.     default:
20.       break;
21.   }
22. }
```

Ha valamilyen ismétlődő feladatot szeretnénk végeztetni a számítógéppel, ciklust szervezünk. Az első ciklusunk az elöltesztelő, 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.  cout << "Üdv én egér vagyok. Cincogjak neked? (i/n) ";
 9.  string valasz;  cin >> valasz;
10.  while (valasz == "i")
11.  {
12.    cout << " cin- cin \tMég? (i/n)";
13.    cin >> valasz;
14.  }
15.  cout << "Szia." << endl;
16.
```

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

Ha nem „i”-t válaszolunk, akkor a 10. után a 4. sorban nem teljesül a feltétel, onnan a 11. sorra ugrik.

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

Írjunk programot, amely addig kérdezgeti, hogy hány évig tartott a harmincéves háború, amíg a felhasználó ki nem találja! Ne felejtsük megfelelő típusúra alakítani a felhasználó válaszát! 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 valasz;
 9.  int proba = 0;
10.  do
11.  {
12.    if (proba >= 3)
13.      cout << "Súgok: 1618-tól 1648-ig tartott." << endl;
14.    cout << "Hány évig tartott a harmincéves háború? ";
15.    cin >> valasz;
16.    proba += 1;
17.    if (valasz > 30)
18.      cout << "Rövidebb volt." << endl;
19.    else if (valasz < 30)
20.      cout << "Hosszabb volt." << endl;
21.  } while (valasz != 30);
22.  cout << "Ügyes! Nem gondoltam volna..." << endl;
```

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.

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 sorszámukkal – tömbön belül elfoglalt helyükkel – adhatjuk meg. Az első sorszám a 0.

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

Van, aki jobban szereti a méretbéli szabadságot. A dinamikus tömb, a **vector<>** bővíthető adatsorozat, gyakran nevezik listának. Ennek használatához a program elején meg kell adni az adattípust leíró programkódot, a **<vector>**-t. 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.

```
2. #include <vector>
3. …
4. vector<típusnév> változónév = new vector<típusnév>();
5. változónév.push.back(adat);
```

Természetesen adódik, hogy egy tömb vagy lista szöveg típusú adatokat tároljon. Egy karakter elérése `változónév[adatsorszám][karaktersorszám]` 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, amelyet leggyakrabban tömbök tömbjeként adunk meg

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

A tömb, a **vector<>**, 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, a **vector<>** és a **string** közös jellemzője, hogy bármelyik adatuk bármikor elérhető és módosítható az indexen keresztül.

6. példa: Szökőévek 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 szökőéveket!

```
 6.  int ideiev = 2023;
 7.  for (int ev = 1848; ev <= ideiev; ev++)
 8.  {
 9.    if (ev % 4 == 0 && (ev % 100 != 0 || ev % 400 == 0))
10.      cout << ev + ".";
11.  }
```

Szöveg (string)

A leggyakrabban használt összetett adattípus a **string**, ami egy karaktersorozatot tartalmazó objektum. A karaktersorozat hosszát az objektum (pl. **string** s) `s.size()`, illetve `s.length()` függvénye adja meg. A karaktersorozatot az `s.c_str()` függvénnyel nyerhetjük ki az objektumból. Az egyes karakterek egyenként is olvashatók, módosíthatók (az objektum kiválasztott karaktere a benne lévő karaktersorozat megfelelő karaktere). A programunkban beállíthatjuk, hogy az ékezetes karaktereket – UTF-8 kódolást – is értelmezze:

```
    setlocale(LC_ALL, "HUN");
```

Két szöveg összehasonlítható a szokásos '==' és a '!=' relációkkal. Az ábécé szerinti viszonyuk jellemzéséhez használhatjuk a '<' '<=', '>=' és '>' relációkat, amelyekkel az első eltérő karakterpár kódja adja a vizsgálat logikai (igaz-e) eredményét. Az objektumok, így a stringek összehasonlítására is jellemzőbb az összehasonlító – `compare()` – függvény, ami az adott string objektumhoz viszonyítja a paraméterként megadott másik objektumot. Ennek eredménye -1, 0 vagy 1 lehet. Az alábbi két értékadás egyenértékű:

```
 6.  string s = "egyik"
 7.  bool a = (s.compare("masik") == -1);
 8.  bool b = (s < "masik");
```

Két szöveg, illetve szöveghez karakter összefűzhető a '+' jellel, továbbá használhatjuk a szokásos szövegfüggvényeket: `s.substring(int poz int db)`, `s.erase(int poz, int db)`, `s.insert(int poz, string ezt)`.

Fontos megjegyezni, hogy az ábécé szerinti összehasonlításban a karakterek kódja a meghatározó, nem a magyar nyelvtan szabályai, ezért például a 'z' után található az 'á'.

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

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

```
 6.  setlocale(LC_ALL, "HUN");
 7.  cout << "Írd be a megfordítandó szót: ";
 8.  string szo;  cin >> szo;
 9.  string vissza = "";
10.  for (int i = 0; i < szo.size(); i++)
11.    vissza = szo[i] + vissza;
12.  cout << "A szó megfordítva: " + vissza;
```

Számláló (iterátor) kezdőértékkel

Indexével kiválasztott karakter

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

```
13.  char betuk[50];
14.  for (int v = szo.length() - 1; v >= 0; v--)
15.  {
16.    int e = szo.length() - 1 - v;
17.    betuk[e] = szo[v];
18.  }
19.  betuk[szo.length()] = '\0';
20.  for (int i = 0; betuk[i] != '\0'; i++)
21.    cout << betuk[i]);
```

"Elég nagy" karakterek tömb létrehozása

Visszafelé számlálás

{ }, ha a ciklusmagban több utasítás van.

Tömb (Array): azonos típusú adatok tárhelye

A fenti megoldásnál van egyszerűbb is, de ezen érdemes megfigyelni három dolgot:

1. A tömb mérete konstans, a program létrejöttének idején tudni akarja a fordító program, hogy mekkora területet készítsen elő.
2. A 0-tól kezdődő számozás miatt az utolsó karakter pozíciója a `szo.length() - 1`. Erre figyelni kell akkor is, amikor visszafelé haladva beállítjuk a kezdőértéket és akkor is, amikor kiszámítjuk a másik irányból szükséges pozíciót.
3. A tömb nagyobb, mint a `szo`, tudnunk kell, hogy a megfordított szónak hol a vége. Erre egyik megoldás, hogy – akár külön változóban tárolva – a `szo` hosszát használjuk. Másik gyakori, itt is alkalmazott megoldás, a végjel használata. A karaktersorozatok végét egy `0` értékű bájttal jelezzük, amit az olvashatóság kedvéért `'\0'` karakterként adunk meg. A karaktersorozattal megadott szöveg érdemi hossza egyenlő a végjel pozíciójával, a szöveg fizikai karaktereinek száma a hosszánál 1-gyel nagyobb.

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

Tároljuk el egy vizsgára felkészülés próbadolgozatainak pontszámait! A pontszámokat egy sorban, szóközökkel 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.  int main()
 7.  {
 8.    string adatsor = "9 19 57 73 100"; //futtatáskor megadva: getline()
 9.    string adatok[10]; //legyen kellően sok hely
10.    int szodb = 0;
11.    adatok[szodb]="";
12.    for(unsigned i = 0; i < adatsor.length(); i++) {
13.      if(adatsor[i] !=' ')
14.        adatok[szodb] += adatsor[i];
15.      else {          /*ha szóköz, akkor következő adatot kezdi írni*/
16.        szodb++;
17.        adatok[szodb] = "";
18.      }
19.    }
20.  szodb++; /*az utolso szó után nincs szóköz*/
21.  /*szövegek összehasonlítása*/
22.    for (int i = 1; i < szodb; i++)
23.      if (adatok[i] > adatok[i - 1])  cout << '+';
24.      else  cout << '-' //;
25.     cout << endl;
```

Ez a megoldás azért nem jó, mert string adatként például a 9-nél kisebb a 19. A helyes megoldáshoz az adatokat át kell konvertálni számmá.

```
26.  /*egésszé alakítás, számok összehasonlítása*/
27.    int pontszamok[10];
28.    for (int i = 0; i < szodb; i++)
29.      pontszamok[i] = stoi(adatok[i]);
30.    for (int i = 1; i < szodb; i++)
31.      if (pontszamok[i] > pontszamok[i - 1])  cout << '+';
32.      else  cout << '-';
33.    cout << endl;
34.  }
```

Így már jó az eredmény, de elgondolkodtató, hogy több, mint 20 sor a megoldás és talán eddig soha nem volt rá szükség. Ha a programkódban adjuk meg az adatot, akkor egész számokat adnánk meg az adatsor helyett. Ha konzolról várjuk az adatokat, akkor nem a `getline()`-t használnánk, hanem az extractort (`>>`). De azért előfordulhat, hogy hosszabb szöveget, mondatot kellene feldarabolni, ezért a C++ nyelvben van egy olyan szövegtároló, ami a konzolhoz hasonlít, de a konzol helyett a kimenete „visszakanyarodik” a programba. A **`cin`** és **`cout`** az **`iostream`** két csatornája. Egy stringstream típusú adat egy olyan csatorna, amibe az inserterrel (`<<`) betöltjük az adatot, az extractorral (`>>`) pedig a „másik végén” kivesszük. Használatához be kell tölteni a **`<sstream>`** csomagot.

```
1.  #include <iostream>
2.  #include <sstream>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.    string adatsor = "9 19 57 73 100";  //futtatáskor megadva: getline()
9.    int pontok[10];       //legyen kellően sok hely
10.   stringstream sstr;
11.   sstr << adatsor; //inserterrel betesszük a stringet a csatornába
12.   int szodb = 0;
13.   while (sstr >> pontok[szodb]) //extractor kivesz és egésszé alakít
14.     szodb++;                    //ha sikeres, növeli a db-t
```

A ciklusnak akkor van vége, amikor a `sstr`-ből az extractor nem tud újabb számot kivenni, ezért a `szodb` pontosan a kivett számok darabszáma.

Egy ciklus feltétele általában nem lehet utasítás, csak logikai kifejezés. C++ nyelvi specialitás, hogy egy hibát adó utasítás **`false`** logikai értéket jelent a ciklusfejben. Hasznos lehet tudni, hogy ha konzolról olvas be a program adatot, akkor nem lesz vége, hanem várja a következő adatot. A befejezést CTRL+D billentyűkombinációval jelezhetjük.

A folytatás azonos a korábbi megoldásokkal.

```
15.   for (int i = 1; i < szodb; i++)
16.     if (pont[i] > pont[i - 1])  cout << '+';
17.     else  cout << '-';
18.   cout << endl;
19. }
```

Lista (vector<>)

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

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

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  setlocale(LC_ALL, "Hun");
  string dolgozat = "5543521H3545h555-4H5";
  cout << "Csoportlétszám: " << dolgozat.length() << " fő" << endl;
  vector<int> jegyek;
  for (char c : dolgozat)
  {
    if (c >= '1' && c <= '5')
    {
      int jegy = c - '0';
      jegyek.push_back(jegy);
    }
  }
  for (int jegy : jegyek)
     cout << jegy << " " << endl;
  cout << endl;
  cout << "A dolgozatot megírta: " << jegyek.size() << " fő";
}
```

A vector<> kódcsomagja kell.

Első alkalommal az értéke ’5’

karakter kódok különbsége

A jegyek lista végéhez hozzáfűzi’

string hossz length vagy size

vector hossza csak size

A C++ nyelvben a `for (){}` vezérlési szerkezetnek kétféle alkalmazása van. Az ismertebb változatában megadhatjuk a ciklusváltozó kezdőértékét, a ciklusba belépés feltételét és a léptetés módját. Ez használható bármilyen adatsorozat elemeinek közvetlen eléréséhez az elem indexén keresztül. A másik változat a felsorolható elemek bejárását teszi lehetővé, a lista minden elemét sorba veszi. Ezt szokták – más nyelvekben használatos kifejezést átvéve – foreach-ciklusnak nevezni.

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 (konténer – container – eszközöket), derítsük ki, hogyan használhatók!

10. példa: A három kismalac háza

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

```
string malacok [3][2] {{ "Töfi", "szalma" },
                       { "Röfi",   "fa"   },
                       { "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!

```
 8.  for (int m = 0; m < 3; m++)
 9.  {
10.    for (int i = 0; i < 2; i++)
11.    {
12.      cout << malacok[m][i] << " ";
13.      if (i == 1 && malacok[m][i] == "kő")
14.        cout << "A farkas nem tudja bántani " << malacok[m][0] << " t.";
15.    }
16.    cout << endl;
17.  }
```

Megoldható a feladat listába tett listával és foreach-ciklus használatával is!

```
 5.  vector<vector<string>> malacok{      { "Töfi", "szalma" },
 6.                                       { "Röfi", "fa" },
 7.                                       { "Döfi", "kő" }
 8.                                   };
```

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.

```
 9.  for (vector<string> malac : malacok)
10.  {
11.    for (string adat : malac)
12.    {
13.      cout << adat << " ";
14.      if (adat == "kő")
15.        cout << "A farkas nem tudja bántani " << malac[0] << " t.";
16.    }
17.    cout << endl;
18.  }
```

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, de 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 adatleíró (pontosabban objektumot leíró) eszköze az osztály (**`class`**), amiről kiegészítő ismeretként volt már volt 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 egy-egy malac első adata `malac[0]`, ehelyett sorrendiségtől függetlenül, `malac.Nev` – 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:

```
 5.  struct Malac
 6.  {
 7.    string Nev;
 8.    string Hazanyag;
 9.  };
10.  int main()
11.  {
12.    setlocale(LC_ALL, "Hun");
13.    Malac malacok[3];
14.    malacok[0].Nev = "Töfi";
15.    malacok[0].Hazanyag = "szalma";
16.    malacok[1] = Malac{"Röfi", "fa" };
17.    malacok[2] = Malac{"Döfi", "kő" };
18.  }
```

A `malacok[0]` két adata egymás alatt szerepel, a `malacok[1]` és `malacok[2]` adatainak beírásakor viszont más módszer látható. A kapcsoszárójelben megadjuk az inicializáló adatokat, amiből a sorrendnek megfelelően kapnak értéket az egyes adattagok.

A sorrend betartása nem mindig könnyű, ezért érdemes konstruktort, azaz létrehozási útmutatót írni. Ebben tetszőleges módon megadhatjuk a felhasználandó adatokat, majd megírjuk hogyan értelmezze a programunk ezeket az egyes tulajdonságokra.

A konstruktor megírásával előírjuk, hogy a tulajdonságoknak a létrehozás pillanatában legyen értéke, ezért nem tud létrejönni olyan adat, amiben nincs megadva a kezdőérték. Ennek kivédésére a konstruktor paramétereinek célszerű megadni egy, a hiánya esetén használandó értéket.

```
 5.  struct Malac
 6.  {
 7.    string nev;
 8.    string hazanyag;
 9.    Malac(string anyag = "", string neve = "")
10.    {
11.      nev = neve;
12.      hazanyag = anyag;
13.    }
14.  };
15.
16.  int main()
17.  {
18.    setlocale(LC_ALL, "Hun");
19.    Malac malacok [3];
20.    malacok[0] = Malac("szalma", "Töfi");
21.    malacok[1] = Malac("fa", "Röfi");
22.    malacok[2] = Malac("kő", "Döfi");
```

Adattagok (publikusak)

Ha nincs megadva, akkor ez!

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

Tömb konstruktor

Malac konstruktorok

```
23.  for (int ez = 0; ez < 3; ez++)
24.  {
25.    cout << malacok[ez].nev << " " << malacok[ez].hazanyag << " ";
26.     if (malacok[ez].hazanyag == "kő")
27.       cout << "A farkas nem tudja bántani " << malacok[ez].nev << "t";
28.     cout << endl;
29.   }
```

PONT

A `malacok[ez]` egy malac. Ha listaelem lenne, akkor a foreach-ciklussal írnánk, akkor `malac.nev` szerepelhetne a `malacok[ez].nev` helyett.

Kiegészítés Python-utánzóknak, Pythonról áttérőknek (map)

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++ `map<,>` használatával a „listában szótárak" megoldás:

A programfájlhoz hozzáadjuk a `<map>` (szótár) csomagot és a `<vector>` (lista) csomagot is:

```
2.  #include <map>
3.  #include <vector>
```

Célszerű először a magyar ékezetes karakterek használatát beállítani, majd …

```
5.  vector<map<string,string>> dicmalac{
6.    {{"név","Töfi" },{"ház","szalma" }},
7.    {{"név","Röfi" },{"ház","fa" }},
8.    {{"név","Döfi" },{"ház","kő" }}
9.  };
10. for (map<string,string> malac : dicmalac)
11. {
12.   cout << malac["név"] << " házának anyaga: " << malac["ház"] << ".";
13.   if (malac["ház"] == "kő")
14.     cout << " A farkas nem tudja bántani " << malac["név"] << "t.";
15.   cout << endl;
16. }
```

A fenti példában 3 map objektumunk van, mindháromnak 2-2 adata, minden adat egy pár: van Key (`first`) és Value (`second`) értéke. A Key egyedi, a Value értékek lehetnek azonosak. (Példánkban a „ház" nem lehet „név", de a malac neve is lehet „szalma". Az egyetlen megszorítás a Python szótárához képest, hogy a Key és a Value típusa kötött, a map létrehozásakor meg kell adni.

A `map<,>` a tömbhöz és `vector<>`-hoz hasonlóan bejárható objektum. csak egy „picit" másképp. Lássuk erre a tizenegyedikes tankönyv példájának C++ nyelvű megoldását:

```
5.  #include <iostream>
6.  #include <map>
7.  using namespace std;
8.  int main()
9.  {
10.   setlocale(LC_ALL, "Hun");
11.   map<string, string> ember;
12.   ember["név"] = "Debora";
13.   ember["magasság"] = "167";
14.   ember["IQ"] = "131";
```

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

Bejárhatók a szótár elemei foreach-ciklussal. Egy-egy szótárbejegyzés két adatból áll, két tulajdonsága van: a `first` a kulcs, a `second` az érték.

```
15.   for (pair<string, string> adat : ember)
16.     cout << adat.first << " értéke: " << adat.second << "; ";
17.   cout << endl;
```

Bejárható a szótár az egyes bejegyzésekre rámutatással. Ez nem a tömbnél vagy a vector-nál használható index (sorszám), hanem egy mutató – hivatalos nevén pointer – ami az elem címét tartalmazza. Nem a pointernek van kulcs és érték adata, ezért a **pont operátor** (`.`) itt nem megfelelő. A pointerrel hivatkozott címen lévő adatrészt a **nyíl operátor**ral (`->`) operátorral érhetjük el.

```
18.   for (map<string,string>::iterator it = ember.begin();
                                          it != ember.end(); it++)
19.     cout << it->first << " értéke: " << it->second << ", ";
20.   cout << endl;
```

Bármelyik módot használjuk, a szintaktika bonyolult, az eredmény meglepő: A map-et nem a beírás sorrendjében kapjuk vissza a bejáráskor, hanem az első adat szerint rendezve. A map elemei a két adat mellett mutatókat– balgyerek, jobbgyerek, szülő – is tartalmaznak, amelyekkel a kulcs alapján bináris rendezőfát képez. A bejáráskor a bináris fa elemein megy sorba (egyik ág – adat – másik ág sorrendben). A bejárás nem egyszerű léptetés a memóriában, hanem „ugrás a mutatottra”. Ha egyedi sorrendet szeretnénk megadni, akkor célszerű a kulcsokat felvenni egy sorozatba, például tömbbe. Ezeket egymás után megadva, a mutatók mentén kikeresett kulcshoz tartozó értékétet ki tudjuk írni:

```
21.   string keys[3] {"név","IQ","magasság"};
22.   for (unsigned i = 0; i < 3; i++)
23.     cout << keys[i] << " értéke: " << ember[keys[i]] << " * ";
24.   cout << endl;
25. }
```

12. példa: Kedvenceink (vector<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!

```
 5. struct Ember
 6. {
 7.   string nev;
 8.   string telj;
 9.   int ev;
10. };
```

```
11.  int main()
12.  {
13.    vector<Ember> nagymesterek {
14.      {"Polgár Judit", "Garri Kaszparov legyőzése", 2002},
15.      {"Kempelen törökje", "Napóleon legyőzése sakkban", 1809},
16.      {"Beth Harmon","TV-sorozattá lett az \"élete\"", 2020} };
17.    for (Ember ember : nagymesterek)
18.      cout <<ember.nev<<": "<<ember.telj<<" ( "<< ember.ev <<");"<< endl;
19.  }
```

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 helyettesítsük a fenti mintaprogram 17–18. sorát egy kiírást végző eljárással

Kész kódból eljárás, függvény készítése: Refactoring

Az áttekinthetőség érdekében most a `main()` függvény előtt csak deklaráljuk az eljárást, utána írjuk meg az eljárás definícióját. Egyúttal az új emberek felvételéhez is deklarálunk egy függvényt.

```
10.  …}; /*struct vége*/
11.    /****** fgv. deklarációk *******/
12.  void nagymestereket_listaz(vector<Ember>);
13.  Ember adatokat_beker();
14.    /*a végrehajtás belépési pontja*/
15.  int main()
16.  {
17.    setlocale(LC_ALL, "Hun");
18.    vector<Ember> nagymesterek {
19.      {"Polgár Judit", "Garri Kaszparov legyőzése", 2002},
20.      {"Kempelen törökje", "Napóleon legyőzése sakkban", 1809},
21.      {"Beth Harmon","TV-sorozattá lett az \"élete\"", 2020} };
22.    cout << "A nagymesterek listája most:" << endl;
23.    nagymestereket_listaz(nagymesterek);       /*fgv. felhasználása*/
24.  }
```

Ha a program elején deklaráljuk a függvényt, a fordító program keresi a definíciót.

```
39.    /*fgv. definíciók a main után*/
40.  void nagymestereket_listaz(vector<Ember> nm)
41.  {
42.    for (Ember ember : nm)
43.      cout <<ember.nev<<": "<<ember.telj<<" ( "<< ember.ev <<");"<< endl;
44.  }
```

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**.

Az adatok bekérésekor figyelni kell arra, hogy a név és a teljesítmény is több szóból áll. Ezért az első két adat beolvasásához sor-olvasást, `getline()`-t érdemes használni. A harmadik adat évszám, amihez az extractor (`>>`) célszerű, de ez „bent hagyja a csatornában” a bevitelt lezáró ENTER-t, ezért ezt utólag ki kell olvasni.

```
45.  Ember adatokat_beker()
46.  {
47.    setlocale(LC_ALL, "Hun");
48.    cout << "Mi a neve? ";
49.    string nev; getline(cin, nev); //több szó is lehet
50.    cout << "Mi a legjelentősebb teljesítménye? ";
51.    string kunszt; getline(cin, kunszt);
52.    cout << "Melyik évben érte ezt el? ";
53.    int ev; cin >> ev;
54.    string qka; getline(cin, qka); //sorvégét kiolvasni, eldobni
55.    return Ember{nev, kunszt, ev}; //kapcsos zárójellel létrehozás
56.  }
```

A `nagymestereket_listaz()` eljárást egy kész kódrészlet áthelyezésével hoztuk létre, átszerveztük refaktorizáltuk a kódot. Az `adatokat_beker()` függvénnyel továbbfejlesztjük a programunkat. A függvény definiálása után `main()` függvényt kiegészítjük a lista bővítésének lehetőségével.

A kiegészítésben – most hátul-tesztelő ciklust használva – a felhasználó „n” válaszáig egy-egy adat bekérése és az aktuális lista kiírása szerepel. Mivel az `adatokat_beker()` függvényben `getline()`-t használtunk, ez kihat erre a programrészre is.

> Bár csak egy betűt várunk, ha az extractorral (`>>`) olvassuk be, akkor ott marad az enter, amit a következő, nevet bekérő `getline()` fog kiolvasni. További hatásként tulajdonság lesz a beírt névből és a tulajdonság első szavát próbálja számmá alakítani a `>>`.
> Ebbe „belebukik”, így a hibás adat érzékelése miatt „bezárja a cin-t” – a `cin.fail()` értéke igaz lesz –. További olvasás csak a `cin.clear()` kiadása után lenne lehetséges.
> Ezért – tekintettel arra, hogy az `adatokat_beker()` függvényben `getline()`-t kell használni – a főprogramban is `getline()`-t használunk.

```
25.   string bovitjuk;
26.   do
27.   {
28.     cout << "Akarsz új nagymestert megadni? (i/n) ";
29.     getline(cin, bovitjuk); //getline lesz utána is, ne maradjon semmi
30.     if (bovitjuk != "n")
31.     {
32.       nagymesterek.push_back(adatokat_beker());
33.       cout << "A nagymesterek listája ilyen lett:" << endl;
34.       nagymestereket_listaz(nagymesterek);
35.     }
36.   } while (bovitjuk != "n");
37. }
```

Változók láthatósága függvényen belül és kívül

Az első szabály, hogy egy változó azon a blokkon belül látható – és használható – amin belül megadtuk a típusát és nevét. A blokk határai a kapcsos-zárójelek, illetve a vezérlési struktúra esetén a struktúra határai. Például blokk kezdete lehet „for (”, a vége pedig az egyetlen utasítást tartalmazó ciklusmag végét jelző pontosvessző.

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

Eljáráson, függvényen kívül – de jelen tudásunk mellett a fájlon belül – elnevezett változót minden eljárásban és függvényben tudunk használni, a program szempontjából globális változók. A változó deklarálása meg kell előzze a felhasználás helyét. A mennyiben nem adunk azonnal kezdőértéket, 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: Globális változók használata

```
5.  const int MaxN=1000;
6.  int tomb [MaxN];
7.  int N;
8.  double dt[1000];
```

A tömb méretét itt meg kell adni, később, a program futása közben nem módosítható. Ezért itt is csak szám vagy konstans érték lehet a méret. A konstans használata több, azonos méretű tömb esetén praktikus, mert ezzel szükség esetén egy helyen elegendő módosítani a kódot. A deklarált változók – itt 2001 db – alapértelmezett értéket kapnak. Számok esetén `0`, `0.0`, szöveg esetén `""` lesz az érték.

```
9.  void masikeljaras()    /*nincs paramétere*/
10. {
11.   dt[N]= 2;       /*Mennyi N értéke? */
12.   N= 1004;
13. }
```

A `masikeljaras()` a globális változók értékét módosítja. A hívás (felhasználás) helyén érvényesül a tevékenysége, de a kódban a hívás előtt kell deklarálni. Mos a definíció is a hívás előtt van.

```
14.  int main()
15.  {
16.    cout << "tomb[0]: " << tomb[0] << endl; /*default érték: 0*/
17.    tomb[0] = 3;       /*módosítás*/
18.    N = tomb[0] + 1;   /*eddig memóriaszemét, ezután: 4*/
19.    cout << "N: " << N << endl; /*itt már értelmes*/
20.    cout << "dt[4]: " << dt[4] << endl; /*default érték: 0 */
21.    /*N: 4*/ masikeljaras();
22.    cout << "dt[4]: " << dt[4] << endl; /*eljarasban kapott érték*/
23.    cout << "dt[" << N << "]: " << dt[N] << endl; /*N: 1004 módosult*/
24.    cout << dt[N] << endl; /*dt[1004]: memóriaszemét, túlindexelés*/
25.  }
```

A C++ (egyszerű) tömbjének nincs méret vagy hossz tulajdonsága. A kezdetben megadott mérettel biztosítjuk az adataink számára a helyet, de nem szabunk határt az hivatkozásnak. Inkább a szükségesnél nagyobb méretet foglaljunk le globálisan és jegyezzük fel – szintén globális változóban – a felhasznált méretet, a tárolt adatok aktuális számát.

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

14. példa: Függvény paraméterezése, lokális változók átadása

Példaként hozzunk létre egy számot, egy rekordot (**struct**), amiben van szöveg és szám is, valamint egy kételemű egészekből álló tömböt. Figyeljük meg, hogyan használhatók ezek az adatok más függvényekben.

A sorok sávozása helyett most a fehér hátterű blokk a `main()` részeit mutatja, a szürke hátterű blokkokban az előtte definiált kódrészletek lesznek olvashatók.

A rekord típus definíciója után rögtön érdemes megírni a változók kiírásának eljárását. Ehhez a változókat tetszőleges módon adhatjuk át, mivel a kiírás során már nem kívánunk változtatni az értékükön.

```
 5.  struct R {string J; int K;};
 6.  void kiir(int n, R r, int t[], string s, vector<int> v)
 7.  {
 8.    cout << n << " " << r.J << " " << r.K << " " << t[0] << " " << t[1];
 9.    cout << " " << s << " " << v[0] << " " << v[1] << endl;
10.  }
```

A főprogram a kódunk legvégén található:

```
40.  int main()
41.  {
42.    int n = 3;
43.    R r {"r", 0};
44.    int t[2] {5, 4};
45.    string s = "szoveg";
46.    vector<int> v {1,2};
47.    int x = 0; //függvény visszaadott értékéhez
48.    cout << x << '\t'; kiir(n, r, t, s, v); //0 3 r 0 5 4 szoveg 1 2
```

- Ha csak olvasni akarjuk a változó értékét, akkor elegendő a típust és az argumentum nevét megadni.

- Ha a függvényen belül módosítjuk az adatot, akkor már jobban oda kell figyelni. A paraméterek helyén változóink másolata lesz az argumentum, kivéve a tömböt, amelyiknél a tömb elejére mutató hivatkozás másolata lesz az argumentum, emiatt az eredeti adatokat használjuk.

```
12.  int felhasznal(int adat, R rek, int tomb[], string s, vector<int> v)
13.  {/*a tomb hivatkozás (méretét nem tudjuk)*/
14.    tomb[0] += tomb[1];
15.    rek.K--;
16.    adat += tomb[0] + rek.K;
17.    s = "duma";
18.    v[1] *= 10;
19.    rek.J = s;
20.    return adat;
21.  }
```

A felhasználást követően a változók tartalma:

```
49.  int x = felhasznal(n, r, t, s, v);
50.  cout << x << '\t'; kiir(n, r, t, s, v); //11 3 r 0 9 4 szoveg 1 2
51.  t[0] = 5; //Visszaállítás mert t[0] módosult, a többi nem.
52.
```

Az x értéke módosult, az eredmény azt mutatja, hogy a függvényen belül az adatok (másolatának) értéke megfelelően változott. A kiírt eredményekből látszik, hogy csak a `t[0]` értékének változása maradandó.

- Ha módosítani akarjuk az átadott (nem tömbelem) változót, akkor a memória címét, azaz az adat referenciáját kell átadni. Ezt a típus után írt **&** referencia-jellel jelezzük a függvény deklarációban és definícióban. 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.

```
22.  int modosit(int& adat, R& rek, int tomb[], string& s, vector<int>& v)
23.  {          /*az "int& tomb[]" vagy "int tomb[]&" értelmetlen)*/
24.    tomb[0] += tomb[1];
25.    rek.K--;
26.    adat += tomb[0] + rek.K;
27.    s = "duma";
28.    v[1] *= 10;
29.    rek.J = s;
30.    return adat;
31.  }
```

A felhasználást követően a változók tartalma:

```
53.  int y = modosit(n, r, t, s, v);
54.  cout << y << '\t'; kiir(n, r, t, s, v); //11 11 duma -1 9 4 duma 1 20
55.
```

Természetesen nem követelmény, hogy a hivatkozással megadott paraméter értéke a függvényen belül módosuljon. Gyakran nem is a módosíthatóság miatt használják a programozók a referenciaként történő paraméterátadást, hanem a hatékonyság növelése érdekében, mivel

a referenciával megspórolható egy adat másolása, ami időt és memóriát is igényel. Ökölszabályként mondhatjuk, hogy ha egy adat memóriamérete nagyobb, mint a rá mutató pointer (hivatkozás) mérete, akkor hatékonyabb a referenciáját használni paraméterként.

Másik szempont a kód biztonságos használata. Ha referenciájával adunk át egy adatot, akkor az módosulhat. A függvényt meghívó oldaláról láthatatlan a függvény belső működése. Előfordulhat, hogy a függvényen belül másik függvényt is meghívnak … és valahol, a sokadik hívásban a referenciaként átadott adat megváltozik. Ki fogja végig ellenőrizni az összes függvénydeklarációt? Senki. A fordító program, de az nem ismeri a programozó elképzeléseit, szándékát. Jeleznünk kell, hogy a hivatkozott adatot ne lehessen módosítani. Ezt írhatjuk elő a `const` kulcsszóval:

```
33.  void ki (int const& n, R const & r, const int t[],
                const string& s, const vector<int> & v)
34.  {
35.  cout << n << " " << r.J << " " << r.K << " " << t[0] << " " << t[1];
36.  cout << " " << s << " " << v[0] << " " << v[1] << endl;
37.  }
```

Látható, hogy a `const` és típus sorrendje felcserélhető, a `&` bármelyikkel egybeírható, de külön is írható és `const` megelőzi az `&`-et.

Globális vagy lokális legyen a változó?

A fentiek alapján, legegyszerűbb minden változót a program elején globálisan létrehozni. Mégsem lenne jó ez a megoldás, mert a megértést, olvashatóságot az segíti, ha a változókat az első használathoz közel hozzuk létre. Ezért azokat a változókat, amiket egy függvényen belül használunk, biztosan lokálisan célszerű létrehozni.

Mérlegelést igényel a több függvényben használt változó, mert ezeket globálisan létrehozva nem kell foglalkozni a függvény paraméterezésével. Azonban a függvényeket sokszor azért írjuk, mert ugyanazt a kódot akarjuk többször, más-más adatokkal használni. Például, ha a kedvenceinket nem egy listában tároljuk, hanem külön listákat készítünk a sportolóknak, tudósoknak, művészeknek … akkor célszerű a listát paraméterként megadni. Ha a programunk egyetlen adatsorozat kezeléséről, az ezzel kapcsolatos kérdések megválaszolásáról szól, akkor ezt az egyetlen adatsorozatot (és hozzá tartozó jellemzőket) globálisan célszerű megadni.

A globális változó létrehozásánál szempont lehet, hogy a program a globális adat statikusan (a program elejétől a végéig) biztosan a memóriában lesz, amiből „korlátlan” méretet le lehet foglalni. Egy függvényen belül létrehozott tömb csak a függvényhívás idejére foglalja a memóriát, ami látszólag takarékosabb megoldás, de többlet feladatot jelent a változó memóriaméret kezelése, esetleg „menet közben” túlzott megnövekedése, ezért erre a fejlesztőkörnyezetekben méretkorlátozás lehet.

Vittük 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 típusalgoritmusokra van függvény is, ezek

leginkább az <algorithm> csomagban érhetők el. Ha nem lennének, akkor a programozók saját kód-tárában lennének elmentve a gyakori feladatokra a megoldások, saját készítésű csomagokat adnának a programjukhoz. Jó, hogy nem kell a részletekbe belemenni, de a „mindenre is jó” megoldás indirekt, pointerekkel bűvészkedik. Megértés nélkül nehéz jól használni a pont (.), a csillag (*) és a nyíl (->) operátort és az ezeket használó függvényeket. Ezért nem a haladó programozási nyelvismeret felé megyünk el, hanem az egyes feladattípusok elemi eszközökkel történő megoldásának elvét tanulmányozzuk, hogy mindig az épp felmerülő problémának megfelelően tudjuk alkalmazni, kódolni a típusalgoritmusokat. A nagyon érdeklődők ennek a fejezetnek a végén tanulmányozhatják az egyes feladatok „rövid” megoldását.

A tankönyvtől eltérően, a megoldásokat csak tömb típusra adjuk meg, a vector<> használatát ismerők számára az átírás nem jelenthet gondot. Az egyes algoritmusokra külön függvényeket írunk, aminek bemeneti paraméterként adjuk át a feldolgozandó tömböt, függetlenítve a konkrét feladat adataitól. Az általunk írt függvények a main() függvény előtt lesznek, de ezen belül a sorrendjük tetszőleges, ezért a sorok számozása csak az adott blokkon belül releváns.

A tanult típusalgoritmusok többségében az elágazás és ciklus magja csak egy utasítást tartalmaz. A ciklusmagot a kapcsos-zárójelek közé kell írni, de ez a zárójelpár egy (1 db) utasítás esetén felesleges, elhagyható. 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 érdemes megtartani a zárójeleket, mert egy későbbi továbbfejlesztés ebből az állapotból könnyebb.

Sorozatszámítás

A sorozatszámítás algoritmusa arra jó, hogy egy bejárható objektum elemeit összeadjuk, összeszorozzuk, egymásután fűzzük … azaz egy eredményértéket képezzünk. Például:

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.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    cout << "Összesen: " << sorozatszamitas(honapnapok) << " " << endl;
5.  }
```

A sorozatszámítás függvénye:

```
1.  int sorozatszamitas(int tomb[])
2.  {
3.    int szum = 0; /*műveletfüggő kezdőérték*/
4.    for (int i = 0; i < 12; i++)
5.      szum += tomb[i]; /*kumuláció, halmozás művelete*/
6.    return szum;
7.  }
```

Eldöntés

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

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

```
1.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    if (van28(honapnapok))
5.      cout << "Van 28 napos hónap." << endl;
6.    else
7.      cout << "Nem találtunk 28 napos hónapot." << endl;
8.  }
```

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.  bool van28(int tomb[])        /*1. megoldás*/
2.  {
3.    bool van_28 = false;
4.    for (int i = 0; i < 12; 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.  bool van28(int tomb[])        /*2. megoldás*/
2.  {                                      /*Lehetne:          */
3.    bool van_28 = false;                 /*int db = 0        */
4.    for (int i = 0; i < 12; i++)
5.      if(tomb[i] == 28)
6.        van28 = 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 lépjünk be, ha a van28 értéke `false` (azaz a tagadása igaz).

```
1.  bool van28(int[] tomb)        /*3. megoldás*/
2.  {
3.    bool van_28 = false;
4.    for (int i = 0; i < 12 && !van_28; i++)
5.      if(tomb[i] == 28)
6.        van28 = 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 *olvashatóság*án 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.  bool van28(int tomb[])          /*4. megoldás*/
2.  {                               !(tomb[i] == 28) másképp: tomb[i] != 28
3.    int i;
4.    for (i = 0; i < 12 && !(tomb[i] == 28); i++)
5.    {}
6.    return i < 12;
7.  }
```

Szövegesen: Úgy döntöm el, hogy van-e 28 napos hónap, hogy elindulok a tömb elejéről 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_hossza az első. A programozásban nem kommutatívak 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.  bool van28(int tomb[])          /*5. megoldás*/
2.  {                               (1) létezik, (2) ÉS, (3) NEM, (4) jó
3.    int i = 0;
4.    while (i < 12 && !(tomb[i] == 28))
5.      i++;                        kérem a következőt (mert ez nem jó)
6.    return i < 12;
7.  }                               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.  bool van28(int tomb[])          /*6. megoldás*/
2.  {
3.    int i;
4.    for (i = 0; i < 12; i++)
5.      if(tomb[i] == 28)           Nem a 4. sorra ugrik ellenőrzésre,
6.        break;                    hanem a 7. sorra, pánikszerűen.
7.    return i < 12;
8.  }
```

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.  bool van28(int tomb[])        /*7. megoldás*/
2.  {
3.    for (int i = 0; i < 12; i++)
4.      if(tomb[i] == 28)
5.        return true;
6.    return false;
7.  }
```

Bár ez a megoldás sem strukturált, de egyértelmű, hogy a **return** nem csak az adott ciklusból lép ki, hanem a függvényből is, eközben „takarít maga után”.

Ha ezt a megoldást nem függvényként írjuk meg, hanem a **return** helyett kiírjuk a választ, akkor hibás lesz a megoldásunk, mert annyiszor írja ki a **true** érték helyett a választ, ahány esetben előfordul, végül a **false** eset válaszát is kiírná.

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 algoritmusa akkor használható, ha tudjuk, hogy van adott tulajdonságú elem az adatsorozatunkban és kíváncsiak vagyunk, hogy hányadik sorszámú 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 szökőévben, az programhibát okozna. A strukturált programozás követelményének eleget téve – elöl-tesztelő, feltételes ciklust használunk.

```
1.  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.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    cout <<"A(z) "<< harminc(honapnapok)+1 <<". hónap 30 napos."<<endl;
5.  }
```

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

Keresés

A keresés algoritmusa 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.  void melyik28(int tomb[])     /*3. megoldás átalakítva*/
2.  {
3.    int melyik = -1;   /*bool van_28 = false;*/
4.    for (int i = 0; i < 12 && melyik == -1; i++) /*!van_28*/
5.      if(tomb[i] == 28)
6.        melyik = i;     /*van28 = true;*/
7.    if(melyik > -1)    /*return van_28;*/
8.      cout << "A(z) " << melyik + 1 << ". hónap 28 napos." << endl;
9.    else
10.     cout << "Nem találtunk 28 napos hónapot." << endl;
11. }
```

Felhasználása:

```
1.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    melyik28(honapnapok);
5.  }
```

A második megoldásban a keresés kiválasztás jellegét emeljük ki, a függvény felhasználójára bízzuk, 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.  int melyik28(int tomb[])      /*5. megoldás*/
2.  {
3.    int i = 0;
4.    while (i < 12 && !(tomb[i] == 28))
5.      i++;
6.    return i;       /*< 12 ? */
7.  }
```

A felhasználásnál kell figyelni!

```
1.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    int melyik = melyik28(honapnapok);
5.    if(melyik < 12)
6.      cout << "A(z) " << melyik + 1 << ". hónap 28 napos." << endl;
7.    else
8.      cout << "Nem találtunk 28 napos hónapot." << endl;
9.  }
```

A következő megoldásban a függvény logikai értéket ad vissza, de futása során – ha tudja – átadja a keresett indexet egy paraméterén keresztül.

```
1.  bool melyik28(int tomb[], int& ez)    /*7. megoldás*/
2.  {
3.    for (int i = 0; i < 12; 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.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    int melyik;
5.    if(melyik28(honapnapok, melyik))
6.      cout << "A(z) " << melyik + 1 << ". hónap 28 napos." << endl;
7.    else
8.      cout << "Nem találtunk 28 napos hónapot." << endl;
9.  }
```

Kiegészítés Pythonról áttérőknek és C++ nyelvészeknek

Ha nagyon megerőltetnénk 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 `map` elem) mellett, a `tuple` adattípus is használható. A `<tuple>` csomag betöltése után a Python-hoz hasonló módon használhatjuk a `tuple` típust. A `tuple` egy átmeneti objektum, több hasznos metódusa van, ebből most kettőt használunk. A felhasználás oldaláról indulva:

```
1.  int main()
2.  {
3.    int honapnapok[12]
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    tuple<bool, int> hol28 = melyik28(honapnapok);
5.    if(get<0>(hol28))
6.      cout <<"A(z) "<< get<1>(hol28) + 1 <<". hónap 28 napos."<< endl;
7.    else
8.      cout << "Nem találtunk 28 napos hónapot." << endl;
9.  }
```

Két változóba várjuk a függvény eredményét

Részek elérése

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

```
1.  tuple<bool, int> melyik28(int tomb[])       /*5. megoldás*/
2.  {
3.    int i = 0;
4.    while (i < 12 && !(tomb[i] == 28))
5.      i++;
6.    return make_tuple(i < 12, 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 megszámolás algoritmusa megmondja, hogy hány adott tulajdonságú elem van a bejárható objektumban.

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

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

```
1.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    cout << darab(honapnapok, 30) << "db. 30 napos hónap van." << endl;
5.  }
```

```
1.  int darab(int tomb[], int ertek)
2.  {
3.    int db = 0;
4.    for (int i = 0; i < 12; i++)
5.      if(tomb[i] == ertek)
6.        db++;
7.    return db;
8.  }
```

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

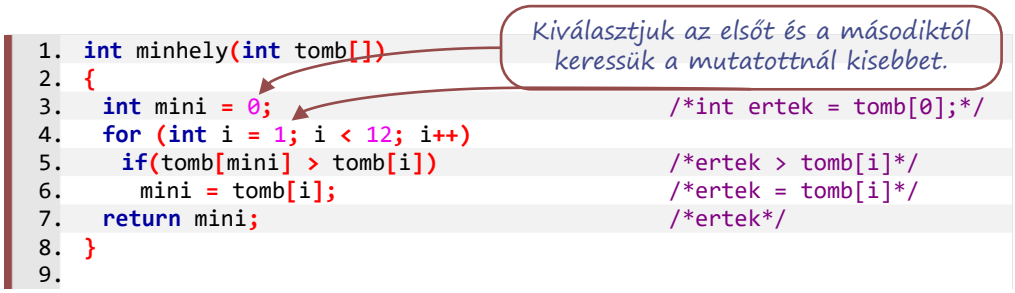
Szélsőérték-kiválasztás: maximum- és minimumkiválasztás

A maximum- vagy a minimumkiválasztás algoritmusa 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.  int main()
2.  {
3.    int honapnapok[12]
          { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4.    cout << "A legrövidebb hónap " << honapnapok[minhely(honapnapok)]);
5.    cout << " napos." << endl;
6.  }
```




```
1.  int minhely(int tomb[])
2.  {
3.    int mini = 0;                          /*int ertek = tomb[0];*/
4.    for (int i = 1; i < 12; i++)
5.      if(tomb[mini] > tomb[i])             /*ertek > tomb[i]*/
6.        mini = tomb[i];                    /*ertek = tomb[i]*/
7.    return mini;                           /*ertek*/
8.  }
9.
```

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.

Kiegészítés Pythonról áttérőknek és C++ nyelvészeknek

Ennek a fejezetnek a megértéséhez szükséges az, amit a `vector`, a `map` és a `tuple` adattárolókkal – *konténerekkel* – kapcsolatban már megfigyelhettünk. Ezen adattárolók objektumokban, az adatsorozat adatain kívül a bejárásukhoz, új adat beszúrásához, adat törléséhez megfelelő belső adatok, mutatók, eljárások és függvények állnak rendelkezésre. Ezekhez képest a tömb csak egy egyszerű adatsorozat, ami egyetlen mutató – *pointer* – , amely az első adat memóriatartományát adja meg. Ebből számolható ki, hogy a többi adat melyik memóriacímen található – ez történik a tömb név és index megadásával. A `[]` szelektor helyettesíthető egy összeadással és egy *dereferálás*sal. Mi a *dereferálás*? Az az érték (adat), amire a *pointer* – ami egy hivatkozás –mutat. Jele a *pointer* elé írt `*` operátor. C++ nyelvtanban a `tomb[index]` egy olvashatóbb formája a `*(tomb + index)` kifejezésnek.

Az alábbi kód kiírja a honapnapok értékeit:

```
1.  for(unsigned i = 0; i < 12; i++)
2.    cout << *(honapnapok + i) << ", ";
3.  cout << endl;
```

Néhány esetben könnyebb a paraméterként átadott feltételt függvényként megírni, erre mutat példát a `pred30()`.

```
1.  bool pred30(int a)
2.  {
3.    return a == 30;
4.  }
```

Másik megoldás a *lambda* kifejezések használata. – A *lambda* kifejezés kb. kódba integrált névtelen függvény. Ilyenkor nem előre írjuk meg a felhasznált függvényt, hanem egy paraméterátadás helyén.

Az egyes feladatok megoldásához szükséges az `<alrorithm>` csomag – ebben találhatók a típusfeladatok általánosított megoldásai és a `<numeric>` csomag a számításokkal.

A feladatok megoldása:

```
/*int szum(honapnapok)*/
cout <<
  accumulate (honapnapok, honapnapok+12, 0) << endl;
/*bool létezik(honapnapok, 28) lambda és bool?string:string */
cout << (
  any_of(honapnapok, honapnapok+12, [](int h){return h == 28;}
       )? "Van":"Nincs") << " 28 napos hónap." << endl;
/*pointer hol.van(honapnapok, 30) fgv-pointer és pointer műveletek.*/
cout<<
  find_if(honapnapok, honapnapok+12, pred30) - honapnapok + 1
     << ". hónap 30 napos." << endl;
/*pointer hol.van(honapnapok, nap) lambda és nincs => 13.*/
int nap = 29;
int hanyadik =
  find_if(honapnapok, honapnapok+12, [nap](int h){return h == nap;})
          - honapnapok;
if (hanyadik < 12)
  cout<< hanyadik + 1 << ". hónap " << nap << " napos." << endl;
else
  cout << "Nincs " << nap << " napos hónap" << endl;
/*int darabha(honapnapok, nap) lambda*/
nap = 30;
cout <<
  count_if(honapnapok, honapnapok+12, [nap](int h){return h == nap;})
    << " db. 30 napos hónap van" << endl;
/*int maxérték(honapnapok) - * nélkül maxhely*/
cout <<
  *max_element(honapnapok, honapnapok+12) << endl;
```

Konténerekkel, lista típusokkal kicsit másképp kell bánni: adatsorozatot tartalmazó objektumok kezdőcíme a `nev.begin()`, a vége a `nev.end()` pointerekkel adható meg.

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 nullafokos 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 nullafokos hajnal volt?
5. Mikor volt először nullafokos 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 hiányzási adatok kétdimenziós tömbjét a `main()` függvényen belül adjuk meg.

```
100.  int main()
101.  {
102.    int hianyzok[4][ 5]{ /*táblázat eleje*/
103.               /*1. sor*/  { 3, 4, 0, 0, 1 },
104.               /*2. sor*/  { 2, 3, 0, 0, 0 },
105.               /*3. sor*/  { 4, 3, 2, 3, 4 },
106.               /*4. sor*/  { 0, 0, 1, 0, 0 } /*táblázat vége*/};
107.    setlocale(LC_ALL, "Hun");
108.
```

Például a második hét keddjén három fő hiányzott.

```
110.    cout << "Heti hiányzás statisztikája" << endl;
111.    cout << "1a. Össz hiányzás: " << tombszum(hianyzok) << " ." << endl;
112.    cout << "1b. Összes hianyzas: " << szumszum(hianyzok) << "." << endl;
113.    if (volthet0(hianyzok))
114.      cout << "2. Volt hiányzásmentes hét." << endl;
115.    else
116.      cout << "2. Nem volt hiányzásmentes hét." << endl;
117.    if (volthetkevesebb(hianyzok, 5))
118.      cout << "3. Volt 5-nél kevesebb hiányzásos hét." << endl;
119.    else
120.      cout << "3. Ötnél mindig több volt a hiányzás." << endl;
121.    cout << "4. A(z) " << maxhet(hianyzok) + 1;
122.    cout << ". héten volt a legtöbb hiányzás." << endl;
123.    cout << "5. A " << kivalaszt1(hianyzok) + 1;
124.    cout << ". héten volt 1 hiányzás." << endl;
125.    cout << "6. ";
126.    hetfo_kedd(hianyzok);
127.  }
```

Az eldöntés vagy keresés típusalgoritmussal megoldható feladatokban gyakori, hogy az eredményt csak egy értékadásra használjuk. A 113–116. sorban a válaszokban a `"Volt"` és `"Nem volt"` ez az érték, amit egy elágazásban adtunk meg, de helyette használhatjuk a **háromoperandusú operátor**t: 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 op2 értéket írjuk.

```
113.  cout << "2. " << (volthet0(hianyzok) ? "Volt" : "Nem volt");
114.  cout << " hiányzásmentes hét." << endl;
```

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

```
160.  string valasz = volthet0(hianyzok) ? "Volt" : "Nem volt";
```

Hasonlóan átírható a 117–120. 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, de a végrehajtás szempontjából ez olyan, mintha az adatokat egyetlen sorba írtuk volna fel, egyetlen ciklusban, ahol 4*5 lehetne a lépések száma.

```
 5.  int tombszum(int tomb[4][5])
 6.  {
 7.    int szum = 0;
 8.    for (int tr = 0; tr < 4; tr++)      /*soronként 4-ig*/
 9.      for (int td = 0; td < 5; td++)    /*naponként 5-ig*/
10.        szum += tomb[tr][td];
11.    return szum;
12.  }
```

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.

```
14.  int sorszum(int tomb[4][5], int sor)
15.  {
16.    int szum = 0;
17.    for (int i = 0; i < 5; i++)
18.      szum += tomb[sor][i];
19.    return szum;
20.  }
```

A paraméterben megadott sorban.

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

```
22. int szumszum(int tomb[4][5])
23. {
24.   int szum = 0;
25.   for (int i = 0; i < 4; i++)
26.     szum += sorszum(tomb, i);
27.   return szum;
28. }
```

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:

```
30. bool volthet0(int tomb[4][5])
31. {
32.   int ez = 0;
33.   while (ez < 4 && sorszum(tomb, ez) != 0)
34.     ez++;
35.   return ez < 4;
36. }
```

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

Ugye látjuk, hogy egy reláció módosítása a lényegi változtatás az előző kódhoz képest?

```
38. bool volthetkevesebb(int tomb[4][5], int korlat)
39. {
40.   int ez = 0;
41.   while (ez < 4 && sorszum(tomb, ez) >= korlat)
42.     ez++;
43.   return ez < 4;
44. }
```

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, ami itt függvénnyel kiszámított érték. Az ismételt kiszámítást elkerülendő, az első használat alkalmával változóban tároljuk az értéket.

```
46. int maxhet(int tomb[4][5])
47. {
48.   int maxi = 0;
49.   int maxe = sorszum(tomb, 0);
50.   for (int i = 1; i < 4; i++)
51.   {
52.     int akt = sorszum(tomb, i);
53.     if (maxe < akt)
54.     {
55.       maxe = akt;
56.       maxi = i;
57.     }
58.   }
59.   return maxi;
60. }
```

Kétszer használt i-edik összeg

Minden ciklusléptésben 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.

```
62.  int kivalaszt1(int tomb[4][5])
63.  {
64.    int i = 0;
65.  /*hibás, ha nincs megfelelő sor!*/
66.    while (sorszum(tomb, i) != 1)
67.      i++;
68.    return i;
69.  }
```

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 sorozatszámítást „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.

```
71.  void hetfo_kedd(int tomb[4][5])
72.  {
73.    int hetfo = 0;
74.    int kedd = 0;
75.    for (int het = 0; het < 4; het++)
76.    {
77.      hetfo += tomb[het][0];
78.      kedd += tomb[het][1];
79.    }
80.    string valasz;
81.    if (hetfo > kedd)
82.      valasz = "Hétfői napokon hiányoznak többen.";
83.    else if (hetfo < kedd)
84.      valasz = "Keddi napokon hiányoznak többen.";
85.    else
86.      valasz = "Ugyanannyi a hiányzás a hét két napján.";
87.    cout << valasz << endl; /*ha függvény lenne: return valasz;*/
88.  }
```

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, globális változóként hozzuk létre.

```
 5.  struct csucs
 6.  {
 7.     string nev;
 8.     string hegys;
 9.     int magas;
10.  }
11.
12.  csucs csucsok[7] {
13.     {"Kékes", "Mátra", 1014 },
14.     {"Hidas-bérc", "Mátra", 971 },
15.     {"Galya-tető", "Mátra", 964},
16.     {"Szilvási-kő", "Bükk-vidék", 961 },
17.     {"Péter hegyese", "Mátra", 960 },
18.     {"Kettős-bérc", "Bükk-vidék", 958 },
19.     {"Istállós-kő", "Bükk-vidék", 958 }
20.  };
```

A **csucs** struktúrának nem írtunk konstruktort, így az alapértelmezett konstruktor az adattagok sorrendjében állítja be az értékeket.

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()` függvényen belül válaszolunk.

1. **Hány csúcs található a Bükk-vidéken?**

```
25.  int main()
26.  {
27.    setlocale(LC_ALL, "Hun");
28.    int db = 0;
29.    for (int i = 0; i < 7; i++)
30.      if (csucsok[i].hegys == "Bükk-vidék")
31.        db++;
32.    cout << "A Bükk-vidék " << db << " csúcsa van a listában." << endl;
```

2. **Mennyi a mátrai csúcsok magasságának átlaga?**

```
33.    double szum = 0;
34.    db = 0; //újrahasznosított
35.    for (int i = 0; i < 7; i++)
36.      if (csucsok[i].hegys == "Mátra")
37.      {
38.        szum += csucsok[i].magas;
39.        db++;
40.      }
41.    cout << "A mátrai csúcsok átlagosan " << round(szum / db)
          << " méteresek." << endl;
```

Az eredmény egész számra kerekítéséhez a `<cmath>` csomag kell.

A 3. és 4. kérdésnek a megválaszolásához függvényt írunk. a `main()` előtt deklaráljuk a függvényeket, majd a `main()`-t követően jönnek a definíciók.

```
22.  bool vanezres();
23.  csucs bukkleg(string);
```

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

Az itt bemutatott megoldás egyik érdekessége, hogy a két lehetséges válasz egy szóban tér el egymástól, háromoperandusú feltételes értékadást használunk az elágazás kiírása helyett. A `main()` folytatása:

```
33.    cout << vanezres() ? "Van" : "Nincs"
             << " 1000 méternél magasabb csúcs"<< endl;
34.    cout << "A Bükk-vidék legmagasabb csúcsa: "
             << bukkleg("Bükk-vidék").nev << endl;
35.  }/*main() vége*/
36.
37.  bool vanezres()
38.  {
39.    int i = 0;
40.    while (i < 7 && !(csucsok[i].magas > 1000))
41.      i++;
42.    return i < 7;
43.  }
```

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 hegységnév esetén is értelmesen – válaszolunk.

```
45.  csucs bukkleg(string hegyseg)
46.  {
47.  /*első csúcs keresése*/
48.    int maxi = 0;
49.    while (maxi < 7 && csucsok[maxi].hegys != hegyseg)
50.      maxi++;
51.    if (maxi < 7) /*Ha van egy Bükk-vidéki csúcs,*/
52.  /*A legnagyobb kiválasztása*/
53.      for (int i = maxi + 1; i < 7; i++)
54.        if (csucsok[i].hegys == hegyseg &&
               csucsok[i].magas > csucsok[maxi].magas)
55.          maxi = i;
56.  /*Ha van Bükk-vidéki csúcs, akkor a legmagasabb nevének megadása*/
57.    if (maxi < 7)
58.      return csucsok[maxi];
59.    else /*Hibajelzéshez értékek*/
60.      return csucs{"#Nincs csúcs#", hegyseg, 0};
61.  }
```

Itt a 57–60. 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 < 7)  ? csucsok[maxi] : csucs{"#Nincs csúcs#",hegyseg,0};
```

23. példa: ***Kiegészítés*: Kutyaoltás (szótárral)**

Az egyik tipikus eset, amikor szótárt érdemes használni az, ha 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 a következő példa. Az állatorvostól kapott, egy hétre vonatkozó adatokat szótárban tárolták:

```
 1. #include <iostream>
 2. #include <map> /*KELL*/
 3. using namespace std; {
    …
40. int main()
41. {
42.   setlocale(LC_ALL, "Hun");
43.   map<string, int> oltasok
44.   {
45.     {"hétfő",8},{"kedd",14},{"szerda",2},{"csütörtök",3},{"péntek",13}
46.   };
```

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?**

Egy-egy pénteki, illetve szombati plusz oltás bejegyzésekor, ha az adott nap szerepel már a szótárban, akkor ennek értékéhez kell hozzáadni 1-et. Ha még nem szerepel, akkor elvileg nem tudjuk növelni az aznapi oltások számát. Ilyenkor a kulcs keresése nem ad találatot, pontosan lehet tudni, hogy hova, milyen kulccsal kell hozzáadni a szótárelemet, ezért az rögtön létrejön, az érték pedig a típusnak megfelelő alapértelmezett érték, 0 lesz.

```
48.   oltasok["péntek"]++; /*növeli a second értéket, most már 14*/
49.   oltasok["szombat"]++; /*létrehozza {szombat, 0}-t majd növel*/
```

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

```
50.   cout << "1. A héten " << osszes(oltasok) <<
                                  " kutya kapott oltást" << endl;
51.   cout << "2. " << (volt(oltasok, 10) ? "Volt" : "Nem volt") <<
                                  " legalább tízkutyás nap." << endl;
52.   cout << "3. A legkevesebb kutya ezen a napon kapott oltást: " <<
                                  min(oltasok) << endl;
53. }
```

A függvényekben foreach-ciklust használunk, mert a `map` binárisfa jellegű adatsorozat. Ebből következik, hogy az eldöntés/keresés típusalgoritmusokban `break;`/`return;` megszakítás kell.

1. **Hány kutyát oltott be a héten az állatorvos?**

```
 5. int osszes(map<string, int> oltasok)
 6. {
 7.   int szum = 0;
 8.   for (pair<string, int> oltas : oltasok)
 9.     szum += oltas.second;
10.   return szum;
11. }
```

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

```
13.  bool volt(map<string, int> oltasok, int minimum)
14.  {
15.    bool volt = false;
16.    for (pair<string, int> oltas : oltasok)
17.      if (oltas.second >= minimum)
18.      {
19.        volt = true;     /*19-20. sor helyett lehetne: return true;*/
20.        break;
21.      }
22.    return volt;         /*itt pedig return false; és a 15. sor nem kell*/
23.  }
```

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

A bejáró-ciklus miatt az első adat lekérdezése nem idevaló. Helyette adjunk meg egy lehetetlenül nagy „minimális” értéket. A legnagyobb lehetséges egész érték az adattípus méretétől függ, ez pedig attól, hogy hogyan van megadva a programozási nyelvben. Ezért elnevezés van rá: `INT_MAX`. (Ez a tulajdonsága az `int` típusnak statikus, a C++ nyelv átírásával módosítható.)

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

```
24.  string min(map<string, int> oltasok)
25.  {
26.    string minnap = "Hiányzik";
27.    int minertek = INT_MAX;      /*az int legnagyobb lehetséges értéke*/
28.    for (auto oltas : oltasok)
29.      if (oltas.second < minertek)
30.      {
31.        minnap = oltas.first;
32.        minertek = oltas.second;
33.      }
34.    return minnap;      /*csak a nap neve kell*/
35.  }
```

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!

Elmélkedés: Szöveges és bináris fájlok – Bájtok és bitek értelmezése

Ha elég messziről – vagy inkább egy jegyzettömbben – tekintünk a számítógépeken tárolt fájlokra, akkor alapvetően kétfélét különböztetünk meg. Az egyikben szöveg van, méghozzá 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` fájltípus, valamint a weboldalak kódját tartalmazó `html` és `css` kiterjesztésű állomány és még jó néhány fájltípus. Szöveget tartalmazó fájlokat készíthetünk az úgynevezett egyszerű szerkesztőkkel, például a Windows Jegyzettömbjével, de ilyen fájlt ír minden IDE. Code::Block projektben az obj és bin mappán kívül minden fájl szöveges.

A „másik" fájltípus – a jegyzettömbben 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` fájlokat, 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 példákban – 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` (régi Mac), `\n\r` (régi 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 operációsrendszeren 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 ősibb 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. Érdemes tudni, hogy a fájl tizenöt sorból áll, és Latyak kacsa 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 kacsa 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 kacsa 2
```

Ismétlés: Szöveges fájlban tárolt adatok felhasználása konzolon keresztül

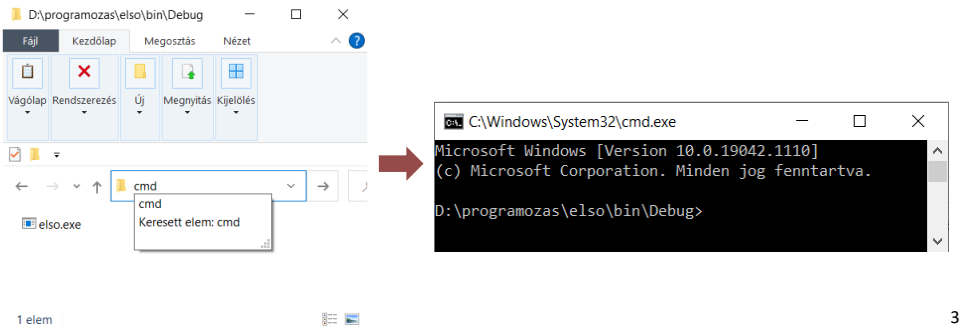
A kilencedikes jegyzetben már volt szó arról, hogy hogyan lehet hosszabb (szöveges) adatsorokat felhasználni a programunkban. Az egyik módszer a fájlból konzolra másolás, a másik a konzol átirányítása.

A fájlból konzolra másoláshoz egy egyszerű szöveges fájlban eltároljuk, amit majd be szeretnénk írni. A program adatsorokat olvas, ezért a beíráskor ennek megfelelően, enterrel tördeljük 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-e!

- 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 az `.exe` 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 használni mindkettőt, 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 útvonalát a gyökérkönyvtártól elindulva adjuk meg, mert így csak az adott gépen működhet a programunk (vagy ott se, egy mappanév módosítása után). Relatív, a programhoz képest legegyszerűbb elérési útvonal az, amikor nem kell útvonalat írni, csak a fájl nevét. Ezt úgy érjük el, ha a `main.cpp` fájl mellé másoljuk a szöveges fájlt.

Amikor a `main.cpp` kódjából elkészül a programfájl, akkor a beolvasandó fájl elérési útvonala a `main.cpp`-hez képest relatív. De erről az elkészült `.exe` fájl nem tud. Ezért amikor parancssorból vagy fájlkezelőből indítjuk el a kész programot, nem találja a beolvasandó fájlt. A gyakorlatban ez azért nem okoz gondot, mert a programozó mindig teszt fájllal dolgozik, a felhasználó viszont csak az `.exe` fájlt kapja meg, emellé másolja az adatfájlt. Ha mindkét módot szeretnénk használni, akkor *az adatfájlt a* `main.cpp` *és a* `.exe` *mellé is be kell másolni*.

[3] A kilencedikes jegyzetben szereplő kép.

Fájl beolvasása, adatok tárolása

A fájlok kezeléséhez a `<fsstream>` csomagra lesz szükségünk. A megnevezés hasonlósága nem véletlen: Az `<iostream>` a konzollal folytatott kommunikációt tartalmazza, az `<fstream>` a fájlokkal, a `<sstream>` pedig belső kommunikációt tesz lehetővé egy speciális karaktersorozat segítségével. Nem csoda, hogy a fájlolvasó és -író eszköz használata – miután létrehoztuk őket – megegyezik a `cin`, illetve `cout` használatával.

A fájlbeolvasás eszköze egy `ifstream` típusú eszköz, a fájlba íráshoz `ofstream` kell. A „változó név” valójában az eszköz neve, az analógiát kihasználva például `fin`, illetve `fout` is lehet. Vagy bármi más … Rögtön hozzá is rendelhetünk a névhez egy fájlt, zárójelben szövegként megadva a fájl nevét (és elérési útját), de külön utasításként a `.open()` függvény is használható.

Különösen a fájl írásánál fontos, hogy le is zárjuk a kapcsolatot, mert ez jelenti a háttértárra mentést, de az olvasásnál is akadályozhatja más programok számára a fájl használatát, ha nem zárjuk le a fájlt. A lezárás utasítása a `.close()`. Ilyenkor nem kell megadni a fájl nevét, hiszen azt tudja az eszközünk (ha véletlenül mást írnánk, az programhibát okozna). A lezárás után elvileg a filekezelő eszköz újra használható: az `.open()`-nel megnyithatunk egy másik fájlt.

A fájl olvasása a `cin`-nel azonos: a fájl elejétől sorban olvassa a sorokat, karaktereket. Nem tud sem kihagyni, sem visszalépni. Gyakori feladat a fájl végének felismerése, erre háromféle megoldás létezik:

- A sikertelen fájlból olvasást `false` logikai értéknek érzékelik a vezérlési struktúrák. Ezért gyakran látható a `while()` vagy `if()` logikai kifejezése helyén beolvasás. Ezt használtuk már a 8. példában, a `stringstream`-ből olvasáskor is.
- Amikor a beolvasás azért hiúsul meg, mert nincs mit olvasni, a stream `.eof()` függvénye `true` értékű lesz. Ezzel csak az a probléma, hogy a hiányzó adat előbb okoz problémát, csak ezt követően lehet vizsgálni, hogy a hiba a fájl vége miatt volt-e.
- A streamben lévő első bájtot meg lehet „nézni”. A `.peak()` függvény megmutatja mi lesz a következő olvasás első bájtja. A fájl végét egy vezérlő karakter jelzi, aminek C++ neve `EOF`. Óvatosan közelítve a fájl végéhez, vizsgáljuk, hogy a `.peak() != EOF`.

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.

```
1.  #include <iostream>
2.  #include <fstream>     /*Nem szabad elfelejteni, hogy ez is kell!*/
3.  using namespace std;
4.
5.  void beki()
6.  {
7.    ifstream fin;                       /*vagy fin("allatok.txt");*/
8.    fin.open("allatok.txt");
9.    string adat;
10.   while (fin.peek() != EOF)           /*amíg nincs 'vége' jelzés*/
11.   {
12.     fin >> adat;
13.     cout << adat << " ";
14.   }
15.   fin.close();                  /*végül bezárja*/
16. }
```

A fájlbeolvasás hibáit nem jelzi a program, csak abból látható, hogy nem kapnak új értéket a változók. A fájl megnyitásának sikerességét ellenőrizhetjük, jelezhetjük a hibát a felhasználónak az `.is_open()` függvény felhasználásával.

25. példa: Fájlból beolvasott adatok tárolása

A fájlban lévő adatokat tárolásához 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.

```
18. void be15()
19. {
20.   ifstream fin("15allat.txt");
21.   int N; fin >> N;
22.   string sorok[50];
23.   for (int i = 0; i < N; i++)
24.     getline(fin, sorok[i]);
25.   fin.close();
26.   for (int i = 0; i < N; i++)
27.     cout << sorok[i]) << endl;
28. }
```

2. Egyszerre beolvassunk mindent, majd a programban sorokra tördeljük

Az `ifstream` nem csak sort tud olvasni, hanem teljes fájlt is, csak a `getline()` harmadik paraméterét kell ügyesen megadni. Az eredmény egyetlen `string`, amit ezután egy `stringstream`-be tudunk betölteni, majd ebből akár szavanként kivenni.

```
30. void beegybe()
31. {
32.   ifstream fin("allatok.txt");
33.   string szoveg; getline(fin, szoveg, '\0');
34.   cout << "A beolvasott fajl" << endl;
35.   cout << szoveg << endl;
36.   string tesztt[45];
37.   stringstream fajlszeru(szoveg);
38.   int i = 0;
39.   while (i < 45)
40.   {
41.     fajlszeru >> tesztt[i];
42.     i++;
43.   }
44.   cout << "szavanként kiírás" << endl;
45.   for (int i = 0; i < 45; i++)
46.     cout << tesztt[i] << endl;
47.   fin.close();
48. }
```

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

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

```
49.  void betarba()
50.  {
51.  vector<string> tesztv;
52.  ifstream fin("allatok.txt");
53.  while (fin.peek() != EOF)
54.  {
55.    string temp;
56.    getline(fin, temp);
57.    tesztv.push_back(temp);
58.  }
59.  fin.close();
60.  for (string s : tesztv)
61.    cout << s << endl;
62.  }
```

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 a futási időbéli 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 bájt lefoglalása számít).

```
63.  struct allatka
64.  {
65.  string nev;
66.  string faj;
67.  int kor;
68.  };
69.  allatka allatos[100];
```

Az itt következő megoldásban az adatok tárolására egy kellően nagy méretű, minden más programrész által használható, globális 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.

```
71.  int beminimal()
72.  {
73.    ifstream fin("allatok.txt");
74.    int db = 0;
75.    while (fin.peek() != EOF)
76.    {
77.      fin >>allatos[db].nev >> allatos[db].faj >> allatos[db].kor;
78.         db++;
79.    }
80.    fin.close();
81.    return db;
82.  }
83.
```

A függvény felhasználása a `main()` eljárásban:

```
166.  int nminimal = beminimal(); /*közben módosítja az allatos tömböt is*/
167.  for (int i = 0; i < nminimal; i++)
168.  cout << allatos[i].nev << " ";
169.  cout << endl;
170.
```

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 függvényben vagy eljárásban. Jelen esetben a `main()` eljárásban hozzuk létre a 100 elemű `allatkak` tömböt és ebbe töltjük be az adatokat a `beapro()` eljárással.

```
171.  allatka allatkak[100]; /*van 100 adat, alapértelmezett értékkel*/
172.  int napro; /*Nincs értéke, módosítani kell*/
173.  beapro(allatkak, napro);
174.  for (int i = 0; i < napro; i++)
175.  cout << allatkak[i].nev << " ";
176.  cout << endl;
```

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

A `beapro()` eljárás az előző megoldáshoz képest alig van változik:

```
84.  /*kisallatok: megfelelő méretű allatka adatokra készített tömb*/
85.  void beapro(allatka kisallatok[], int& db)
86.  {
87.    ifstream fin("allatok.txt");
88.    db = 0;
89.    string sorelso;
90.    while (fin >> sorelso)
91.    {
92.      kisallatok[db].nev = sorelso;
93.      fin >> kisallatok[db].faj;
94.      fin >> kisallatok[db].kor;
95.      db++;
96.    }
97.    fin.close();
98.  }
```

6. Összetett adat létrehozása fájlból beolvasott sorból, egy lépésben.

Az előző két megoldásban a függvényben bontottuk fel elemeire a sorokat és egyenként adtuk át az **`allatka`** adatnak. Programunk strukturáltabb lesz, ha az egy állatról szóló sort a függvényünkben egyetlen **`string`** adatként használjuk és a struktúrában adjuk meg, hogy az egyes részei mit jelentenek. Ehhez az kell, hogy konstruktort, (létrehozót) írjunk a struktúrához. A korábbi – ismétlő – feladatokban a konstruktor szignatúrájában (bemenő adatainál) az adattagok sorrendjének megfelelően adtuk meg az egyes értékeket. Most olyan konstruktor lenne praktikus, ami az adatsort – egy **`string`**-et – használna az adat előállítására.

Azzal, hogy megadjuk, hogyan értelmezze a megadott adatokat a `struct`, elveszítjük az alapértelmezett – változók sorrendjében megadva vagy üresen hagyva – konstruktort, ezért ezt is meg kell írni. (Pontosabban: akkor kell megírni, ha van olyan helyzet, amikor „üresen” akarjuk létrehozni. Tipikusan ilyen, ha csak deklaráljuk, amikor tömböt hozunk létre a későbbi tárolás céljából.

A C++ nyelvben van az összetett adatok kezdőértékadására egy rövid megoldás az inicializáló lista. Ebben a kettőspontot követően felsorolhatjuk, hogy melyik adatnak milyen értéket adunk.

```
 99.  struct allat
100.  {
101.    string nev;
102.    string faj;
103.    int kor;
104.    allat(string sor)
105.    {
106.      stringstream ss(sor);
107.      ss >> nev >> faj >> kor;
108.    }
109.    allat():nev(""),faj(""),kor(0)      //inicializáló lista minta
110.    {} // ^paraméter nélküli konstruktor, pl. tömb létrehozáshoz kell
111.  };
```

Az alapértelmezett – paraméter nélküli – konstruktor inicializálás nélkül is használható, de meglepetéseket tartogat. Például a kor értéke az lesz, amit a memóriában talál.

Ezután a beolvasáshoz az első három módszer bármelyikét használhatjuk. 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 adatstruktúra:

```
112.  vector<allat> allatlist;      /*Csak megnevezés, globális.*/
113.  void belistbe()
114.  {
115.    ifstream fin("allatok.txt");
116.    string sor;
117.    while (getline(fin,sor))
118.       allatlist.push_back(allat(sor));
119.    fin.close();
120.  }
121.
```

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

```
177.  belistbe();
178.  for (allat a : allatlist)
179.  cout << a.nev << endl;
```

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

```
180.  const int MaxN = 100;          /*A program legelején szokott lenni.*/
181.  allat allataim[MaxN];          /*Ehhez kell az allat() konstuktor*/
182.  int Ndb = 0;                   /*Komoly galibát okoz, ha ez nem 0*/
183.  beNdb("allatok.txt", allataim, Ndb);
184.  for (int i = 0; i < Ndb; i++)
185.  cout << allataim[i].nev << "; ";
186.  cout << endl;
```

Az `Ndb`-nek az értékadása megoldható lenne függvényérték visszaadásával is.

```
122.  void beNdb(string fn, allat allatok[], int& N)
123.  {
124.    ifstream fin(fn.c_str());
125.    string sor;
126.    while (getline(fin,sor)){
127.      allatok[N] = allat(sor);
128.      N++;
129.    }
130.    fin.close();
131.  }
```

Figyeljük meg: az **`ifstream`**-nek nem jó a **`string`** típusú változó, mert valójában karakterek tömbjét, karaktersorozatot vár – a nyelv alapjául szolgáló C nyelvnek megfelelően. Ezért az átadott **`string`**-ből a `c_str()` függvénnyel „kiszedjük” a karaktereket.

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 előző példa 1–3. megoldásaiban 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 4–7. megoldások lefuttatása után a főprogramban is megírhatjuk legidősebb kiválasztását, vagy bármelyik adatsorozatot átadhatjuk egy függvénynek.

Figyeljünk arra, hogy a 4., 5. és 7. beolvasás során nemcsak a tömböt kell átadnunk, hanem a beolvasott adatok számát is. Ezekben az adattároló jóval nagyobb, mint a beolvasott adatsorok 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 6–7. feladatok **`allat`** struktúráját.

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

```
132.  allat legoregebb(allat allatok[], int N)
133.  {
134.    int maxi = 0;
135.    for (int i = 0; i < N; i++)
136.      if (allatok[i].kor > allatok[maxi].kor)
137.        maxi = i;
138.    return allatok[maxi];
139.  }
```

27. példa: Legöregebb állat adatainak fájlba írása

```
161.  fajlbair(legoregebb(allataim, Ndb), "oreg.txt");
```

A szövegfájlok írása egyszerűbb az olvasásuknál. A fájl megnyitásához `ofstream` kell, ennek konstruktorában megadjuk a fájl nevét. Ha a fájl már létezik, akkor az törlődni fog, az `ofstream` új fájlt hoz létre. A fájlba írás eljárása és tulajdonságai is pontosan megegyeznek a `cout`-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. Ha elfelejtjük a `close()`-t, akkor nem lesz kimeneti fájl.

A fájlnév átadása során itt is figyelni kell arra, hogy a `string` objektum (összetett adat), ezért a név `c_str()` függvénnyel adjuk meg az egyszerű karaktersorozatot.

```
140.  void fajlbair(allat ez, string fajlnev)
141.  {
142.    ofstream fout(fajlnev.c_str());
143.    fout << ez.nev << " " << ez.kor);
144.    fout.close();
145.  }
```

Az útvonal nélkül megadott fájl abba a mappába kerül, amelyikben a beolvasáshoz útvonal nélkül megadott fájl is található.

Ebben a fejezetben a fájlból olvasásra az `fin` változót használtuk, az írásra a `fout`-t. Ha interneten vagy szakkönyvben olvasunk fájlműveltekre C++ kódokat, akkor valószínűleg nem ezekkel fogunk találkozni. A modern programozó csapatban dolgozik, a közös munkát könnyíti meg, ha az elnevezések szabványosak és érthetőek. Ebben a jegyzetben az oldalszélességet és az olvashatóságot is figyelembe véve a nevek hossza (rövidsége) is szempont. A C++ kódolási szokás szerint a `struct`, `int` és a függvénynevek kisbetűvel kezdődnek, a többszavas elnevezéseket aláhúzással szokták kapcsolni. Egy változó neve sokszor a típusának rövidítése szokott lenni. Egyszerű adat esetén az első betű, például `string s`, `char c`. Érdemes ezekhez a jelölésekhez alkalmazkodni, mert megkönnyíti a programkód értelmezését, de az sem elvetendő szempont, hogy a saját elnevezések magyar (ékezetmentes) fogalomhoz kapcsolódjanak, mert így jobban látszik, hogy mi a saját elnevezésünk és mit vettünk át.

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 fájlnév (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.

Már korábban is használtunk ékezetes karaktereket a válaszaink kiírásához. A helyes megjelenítéshez (különösebb magyarázat nélkül) a `setlocale(LC_ALL, "Hun");` tette lehetővé. Most szeretnénk változóban tárolni, felhasználótól bekérni, fájlból beolvasni az ékezetes karaktereket tartalmazó szövegeket, ezekkel műveleteket végezni, kiírni képernyőre, fájlba.

A változóban tárolt, konzolról és fájlból bekért ékezetes adatok képernyőre kiírása furcsa eredményt ad:

```
 5. int main()
 6. {
 7.   setlocale(LC_ALL, "Hun");    //másik verzió ezt kommentbe téve
 8.   ifstream fin("teszt.txt");
 9.   ofstream fout("setloc.txt"); //másik verzióban másik fájlnév
10.   string minta = "áéíóöőúüű";
11.   string cons, fajl;
12.   cin >> cons;
13.   fin >> fajl;
14.   cout <<" minta: "<< minta <<" beírt: "<< cons <<" fájlból: "<< fajl;
15.   fout <<" minta: "<< minta <<" beírt: "<< cons <<" fájlból: "<< fajl;
16.   fin.close();
17.   fout.close();
18.   return 0;
19. }
```

Az ANSI kódolású `teszt.txt` tartalma „`íáéűúőóüö`”. Futtatáskor „`öüóőúűéáí`” tesztadatot beírva a konzol, illetve a kimeneti fájl tartalma:

```
öüóőúűéáí
minta: áéíóöőúüű beírt: "?˘<ŁŰ' ˘ fájlból: íáéűúőóüö
```

```
 minta: áéíóöőúüű beírt: ”?˘‹Łű, ˘ fájlból: íáéűúőóüö
```

Módosítva a 7. és 9. sort:

```
öüóőúűéáí
minta: ßÚÝ˘÷§˙Řű beÝrt: öüóőúűéáí fßjlb˘l: ÝßÚű˙§˘Ř÷
```

```
 minta: áéíóöőúüű beírt: ”?˘‹Łű, ˘ fájlból: íáéűúőóüö
```

Azt látjuk, hogy a magyarítás ahhoz kell, hogy a kódban lévő ékezetes karakterek helyesen jussanak ki a konzolra és a fájlba, de nem használható a konzolról beírt adatokhoz. Az eredményt befolyásolja a forrás fájl kódolása. Az ANSI azért jó, mert a kimenet ANSI kódolású, így helyes lesz az átvitel. Például egy UTF-8 kódolású fájlt beolvasva a kimeneten más karaktereket látunk, az ANSI megfelelőket. A konzolról beolvasott ékezetes karakterek OEM 852-es kódolással jelennek meg helyesen.

Ha módosítjuk a teszt.txt kódolását OEM 852-re, akkor a fájlok eléréséhez is ezt kell beállítani. Ezt például úgy érhetjük el, hogy a `setlocale()` helyett a `system("chcp 852")` kódot írjuk be, valamint a fájl kódolását módosítjuk (például Notepad++ erre alkalmas). Ezzel a konzollal és fájlokkal történő kommunikáció azonos kódkészletű lesz, de a Code::Blocks kódfájlja más kódolást használ.

Összességében azt látjuk, hogy az a biztos, ha nem használunk ékezetes karaktereket. Ha mégis, akkor azt vagy csak a fájlban, vagy csak a konzollal történő kommunikációban, vagy

csak a kódban és ehhez állítjuk be a környezetet, a fájlformátumot. A megoldás az UTF-8 kódolás alapértelmezetté tétele a programozási nyelvben, az IDE szerkesztő felületén, az operációs rendszer konzol/terminál ablakában, a szövegfájlok kódolása során … mindenhol.

28. példa: Ékezetes állatok olvasása fájlból és kiírása fájlba

Készítsünk egy új, UTF-8 kódolású szöveges fájlt `allatok.txt` néven:

```
Hápi kacsa 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 eltárolása nélkül adjuk meg. Ehhez egyszerre nyitjuk meg a fájlt olvasásra és egy másik fájlt írásra.

A megoldáshoz készítsünk új programot, ne felejtsük el, hogy a fájlt elérhetővé kell tennünk és a `<fstream>` csomagra is szükségünk lesz.

```
 5. void ekezetes_olvasir()
 6. {
 7.   ifstream fin("allatok.txt");
 8.   ofstream fout("nevsor.txt");
 9.   string nev, faj;
10.   int kor;
11.   while (fin >> nev)
12.   {
13.     fin >> faj >>  kor;
14.     cout << nev << ", ";
15.     fout << nev << " " << faj << endl;
16.   }
17.   fin.close();
18.   fout.close();
19.   cout << "\b\b." << endl;
20. }
21.
```

Futtatást próbáljuk ki többféle kódolással:

```
22. int main()
23. {
24.   //system("chcp 65001"); //system("chcp 852");
      //system("chcp 1250"); //setlocale(LC_ALL, "Hun");//
25.   ekezetes_olvasir();
26.   return 0;
27. }
```

Ellenőrizzük, hogy a `nevsor.txt` létrejött-e, és helyesen szerepelnek-e benne a nevek. Válasszunk ki a 24. sorból egyenként a kódtáblákat, figyeljük meg a hatását! Jó eséllyel minden

esetben helyes lesz a létrejövő fájl, de csak a "chcp 65001", az UTF-8 kódolás beállításával lesz helyes a kiírás a képernyőre.

Vajon a bájtok változnak, vagy csak a megjelenítés? Ennek kiderítéséhez a szöveg helyett karakterek szintjén vizsgáljuk a fájl tartalmát.

Írjunk programot bájtok vizsgálatára!

Az extracror (>>) karakterenként is tudja olvasni az adatokat, ha **char** típusú változót teszünk mögé. Olvassuk egyenként a karaktereket, jelenítsük meg az ékezetes karakterek kódját …

29. példa: Fájlból olvasás karakterenként

Az előző példában szereplő állatok.txt fájlt olvassuk be a **char** típusú változóba. Az olvashatóság értekében, 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 szóközök között a kódját jelenítjük meg.

```
 6.  void karakterolvasas()
 7.  {
 8.    ifstream fin("allatok.txt");
 9.    char c;
10.    while (fin >> c)
11.    {
12.      if ('A' <= c && c <= 'Z')
13.        cout << endl; /*nagybetű előtt új sort kezd*/
14.      if (('A'<=c && c<='Z') || ('a'<=c && c<='z') || ('0'<=c && c<='9'))
15.        cout << c;
16.      else
17.        cout << " " << c << " ";
18.    }
19.    fin.close();
20.  }
```

Az allatok.txt tartalma:

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

```
Active code page: 65001

H ♦  ♦ pikacsa1
Morg ♦  ♦ kutya12
Hekuskutya4
B ♦  ♦ g ♦  ♦ birka3
F ♦  ♦ tfalkecske5
M ♦  ♦ z ♦  ♦ sszarvasmarha3
H ♦  ♦ n ♦  ♦ rliba2
K ♦  ♦ l ♦  ♦ ncmalac1
```

A képernyőképen látható, hogy a szóközöket nem sikerült megjeleníteni. Mivel az új sort a kódban írtuk elő és nem látható a sorok végén, hogy mitől kezd új sort a fájlban lévő szöveg, megállapíthatjuk, hogy ezeket a karaktereket is elnyeli az extractor, nem tekinti eltárolható karakternek.

Az ékezetes karakterek viszont nem egy, hanem két karakterként jelennek meg. Az egyik rombusz, a másik láthatatlan. A különböző ékezetes karaktereknek egyforma a kódja? Módosítsuk a 17. sort, írjuk ki a c-t, egész számként:

```
17.       cout << " " << (int)c << " ";
```

Az eredmény:

Az allatok.txt tartalma:

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

```
Active code page: 65001

H -61  -95 pikacsa1
Morg -61  -77 kutya12
Hekuskutya4
B -61  -87 g -59  -111 birka3
F -59  -79 tfalkecske5
M -61  -70 z -61  -77 sszarvasmarha3
H -61  -83 n -61  -95 rliba2
K -61  -68 l -61  -74 ncmalac1
```

Tényleg két karakter. És mindkét érték minden esetben negatív. Az első karakter -61 vagy -59, a második -68 (ü) és -111 (ő) közötti érték. Hogyan lehet negatív egy karakterkód? Jöjjön egy kis elmélet: negatív számok bináris tárolása.

A negatív egész értékeket kettes-komplemens módban ábrázolja (és értelmezi) a processzor. Példaként, a `61` 8-bites bináris ábrázolása `00111101`. A negatív érték ehhez képest: minden bit ellentettje +1: `-61` 8-bites ábrázolása `11000010 + 1`, azaz `11000011`. Ez `195`. A karakterkód értéket ennél gyorsabban is megadhatjuk, mivel a számításból látszik, hogy a negatív szám abszolútértéke és a bináris pozitív megfelelőjének összege 256. Naná, ezért komplemens a módszer neve …

Miért lett negatív az érték? Nagyjából a C/C++ nyelv egyik ősi szabványa miatt. C-ben nem volt byte. A számok és karakterkódok közös adattípusa a char, ami egyben egybájtos egész szám is. Az ASCII 7 bites, a karakterkódjai pozitív értékek. A többi csak idővel lett karakterré. Azután idővel nem is egy bájt, hanem több bájtossá. Persze a nyelv is fejlődött … Az egy bájtos karakter típusa `unsigned char`. A C++ 2011-es verziója óta létezik a char16_t és a char32_t és a 2020-as nyelvújítás óta (elvileg) a char8_t típus is, de ezek írásához, olvasásához az összes függvényt és operátort másképp kell használni.

A 9. sor módosításával már pozitív értékű karakterkódokat kapunk

```
9.  unsigned char c;
```

Az eredmény:

Az allatok.txt tartalma:

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

```
Active code page: 65001

H 195  161 pikacsa1
Morg 195  179 kutya12
Hekuskutya4
B 195  169 g 197  145 birka3
F 197  177 tfalkecske5
M 195  186 z 195  179 sszarvasmarha3
H 195  173 n 195  161 rliba2
K 195  188 l 195  182 ncmalac1
```

Utánajárhatunk, hogy ezek a kódok valójában mit jelentenek. A különböző kódtáblákban a korábban már látott speciális karaktereket láthatjuk. Például a 195-ös kód több kódtáblában 'Ã', de gyakori a 'Ă' is. A 437-es kódlap szerint, amelyet az eredeti IBM PC-k használtak, ez egy grafikai jel: ├. A helyes megjelenés próbálgatásánál pont ezeket láthattuk.

Feltűnő, hogy az első bájt általában 195-ös (-61 volt) értékű, de van két 197-es. Ez az 'ő' és az 'ű'. Az a két betű, amelyik a betűtípusoknál a legtöbb gondot okozza. A 195 és 197 két altáblát jelent. A vesszős, kalapos, kétpontos magánhangzók a 195-ös altáblában találhatók, de a kétvesszősnek nem jutott hely. Ha ezek bármelyike szerepel a Code::Blocks-ban írt kódban, akkor egy sárga felbukkanó ablak figyelmeztet az „illegális karakter"-re, ráadásul a `setlocale()` sem működik helyesen.

A bináris olvasó

A karakterkódok már értelmet kaptak, de hogyan lehet látni (olvasni) a láthatatlant, a szóközt és a vezérlő karaktereket? Lényegét tekintve a [0; 32] tartomány kódjait nem tudjuk megjeleníteni az extractor és inserter használatával.

30. példa: Szöveges fájl olvasása binárisan

Az értelmezés nélküli olvasáshoz „szólni kell", hogy binárisan szeretnénk majd olvasni.

```
 8.    ifstream fin("allatok.txt", ios::binary);
```

Továbbá nem szabad használni az extractort, mert azt úgy tervezték, hogy megtalálja a határoló karaktereket, amiket – mivel határolók és nem az adathoz tartoznak – eldob. Helyette a read() függvényt használjuk, ami „bájt sorozatot" olvas, C++ nyelven `char` tömböt. Mivel karakterenként szeretnénk olvasni, ezért létrehozunk egy egyelemű karakter tömböt a c karakter helyett:

```
 9.    char b[1];
```

Kiegészítés: Nem ez a normális megoldás, ez a már tanult fogalmakból építkező megoldás. Ha a dokumentációt nézzük, ott `b` típusa `char*`, azaz pointer. A tömböt is az első adatra mutató pointerrel adjuk meg, ezt használjuk most ki. Az adattípusra mutató pointernek azonban a tömbbel ellentétben nincs rögzített mérete, ezt a program futása során lehet lefoglalni, felszabadítani.

Továbbra is a ciklusfejben végezzük el a beolvasást, mert most is az ifstream üressége jelenti az ciklus befejezését.

```
10.    while (fin.read(b,1))
```

Itt a `b` a karakter helyét jelöli, ezért amikor a karakterre lesz szükségünk, akkor az tömb `b[0]` eleme lesz. A teljes eljárás:

```
 6.  void byteolvasas()
 7.  {
 8.    ifstream fin("allatok. txt", ios::binary);
 9.    char b[1];
10.    while (fin.read(b,1))
11.    {
12.      if ('A' <= b[0] && b[0] <= 'Z')
13.        cout << endl; /*nagybetű előtt új sort kezd*/
14.      if (('A' <=b[0] && b[0] <= 'Z') || ('a' <=b[0] && b[0] <= 'z') ||
             ('0' <= b[0] && b[0] <= '9'))
15.        cout << b[0];
16.      else
17.        cout << " " << (int)b[0] << " ";
18.    }
19.    fin.close();
20.  }
```

Az eredmény:

```
Active code page: 65001

H -61  -95 pi 32 kacsa 32 1 13  10
Morg -61  -77  32 kutya 32 12 13  10
Hekus 32 kutya 32 4 13  10
B -61  -87 g -59  -111  32 birka 32 3 13  10
F -59  -79 tfal 32 kecske 32 5 13  10
M -61  -70 z -61  -77 s 32 szarvasmarha 32 3 13  10
H -61  -83 n -61  -95 r 32 liba 32 2 13  10
K -61  -68 l -61  -74 nc 32 malac 32 1 13  10
```

Az ékezetes karakterek egész szám értékét kiírva most is negatív szám lesz az eredmény, de most nem lehet a **char** helyett **unsigned char**, mert azt a `read()` nem ismeri. Megjelent a szóköz kódja: 32 és az enter helyén a 13-as (`'\r'`, CR, Carriage Return) és 10-es (`'\n'`, LF, Line Feed).

Ha a 17. sorban az egész helyett a karaktert írjuk ki, akkor minden szóköz helyett két szóköz, minden enter helyett két enter jelenik meg. Ráadásul a sorok első két betűjét le is törli.

```
Active code page: 65001

  ♦  ♦ pi   kacsa   1

 rg ♦  ♦    kutya   12

 kus   kutya   4
```

De miért?!?

Lépésenként futtatva, látható, hogy kiírja a sort (például „Hekus kutya 4” Ezután jön a `'\r'`, ami előtt és után is kiír 1-1 szóközt. Csakhogy a `'\r'` visszaviszi a kurzort a sor elejére, ezért az utána következő szóköz felülírja a korábban beírt első karaktert, majd a `'\n'` előtti szóköz felülírja a második karaktert még a soremelés előtt.

A soremelést a következő sor első nagybetűje követi, ami a 13. kódsor miatt még egy **endl**-t is tartalmaz. Ezért két soremelést követően a sor elején folytatódik a kiírás. Gondoljuk végig és próbáljuk ki, mi lesz a konzolon, ha a 13. sorban új sor helyett egy betűt írunk ki!

Kitekintés: Bináris fájlok olvasása és írása

Ez a fejezet tényleg nagy kitekintés lesz a tankönyvi anyaghoz képest. Az előző fejezetben láthattuk, a legegyszerűbb adatnak, a karakternek az olvasása, írása és értelmezése is programozói kihívás. Az extractor (`>>`) és az inserter (`<<`) e kihívás jelentős részére ad megoldást,

nekünk csak a specialitások figyelembevétele marad. Konzolon keresztül minden esetben karaktersorozattal folyik a kommunikáció, ezt a kommunikációs formát azonos módon át lehet tenni szövegfájlok kezelésére. Ebben a fejezetben azt nézzük meg, – akár megértve, akár csak átmásolva a megfelelő kódrészleteket, – hogy hogyan lehet binárisan tárolt adatokat beolvasni, értelmezni. … és ha már idáig eljutunk, akkor módosítani és kiírni. „Látványos" bináris fájlunk a `bmp` formátumú kép lesz.

Amikor egy fájlban binárisan tárolunk adatokat (adatot, nem programéli utasításokat), akkor a tárolt bájtok a programban használt formátumban őrizik meg az adatot. Az egybájtos egész 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 bájtsorrendben történt: little-endian[4] 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 bájtsorrendben vannak a bájtok (little-endian vagy big-endian) és milyen típusúként 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 (a 32-es kód nyomtatási szempontból határeset, ez a szóköz). Ezeket az egybájtos értékeket (nyomtatható karaktereket) egy binárisan kódolt fájlból karakterként kiolvasva ugyanazt kapjuk, mintha szöveges fájlból olvasnánk ki. Ezért egy binárisan kódolt fájlt 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 szövegként 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 környezetfüggő, például „□`[NUL][NUL][NUL]`" lehet a szöveges megjelenése; a `260`-nak `[EOT][SOH][NUL][NUL]`, mert az `[EOT]` binárisan `00000100`, a `[SOH]` binárisan `00000001`, így a négy bájtot little-endian sorrendben értelmezve 4 + 1*256 + 0 + 0.

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 színkészlettel mentjük el (fekete-fehér, 16 színű vagy 256 színű képként). A 24 bites bitkép a képernyőn látható kép adatait változtatás nélkül tartalmazza, ezért fogunk `bmp` képeket vizsgálni.

[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).

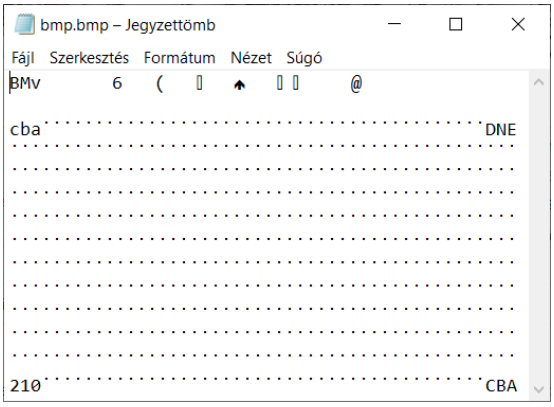
Bármelyik képünket elmenthetjük `bmp` formátumban, de minél nagyobb és színesebb egy kép, annál nehezebb kiismerni magunkat a rengeteg bájt között. Ezért vizsgálódásunkhoz készítünk egy kicsi, vélhetően jellemző jeleket tartalmazó képecskét `bmp.bmp` néven.

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 néggyel, akkor lyukat hagy a mentéskor. Ugyancsak praktikus viszonylag kis méretet beállítani, hogy kevés kódot kelljen értelmeznünk.

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

Először szövegszerkesztőben szeretnénk majd megnyitni és látni a bájtok értékét, ezért színkomponenseknek válasszuk nyomtatható karakterek kódjait. Például legyen a bal-fölső pixel RGB kódja (48, 49, 50), ezek a 0, 1, 2 ASCII kódjai; a jobb-fölső 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 színek sem mondanak sokat, sötétszürke pontok …

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



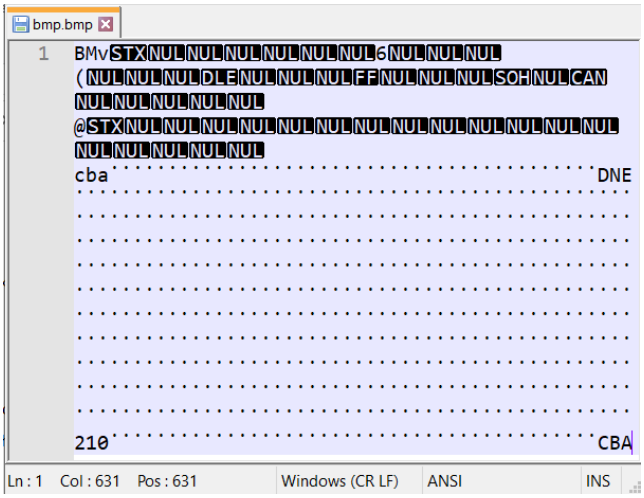
Az értelmezhető karakterek:

- 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 közbenső 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 kódok alapján tett megfigyeléseinket ellenőrizhetjük, ha rákeresünk az interneten a BMP-formátum leírására. A `*.bmp` fájl elején fejlécek találhatók, ami a kép tulajdonságairól tájékoztatnak, ezt követi a bitmap, a pixelek táblázata. Az elmentett képen nem RGB, hanem ábécé sorrendben, BGR sorrendben vannak elmentve a színkomponensek. Az is a szabvány része, hogy az adatokat soronként kezeli, és lentről felfelé veszi őket sorra (mintha koordináta-rendszerben nézné, mentéskor alul van a 0. sor).

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



Előforduló vezérlő karakterek:

- 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 jegyzettömbben láthatókkal.

A vezérlőkarakterek kódjaiból következtethetünk a tartalomra is: A DLE a szélesség, az FF a magasság adata, a CAN valószínűleg a színmélység jelölése. Ennél pontosabb értékeket akkor kapunk, ha figyelembe vesszük az adatok méretét, azt, hogy egy adatot hány bájton tárolunk.

- 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: 118+2*256+ 0 + 0 .
- A „6[NUL][NUL][NUL]” értéke ugyanígy számolva 54. Ez a teljes fejléc mérete, másképp: az 54. bájton kezdődik a bitmap. Ez Notepad++-ban ellenőrizhető, ha a ’c’ elé kattintunk, a státuszsorban láthatjuk, hogy az 55. karakter következik 1-től számolva.
- A „[DLE][NUL][NUL][NUL][FF][NUL][NUL][NUL]” egymás után két szám, a kép szélességének és magasság értéke.
- 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 + 0.
- Egy 4 bájtos 0 érték után jön a „@[STX][NUL][NUL]”: 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 fájlméret 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.

- Készítsünk több példányban más-más néven mentést a bmp.bmp-ről!

4. Nyissuk meg jegyzettömbben 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 fájlsérülést 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 bájtjait jelölő pöttyökről nem látjuk, hogy valóban milyen karakterek.

Fehér pixel? Három 255-ös érték. 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.

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

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 színkomponensek értékét. Csak ellenőrzésképpen néhány adatot kiírunk konzolra. Végül nem felejtjük el bezárni a fájl olvasót.

```
1.  #include <iostream>;
2.  #include <fstream>
3.  using namespace std; //…
4.  struct pixel        /*ebből van a bitmap*/
5.  {
6.    char R;
7.    char G;
8.    char B;
9.  };
10.
11. int main()
12. {
13.   setlocale(LC_ALL, "Hun");
14.   cout << "bmp.bmp fájl olvasása, Red kiírása" << endl << endl;
15.   ifsream fs("bmp.bmp", ios::binary);
16.   char fej[54];
17.   fs.read(fej, 54); /*ha nem fog változni, így is jó*/
18.   pixel bitmap[12][16];
19.   char in[3];
```

```
20.   for (int sor = 11; sor >= 0; sor--)        /*első sor az alsó…*/
21.     for (int hely = 0; hely < 16; hely++)
22.     {
23.       bfin.read(in, 3);
24.       bitmap[sor][hely].B = in[0];
25.       bitmap[sor][hely].G = in[1];
26.       bitmap[sor][hely].R = in[2];
27.     }
28.   for (int sor = 0; sor < 12; sor++)          /*elejétől végig bejár*/
29.   {
30.     for (int hely = 0; hely < 16; hely++)
31.       cout << bitmap[sor][hely].R << " ";     /*csak az R kiírása*/
32.     cout << endl;
33.     }
34.  bfin.close();
```

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 a méretet. Erre a következő példában látunk profibb megoldást.

A 30–35. sorban az R értékeket azért érdemes kiírni, mert így tudjuk ellenőrizni, hogy tényleg a jól tároltuk el az adatokat. Az R komponens: 0, A, a, E. Ezekből és helyes sorrendjükből már lehet következtetni a többi érték helyességére is.

Mindegyik pixelből célszerű kiírni egy adatot és a méretnek megfelelően sorokra bontani, mert így ellenőrizhető, hogy jók-e a ciklusok paraméterezése. A 34. sor csak a soronkénti kiírás miatt kell.

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

```
35.   for (int sor = 1; sor < 6; sor++)
36.     for (int hely = 1; hely < 9; hely++)
37.     {
38.       bitmap[sor][hely].R = -1;         /*pozitív byte értékkel 255*/
39.       bitmap[sor][hely].B = -1;         /*eddig is 255 volt mindhárom*/
40.       bitmap[sor][hely].G = 0;
41.       if (sor != 3 || hely < 3 || hely > 6)
42.         bitmap[sor][hely].B = -128;     /*pozitív byte érték: 128*/
43.     }
```

Mivel tudjuk, hogy kezdetben a pixelek nagyrésze fehér, ezért a 41. és 42. 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 3–6. helyén. A többi helyen kevesebb lesz a kék, így ott pirosabb lesz a téglalap.

Az eredmény fájlba írása – a beolvasás után – már gyerekjáték.

```
44.   ofstream bfout("piros.bmp", ios::binary);
45.   bfout.write(fej, 54);
46.   for (int sor = 11; sor >= 0; sor--)
47.    for (int hely = 0; hely < 16; hely++)
48.    {
49.     bw.Write(bitmap[sor, hely].B);
50.     bw.Write(bitmap[sor, hely].G);
51.     bw.Write(bitmap[sor, hely].R);
52.    }
53.   bfout.close();
54.  }
```

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

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 19-20. sorát az itt látható 12 sorral (ami valójában 24 sor) helyettesíthetjük:

```
char bm[2];        bfin.read(bm, 2);              //BM
int fmeret;        bfin.read((char*)&fmeret, 4);  //630  v.STX.NUL.NUL
char szabad[4]; bfin.read(szabad,4);              //0
int mapkezd;    bfin.read((char*)&mapkezd, 4);//54     6.NUL.NUL.NUL
int infsize;    bfin.read((char*)&infsize, 4);//40     (.NUL.NUL.NUL
int Szeles;        bfin.read((char*)&Szeles, 4); //16  DLE.NUL.NUL.NUL
int Magas;      bfin.read((char*)&Magas, 4);  //12  FF.NUL.NUL.NUL
short ki;       bfin.read((char*)& ki, 2);        //1   SOH.NUL
short bppix;    bfin.read((char*)& bppix, 2);  //24  CAN.NUL
int tomore;        bfin.read((char*)&tomore, 4); //0
int mapsize;    bfin.read((char*)&mapsize, 4);//576  @.STX.NUL.NUL
char egyeb[16]; bfin.read(egyeb, 16);     //0
/*ellenőrizzük néhány adat kiírásával*/
cout<<bm[0]<<bm[1]<<" "<< fmeret <<" "<< Szeles <<" "<< Magas << endl;
```

A megoldás inkább munkaigényes, mint nehéz. Pontosan kell követni a BMP fájl specifikációját és a sormintát.

Kiegészítés: Arról már volt szó, hogy a `read()` első paramétere pointer és arról is, hogy a tömbök neve is az. A többi adattípusnál a változónév az adatot nevezi meg. Ha a címét, referenciáját – azt az adatot, ami rámutat szeretnénk megkapni, akkor a változónév elé kell írni az `&` referencia jelet. Ez a jelölés megfelel a függvények paramétereinél használt jelzésnek. Nagyobb trükk az előtte lévő `(char*)`. A zárójelben lévő adattípus az „explicit konverzió” jelölése, azaz megadjuk, hogy – akárhogy is, de – az utána levő adat típusát módosítsa a zárójelben lévőre. Jelen esetben az általában egy egész szám típusú adatra mutató referenciát tekintsen karaktersorozatra mutató pointernek.

Ezt követően választhatunk, hogy egy nagyon nagy bitmap-tömböt hozunk létre, aminek a bal felső sarkában elhelyezzük a pixeleket vagy dinamikusan, `vector<>`-ral foglalunk le pontos méretű területet.

1. Ha tömböt használunk, akkor azt a nagy mérete miatt a `main()` előtt(!) kell lefoglalni, jó nagy konstans méretet megadva:

```
11.  pixel bitmap[768][1024];
```

   Ezt követően a kódban le lehet cserélni a 12 és 16 értékeket `Magas`, illetve `Szeles` változókra (és a 11-es `Magas - 1`-re).

2. A `vector<>` használatához a 3. sorban fel kell venni a `<vector>` csomagot. A fej beolvasását követően létrehozzuk a kétdimenziós „listában listát”. Rögtön a pixeleket is inicializáljuk, így nem lesz szükség a `push_back()`-re, csak a már létező pixelek módosítására:

```
11.  vector<vector<pixel>> bitmap(Magas, vector<pixel> (Szeles, {0,0,0}));
```

   Ennél a megoldásnál is lecseréljük a konstansokat a `Magas` és `Szeles` változókra.

Mindkét megoldás lehetőséget ad arra, hogy módosítsuk a fájl méretét (a tömb korlátozott méretig, de készen áll, a `vector<>` „korlátlanul” bővíthető a `push_back()` függvénnyel). Azonban ez elronthatja a fájlt, mivel a mérettel együtt a `fmeret` és a `mapsize` is változik. További veszélyt rejt, ha a módosítás során nem osztható 4-gyel az egy sorban lévő bájtok száma, mert ilyenkor a sorvégeket megfelelő számú 0 kódú bájttal ki kell egészíteni, ami szintén hatással van a fájlméretre. 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 bájton, így már `mapkezd` 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 fehér hozzáadott képpont esetén párosra, a fekete hozzáadott képpont esetén páratlanra állítjuk. Aki tudja az elrejtés szabályát, az ki tudja nyerni az információt.

A módosított kép mentésekor a fejlécet pontosan kell kiírni. Az eredeti 45. sor helyett ez újabb 12 sor:

```
    bfout.write(bm, 2);                     //BM
    bfout.write((char*)&fmeret, 4); //630   v.STX.NUL.NUL
    bfout.write(szabad,4);                  //0
    bfout.write((char*)&mapkezd, 4);        //54    6.NUL.NUL.NUL
    bfout.write((char*)&infsize, 4);        //40    (.NUL.NUL.NUL
    bfout.write((char*)&Szeles, 4); //16    DLE.NUL.NUL.NUL
    bfout.write((char*)&Magas, 4);          //12    FF.NUL.NUL.NUL
    bfout.write((char*)& ki, 2);            //1     SOH.NUL
    bfout.write((char*)& bppix, 2); //24    CAN.NUL
    bfout.write((char*)&tomore, 4); //0
    bfout.write((char*)&mapsize, 4);        //576   @.STX.NUL.NUL
    bfout.write(egyeb, 16);         //0
```

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 előállítása 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 „matematikus” megfogalmazással mondhatjuk úgy is, hogy az adatsorozat elemeihez valamilyen hozzárendelési szabállyal, hozzárendelő függvénnyel másik elemet rendelünk.

Mindez mondatszerű 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 feltétlen 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, az egészeket tartalmazó tömb és az elemszám. 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.

```
 5. void ki(string szeparator, int adatsor[], int N)
 6. {
 7.   string ki = to_string(adatsor[0]); /*N > 0*/
 8.   for (int i = 1; i < N; i++)
 9.     ki += szeparator + to_string(adatsor[i]);
10.   cout << ki;
11. }
```

2. Szövegként megadott számok értékké alakítása

Begépelésekor a program szövegként kapja meg az adatokat. Ezeket egyenként, másolás típusalgoritmus használatával tehetjük be a kívánt típusú elemeket tartalmazó tömbbe. A megoldás most is eljárás lesz, az eredményt egy létező tömbben tároljuk:

```
12. void masol1(string adatsor[], int N, int kimenet[])
13. {
14.   for (int i = 0; i < N; i++)
15.     kimenet[i] = stoi(adatsor[i]);
16. }
17.
```

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 készítünk `stringstream` és a >> extractor operátor felhasználása. Eközben kétszer használjuk a másolása algoritmusát. (A megoldáshoz szükséges a `<sstream>` csomag.

```
18. void masolasthasznal()
19. {
20.   string adatsor = "1 5 2 3 4";
21.   stringstream ss(adatsor); //karakterenként bemásolja a sstreambe
22.   string adatok[50];
23.   int db = 0;
24.   while(ss >> adatok[db]) db++; //szavanként kimásolja az adatokba
25.   int szamok[50];
26.   masol1(adatok, db, szamok); //szövegeket egészekre másol
27.   ki(" ", szamok, db);
28. }
```

4. Adatsorozat stream-be másolása

Végül egy nagyon hasznos, de nyelvspecifikus megoldás az eredmények kiírásához. Az inserter (`<<`) operátort „meg lehet tanítani” arra, hogy egy adott típusú adatot hogyan írjon ki egyetlen változóként. Példánkban egy `vector<int>` elemeit pontosvesszővel és szóközzel elválasztva írjuk ki, de bármilyen más adatsorozattal is hasonló a megoldás.

```
29.  ostream& operator<<(ostream& os, vector<int> v)
30.  {
31.    os << v[0];
32.    for (unsigned i = 1; i < v.size(); i++)
33.      os << " " << v[i];
34.    return os;
35.  }
```

Ezt követően egy `vector<int>` kiírása például a főprogramban:

```
    vector<int> lista{1, 3, 5, 2, 4};
    cout << lista << endl;
```

Tömb kiírására ez a megoldás még nem lesz jó, mert tudni kellene a kiírandó adatok számát is. Ezt vagy konstansként írjuk be a függvénybe, vagy valamilyen módon egy adatba kell szervezni a tömböt és elemszámát. számokat tartalmazó tömb esetén fel lehet használni erre a tömb 0. elemét (1-től számozva az adatokat). Lehet `struct`-ba (`class`-ba) tenni a darabszámot és a tömböt – ez a `string` típus megalkotásának az elve. Vagy használhatunk a célnak megfelelő speciális – már megírt – adatstruktúrát: `map`-ot vagy `tuple`-t. Ez utóbbira a következő példában látható megoldás.

34. példa: Kiegészítés: Másolás haladó nyelvi eszközökkel

Ahogy a korábbi típusalgoritmusoknál is láthattuk, a C++ nyelvben az egyes gyakran használt algoritmusok kódolása helyett a konténer – például `vector<>` – típusú adatsorozatokhoz „előregyártott” függvényeket is használhatunk. Ahogy eddig is, ezek ismerete nem tananyag, de van, aki nagyon szereti használni. A példákhoz az alábbi kiegészítő csomagokra lesz szükség:

```
1.  #include <vector>
2.  #include <cmath> /*csak a négyzetgyök számítása miatt*/
3.  #include <algorithm> /*típusalgoritmusok csomagja*/
4.  #include <numeric> /*az accumulate-hez kell*/
5.  #include <map> /*a tuple (értékpár) ebben van*/
6.  using namespace std;
   /*itt lesznek az eljárások és függvények… utána:*/

7.  int main()
8.  {
9.    setlocale(LC_ALL, "Hun");
10.   string adatsor[]{"1", "5", "2", "3", "4"};
11.   masolasok(adatsor, 5); /*kipróbálás helye*/
12.   return 0;
13.  }
```

A példákat egy – `masolasok()` – eljárásba tesszük, de előtte megírjuk az egyes feladatokhoz tartozó „hozzárendelő függvény”-eket.

Ahhoz, hogy `string`eket tartalmazó adatsorozatból számokat tegyünk egy `vector<int>`-be, érdemes megírni az átalakító-hozzáfűző függvényt.

```
14.  void add(vector<int>& v, string s) /*& jel miatt v-be teszi s-t*/
15.  {
16.    v.push_back(atoi(s.c_str()));
17.  }
```

Az adatsorozat elemeinek szöveggé alakítása és összefűzése (konkatenációja) mindig

problémás az adatok közötti elválasztó jel miatt. A művelet előkészíthető egy hozzáfűzés függvénnyel:

```
18.  string fuz(string s, int b) /*trükk: a kimenet elé is az s-t írjuk*/
19.  {
20.    return s+ ", " + to_string(b);
21.  }
```

Rögtön alkalmazzuk is. A `join()` függvényünkben lényegében összegzést alkalmazunk. Filozófiai kérdés, hogy most karaktersorozatot hozunk létre számokból (másolás típusalgoritmus) vagy egy szöveges adatot (összegzés típusalgoritmus).

```
22.  string join(vector<int>& vi) /*trükk: a kimenet elé is az s-t írjuk*/
23.  {
24.    return
        accumulate(next(vi.begin()), vi.end(), to_string(vi[0]), fuz);
25.  }
```

Magyarul: a sorozat 0. eleméhez hozzágyűjti a sorozat eleje utánitól a sorozat végéig az elemeket. A gyűjtéshez „gyűjtő @=adat” jellegű függvényt kell megadni. Ez most a `fuz()`.

Ha most visszanézzük az inserter (`<<`) „okosító” függvényünket, láthatjuk, hogy ott is hasonló műveletet adtunk meg. Az `operator@(left, right)` függvény a `left @ right` megfelelője. Operátor minden, ami a kódokban piros. Ezek jellemzően a szomszédaikkal operálnak. Többségük valójában kétváltozós függvény, ahol megszokásból a két változó közé írjuk az operátor, azaz a műveletvégzés jelét. Egyváltozós operátor például a tagadás (`operator!`), a növelés (`operator++`), a negálás (`operator-`). Jelentős a páros operátorok száma is, melyek a `right` változót közrefogják. Ilyen például az indexelő `operator[]`, ami az egyváltozós `operator*` műveletet ötvözi egy címszámítással.

Az egyik legismertebb műveleti jel az `+`. Az `operator+()` függvényt nagyon sok adattípusra megírták. Használjuk az egészek, lebegőpontos adatok, ezek keverékének az összeadására, szövegek és karakterek összefűzésére. Programozók írták a különböző értelmezéseit, és programozható rá újabb értelmezés is – ezt nevezik egy függvény **overloading**-jának. Például törtek, síkbeli vektorok, emberek összeadását is megadhatjuk, de elvárt, hogy értelmes legyen a programunk, ha a ’+’ jelent valamit, akkor a ’+=’ is értelemszerűen jelentsen valami hasonlót …

Az inserter és extractor operátorok a C++ nyelv jellegzetességei. E két operátornak kötött – i/o stream – a visszaadott adat típusa és a `left` adata. (Más környezetben egész más műveletet végez: biteket shiftel.) Az egymásutáni végrehajtások lehetőségét az adja, hogy a visszaadott érték és a `left` paraméter referencia (`&`) a streamre, ezért az menet közben módosul és utána a módosult stream lesz az eredmény, ami egyben a következő használat paramétere is.

Ahogy korábban jeleztük, adjunk új értelmet az inserternek. Ebben a megoldásban egy `tuple` típusú adatot fogunk kiírni, aminek a második adata egy tömb, az első egy szám – a tömbben tárolt adatok száma. Bár most a tömb is egészeket tárol, ez könnyen lecserélhető bármilyen más típusra. A megoldás – a megértés kedvéért – túl beszédes:

```
26.  ostream& operator<<(ostream& os, tuple<int, int*> t)
27.  {
28.    os << get<0>(t) << " db adat:";
29.    for (unsigned i = 0; i < get<0>(t); i++)
30.      os << ", " << get<1>(t)[i];
31.    return os;
32.  }
```

Írjunk egészeket tároló tömb beolvasására is egylépéses megoldást! Ehhez egy másik logikát használunk: A beolvasandó első adat lesz a további adatok száma, ezt követően szóközzel vagy vezérlőkarakterekkel elválasztva írjuk be a számokat. A darabszámot a tömb 0. helyére írjuk, 1-től indexelve tároljuk az adatokat.

```
33. istream& operator>>(istream& os, int* t)
34. {/*db és db darab egész szóközzel, vezérlő karakterre elválasztva*/
35.   is >> t[0];
36.   for (unsigned i = 1; i <= t[0]; i++)
37.     is >> t[i];
38.   return is;
39. }
```

Teszteljük mindkét új operátorunkat a `main()`-ben! Sejthető, hogy a kétféle elven készült megoldásból nem lesz hibátlan az eredmény.

```
int tomb[100];
cin >> tomb;
cout << make_tuple(tomb[0], tomb) << endl;
cout << "utolsó adat:" << tomb[tomb[0]] << " ";
cout << "utolsó után:" << tomb[tomb[0] + 1] << endl;
```

Futtatás:

```
3       4
5 6 7 8 9
3 db adat:; 3; 4; 5
6 4941664
```

3 adat: 4, 5, 6. A 3 után TAB, a 4 után ENTER, a többi szám után szóköz van, végül a 9 után ENTER-rel véglegesítjük a bevitelt. Az első három adatot írja ki: 3, 4, 5 (0-tól indexelve). A 4. adat 6, utána memória szemét. A 7–9 a **istreamben** maradt.

A kiírás, beolvasás és szöveggé egyesítés után néhány adattípusok közötti másolás következik. Az eddigieket is felhasználva példák láthatók tömbből vektor létrehozására, listákból más adattípusú listába másolásra, listából tömbbe másolásra, és matematikai művelettel létrehozott másolatra. Két példában az adatsorozatot karaktersorozattá alakításához a függvény-paraméter helyére névtelen függvényt, **lambda-kifejezés**t adunk meg.

```
40. void masolasok(string adatsor[], int N)
41. {
42.   /*létrehozáskor tömbből listába másolás*/
43.   vector<string> adatlista{adatsor, adatsor + N};
44.   /*vectorok között: stringekből egészekre alakítás*/
45.   vector<int> szamlista;
46.   for_each(adatlista.begin(), adatlista.end(),
                [&](auto a){ add(szamlista, a);});
47.   /*listából tömbbe másolás*/
48.   int szamok[5];
49.   copy(szamlista.begin(),szamlista.end(), szamok);
50.   cout << szamlista << endl;                  /*vector kiírása*/
51.   cout << make_tuple(5, szamok) << endl;      /*tömb kiírása*/
52.   /*egészek karaktersorozatba másolása – fuz() használata*/
53.   string s = accumulate(next(szamok), szamok + 5, /*eleje, vége*/
                            to_string(szamok[0]), fuz); /*kezdőérték, fgv*/
54.     cout << "egy string " << s << endl;
55.   /*A szamlista minden elemének négyzetgyöke*/
56.   vector<double> gyoklista;
57.   for_each(szamok, szamok + 5),
                          [&](auto a){gyoklista.push_back(sqrt(a*1.0));});
```

```
58.   /*double típusú adatok karaktertömbbe másolása*/
59.   cout << "gyökök: " << accumulate(next(gyoklista.begin()),;/*2.-tól*/
        gyoklista.end(), to_string(gyoklista[0]), /*vége, kezdőérték:*/
        [](string& a, double b) /*lambda paraméterei:*/
         {return a + string(", ") + to_string(b);}); /*lambda definíció:*/
60.   cout<< endl;
61. }
```

Virtuózok (és akik a funkcionális programozást részesítik előnyben) szeretnek egész programokat függvénykombinációval, „egysorban” megadni. A C++ nyelv a C nyelv bővítésével keletkezett, annak a kifejezéseit megtartotta. Ezért az „egysoros” megoldások általában csak elméletileg vannak egy sorban.

Nagy a különbség a Python-féle kész függvények és a C++ függvényei között. A Python nyelv készítésekor a cél egy könnyen tanulható, használható egyszerű nyelv volt. A C++ megoldások lényege az általánosítás, a profi programozók számára egy hatékony eszköz biztosítása.

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 egészre kerekített 49 százalékáról lemondhat. Adjuk meg a taxis adózott bevételeinek a listáját!

```
5.  void taxis_bevetelek()
6.  {
7.  int bevetelek[5] { 1, 5, 2, 3, 4 };
8.    int[] adozott[5];
9.    for (int i = 0; i < 5; i++)
10.   {
11.     int ado = round(bevetelek[i] * 0.49);
12.     adozott[i] = bevetelek[i] - ado;
13.   }
14.   for (int i = 0; i < 5; i++)
15.     cout << adozott[i] << " ";
16. }
17.
```

Kódunk 9–13. sorában és a 14–15. sorában alkalmazzuk a másolás típusalgoritmust. A kerekítéshez a `<cmath>` csomagra van szükség.

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!

```
5.  void rokalibai()
6.  {
7.    int libak [5] { 1, 5, 2, 3, 4 };
8.    int sajat [5];
9.    for (int i = 0; i < 5; i++)
10.     sajat[i] = libak[i] > 3 ? 0 : libak[i];
11.   for (int i = 0; i < 5; i++)
12.     cout << sajat[i] << " ";
13. }
```

```
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 (kell a `<vector>`), 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 majd tömböt.

```
14.  struct allathang
15.  {
16.    string fajta;
17.    string hang;
18.    allathang(string f, string h)
19.    {
20.      fajta = f;
21.      hang = h;
22.    }
23.  };
```

A feladat megoldásához meg kell adnunk, hogy egy-egy állatfajta mit mond, erre készítjük el az `allathangja()` függvényt, állathangokra csak ezen belül lesz szükségünk.

```
24.  string allathangja(string allat)
25.  {
26.  allathang hangok[9] {
           {"kacsa","háp"}, {"kutya","vaú"}, {"birka","bee"},
           {"kecske","mek"}, {"szarvasmarha","mú"}, {"liba","gá"},
           {"malac","röf"},{"kakas","qqriq"}, {"macska","nyaú"}};
```

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 < 9 && hangok[ez].fajta != allat)
30.      ez++;
31.    if (ez < 9)
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, hogy ez másolás vagy az összegzés típusalgoritmusnak felel-e meg.

```
36.  string vec2str(string szeparator, vector<string> adatsor)
37.  {
38.    string ki = adatsor[0];
39.    for (int i = 1; i < adatsor.size(); i++)
40.      ki += szeparator + 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 (<fstream> is kell). 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.  void tanyahangok()
44.  {
45.    vector<string> koszontesek;
46.    ifstream fin ("allatok.txt");
47.    string allat, fajta, kor; /*kor nem kell => string*/
48.  setlocale(LC_ALL, "Hun");
49.    while (fin >> allat)
50.    {
51.      fin >> fajta >> kor;
52.      koszontesek.push_back(allathangja(fajta));
53.    }
54.    fin.close();
55.    cout <<vec2str(" ", koszontesek) << endl;
56.  }
```

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 típusalgoritmusa 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 algoritmikusbéli 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 típusalgoritmusa abban különbözik a kiválogatásétól, 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 elemsorozatunkat: az egyik gyűjteménybe kerülnek a kicsik, a másikba a közepesek, a harmadikba a nagyok.

38. példa: A farkas és a róka libalakomája

A korábban (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.

```
57.  void kiir(string ragadozo, int ltomb[], int N)
58.  {
59.    cout << "A " << ragadozo << "libái:";
60.    int szum = 0;
61.    for (int i = 0; i < N; i++)
62.    {
63.      cout << " " << ltomb[i] << ",";    /*előtte szóköz, utána vessző*/
64.      szum += ltomb[i];    /*ha már nézem, akkor összegzés tétele is */
65.    }
66.    cout << "\b" << endl; /*utolsó vessző törlése*/
67.    cout<<"A libák száma:"<< N <<", össztömege: "<< szum <<" kg."<<endl;
68.  }
```

A kiírás lényegében másolás típusalgoritmus, amit összevontunk az összegzés típusalgoritmussal, mert a feladat az összeget is kérte.

```
69.  void kiir(string ragadozo, vector<int> llista)
70.  {
71.    cout << "A " << ragadozo << "libái:";
72.    int szum = 0;
73.    for (unsigned i = 0; i < llista.size(); i++)
74.    {
75.      cout << " " << llista[i] << ",";
76.      szum += llista[i];
77.    }
78.    cout << "A libák száma: " << llista.size() << ", össztömege: " <<
                szum << " kg." << endl;
79.  }
```

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, továbbá egy változóban – külön – tároljuk, hogy a tömbben épp hány elem van. A farkas libáinak tárolásához – mivel ezt listába gyűjtjük – nem szükséges előzetesen a méret.

```
80.  void rokalibakette()
81.  {
82.    int libak[5] { 1, 5, 2, 3, 4 };
83.    int rokalibai[5];            /*akár mind beleférjen!*/
84.    int rlDB = 0;                /*ennyi libája van a rókának*/
85.    vector<int> farkaslibai;
86.    for (int i = 0; i < 5; i++)
87.      if (libak[i] <= 3)
88.      {
89.        rokalibai[rlDB] = libak[i]; /*pl. rlDB = 0. helyre 1-et*/
90.        rlDB++;                          /*utána már 1 libája van*/
91.      }          /*felülre a rókáét, alulra a farkasét*/
92.      else
93.        farkaslibai.push_back(libak[i]);
```

Figyeljük meg a `kiir()` eljárás használatakor, hogy a Code::Blocks mindkét használati formát felajánlja. Sokszor láttunk már ilyet, de csak akkor érdemes azonos nevet adni két függvénynek, ha ugyanúgy szeretnénk használni. Ellenkező esetben zavaró a névazonosság.

```
94.    kiir("róka", rokalibai, rlDB);
95.    kiir("farkas", farkaslibai);
96.  }
```

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: írt `vec2str()` függvénnyel írjuk ki az eredményt.

A feladatot a terveink alapján abban a projektben érdemes megoldani, amelyikben a `vec2str()` függvény van. Vagy átmásolhatjuk a kódot, vagy – ha minden kötél szakad, meg is írhatjuk újra.

```
97.  struct napiT
98.  {
99.    string nap;
100.   int fok;
101. };
```

Ha sok ehhez 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.

```
102. void hosegreiado()
103. {
104.   napiT elorejelzes [7]{ {"hétfö", 19}, /*nem szereti azz ő-t a C::B*/
105.                          {"kedd", 23}, {"szerda", 26},
106.                          {"csütörtök", 27}, {"péntek", 19},
107.                          {"szombat", 18}, {"vasárnap",18}
108.                        };
```

```
109.   vector<string> riadosnapok;
110.   for (int i = 0; i < 7; i++)
111.     if (elorejeles[i].fok > 25)
112.       riadosnapok.push_back(elorejelzes[i].nap);
113.   cout << "Höségriadós napok: " << vec2str(", ", riadosnapok) << endl;
114. }
```

Feladatok a kilenc típusalgoritmusra és a fájlok használatára

Ebben a fejezetben a tankönyv két – Kópiakészítés és digitális Hamupipőke – gyakorló feladatát dolgozzuk fel. Az eddig tanult kilenc típusalgoritmus: Eldöntés, Keresés, Kiválasztás, Összegzés, Megszámlálás, 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 részekre 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.

A programozási nyelvek és környezetek közötti eltérések, valamint a sokféle karakterkódolás miatt C++ nyelven a megoldás néha jóval nehezebb, ezért alaposabb megfontolást igényel a beolvasás módja (extractor vagy sor olvasás) és az ékezetes karakterek kezelése.

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 típusalgoritmusát érdemes 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. Kihívás, hogy az adatok egy bekezdésben, vesszővel és szóközzel vannak elválasztva.

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 dönteni, 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 (keresés típusalgoritmus) vagy használhatjuk a magánhangzókra a `<string>` csomagban lévő `s.find()` függvényt, ami megadja a találat indexét vagy az „`npos`” értéket. (A keresés megírása hasznosabb gyakorlásra, és lehet, hogy hamarabb megvan, mint a `find()`-ra a mintakód.)

A nagybetűket is kezelő változatban a bekért mondatból a `<cctype>` csomag `tolower()` függvényével lehet karakterenként kisbetűsé 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 gyűjtsük 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 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 allat.faj = „kutya” vagy állat.faj = „macska” akkor
       veszettség-et bővítjük allat.név-vel
     ha allat.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` fájlból[5] gyűjtsük ki azoknak a nevét, amelyeknél elképzelhető ilyen esemény! Az állatok nevéből ezúttal ne következtessünk a nemükre!

A kód eleje megegyezhet az előző feladat megoldásáéval, vagy – ha eljárást írtunk a beolvasásra, azt könnyen újra használatba vehetjük. Ezután a kigyűjtéshez – hasonlóan az eszperente nyelves 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 kacsa kétlábú, azaz rossz.
```

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!

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

(Azaz: esetleg fájlbeolvasás, utána szétválogatás és hozzá eldöntés típusalgoritmus.) Jelenítsük meg vesszővel elválaszott 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 használjuk. 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 'Ű' helyett UUE került a szövegbe. A mondatvégi írásjelek helyett a STOP karaktersor morzekódját küldték át az éteren vagy a kábelen, a mondat belsejében lévőkről pedig lemondtak.

A „Förgeteg közeledik, előz 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 előállítása. Szervezzük ezt ki függvénybe, amelynek paramétere a nagybetűs karakterlánc, visszatérési értéke pedig az átalakított karakterlánc! Az átalakítási szabályokat csak a függvénynek kell ismernie, a megfeleltetéseket 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 bontsuk 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. Különösen az ékezetes karakterek kezelését igénylő feladatoknál lehet hasznos, ha külön projektben készítjük el, mert a kódlap váltásának kellemetlen mellékhatásai lehetnek.

Ne felejtsük el, hogy a fájlból olvasáshoz a fájlokat elérhető helyre be kell másolni!

A névterekből elég azt beírni, amit éppen használunk, az alábbiakra számíthatunk:

```
#include <iostream>;
#include <vector>;
#include <fstream>;
#include <sstream>;
#include <cstdlib>;
#include <ctime>;
#include <string>;
#include <map>;
#include <algorithm>;
using namespace std;

```

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

```
int main()
{
 //seglocale(LC_ALL, "Hun");
 //system("chcp 1250");          //eszperente, tavirat
 //system("chcp 65001");         //kutya_macska
 //szintmeres();                 //40. Gyalogtúra
 //eszperente();                 //41 E terkes mecske…
 //eszperente2();                //41 nagybetűkre is
 //kutya_macska();               //42 Kutya- és macskaoltások
 //tojasrakok();                 //43 Tojásrakók
 //orvell();                     //44 Jók és rosszak
 //hajonaplo();                  //45 Hajónapló
 //Tavirat_1();                  //46 Távirat
 //Tavirat_2();                  //46 haladó
}
```

Tekintettel arra, hogy kódban is szerepel ékezetes szöveg, konzolról kimondottan ékezetes szöveget várunk és fájlból beolvasással járó feladat is van, a helyes megjelenést csak az éppen aktuális részfeladathoz tudjuk biztosítani.

40. példa: Gyalogtúra – mintamegoldás

A feladat megoldását megnehezíti a forrásfájl szerkezete, a vessző-szóköz elválasztás.

```
12.  void szintmeres()
13.  {
14.    const int maxN = 1000;
15.    ifstream fin("tura.txt");
16.    string adatok[maxN];
17.    int N = 0;
18.    while(getline(fin, adatok[N], ’,’))
19.    {
20.      N++;
21.      fin.ignore(1);    //eldobja a szóközt, hogy ne maradjon a szám előtt
22.    }
23.    fin.close();
```

A vesszőig beolvasáshoz `getline()` kell, de ez nem nyeli el a szóközt, ami akadályozza az `atoi()` működését. Egy újabb `getline()` lenne szükséges a szóköz kiolvasására, de a tárolt adatra nincs szükségünk. Az `ignore()` függvény ehhez praktikusabb, tárolás nélkül „eldobja” az adatot.

```
24.    int magassagok int[maxN];
25.    string iranyok string[maxN];        /*7 mérés között 6 szakasz!*/
26.    magassagok[0] = atoi(adatok[0].c_str());
27.    for (int i = 0; i < N - 1; i++) /*Másolás t.*/
28.    {
29.      magassagok[i + 1] = atoi(adatok[0].c_str());
30.      if (magassagok[i + 1] > magassagok[i]) /*rákövetkező feljebb*/
31.        iranyok[i] = "/";
32.      else if (magassagok[i + 1] < magassagok[i])
33.        iranyok[i] = "\\"; /*kettő \ kell*/
34.      else
35.        iranyok[i] = "=";
36.    }
```

A kiíráshoz – amíg nincs a kódban ő vagy ű – használható a setlocale. A ciklusban kiírt adatsor végén lesz egy felesleges szóköz. Akit ez zavar, az enter előtt kitörölheti a backspace-szel.

```
37.    cout << "A magasság változása: ";
38.    for (int i = 0; i < N - 1; i++)
39.      cout << iranyok[i] << " ";
40.    cout << ’\b’ << endl;
41.  }
```

Ha a 3 méteres szintkülönbségtől eltekintünk, akkor a kód két helyen módosul egy kicsit:

```
27.    for (int i = 0; i < N - 1; i++)
28.    {
29.      magassagok[i + 1] = atoi(adatok[0].c_str());
30.      if (magassagok[i + 1] > magassagok[i] + 3)       /*itt +3*/
31.        iranyok[i] = "/";
32.      else if (magassagok[i + 1] < magassagok[i] - 3)  /*itt -3*/
33.        iranyok[i] = "\\";
34.      else
35.        iranyok[i] = "=";
36.    }
```

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:

```
42.  bool bennevan(char c, string ebben)
43.  {
44.    unsigned ez = 0;
45.    while (ez < ebben.size() && ebben[ez] != c)
46.      ez++;
47.    return ez < ebben.size();
48.  }
49.
```

A „fordítás eszperente nyelvre” másolás típusalgoritmussal:

```
50.  void eszperente()
51.  {
52.    cout << "Kérek egy mondatot: ";
53.    string magyar_mondat; getline(cin, magyar_mondat); /*nem 1 szó!*/
54.    string eszperente_mondat = "";
55.    string maganhangzok = "aáeéiíoóöőuúüű";
56.    for (char c : magyar_mondat)
57.      if (bennevan(c, maganhangzok))
58.        eszperente_mondat += 'e';
59.      else eszperente_mondat += c;
60.    cout << "Gyenge eszperente: " << eszperente_mondat << endl;
61.  }
62.
```

Az ékezetes karakterek – különösen az ő és ű – konzolról történő beolvasásához állítsuk a konzol kódlapját 1250-re! A `main()`-be (lényeg, hogy a `cin` használata elé) írjuk be:

```
    system("chcp 1250");
```

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 karaktert, ezért a nagybetűk külön listában (szövegben) felvétele a legjobb megoldás. A karakterenkénti átalakítás kisbetűsre nem csak bonyolultabb, de az ékezetes betűkre nem ad helyes eredményt.

A `bennevan()` függvényt nagyon célszerű megírni, mert itt kétszer felhasználható.

```
63.  void eszperente2()
64.  {
65.    cout << "Kérek egy mondatot: ";
66.    string magyar_mondat; getline(cin, magyar_mondat);
67.    string maganhangzok = "aáeéiíoóöőuúüű";
68.    string Maganhangzok = "AÁEÉIÍOÓÖŐUÚÜŰ";
69.    string eszperente_mondat = "";
70.    for (char c : magyar_mondat)
71.      if (bennevan(c, maganhangzok))
72.        eszperente_mondat += 'e';
73.      else if (bennevan(c, Maganhangzok))
74.        eszperente_mondat += 'E';
75.      else eszperente_mondat += c;
76.    cout << "Erős eszperente: " << eszperente_mondat << endl;
77.  }
```

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]`.

```
80.  struct allat
81.  {
82.    string nev;
83.    string faj;
84.    int kor;
85.    allat(string sor) /*beolvasott adatsorra írt konstruktor*/
86.    {
87.      stringstream ss(sor);
88.      ss >> nev >> faj >> kor;
89.    }
90.  };
91.
```

A mondatszerű 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.

Kiválogatás és szétválogatás esetén sokkal hatékonyabb egy `vector<>`, mint egy tömb. Itt az is a `vector<>` mellett szól, hogy a feladatban semmilyen utalás nem található az állatok számára. Nem csak a kimenet adatsorainak az elemszámait nem ismerjük, de a beolvasandó adatok számát sem.

Mivel két listát kell a végén kiírni, célszerű a példafeladatoknál szereplő `vec2str()` függvényt itt is használni.

```
93.  string vec2str(string szeparator, vector<string> adatsor)
94.  {
95.    string ki = adatsor[0];
96.    for (unsigned i = 1; i < adatsor.size(); i++)
97.      ki += szeparator + adatsor[i];
98.    return ki;
99.  }
```

A fájl olvasásához és a kódba írt szöveg helyes megjelenítéséhez várhatóan a 65001-es kódlap lesz alkalmas.

```
100.  void kutya_macska()
101.  {
102.    ifstream fin("allatok.txt");
103.    vector<string> veszett, parvo;
104.    string sor;
105.    while (getline(fin, sor))
106.    {
107.      allat a (sor);
108.      if (a.faj == "kutya" || a.faj == í"macska")
109.      {
110.        veszett.push_back(a.nev);
111.        if (a.faj == "kutya")
112.          parvo. push_back (a.nev);
113.      }
114.    }
```

```
115.    fin.close();
116.    cout << "Veszettség ellen kap: " << vec2str(", ", veszett) << endl;
117.  }
```

43. példa: Tojásrakók – mintamegoldás

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

```
119.  bool egyike(string ez, vector<string> lista)
120.  {
121.    for (string l : lista)
122.      if (l == ez)
123.        return true;
124.    return false;
125.  }
```

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

```
126.  void tojasrakok()
127.  {
128.    ifstream fin ("allatok.txt");
129.    vector<string> tojok {"kacsa", "liba", "kakas"};
130.    vector<string> tojtNevek;
131.  string temp;
132.    while (getline(fin, temp))
133.    {
134.      allat a(temp);
135.      if (egyike(a.faj, tojok))
136.        tojtnevek.push_back(a.nev);
137.    }
138.    fin.close();
139.    cout << "Tojásrakó fajhoz tartozik: ";
140.    for (unsigned i = 0; i < tojtnevek.size(); i++)
141.      cout << tojtnevek[i] << ", ";
142.    cout << "\b\b." << endl;
143.  }
```

44. példa: Jók és rosszak – mintamegoldás

Ebben a megoldásban is felhasználható az előző feladat `egyike()` függvénye. A `vec2str()` függvényt a jegyzetben előrébb lehet megtalálni. Ami azon túl marad az fájlból olvasás és a szétválogatás típusalgoritmus alkalmazása.

```
144.  void orvell()
145.  {
146.    ifstream fin("allatok.txt");
147.    vector<string> tojok vector<string>(){"kacsa", "liba", "kakas"};
148.    vector<string> jok vector<string>();
149.    vector<string> rosszak vector<string>();
150.    string temp;
```

```
151.   while (getline(fin, temp))
152.   {
153.     allat a(temp);
154.     if (egyike(a.faj, tojok))
155.       rosszak.push_back(a.nev);
156.     else
157.       jok.push_back(a.nev);
158.   }
159.   fin.close();
160.   cout << "Rosszak: " << vec2str(", ", rosszak) << "." << endl;
161.   cout << "Jók: " << vec2str (", ", jok) << "." << endl;
162. }
```

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. De, ha az adatsorozat tárolására tömböt használunk, akkor az adatsor hiánya nem okozhat gondot. Ezért a konstruktorok írásakor mindkét lehetőségre fel kell készülni. Ebben a példában ráadásul az adatsor speciális, ezért kérdéses, hogy érdemes-e erre alapozni a konstruktort – az adatok létrehozását – vagy inkább a beolvasás során bontsuk fel és egyenként adjuk meg az adatokat. Itt erre mutatunk példát.

```
163. struct haladas
164. {
165.   int irany;
166.   int tav;
167.   haladas(int ir = 0, int hossz = 0) /*ha nem adom meg, akkor 0*/
168.   {
169.     irany = ir;
170.     tav = hossz;
171.   }
172. };
```

A mondatszerű leírásban megadott részletből függvény lesz.

```
173. string BJ(int regi, int uj)
174. {
175.   srand(time(0));              /*Ezt jobb lenne a main()-be írni!*/
176.   if ((uj - regi + 360) % 360 < 180)
177.     return "J";
178.   else if ((uj - regi + 360) % 360 > 180)
179.     return "B";
180.   return rand() % 2 == 0 ? "B" : "J";
181. }
```

A tankönyvi feladatok csak listára írt megoldásra készültek, mert a Pythonban nincs tömb adatstruktúra. Ezért – sajnos – nincs utalás az adatsorok lehetséges maximális számára. A `vector<>` ismerete nem alapkövetelmény, tömböt használva a C++ nyelvben előírás a fordításkor ismert méret. Ezért – a feladatból következő realitást figyelembe véve adunk felső korlátot az adatok számára.

Fontos, hogy az általunk megadott érték a kódban könnyen megtalálható, módosítható legyen, ezért konstansként vesszük fel.

Az „elegendően nagy” tömb létrehozásával együtt ját, hogy a benne lévő elemek számát külön változóban tároljuk.

```
183.  void hajonaplo()
184.  {
185.    const int MaxN = 1000;
186.    ifstream fin("hajonaplo.txt");
187.    haladas naplo haladas[MaxN];
188.    int N = 0;
189.    string temp;
190.    while (getline(fin, temp, '-'))
191.    {
192.      naplo[N] irany = atoi(temp.c_str());
193.      fin >> naplo[N] tav;
194.      getline(fin, temp)
195.      N++;
196.    }
197.    ofstream fout ("kormanyzasok.txt");
198.    for (int i = 1; i < N; i++)
199.      fout << naplo[i - 1].irany << " " <<
              BJ(naplo[i - 1].irany, naplo[i].irany) << " "<<
              naplo[i].irany << endl;
200.    fout.close();
201.  }
202.
```

A megoldás kiírásakor figyelni kell az indexelésre: az egymás utáni elemekkel számolás során ne akarjunk a 0. előtti vagy az utolsó utáni adattal dolgozni. Ne lepődjünk meg, ha a képernyőn nem jelenik meg semmi, mert az eredményt fájlba írtuk.

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.

Code::Blocks környezet használata esetén

Mivel a kódba írjuk az ékezetes betűket és konzolon kérjük be az ékezetekkel írt szöveget, az 1250-es kódlapot kell beállítani. Emellett:

- a programot futtató konzolablak betűtípusa (legjobb a Consolas, futtatás során a konzol ablak tulajdonságai – win10: settings \ Defaults \ Apperance – lapon állítható be);
- az ő, ű, Ő és Ű betűk használatát kerüljük a kommentben és inaktív kódban is.
- a kód kódolása: Edit \ File encoding: System default legyen (a sárga figyelmeztető ablak ennek utf-8-ra módosítására hívja fel a figyelmet, vissza kell állítani).
- a kódban az ő, ű, Ő, Ű karakterek ne legyenek pirosan aláhúzva (System default kódolás mellett);
- az Õ, Û, õ, û karaktereket cseréljük le kétvesszősre (System default kódolás mellett);

Ha akár csak egyetlen utf-8 karakter marad a kódban, akkor a fordító System defaultról átvált utf-8-ra, de a konzol nem változik, ezért nem ismeri fel az ékezetes karaktereket, amelyek így rosszul jelennek meg vagy egyáltalán nem jelennek meg.

A C++ `toupper()` függvénye csak az ASCII karakterek átalakítását végzi el, ezért az ékezetes karakterekből a kis- és nagybetűs változatot is fel kell venni.

```
203.  void tavirat_1()
204.  {
205.    const int hdb = 50;
206.    string helyett[] { "Á","AA", "É", "EE", "Í","II", "Ó","OO",
          "Ú","UU", "Ö","OE", "Ő","OOE", "Ü","UE", "Ű","UUE", "á","AA",
          "é", "EE", "í","II", "ó","OO", "ú","UU", "ö","OE", "ő","OOE",
          "ü","UE", "ű","UUE", "."," STOP", "?"," STOP", "!"," STOP",
          ",","", ";","", ":","", "\"",""};
207.    cout << "Mi lesz a távirat szövege?" << endl;
208.    string eredeti;
209.    getline(cin, eredeti);
210.
```

Ki lehet emelni a megoldásból egy függvénybe az egyes karakterek megkeresését a `helyett` 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 (unsigned i = 0; i < eredeti.size(); i++)
214.    {/*Keresés típusalgoritmus: karakter keresése a helyett-ben */
215.      int ez = 0;
216.      string betu = "";
217.      betu += eredeti[i]
218.      while (ez < hdb && betu != helyett[ez])
219.        ez += 2;          /*<= a trükkös tárolás miatt*/
220.      if (ez < hdb)
221.        tavirat += helyett[ez + 1];
222.      else
223.        tavirat += eredeti[i];
224.    }
```

Haladó programozóknak érdekes lehet, hogy ha a keresésre függvényünk van, akkor az hatékonyabbá tehető, ha átírjuk kiválasztásra. Ehhez az kell, hogy biztosan megtalálható legyen a karakter a helyettesíthetők között, ami megoldható úgy, hogy a tömb méretét kettővel megnöveljük és a kiválasztás előtt a keresett karaktert az utolsó két helyre bemásoljuk. 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.

```
225.    cout << "Táviratként: " << tavirat << endl;
226.  }
227.
```

Második mintamegoldásban a haladóbb eszköz használata és a részekre bontás együtt szerepel.

Először szűrjük ki az el az eltűnő karaktereket másolás típusalgoritmussal, ehhez hozzárendelő függvényként használjuk a korábban megírt bennevan() függvényt.

```
string szur(string eredeti)
{
  string mondatközi = ",;:\"-";
  string szurt = "";
  for (unsigned i = 0; i < eredeti.size(); i++)
    if (!bennevan(eredeti[i], mondatkozi)
      szurt += eredeti[i];
  return szurt;
}
```

Trükkös listánkat „normálisan” szótárban vagy karakter struktúra listában illene előállítani. A map<> vagy struct választása az adatszerkezetben és az algoritmusban is strukturáltabb megoldást jelent, a megoldás függvényekre bontásával együtt bemutatható.

```
struct karakter
{
  char utf;
  string telex;
  public karakter(char c, string s) { UTF = c; Telex = s; }
}

```

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 most nem ezt a **karakter** struktúrát használjuk, hanem a C++ „szótár”-át, a map<>-et. Újabb másolás típusalgoritmussal a szűrés után előállítjuk a telexes szöveget. A hozzárendelő függvény most a map<>-ban keresés és nagybetűssé alakítás megfelelő alkalmazása

```
string atalakit(string eredeti)
{
  map<char, string> UtfTelex map<char, string>() {
     {'Á', "AA" },{'É', "EE" },{'Í', "II" },{'Ó', "OO" },{'Ú', "UU"},
     {'Ö', "OE" },{'Ő', "OOE" },{'Ü', "UE" },{'Ű', "UUE" },
     {'á', "AA" },{'é', "EE" },{'í', "II" },{'ó', "OO" },{'ú', "UU"},
     {'ö', "OE" },{'ő', "OOE" },{'ü', "UE" },{'ű', "UUE" },
     {'.', " STOP" }, {'?', " STOP" }, {'!', " STOP" }};
  string szurt = szur(eredeti);
  string tavirat = "";
  for (unsigned i = 0; i < szurt.size(); i++)
    if (UtfTelex.find(szurt[i]) != UtfTelex.end())
      tavirat += UtfTelex[szurt[i]];
    else
      tavirat += szurt[i];
  return tavirat;
}
```

A program maradék része csak egy adatbekérés nagybetűssé másolás – kihasználva, hogy a speciális karakterekre ennek nincs hatása – és eredménykiírás.

```
256.  void tavirat_2()
257.  {
258.    cout << "Mi lesz a távirat szövege?" << endl;
259.    string eredeti = "";
260.    char c;
261.    getline(cin, eredeti);
262.    string nagy = "";
263.    for (unsigned i = 0; i < eredeti.size(); i++)
264.      nagy += toupper(eredeti[i];
265.    cout << "Táviratként: " << atalakit(nagy) << endl;
266.  }
```

MINDENT BELE! – ÖSSZEFÜGGŐ FELADATSOR MEGOLDÁSA

Feladatok

Ahogy egy szép, harmatos reggelen szertenézünk nagy tölgyek alatt megbúvó, fehérre meszelt tanyánkon, 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 ezen 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 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.  #include <iostream>
2.  #include <fstream>
3.  using namespace std;
```

Döntéseinkből következően a megoldást a struktúra elkészítésével kezdjük. Konstruktort nem írunk, az alapértelmezettet fogjuk használni.

```
4.  struct allat
5.  {
6.    string nev;
7.    string faj;
8.    int kor;
9.  };
10.
```

A tömböt a legelején létrehozzuk. Mivel nem tudjuk, hány állattal számoljunk, megpróbáljuk felülbecsülni. Rögtön az első kérdés az állatok száma, és erre a többi részfeladatban is szükség lesz, ezért ezt is globálisan deklaráljuk. Később ezt sem kell paraméterként átadnunk.

```
11.  const int MaxDB = 50;
12.  allat allatok[MaxDB];
13.  int aDB;
14.
```

Az **1. feladat** megoldását az adatok beolvasásával együtt végezzük el. A sor első adatát ciklusfeltételben olvassuk be, ha ez sikerült, akkor a többit a ciklusmagban. Mivel a tömb már rendelkezésre áll, az adatokkal módosítjuk az alapértelmezett értékeket.

```
15.  void beolvas()
16.  {/*Másolás*/
17.    ifstream fin ("allatok.txt");
18.    aDB = 0;
19.    while (fin >> allatok[aDB].nev)
20.    {
21.      fin >> allatok[aDB].faj;
22.      fin >> allatok[aDB].kor;
23.      aDB++;
24.    }
25.    fin.close();
26.  }
```

Függvényünket a `main()` eljárásban hívjuk meg. Kiírjuk a választ és ezután jöhet a 2. feladat:

```
130.  setlocale(LC_ALL, "Hun");
131.  beolvas();
132.    cout << "1. Összesen " << N << " állatunk van." << endl;
133.    cout << "2. A legöregebb neve: " << legoregebb() << endl;
134.
```

A **2. feladat** egy maximumkiválasztás.

```
28.  string legoregebb()
29.  {/*Maximumkiválasztás*/
30.    int maxi = 0;
31.    for (int i = 0; i < aDB; i++)
32.    {
33.      if (allatok[maxi].kor < allatok[i].kor)
34.        maxi = i;
35.    }
36.    return allatok[maxi].nev;
37.  }
38.
```

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

```
135.  cout << "3. Adjon meg egy fajnevet: ";
136.  string faj; cin >> faj;
137.  if (van(faj))
138.  cout << "Van " << faj << " a tanyán" << endl;
139.  else
140.  cout << "Nincs " << faj << " a tanyán" << endl;
141.
```

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.

```
40.  bool van(string faj)
41.  {/*Eldöntés típusalgoritmus*/
42.    int ez = 0;
43.    while (ez < aDB && allatok[ez].faj != faj)
44.      ez++;
45.    return ez < aDB;
46.  }
47.
```

Eddig egyszerű volt a kérdésekre válaszolni, de a 4. feladat 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 fajtáját, 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. Ráadásul 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.)

```
48.  struct fajta
49.  {
50.    string faj;
51.    int db;
52.  };
53.
```

A következő függvény ezt a fajhalmazt fogja feltölteni adatokkal. No, persze most sem tudjuk, hogy hány eleme lesz. (Persze, ha vector<>-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. Lustaságból ezt is globálisan …

```
54.  fajta fajhalmaz[MaxDB];
55.  int fDB;
```

Most jön a neheze:

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

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

```
56.  void fajtak()
57.  {
58.    fDB = 0;
59.    for (int i = 0; i < aDB; i++) /*minden állatot nézni*/
60.    {
61.      int ez = 0;    /*eldöntés: az i-edik állatnak temp-ben van a faja?*/
62.      while (ez < fDB && fajhalmaz[ez].faj != allatok[i].faj)
63.        ez++;
64.      if (ez < fDB)                        /*ha benne van*/
65.        fajhalmaz[ez].db++;       /*növeljük a mennyiséget*/
66.      else                         /*ha nincs benne*/
67.      {        /*a kigyűjtés típusalgoritmus alapján*/
68.        fajhalmaz[fDB].faj = allatok[i].faj;    /*betesszük*/
69.        fajhalmaz[fDB].db = 1;
70.        fDB++;                              /*1-gyel több fajta lett*/
71.      } /*else vége*/
72.    } /*for vége*/
73.  }
```

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

```
142.  fajtak();
143.  cout << "4. Az állatok listája: " << fajnevek(", ") << "." << endl;
144.
```

Ehhez írjuk meg a kívánt elválasztással paraméterezett függvényt. Az eredmény egy lista, amit egyetlen szöveggé fűzzük össze.

```
74.  string fajnevek(string sep)
75.  {
76.    string nevek = fajhalmaz[0].faj;    /*trükk: elsőt sep nélkül*/
77.    for (int i = 1; i < fDB; i++)
78.      nevek += sep + fajhalmaz[i].faj;
79.    return nevek;
80.  }
81.
```

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

```
145.  cout << "Válasszon egy fajnevet: ";
146.  cin >> faj;
147.  cout << (egyeves(faj) ? "Van" : "Nincs") << " egyéves " << faj <<
          " a tanyán" << endl;
```

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.

```
82.  bool egyeves(string faj)
83.  {/*Eldöntés típusalgoritmus*/
84.    int ez = 0;
85.    while (ez < aDB && !(allatok[ez].faj == faj && allatok[ez].kor ==1))
86.      ez++;
87.    return ez < aDB;
88.  }
89.
```

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

```
83.    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 < aDB && allatok[ez].faj != faj || allatok[ez].kor != 1` értelmezése:
`(ez < aDB && 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 < aDB &&` védi a programunkat attól, hogy `ez >= aDB` esetén a másik oldalt kezdje vizsgálni a program. Ez a szabály zárójel nélkül is megakadályozza, hogy a faj-t megnézze, 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.

```
90.  fajta maxfajta()
91.  {
92.    int maxi = 0;
93.    for (int i = 1; i < fDB; i++)
94.      if (fajhalmaz[maxi].db < fajhalmaz[i].db)
95.        maxi = i;
96.    return fajhalmaz[maxi];
97.  }
```

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

```
148.  fajta legtobb = maxfajta();
149.  cout << "5. A " << legtobb.faj << " populáció a legnagyobb: "
          << legtobb.db << " példány." << endl;
150.
```

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. Bár a fajokat és ehhez az egyedek számát is kigyűjtöttük, tudjuk, hogy hány fajta állat van a tanyán (`fDB`) és minden fajtáról tudjuk, hogy mennyi egyed van belőle, a tömbünket a legrosszabb esetre kell felkészítenünk: el kell tudnunk tárolni `MaxDB` különböző fajtájú állatot vagy `MaxDB` állatatot egy fajtából. Az egyedeket így egy kétdimenziós, `MaxDB` széles és `MaxDB` magas táblázatban tudjuk elhelyezni.

```
 98.  allat fajtankent[MaxDB][MaxDB]; /*nagyon pazarló méret*/
 99.  void szetvalogat()
100.  {/*szétválogatás = sok kiválogatás*/
```

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 processzornak) a második megoldás átlagosan fele annyi időt vesz igénybe, de vannak „egyéb körülmények”, amitől az első lesz jobb.

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

```
101.    for (int f = 0; f < fDB; f++)       /*minden fajtára*/
102.    {/* az aktuális fajta kiválogatása*/
103.      int egyed = 0;              /*ennyi van meg már ebből a fajtából*/
104.      for (int i = 0; i < aDB; i++)
105.      {
```

```
106.       if (allatok[i].faj == fajhalmaz[f].faj)
107.       {
108.         fajtankent[f][egyed] = allatok[i];
109.         egyed++;
110.       }       /*elágazás vége*/
111.     }         /*kigyűjtés vége (i)*/
112.   }           /*fajták kiválasztásának vége (f)*/
113. }             /*eljárás vége*/
```

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 praktikusan 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 (sorban) van és minden sorról tudjuk a darabszámot, ugyanazt a függvényt tudjuk használni akár az összes állatra is és egy-egy fajtára is: paraméternek megadjuk az állatokat tartalmazó (egydimenziós) tömböt és darabszámot.

```
113. double atlag(allat allatok[], int db)
114. {/*Összegzés*/
115.   double szum = 0;
116.   for (int i = 0; i < db; i++)
117.     szum += allatok[i].kor;
118.   return szum / db;
119. }
120.
```

A feladat megoldása a függvény meghívása minden fajtára:

```
121. void atlageletkorok()
122. {/*Másolás konzolra*/
123.   for (int i = 0; i < fDB; i++)
124.      cout <<  setfill('.') << setw(14) << left << fajhalmaz[i][0].faj
                << right << setw(4) << setprecision(3)
                << atlag(fajtankent[i], fajhalmaz[i].db) << endl;
125. }
```

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 6. feladatban a kigyűjtést (110–118. 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:

```
151.  cout << "6. Az egyes fajták átlagéletkora:" << endl;
152.  szetvalogat();
153.  atlageletkorok();
154.  cout << "7. Fajtánként fájlokba írás" << endl;
155.  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 fájlnév egy karaktersorozat, bármilyen eljárás, aminek az eredménye karaktersorozat, megteszi. A szétválogatás elemei az egyes sorokban olyan állatok, amiknek a `faj` tulajdonsága megegyezik a `fajhalmaz` tömb megfelelő elemének `faj` adatával. A `fajhalmaz faj` épp megfelel a fájlnév elejének, ehhez kell a kiterjesztést hozzáfűzni majd karaktersorozattá alakítani. A kiírás sorainak száma pedig ugyanennek az adatnak a `db` tulajdonsága.

Másik kérdés, hogy hány `ofstream` változó kell a fájlba íráshoz. Az biztos, hogy változónévből elég egy (`fout`). Az is biztos, hogy ehhez a cikluson belül rendeljük hozzá a fájl nevét és ugyanazon a cikluson belül le is zárjuk a fájlt. Ha a `fout` deklarációja így (140. sor), a cikluson kívül van, akkor ugyanazt a változót használjuk újra és újra, reinkarnálódik a `fout`. Ha a 143. sor elején deklaráljuk az `fout` típusát, akkor minden ciklusléptésben 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.

```
127.  void kiir()
128.  {/*Másolás, fájlbaírás*/
129.    ofstream fout;
130.    for (int i = 0; i < fDB; i++)
131.    {
132.      fout.open((fajhalmaz[i].faj + ".txt").c_str());
133.      for (int a = 0; a < fajhalmaz[i].db(); a++)
134.        fout << fajtankent[i][a].nev << endl;
135.      fout.close();
136.    }
137.  }
```

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 műszaki megoldások igényeihez igazítják a nyelv szabályait, de a felhasználás területét tekintve a nyelv univerzális. 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 tömb helyett `vector<>` típus szerepel.
- Az eljárásokat és függvényeket C++ függvények helyettesítik, így gyakorlatilag értelmetlenné vált a feladat részekre bontása. A teljes kód a `main()` eljáráson 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.  #include <iostream>
2.  #include <fstream>
3.  #include <vector>
4.  #include <map>
5.  #include <algorithm> /*típusalgoritmusok template függvényei*/
6.  #include <numeric>   /*accumulate és egyébb rekurzív összegzésekhez*/
7.  #include <iomanip>   /*extra formázás a táblázatos kiíráshoz*/
8.
9.  using namespace std;
10.
```

Komolyabb használat előtt ajánlott a fenti csomagok dokumentációját áttekinteni. Most csak a feladatokkal kapcsolatosan használunk néhány eszközt belőlük, de hasonló szintaktikával az összes típusalgoritmusra van elemtípustól független sablon megoldás.

```
11.  struct allat
12.  {
13.    string nev;
14.    string faj;
15.    int kor;
16.  };
17.
18.  int main()
19.  {
20.    setlocale(LC_ALL, "Hun");
```

A beolvasásnál, mivel nem ismerjük a tanyán élő állatok számát, könnyebb `vector<>`-t használni. Ez

```
21.    vector<allat> allatok;
22.    ifstream fin("allatok.txt");
23.    allat uj;
24.    while (fin >> uj.nev)
25.    {
26.      fin >> uj.faj >> uj.kor;
27.      allatok.push_back(uj);
28.    }
29.    cout <<"1. Összesen "<< allatok.size() <<" állatunk van."<< endl;
```

A legöregebb állat kiválasztása mutatóval – iterator – történik. A mutató pontos típusa olyan összetett, hogy reménytelen a kitalálása. Szerencsére a fordító program az értékadás jobb oldala alapján kitalálja, hogy mire van szükség.

```
30.    auto legoregebb = max_element(allatok.begin(), allatok.end(),
       [](allat a, allat b)
31.    {
32.      return a.kor < b.kor;
33.    });
34.    cout << "2. A legöregebb neve: " << legoregebb->nev << endl;
```

Figyeljük meg, hogy az eredmény kiírásához a nyíl-operátor szükséges! A lambda kifejezés használatára jellemző a végén megjelenő záró operátorok halmozása: `;});`.

Az eldöntés típusalgoritmusára három függvény van: any_of, all_of, none_of.

```
35.   cout << "3. Adjon meg egy fajnevet: ";
36.   string faj;
37.   cin >> faj;
38.   van = any_of(allatok.begin(), allatok.end(),[&](allat a)
      {
        return a.faj == faj;
      });
39.   cout << (van ? "Van" : "Nincs") <<" "<< faj <<" a tanyán."<< endl;
```

A fajták listája és ehhez az egyedek száma tipikusan szótár, `map<>` struktúrát igényel. Kivéve, ha az eredeti sorrend (első előfordulása egy fajnak) számít, mert a map a fajok neve szerint ábécébe rendez.

```
40.   map<string, int> fajhalmaz;
41.   for (allat a : allatok)
42.     fajhalmaz[a.faj]++;
```

Az összegzés típusalgoritmust megvalósító `accumulate()` függvényt használhatjuk a kimeneti `string` előállítására:

```
43.   string fajtak = accumulate(next(fajhalmaz.begin()), fajhalmaz.end(),
        fajhalmaz.begin()->first,[](string s, pair<string, int> f)
      {
        return s += ", " + f.first;
      });
44.   cout << "4. Az állatok listája: "<< fajtak << "." << endl;
```

Újra eldöntés … Az eddigiek alapján már megfigyelhető, hogy a lambda kifejezés függvénynévhelyettesítőjében (`[]`) akkor van `&` jel, ha a definícióban külső adatot kell felhasználni. Jelen esetben a `faj` változót.

```
45.   cout << "Válasszon egy fajnevet: ";
46.   cin >> faj;
47.   van = any_of(allatok.begin(), allatok.end(),[&](allat a)
      {
        return a.faj == faj && a.kor == 1;
      });
48.   cout<<(van ? "Van" : "Nincs")<<" egyéves "<<faj<<" a tanyán."<<endl;
```

Ismét maximum-kiválasztás … Látható, hogy a lambda kifejezés mindig egy logikai értéket visszaadó függvény, ami megadja a nagyságviszonyt.

```
49.   auto legtobb = max_element(fajhalmaz.begin(), fajhalmaz.end(),
      [](pair<string, int> x, pair<string, int> y)
      {
        return x.second < y.second;
      });
50.   cout << "5. A " << legtobb->first << " populáció a legnagyobb: "
           << legtobb->second << " példány." << endl;
```

A minimál megoldásban most jön a 2D tömb létrehozása, aminek az „átirata” listában lista lenne. Ez következik most, de utána lesz egy jobb megoldás. A 2D listát arra használjuk – eléggé pazarló módon –, hogy kiírjuk a legnagyobb populáció tagjait.

```
51.   cout<<"névszerint a "<<legtobb->second<<" "<<legtobb->first<<": ";
52.   vector<vector<allat>> fajtankent;
53.   for (pair<string,int> f : fajhalmaz)
54.   {
55.    vector<allat> uj;
56.    copy_if(allatok.begin(), allatok.end(), back_inserter(uj),
        [&](allat x)
        {
          return x.faj == f.first;
        });
57.    fajtankent.push_back(uj);
58.   }
```

Eddig tart a listák listájának az elkészítése. Érdemes megfigyelni a feltételes másolás paraméterezését: A harmadik paraméter a másolat helye (az `uj` végéhez), a negyedik paraméter a lambda kifejezés. A nevek kiírását szinte hagyományosan adjuk meg. Így sokkal rövidebb, egyszerűbb …

```
59.
60.   for (vector<allat> v : fajtankent)
61.    if (v[0].faj == legtobb->first)
62.     for (allat a: v)
63.      cout << a.nev << ", ";
64.   cout << "\b\b."<<endl;
```

Az egyes fajták átlagéletkorát először eltárolás nélkül adjuk meg. A <fajta,egyedszám> szótárt járjuk be, minden fajtára elvégezzük a feltételes összegzést és az eredményt elosztjuk az egyedszámmal:

```
65.   cout << "6. Az egyes fajták átlagéletkora:" << endl;
66.   for (pair<string,int> f : fajhalmaz)
67.   {
68.    double atlag = accumulate(allatok.begin(), allatok.end(), 0.0,
        [&](double x,  allat y)
        {
          return x + (y.faj == f.first? (double)y.kor : 0.0);
        });
69.    cout<<left<<setw(14)<< f.first <<setw(4)<< atlag/f.second << endl;
70.   }
```

A kiírásban beállított balra igazítás a program végéig (vagy módosításig) érvényes lesz. Az egyes adatok előtt megadott szélesség csak az adott adatra érvényes.

Ebben a feladatban is két adatot kell összerendelni, a fajtát és az átlagot. Erre legjobb a szótár:

```
71.   cout << "ugyanez map használatával" << endl;
72.   map<string, double>fajatlag;
73.   for (allat a : allatok)
74.    fajatlag[a.faj] += a.kor;
75.   for (pair<string,int> f: fajhalmaz)
76.    fajatlag[f.first] /= f.second;
77.   for (pair<string, double> a: fajatlag)
78.    cout << setw(14) << a.first << setw(4) << a.second << endl;
```

Jól érzékelhető, hogy ebben a megoldásban nem a sablonfüggvények, hanem a map<> beépített keresőfája adja a kényelmes kódolást. Egy ciklusban létrejön a szótár a korok összegével, a következő ciklusban elosztjuk az értékeket a korábban kiszámolt darabszámokkal, végül, a

harmadik ciklusban kiírjuk az eredményeket. Mindhárom esetben foreach-ciklust használtunk, ami a C++ nyelv speciális for-ciklusa. Ennek a ciklusnak a függvénysablonja a `for_each()`.

Nézzük, hogyan változik a megoldás, ha pontosan ugyanezt a megoldást `for_each()` függvényekkel írjuk meg:

```
79.    cout << "ugyanez for_each használatával" << endl;
80.    map<string, double>faja;
81.    for_each(allatok.begin(),allatok.end(),
         [&](allat a) {faja[a.faj] += a.kor;});
82.    for_each(fajhalmaz.begin(), fajhalmaz.end(),
         [&](pair<string,int> fh) {faja[fh.first] /= fh.second;});
83.    for_each(faja.begin(), faja.end(), [](pair<string, double> fa)
         {
           cout << setw(14) << fa.first << setw(4) << fa.second << endl;
         });
84.
85.    return 0; /*main() vége*/
86.  }
```

Látható, hogy nem lett egyszerűbb se és rövidebb se a kód. Az egyes utasítások olyan hosszúak, hogy átláthatatlanok egy sorban. Ha Code::Blocks-ban nézzük, ott a formázás több utasítássorba teszi, ha strukturálisan nézzük, akkor a jegyzetben látható sorszámozás (közben sortörésekkel) jelent egy-egy utasítást. Mondhatjuk, hogy egy típusalgoritmust igénylő feladat megoldása funkcionális eszközkészlettel egy sor, de ez a sor hosszabb, mint a típusalgoritmus körülbelül öt sora.

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 közöseket 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 sorbarendezbe, 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 Százszorszépvárosban

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

A közös megállók nevére van szükségünk, azaz metszetet kell képeznünk. A feladat szövege alapján feltételezhető, 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 `vector<>` típus miatt szükség lesz a `<vector>` csomagra. A bővíthetőség miatt az adatokat globálisan vesszük fel, a megoldásra eljárást készítünk a `main()` függvény előtt, amit a program végén lévő `main()`-ben hívunk meg, de ezt a jegyzetben most már nem írjuk ki.

```
 6.  string vill92 [7] {"Kiss u.", "Zöld fasor","Piros tér","Malom-patak",
 7.                     "Puccos u.", "Remegő-erdő", "Fapapucs sgt."};
 8.  string busz19 [8] {"Nagy u.", "Balambér tér", "Szent Lajos hídja",
 9.                       "Által-ér", "Reghős Bendegúz sz.k.",
10.                       "Puccos u.","Remegő-erdő", "Pöszke-liget"};
```

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

```
30.  void atszallas()
31.  {
32.    vector<string> atszall;
33.    for (int ez = 0; ez < 7; ez++)
34.    {
35.      if (bennevan(vill92[ez], busz19, 8))
36.        atszall.push_back(vill92[ez]);
37.    }
38.    cout << "Átszállási lehetőségek: ";
39.      kiir(atszall);
40.  }
```

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

```
12.  bool bennevan(string elem, string tomb[], int N)
13.  { /*Eldöntés típusalgoritmus*/
14.    int itt = 0;
15.    while (itt < N && elem != tomb[itt])
16.      itt++;
17.    return itt < N;
18.  }
```

Az eredmény kiírására is érdemes függvényt írni.

```
20.  void kiir(vector<string> ezt)
21.  {
22.    for (unsigned i = 0; i < ezt.size(); i++)
23.      cout << ", " << ezt[i];
24.    cout << endl;
25.  }
```

Unió

48. példa: Fricska és Kökény első szavainak jegyzése

A szomszédban lakó két bölcsis – Nagy Fricska és Zsémbes Kökény – első hét, illetve nyolc szavát feljegyezték anyukáik. Közösen tartott névnapi zsúrjukon 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 tömbök másolásával kapcsolatos sajátosságára. Ha egy tömbnek 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 paraméterében adjuk így át az adatokat, de a duplikáláshoz vagy másolás algoritmus kell vagy egy ezt megvalósító függvény.

```
166.  string fricska [7] {"anya", "maci", "baba", "apa", "kaja", "ló",
167.                       "erősáramú feszültségszabályzó"};
168.  string kokeny [8] {"apa", "autó", "anya", "víz", "maci", "könyv",
169.                       "nagyi", "pörgettyűs tájoló" };
```

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

```
30.  void szojegyzek()
31.  {
32.    vector<string> szojegyzek;
33.    for (int i = 0; i < 7; i++)
34.      szojegyzek.push_back(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.

```
35.    for (int i = 0; i < 8; i++)
36.      if (!bennevan(kokeny[i], fricska, 7))
37.        szojegyzek.push_back(kokeny[i]);
38.    cout << "A kész szójegyzék: ";
39.    kiir(szojegyzek);
40.  }
```

És a többi halmazművelet

A metszeten és unión túl, igény lehet két halmaz különbségére, ami a metszettől annyiban tér el, hogy akkor kerül be az elem az eredménybe, ha a `bennevan()` hamis. Ezeket felhasználva a szimmetrikus különbség is előállítható: az unióból kivonjuk a metszetet.

Lényegében minden halmazműveletre van függvény C++ nyelven az `<algorithm>` csomagban, de ezek rendezett adatsorozatot kezelnek. A fenti két feladat két tömbjét először rendezni kellene, azt követően használható a set_intersection, a set_union, a set_difference és a set_symmetric_difference.

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 memóriafoglalásban 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 eljátsszuk 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 megérhető – 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 algoritmusa

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
egyik := másik
másik := temp
```

Csere algoritmus C++ kód részlete:

```
1. Adattipus temp = egyik;
2. egyik = masik;
3. masik = 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 alapművelete, ezért sok programozási nyelvben van rá eljárás, általában `swap` néven, de ennek a használata mindenképp nagy körültekintést 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:

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

Egy tömb két helyén lévő értékek cseréje:

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

Ha két egész számot adunk felcserélésre, akkor **&** 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, amelyikre mutatnak. A sikerhez azonban az is hozzá tartozik, hogy a tömb másolata valójában a tömb elejére mutató hivatkozás másolata, így a tömbelemek **&** 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 típusalgoritmusa. A szétválogatás stratégiája: az adatsorozatot mindkét végéről egymásfelé haladva járjuk be és a nem odavaló elemeket cseréljük ki.

Például: egy osztály tanulói névsorát kell átrendeznünk úgy, hogy elöl legyenek a lányok, hátul a fiúk. A megoldás: az elölről az első fiút kicseréljük hátulról az első lánnyal. Ezt addig ismételjük 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 elölre kerüljenek a rosszak (kétlábúak), a végére a jók (négylábúak). A válogatás során elölről 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:

```
allat allatok[50]; /*az 44. feladat megoldásából átvéve*/
int aDB;
int main() /*_orvellhelyben */
{
  vector<string> tojok{ "kacsa", "liba", "kakas" };
  beolvasas()                          /*megírt eljárás*/
          /*szétválogatás mutatóinak a kezdőértéke:*/
  int eleje = 0;
  int vege = aDB - 1;
  int rosszdb;          /*az első rész hossza, itt a kétlábúak száma*/
  while (eleje < vege)
  {
```

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.

```
          /*négylábút keres elölről*/
    while (eleje < vege && egyike(allatok[eleje].faj, tojok))
      eleje++;
    if (eleje < vege) /*ha nem értek össze*/
    {
```

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

```
          /*kétlábút keres a végétől visszafelé*/
      while (eleje < vege && !egyike(allatok[vege].faj, tojok))
        vege--;
    }/*elágazás vége; lehet, hogy közben összeértek*/
```

Ezen a ponton, ha csak négylábú állataink vannak, akkor az `eleje` értéke `0`, a `vege` 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 < vege`, akkor kicseréljük a mutatott két elemet.

```
    if (eleje < vege) /*ha mindkét irányból talált nem odavalót*/
    {     /*felcserélés*/
      allat temp = allatok[eleje];
      allatok[eleje] = allatok[vege];
      allatok[vege] = temp;
    }
```

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 (`eleje == aDB - 1`), akkor ennél eggyel több kétlábú van. Egyébként, amikor az elejét léptetjük, akkor a vége egy négylábúra mutat: vagy a legelsőre, vagy az éppen kicserélt elemre, ami emiatt négylábú. 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, tojok)) /*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.   cout << "Rosszak: ";
35.   for (int i = 0; i < rosszdb; i++)
36.     cout << "( " << allatok[i].nev << ", " << allatok[i].faj << ") ");
37.   cout << "\nJók: ";
38.   for (int i = rosszdb; i < aDB; i++)
39.     cout << "[" << 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 bújik meg. Ezért fontos lesz, hogy a rendezés stratégiájának kitalálása után a kódolásnál minden apróságra figyeljünk.

A megoldás 8–28. sorát okosabban, rövidebben is megírhatjuk.

```
 8.   /*******************szétválogatás okosabban******************/
 9.   int eleje = 0;
10.   int vege = aDB - 1;
11.   int rosszdb;          /*az első rész hossza, itt a kétlábúak száma*/
12.   while (eleje < vege)
13.   {
```

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.

```
14.         /*négylábút keres elölről*/
15.     while (eleje < aDB && egyike(allatok[eleje].faj, tojok))
16.       eleje++;
17.         /*kétlábút keres a végétől visszafelé*/
18.     while (vege >= 0 && !egyike(allatok[vege].faj, tojok))
19.       vege--;
```

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 `vege` `-1` lett, akkor az `eleje` `0` maradt, ha az `eleje` `aDB` lett akkor a `vege` `aDB - 1`. Mindkét esetben igaz, hogy `vege < eleje`. A feldolgozandó elemek az `[eleje; vege]` zárt intervallumban vannak. Ha ez létezik, azaz, ha `eleje < vege`, akkor a két helyen lévő adat jó helyre kerül, ezért mindkét irányból szűkíthetjük a vizsgálandó tartományt.

```
20.     if (eleje < vege) /*ha mindkét irányból talált átrakandót*/
21.     {     /*felcserélés*/
22.       Allat temp = allatok[eleje];
23.       allatok[eleje] = allatok[vege];
24.       allatok[vege] = temp;
25.         /*továbblép, mert az eleje és a vége helyen már kész*/
26.       eleje++; vege--;
27.     }
```

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

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

```
28.    }
29.    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.  void cseresRendezes_lepesenkent()
 9.  {
10.    const int N = 5;
11.    int A [N] { 5, 3, 9, 1, 7 };
12.    cout << "Egyszerű cserés rendezés" << endl;
13.    for (int itt = 0; itt < N - 1; itt++)
14.    {
15.      for (int utana = itt + 1; utana < N; utana++)
16.      {
17.        for (int t = 0; t < N; t++)
18.          cout << A[t] << " ";
19.        cout <<"itt: "<< itt + 1 <<", utána: "<< utana + 1 <<"."<< endl;
20.        if (A[itt] > A[utana])
21.        {
22.          cout << " => csere" << endl;
23.          int temp = A[itt];
24.          A[itt] = A[utana];
25.          A[utana] = temp;
26.        }
27.        else
28.        {
29.          cout << " => nem cserélünk" << endl;
30.        }
31.      }
32.    }
```

```
33.   cout << "Vége: ";
34.   for (int t = 0; t < N; t++)
35.     cout << A[t] << " ";
36.   cout << endl;
37. }
```

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 eggyel 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 eggyel 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. void cseresRend_algoritmus(int A[], int N)
 9. {
10.   for (int itt = 0; itt < N - 1; itt++)
11.     for (int utana = itt + 1; utana < N; utana++)
12.       if (A[itt] > A[utana])
13.         csere(A, itt, utana);
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 visszamásolunk 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 találtnál 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 értékűnél 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 értéket egy másik sorozatba átmásoljuk és ezt az értéket

nagyobbra (esetleg egyenlőre) állítjuk a sorozat legnagyobb értékénél. Ez sokkal egyszerűbbnek hangzik, próbáljuk meg kódolni.

```
 8.  void csakmostrend()
 9.  {
10.    const int N = 5;
11.    int A[N] { 5, 3, 9, 1, 7 };
12.    cout << "Másik adatsorozatba átrakással:" << endl;
13.    int temp[N];
14.    for (int db = 0; db < N; db++)
15.    {
16.      int mini = 0;
17.      for (int i = 1; i < N; i++)
18.        if (A[i] < A[mini])
19.          mini = i;
20.      temp[db] = A[mini];
21.      A[mini] = INT_MAX;
22.    }
23.    /*az eredmény visszamásolása!*/
24.    for (int i = 0; i < N; i++)
25.      A[i] = temp[i];
26.    cout << "Vége: ";
27.    for (int t = 0; t < N; t++)
28.      cout << A[t] << " ";
29.    cout << endl;
30.  }
```

Az eljárás neve azért csakmost… mert ezt az életben csak egyszer, tanuláskor szabad kipróbálni. Elkeserítő, hogy az algoritmus lényege a szeméttermelés. 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.  void lyuktomorend()
 9.  {
10.    const int N = 5;
11.    int A [N] { 5, 3, 9, 1, 7 };
12.    cout << "Legkisebbet ki, lyukat a végéről betömni." << endl;
13.    int temp[N];
14.    for (int db = 0; db < N; db++)
15.    {
16.      int utolso = N - 1 - db;          /*hasznos elnevezni*/
17.      int mini = 0;
18.      for (int i = 1; i <= utolso; i++)              /*<=-re figyelni*/
19.        if (A[i] < A[mini])
20.          mini = i;
21.      temp[db] = A[mini];
22.      A[mini] = A[utolso];              /*utolsóval felülírni mini-t */
23.    }
```

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

```
24.   /*az eredmény visszamásolása és kiírása*/
25.   for (int i = 0; i < N; i++)
26.     A[i] = temp[i];
27.   for (int t = 0; t < N; t++)
28.     cout << A[t] << " ";
29.   cout << endl;
30. }
```

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. void minkivRendezes_lepesenkent()
 9. {
10.   const int N = 5;
11.   int A[N] { 5, 3, 9, 1, 7 };
12.    cout << "Minimumkiválasztásos rendezés" << endl;
13.   for (int itt = 0; itt < N - 1; itt++)
14.   {
15.     int mini = itt;                    /*innentől min.-kiválasztás*/
16.     for (int utana = itt + 1; utana < N; utana++)
17.       if (A[mini] > A[utana])
18.         mini = utana;
19.     for (int t = 0; t < N; t++) /*állapot kiírás*/
20.       cout << A[t] << " ";
21.     cout << endl;
22.     if (mini != itt)
23.     {        /*csere a belső ciklus után, csak akkor, ha kell*/
24.       cout << " => csere" << endl;
25.       int temp = A[itt];
26.       A[itt] = A[mini];
27.       A[mini] = temp;
28.     }
29.     else
30.     {
31.       cout << " => nem cserélünk" << endl;
32.     }
33.   }
34.   for (int t = 0; t < N; t++)
35.     cout << A[t] << " ";
36.   cout << endl;
37. }
```

Ugyanez a rendezés a kiírásoktól megszabadítva:

```
 8.  void minkivRend(int A[], int N)
 9.  {
10.    for (int itt = 0; itt < N - 1; itt++)
11.    {
12.      int mini = itt;
13.      for (int utana = itt + 1; utana < N; utana++)
14.        if (A[mini] > A[utana])
15.          mini = utana;
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.  }
```

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

```
 8.  int minkiv(int A[], int tol, int ig)
 9.  {
10.    int mini = tol;
11.    for (int utana = tol + 1; utana < ig; utana++)
12.      if (A[mini] > A[utana])
13.        mini = utana;
14.    return mini;
15.  }
16.
17.  void minkivRend_algoritmus(int A[], int N)
18.  {
19.    for (int itt = 0; itt < N - 1; itt++)
20.    {
21.      int mini = minkiv(A, itt, N);
22.      if (mini != itt)
23.        csere(A, itt, mini);
24.    }
25.  }
26.
```

Megfigyelhető, hogy ez a rendezési algoritmus az egyszerű cserés rendezésünktől 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 „viszünk”, 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.  void bubirendezes_lepesenkent()
 9.  {
10.    const int N = 5;
11.    int A [N] { 5, 3, 9, 1, 7 };
12.    cout << "Buborék rendezés - nagy a végére" << endl;
13.    for (int teteje = N; teteje >= 1; teteje--)
14.    {
15.      for (int bubi = 0; bubi < teteje - 1; bubi++)
16.      {
17.        cout << "Most: ";
18.        for (int t = 0; t < N; t++) /*állapot kiírás*/
19.          cout << A[t] << " ";
20.        cout << endl;
21.        if (A[bubi] > A[bubi + 1])
22.        {
23.          cout << " => " << A[bubi] << " tovább" << endl;
24.          int temp = A[bubi];
25.          A[bubi] = A[bubi + 1];
26.          A[bubi + 1] = temp;
27.        }
28.        else
29.        {
30.          cout << " => " << A[bubi + 1] << " fog menni" << endl;
31.        }
32.      }
33.    }
34.    cout << "Vége: ";
35.    for (int t = 0; t < N; t++) /*állapot kiírás*/
36.      cout << A[t] << " ";
37.    cout << endl;
38.  }
```

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.

```
 8.  void bubirend_algoritmus(int A[], int N)
 9.  {
10.    for (int teteje = N; teteje >= 1; teteje--)
11.      for (int bubi = 0; bubi < teteje - 1; bubi++)
12.        if (A[bubi] > A[bubi + 1])
13.          csere(A, bubi, bubi + 1);
14.  }
15.
```

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 koktél rendezés, angolul **Cocktail sort**.

Beszúró és Törpe rendezés

A beszúró 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 előbbre 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 előrefele 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 előrefelé 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 beszúró rendezés stratégiája tehát: a második elemtől az utolsóig minden elemet kiveszünk és az előtte lévőket – amíg a kiválasztott elemnél nagyobbak – eggyel hátrébb teszszük.

```
 8.  void beszuroRendezes()
 9.  {
10.    const int N = 5;
11.    int A[5] { 5, 3, 9, 1, 7 };
12.    cout << "Beszúró rendezés" << endl;
13.    for (int i = 1; i < N; i++)
14.    {
15.      int ez = A[i];              /*kitessük a vizsgáltat*/
16.      int ide = i;
17.      while (ide > 0 && A[ide - 1] > ez)
18.      {
19.        A[ide] = A[ide - 1];      /*hátrébb másoljuk a nagyobbakat*/
20.        ide--;
21.      }
22.      A[ide] = ez;                /*az így biztosított helyre betesszük*/
23.    }
24.    cout << "Vége: ";
25.    for (int t = 0; t < N; t++) /*állapot kiírás*/
26.      cout << A[t] << " ";
27.    cout << endl;
28.  }
```

Az eddigi rendezési algoritmusokra jellemző volt, hogy két for-ciklusból, egy elágazásból és egy cseréből épültek fel. A beszúró 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 előrefelé 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.  void torpeRendezes()
 9.  {
10.    const int N = 5;
11.    int A[5] { 5, 3, 9, 1, 7 };
12.    cout << "Törpe rendezés" << endl;
13.    int i = 0;
14.    while (i < N - 1)
15.    {
16.      if (A[i] <= A[i + 1])
17.        i++;                /*ha jó a sorrend: index növelése*/
18.      else
19.      {
20.        int tmp = A[i];     /*i. és i+1. elem cseréje*/
21.        A[i] = A[i + 1];
22.        A[i + 1] = tmp;
23.        if( i > 0) i--;     /*vissza, mert az új i. kisebb lehet az
                              előzőnél is, kivéve, ha már a legelején van.*/
24.      }
25.    }
26.    cout << "Vége: ";
27.    for (int t = 0; t < N; t++) /*állapot kiírás*/
28.      cout << A[t] << " ";
29.    cout << endl;
30.  }
```

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, eközben pedig csak méretet vesz, megy fel-alá és cserél a robot.

A sort() és társai

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-eljárás. 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.

A rendezési eljárásokat az `<algorithm>` csomag tartalmazza, a kiíráshoz célszerű függvényt írni.

```
1.  #include <iostream>;
2.  #include <algorithm>
3.  using namespace std;
4.  string ki(int A[], int N)
5.  {
6.    string s = "";
7.    for (int i = 0; i < N; i++)
8.    s += to_string(A[i]) + " ";
9.    return s;
10. }
11. int main()
12. {
13.   setlocale(LC_ALL, "Hun");
```

Először ennek a `T, egész számokat tartalmazó` tömbnek a másolatát rendezzük át.

```
14.  int T[5] { 10, 6, 18, 2, 14 };
15.  int A[5];
```

A tömbök rendezésére a `sort()` eljárás használható, alapértelmezetten növekvően rendezi az adatokat:

```
16.    copy(begin(T), end(T), A);
17.    cout << endl << ki(A,5) << "\b\b." << endl;
18.    sort(begin(A), end(A));
19.    cout << "Növekvő: " << ki(A,5) << "\b\b." << endl;
```

Ebben az egyszerű esetben a rendezés alapja az `A` adattípusára (`int`) értelmezett `<` relációs operátor. A csökkenő rendezéshez harmadik paraméterként kell megadni a rendezési reláció típusát. Korábban már láthattuk, hogy a tömbök esetén a `begin(T)` helyett megfelel a `T`, az `end(T)` helyett a `T + hossz`, ahol a `hossz` egy `unsigned int` típusú érték.

```
20.    copy(T, T + 5, A);
21.    cout << endl << ki(A,5) << "\b\b." << endl;
22.    sort(A, A + 5, greater<int>());
23.    cout << "Csökkenő: " << ki(A,5) << "\b\b." << endl;
```

A négyféle relációt az itt látható `greater<>()` mellett a `greater_equal<>()`, `less<>()`, `less_equal<>()` függvényekkel lehet beállítani. Ha nem számok, hanem szövegek lennének a tömbben, akkor az ábécé alapján lehet rendezni, a típusnak meg kell egyeznie az adatsorozat elemeinek a típusával. Ha ettől eltérő rendezést szeretnénk, azt **komparátor**ral – összehasonlító függvénnyel adhatjuk meg. Ezt vagy előre megírjuk (ekkor csak a nevét kell 3. paraméterként megadni), vagy logikai eredményt adó lambda kifejezésben adjuk meg. Az összehasonlító függvény a sort esetén azt adja meg, hogy mikor van helyes sorrendben két elem, ugyanaz a lambda kifejezés a `max_element()` függvényben a rendezett sorozatunk utolsó elemét adná meg.

Az alábbi példában a hárommal való osztás maradéka alapján rendezzük az adatokat.

```
24.   copy(T, T+ 5, A);
25.   cout << endl << ki(A,5) << "\b\b." << endl;
26.   sort(begin(A), end(A), [](int x, int y){return x % 3 < y % 3;});
27.   cout << "mod 3 nő: " << ki(A,5) << "\b\b." << endl;
```

Ha az összehasonlítás szabályára nincs kész komparátor, akkor írhatunk ennél hosszabb kifejtést is. Ez főleg a többszempontú rendezéseknél hasznos. Akár a **return** előtt lehet több feltételt megadni, akár a **return** után van egy összetett logikai kifejezés … a lényeg, hogy két, a sorozat típusának megfelelő adatra írjunk fel egy sorrendiséget megadó szabályt, aminek a visszaadott értéke logikai **true** vagy **false**.

Az alábbi rendezés első rendezési szempontja a számok 3-mal vett osztási maradék alapján növekvő, amennyiben ez megegyezik, akkor a számok értéke szerint csökkenő rendet kér:

```
28.   copy(begin(T), end(T), A);
29.   cout << endl << ki(A,5) << "\b\b." << endl;
30.   sort(begin(A), end(A), [](int x, int y)
      {return x % 3 < y % 3 || ((x % 3 == y % 3) && (x > y));});
31.   cout<< " 1. mod 3 nő, 2. csökken: "<< ki(A,5) << "\b\b." << endl;
32.
```

Ugyanezeket a rendezéseket majdnem ugyanígy végezhetjük el **vector<>** típusú adatokon, de a megvalósításon látszik, hogy a tömb egy egyszerű adatsorozat, míg a **vector<>** egy adatsorozatot tartalmazó tároló, konténer. A szakmai háttérből a gyakorlati tudnivaló, hogy a **vector<>** több, mint egy adatsorozat, a neve után írt ponttal választható ki megfelelő tagfüggvény: vec**.begin()**.

```
33.   vector<int> V;
34.   copy(begin(T), end(T), back_inserter(V);
35.   sort(begin(A), end(A), [](int x, int y){return x % 3 < y % 3;});
36.   cout<< "V-ben mod 3 nő: ";
37.   for(int v : V) cout << v << ", ";
38.   cout << "\b\b." << endl;
```

A **sort()**-hoz hasonló a **stable_sort()**, ami garantálja, hogy az egyenlők megtartják az eredeti sorrendi viszonyukat. Az **nth_element()** a rendezett sorozat n-edik elemét jelöli ki. Ezt 2. paraméterként, az eleje és vége között adhatjuk meg. A **partial_sort()** részben rendezi az adatokat: megadható, hogy hány elemű legyen a rendezett rész és hogy ez az elejétől vagy a végétől kezdődően legyen-e.

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 Sapientia 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! Gyűjtsük ki a kutyák, a macskák és a kakasok nevét!

1. **Rendezzük a neveket névsorba és írjuk ki!**
2. **Gyűjtsük ki a kutyák, a macskák és a kakasok minden adatát!**
3. **Rendezzük kedvenc állatainkat kor szerint csökkenő sorrendbe!**
4. **Írjuk ki kedvenc állataink nevét és – nevük után zárójelben – a korukat!**
5. **Rendezzük kedvenceinket fajuk szerint, azon belül név szerint! Írjuk ki soronként minden adatukat!**
6. **Rendezzük a tanya összes állatát névsorba sort() eljárással és írjuk ki a névsort!**
7. **Rendezzük át a tanya állatait fajonként, azon belül névsorba a sort() eljárással! Írjuk ki az állatok összes adatát egymás alá!**
8. **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 `sort()` 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 tömbben, 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`** struktúrákat 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.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;

struct allat
{
string nev;
  string faj;
int kor;
};
```

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.

```
15.  void ki(allat a)
16.  {
17.    cout <<setw(13)<< a.nev <<setw(13)<< a.faj <<setw(3)<< a.kor <<endl;
18.  }
```

A feladatok megoldását az adatok beolvasása is megelőzi. A változatosság kedvéért, a kód jelentős része a főprogramban lesz, az egyes részeket megjegyzés sorokkal jelöljük.

```
25.  int main()
26.  {
27.  setlocale(LC_ALL, "Hun");
28.  allat[] mind[15];
29.  ifstream fin("allatok.txt");
30.  for (int i = 0; i < 15; i++)
31.    fin >> mind[i].nev >> mind[i].faj >> mind[i].kor;
32.  fin.close();
```

1. Gyűjtsük ki a kutyák, 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 valójában hány kedvencünk van.

```
33.  //1. kedvenceink neveinek kigyűjtése tömbbe
34.    string nevek[15];  /*mindegyik kedvenc?*/
35.    int N = 15;        /*kedvencek száma*/
36.    for (int i = 0; i < N; i++)
37.    {
38.      if (mind[i].faj == "kutya" || mind[i].faj == "macska"
                                    || mind[i].faj == "kakas")
39.      {
40.        nevek[N] = mind[i].nev;
41.        N++;
42.      }
43.    }
```

2. Rendezzük a neveket névsorba és írjuk ki!

Ha az adataink tömbben vannak, akkor figyelnünk 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. 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 `""` érték van, ez megzavarhatja a sorrendet.

```
45.  //2. nevek rendezése
46.    cout << "Kedvenceink ábécében:" << endl;
47.    /*minimumkiválasztásos rendezés, de csak N-ig*/
48.    for (int ez = 0; ez < N - 1; ez++)
49.    {
```

```
50.     int mini = ez;
51.     for (int i = ez + 1; i < N; i++)
52.       if (nevek[i] < nevek[mini])
53.         mini = i;
54.     if (mini != ez)
55.     {
56.       string temp = nevek[ez];
57.       nevek[ez] = nevek[mini];
58.       nevek[mini] = temp;
59.     }
60.   }
61.   cout << accumulate(nevek + 1, nevek + N, nevek[0],
              [](string a, string b){return a += " " + b;}) << " " << endl;
62.
```

3. Gyűjtsük ki a kutyák, a macskák és a kakasok minden adatát!

Most listába gyűjtjük az adatokat, mert így pont akkora lesz az adatsorunk, amennyi a figyelembe vett elemszám.

```
64.  //3. kedvenceink kigyüjtése listába
65.    vector<allat> kedvencek;
66.    for (int i = 0; i < 15; i++)
67.      if (mind[i].faj == "kutya" || mind[i].faj == "macska"
68.                     || mind[i].faj == "kakas")
69.        kedvencek.push_back(mind[i]);
70.
```

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 adattag legyen az összehasonlítás alapja. Az **allat** típusú változók az = jellel másolhatók, ezért könnyű a tömbelemek cseréje. Sokkal bonyolultabb lenne, ha listákat kellene felcserélni.

```
71.  //4-5. allat egyik tulajdonsága szerint csökkenö rendezés
72.    cout << "Kedvenceink kor szerint:";
73.    /*buborék rendezés csökkenőre - kicsi megy a végére*/
74.    for (unsigned eddig = kedvencek.size(); 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!**

```
87.   for (unsigned i = 0; i < kedvencek.size(); i++)
88.     cout <<" "<< kedvencek[i].nev <<"("<< kedvencek[i].kor <<")";
89.   cout << endl;
90.
```

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:

```
91.  //6. kétkulcsos rendezés
92.   cout << "Kedvenceink faj, azon belül név szerint" << endl;
93.   /*beszúró rendezéssel*/
94.
95.   for (unsigned i = 1; i < kedvencek.size(); i++)
96.   {
97.     allat ez = kedvencek[i];
98.     int ide = i;
99.     while (ide > 0 && (kedvencek[ide - 1].faj > ez.faj ||
100.           (kedvencek[ide - 1].faj == ez.faj) &&
101.            kedvencek[ide - 1].nev > ez.nev)))
102.    {
103.      kedvencek[ide] = kedvencek[ide - 1];
104.      ide--;
105.    }
106.    kedvencek[ide] = ez;
107.  }
108.  for (allat k : kedvencek)
109.    ki(k);
110.
```

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 eljárásként is:

```
94.  maskepp(kedvencek); /*a 107. sorig helyettesít*/
```

A `main()` előtt elfér a ciklusfeltétel deklarációja, utána a kifejtése:

```
145.  void maskepp(vector<allat>& kedvencek) /*referencia átadása*/
146.  {
147.    for (unsigned i = 1; i < kedvencek.size(); i++)
148.    {
149.      allat ez = kedvencek[i];
150.      int ide = i;
151.      while (ide > 0 && a7b(kedvencek[ide - 1], ez))
152.      {
153.        kedvencek[ide] = kedvencek[ide - 1];
154.        ide--;
155.      }
156.      kedvencek[ide] = ez;
157.    }
158.  }
```

Az „a7b” helyett bármilyen más neve is lehet a függvények, de a>b nem. A függvénybe írhatnánk ugyanazt, mint a 98–100. sorban, 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 „sorminta-szerűen”.

```
160.  bool a7b(allat a, allat b)
161.  {
162.    bool nagyobb = false;
163.    if (a.faj > b.faj)
164.      nagyobb = true;
165.    else if (a.faj == b.faj)
166.      if (a.nev > b.nev)
167.        nagyobb = true;
168.    return nagyobb;
169.  }
```

Rövidíthető a kód, ha nem gyűjtjük logikai változóba az eredményt, hanem azonnal visszatérünk vele.

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 fogalmazunk meg.

```
110.  //7. minden állat név - sort és lambda
111.    cout << "Minden állat név szerint:" << endl;
112.    sort(mind, mind + 15, [](allat x, allat y){ return x.nev < y.nev;});
113.    for (int i = 0; i < 15; i++)
114.      cout << mind[i].nev + ", ";
115.    cout << "\b." << endl;
116.
```

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 logikai értéket ad vissza. Ezt megírhatjuk a lambda kifejezésen belül:

```
117.  //8. minden állat két-kulcsos sort, lambda
118.    cout << "\nMinden állat faj, azon belül név szerint:" << endl;
119.    sort(mind, mind + 15, [](allat x, allat y)
120.         {
121.           if (x.faj == y.faj)
122.             return x.nev < y.nev;
123.           else
124.             return x.faj < y.faj;
125.         });
126.    for (int i = 0; i < 15; i++)
127.      ki(mind[i]);
128.
```

Ebben a formában névtelen a függvényünk. Ha többször kell ugyanazt az összehasonlító függvényt megadni, akkor érdemes nevet is adni neki és ez a nevet beírva, delegálni (megbízni a `sort()`-ot a feladattal). Az átadott – vagy paraméterben írt lambda – függvényt hívják delegate függvénynek.

```
119.  sort(mind, mind + 15, hasonlit); /*125. sorig helyettesít*/
```

A függvény csak a fejlécben tér el a lambdától: van visszatérési típusa és neve a `[]` helyett:

```
172.  int hasonlit(allat x, allat y)
173.  {
174.    if (x.faj == y.faj)
175.      return x.nev < y.nev;
176.    else
177.      return x.faj < y.faj;
178.  }
```

Ez a függvény is lehetne pont olyan, mint az `a7b()` a 160–169. sorokban, de így talán átláthatóbb.

9. **Rendezzük a tanya állatait kor szerint csökkenő sorrendbe, ezen belül nevük hossza szerinti sorrendbe! Írjuk ki soronként az adatokat!**

```
130.  //9. mindre két-kulcsos
131.    cout << "\nMind kor csökk., név hossza növ:" << endl;
132.    sort(mind, mind + 15, [](allat x, allat y)
133.         {
134.           if (x.kor == y.kor)
135.             return x.nev.size() < y.nev.size();
136.           else
137.             return x.kor > y.kor;
138.           });
139.    for (int i = 0; i < 15; i++)
140.      ki(mind[i]);
141.
```

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 ciklusléptések 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 ciklusléptések 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 két érték közé esik. 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 `bin_merrevan()` 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:

```
 5.  void bin_merrevan(int ez, int lista[], int N)
 6.  {
 7.    int eleje = 0;
 8.    int vege = N - 1;
 9.    if (ez < lista[eleje])
10.      cout << "a legkisebb előtti, nincs a listában." << endl;
11.    else if (ez > lista[vege])
12.      cout << "a legnagyobb utáni, nincs a listában" << endl;
13.    else if (ez == lista[eleje])
14.      cout << "A " << eleje << ". helyen van " << ez << "." << endl;
15.    else if (ez == lista[vege])
16.      cout << "A " << vege << ". helyen van " << ez << "." << endl;
17.    else
18.    {
```

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.

```
19.      int kozepe;
20.      do
21.      {
22.        kozepe = (eleje + vege) / 2;
23.        if (ez < lista[kozepe])
24.          vege = kozepe; /*a lista[vége] != ez*/
25.        else
26.          eleje = kozepe; /*a lista[eleje] != ez*/
27.      } while (lista[kozepe] != ez && vege - eleje > 1);
28.      if (lista[kozepe] == ez)
29.        cout << "A " << kozepe << ". helyen van " << ez << "." << endl;
30.      else           /*vége - eleje == 1 és sem az eleje, sem a vége.*/
31.        cout << lista[eleje] << " és " << lista[vege]
              << " között lehetne, de nincs a listában." << endl;
32.    }
33.  }
```

A vizsgálat előtt kizártuk, ezt követően, amikor az `eleje` vagy a `vege` értéke változik, tudjuk, hogy ott nem lehet a keresett érték, mert a `kozepe` értékét csak akkor kapják meg, ha a `lista[kozepe]` nem egyenlő a keresett értékkel. Ez akkor is igaz, amikor már szomszédos értékekre mutat az `eleje` és a `vege`. 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[7] { 1, 3, 5, 6, 7, 8, 9 };
for (int i = 0; i < 11; i++) /*tesztadatok*/
{
  cout << i << " keresése" << endl;
  bin_merrevan(i, lista, 7);
}
```

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 felejtsük el, hogy be kell olvasni a fájlt és rendezni is kell. Ezt a két lépést érdemes egyben megoldani. A beolvasásra most egy olyan függvényt írunk, aminek az eredménye az állatok száma, miközben a paraméterében lévő tömb elejét – fajtánként rendezve – tölti fel.

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 az adatot a függvényünk paraméterén keresztül fogjuk átadni, **&** jellel módosíthatóvá tett változóba tesszük ki.

A keresés függvényét a végén ebben a programban teszteljük:

```
58.  /*bináris keresés teszt: van-e postás által kérdezett állat*/
59.  allat allatok[50];
60.  int N = rendezve_beolvas(allatok);
61.  for (int i = 0; i < N; i++)
62.    cout << allatok[i].faj << " ";
63.  cout << endl;
64.  string kerdezett string[6]{"atka", "birka", "egér", "kutya",
                  "szarvasmarha", "zsizsik" };
65.  for (int i = 0; i < 6; i++)
66.  {
67.    allat ez;
68.    if (bin_keres(kerdezett[i], allatok, N, ez))
69.      cout << "A(z) " << ez.faj << " neve " << ez.nev << ", " << ez.kor
     << " éves." << endl;
70.    else
71.      cout << "Nincs " << kerdezett[i] << " a farmon." << endl;
72.  }
```

A hozzávalók közül, először az **allat** struktúra megírása a már rutinfeladat.

```
 5.  struct allat
 6.  {
 7.    string nev;
 8.    string faj;
 9.    int kor;
10.  };
```

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 beszúró 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.

```
13.  int rendezve_beolvas(allat A[])
14.  {/*igazi beszúró rendezés*/
15.    ifstream fin("allatok.txt");
16.    int db =0;
17.    allat ez;
18.    while (fin >> ez.nev)
19.    {
20.      fin >> ez.faj >> ez.kor;
21.      int hely = db;
22.      while (hely > 0 && A[hely - 1].faj > ez.faj)
23.      {
24.        A[hely] = A[hely - 1];
25.        hely--;
26.      }
27.      A[hely] = ez;
28.      db++;
29.    }
30.    fin.close();
31.    return db;
32.  }
33.
```

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 `bin_keres()` függvényünk megkapja a postás által kérdezett fajta nevét, az állatok összes adatát tartalmazó tömböt a benne levő érvényes adatok számával és egy **`allat`** típusú változót, amibe majd az eredményt tesszük, ha megtaláltuk. A függvény logikai értéket ad vissza, az utolsó paraméterben csak **`true`** eredmény esetén lehet értelmes adat. Most jobb megoldás találat esetén az adatról készített másolat átadása, mert ha nincs találat, akkor nem módosul ez az adat.

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.

```
35.  bool bin_keres(string fajta, allat A[], int N, allat& ez)
36.  {
37.    int elso = 0;
38.    int utolso = N;
39.    while (elso < utolso)
40.    {
41.      int kozep = (elso + utolso) / 2;
42.      if (A[kozep].faj == fajta)                  /*talált*/
43.      {
44.        ez = A[kozep];                            /*értékadás*/
45.        return true;                              /*fgv vége*/
46.      }
47.      else if (fajta < A[kozep].faj)
48.        utolso = kozep - 1; /*kisebb => nem középen, előtte*/
49.      else /*fajta > a[kozep]*/
50.        elso = kozep + 1; /*nagyobb => az eleje a közép után*/
51.    }
52.    /*a +-1 miatt elso > utolsó => nincs benne, 'ez' adat nem módosul*/
53.    return false;
54.  }
```

Az 47. sorban – az előtte lévő **return** következtében – az **else** nem szükséges, de talán egy kicsit jobban olvasható így a kód – három eset van: ==, <, >.

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 a programkódunk végén található main() függvényben:

```
170.  /*fricska-kökény: összefuttatás: unió, metszet*/
171.  vector<string> fricska {"anya", "apa", "baba",
172.              "erősáramú feszültségszabályzó", "kaja", "ló", "maci" };
173.  vector<string> kokeny { "anya", "apa", "autó",
174.                  "könyv", "maci", "nagyi", "pörgettyűs tájoló", "víz" };
```

A változatosság érdekében, most `vector<string>` típust használunk, a kiíráshoz a korábban már megírt `vec2str()` függvényt használjuk.

53. példa: Fricska és Kökény szótárainak közös része (metszete)

Készítsük el a két gyerek közös szótárát a szótáraik rendezett listáiból:

```
175.  vector<string> metszet = rendezett_metszet(fricska, kokeny);
176.  cout<<"Fricska és Kökény közös szavai: "<<vec2str(", ",
      metszet)<<endl;
```

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.  vector<string> rendezett_metszet(    vector<string> fricska,
                                          vector<string> kokeny)
10.  {
11.    vector<string> kozos;
12.    unsigned f = 0;
13.    unsigned k = 0;
14.    while (f < fricska.size() && k < kokeny.size())
15.    {
16.      if (fricska[f] < kokeny[k])
17.        f++;
18.      else if (fricska[f] > kokeny[k])
19.        k++;
20.      else /*fricska[f] == kokeny[k] */
21.      {
22.        kozos.push_back(fricska[f]);
23.        f++; k++;
24.      }
25.    }
26.    return kozos;
27.  }
```

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.

```
177.  vector<string> unio = rendezett_unio(fricska, kokeny);
178.  cout << "Fricska és Kökény szótára: " << vec2str(", ", unio) << endl;
```

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 „lemaradó” 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.  vector<string> rendezett_unio(vector<string> fricska,
                                   vector<string> kokeny)
29.  {
30.    fricska.push_back("zzzzz");          /*végére egy nagy érték*/
31.    kokeny.push_back("zzzzz");           /*ugyanaz a nagy érték*/
32.    vector<string> egyben;
33.    unsigned f = 0;
34.    unsigned k = 0;
35.    while (fricska[f] != "zzzzz" || kokeny[k] != "zzzzz")
36.    {
37.      if (fricska[f] < kokeny[k])
38.      {
39.        egyben.push_back(fricska[f]);
40.        f++;
41.      }
42.      else if (fricska[f] > kokeny[k])
43.      {
44.        egyben.push_back(kokeny[k]);
45.        k++;
46.      }
47.      else /*fricska[f] == kökény[k] */
48.      {
49.        egyben.push_back(fricska[f]);
50.        f++; k++;
51.      }
52.    }
53.    return egyben;
54.  }
```

Figyeljünk arra, hogy ha a két `vector<>`-t referenciaként (`&`) adjuk át, akkor a listához hozzáadott elemet távolítsuk is el a végén, mert úgy a módosítás a függvény lefutása után is megmarad. Ha a paraméterbe a változó másolata kerül, akkor nem probléma a módosítás, de a másolás miatt lassabb a programunk. Az utolsó elem eltávolítása – az elem megnevezése nélkül – a `.pop_back()` eljárással lehetséges.

Az összefuttatás nagy előnye az elemenkénti feldolgozás. Ez lehetővé teszi azt is, hogy két – nagyon sok adatsorból álló – fájlból folyamatosan beolvasva, mindig csak egy adatot tartva az operatív memóriában előállítsuk egy harmadik fájlban akár az uniót, akár a metszetet.

Kiegészítés: halmazműveletek rendezett sorozatokkal

A típusalgoritmusok mellett, az `<algorithm>` csomagban a halmazműveletek megfelelőit is megtaláljuk. C++-ban csak rendezett sorozatokra készítették el az eljárásokat, amelyeknek az alapja így az összefuttatás. Ha az egyes kiindulási adatsorozatokban van ismétlődés, akkor ez az eredményben is meg fog jelenni. Ha az adatsorozatok nem rendezettek, akkor előtte a `sort()` eljárással rendezni kell a két halmazt, majd ezt követően jöhet a halmazművelet. Már ennyiből is látszik, hogy ezek az eljárások nem feltétlenül optimálisak, ezért ha az adatsorozat nem rendezett, akkor megfontolandó a használatuk.

Nézzük a metszet, unió, különbség és szimmetrikus különbség előregyártott megoldásait. Látható, hogy szintaktikájuk egyforma, nagyon hasonló a másoláshoz – egy helyett két adatsorozat kezdetét és végét kell megadni, valamint az eredmény sorozat végére történő hivatkozást.

```
179.  vector<string> section;
180.  set_interxection(fricska.begin(), fricska.end(), kokeny.begin(),
                       kokeny.end(), back_inserter(section));
181.  cout << "Szótárak metszete: " << vec2str(", ", section) << endl;
182.
183.  vector<string> uni;
184.  set_union(fricska.begin(), fricska.end(), kokeny.begin(),
                       kokeny.end(), back_inserter(uni));
185.  cout << "Szótárak uniója: " << vec2str(", ", uni) << endl;
186.
187.  vector<string> difference;
188.  set_difference(fricska.begin(), fricska.end(), kokeny.begin(),
                       kokeny.end(), back_inserter(difference));
189.  cout << "Szótárak különbsége: " << vec2str(", ", difference) << endl;
190.
191.  vector<string> symdiff;
192.  set_symmetric_difference(fricska.begin(), fricska.end(),
                       kokeny.begin(), kokeny.end(), back_inserter(symdiff));
193.  cout << "Szimmetrikus differencia: " << vec2str(", ", symdiff) <<
      endl;
```

Ó, ió, Rekurzióóó!

Aki tanult Imagine Logot, vagy blokkprogramozást (pl. Scratch), az lehet, hogy már ismeri a rekurziót, a rekurzív eljárásokat. Logoban 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 blokkprogramozási 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.  void rekhivas(int v)
10.  {
11.    cout << "Hello, " << v << " vagyok. ";
12.    if (v == 0)                          /*rekurzió megszakítása*/
13.    {
14.      cout << "Már megyek is." << endl;
15.      return;                            /*VÉGE! innen nincs tovább.*/
16.    }
17.    cout << "Hívom " << v - 1 << "-t." << endl;
18.    rekhivas(v - 1);                 /*Önmagát hívja, más paraméterrel*/
19.    cout << v-1 << " kimúlt, végem van. Pá: " << v << endl;
20.  }
21.  int main()
22.  {
23.    setlocale(LC_ALL, "Hun");
24.    int pl = 4;
25.    rekhivas(pl);
26.  }
```

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, annyi 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” ágat ki kell írni:

```
20.  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.  int main()
10.  {
11.    int pl = 4;
12.    int f = for_fakt(pl);
13.    cout << pl << "! = " << f << endl;
14.    int r = rek_fakt(pl);
15.    cout << pl << "! = " << r << endl;
16.  }
```

Faktoriális számítás ciklussal:

```
17.  int for_fakt(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.  int rek_fakt(int n)
25.  {
26.    if (n == 0)
27.      return 1;
28.
29.    return n * rek_fakt(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 paraméterekén kell megadnunk 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:

```
70.  int main()
71.  {
72.    setlocale(LC_ALL, "Hun");
73.    int honapnapok [12]{31,28,31,30,31,30,31,31,30,31,30,31};
74.    cout << "Összesen: " << sorozatszamitas(honapnapok) << " "<< endl;
75.     if (van28(honapnapok))
76.      cout << "Van 28 napos hónap." << endl;
77.    else
78.      cout << "Nem találtunk 28 napos hónapot." << endl;
```

```
79.    cout <<"A(z) "<< harminc(honapnapok)+1 <<". hónap 30 napos."<<endl;
80.    int melyik;
81.    if(melyik28(honapnapok, melyik))
82.      cout << "A(z) " << melyik << ". hónap 28 napos." << endl;
83.    else
84.      cout << "Nem találtunk 28 napos hónapot." << endl;
85.    cout << darab(honapnapok, 30) << " db 30 napos hónap van." << endl;
86.    cout << "A legrövidebb hónap " << honapnapok[minhely(honapnapok)];
87.    cout << " napos." << endl;
88.  }
```

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.

1. Hónapok napjaiból az év hossza (Összegzés)

```
5.  int rekurzivszum(int tomb[], int i)
6.  {
7.    if (i == 0)
8.      return tomb[0];
9.    return tomb[i] + rekurzivszum(tomb, i - 1);
10. }
11. int sorozatszamitas(int tomb[])
12. {
13.   return rekurzivszum(tomb, 11);
14. }
```

2. Van-e 28 napos hónap az évben? (Eldöntés)

```
16. bool rekurzivvane(int tomb[], int i)
17. {
18.   if (i == 0)
19.     return tomb[0] == 28;
20.   return (tomb[i] == 28 || rekurzivvane(tomb, i - 1));
21. }
22. bool van28(int tomb[])
23. {
24.   return rekurzivvane(tomb, 11);
25. }
26.
```

3. Hányadik az első 30 napos hónap? (Kiválasztás)

```
27. int rekurzivmelyik(int ertek, int tomb[], int i)
28. {
29.   if (i == -1)
30.     return -1;
31.   return tomb[11 - i] == ertek ? (11 - i) :
                                      rekurzivmelyik(ertek, tomb, i-1);
32. }
```

```
33.  int harminc(int tomb[])
34.  {
35.    return rekurzivmelyik(30, tomb, 11);
36.  }
```

Ha a `11 - 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.

```
37.  bool melyik28(int tomb[], int& ez)
38.  {
39.    ez = rekurzivmelyik(28, tomb, 11) + 1;
40.    return ez > 0;
41.  }
```

5. Hány 30 napos hónap van az évben? (Megszámlálás)

```
42.  int rekurzivdarab(int tomb[], int ertek, int i)
43.  {
44.    if (i == 0)
45.      return tomb[0] == ertek ? 1 : 0;
46.    return (tomb[i] == ertek ? 1 : 0) + rekurzivdarab(tomb, ertek, i-1);
47.  }
48.  int darab(int tomb[], int ertek)
49.  {
50.    return rekurzivdarab(tomb, ertek, 11);
51.  }
```

6. Hányadik hónap a legrövidebb? (Szélsőérték-kiválasztás)

```
53.  int rekurzivminhely(int tomb[], int i)
54.  {
55.    if (i == 0)
56.      return 0;
57.    int eddig = rekurzivminhely(tomb, i - 1);
58.    return (tomb[eddig] > tomb[i]) ? i : eddig;
59.  }
60.  int minhely(int tomb[])
61.  {
62.    return rekurzivminhely(tomb, 11);
63.  }
```

Gyorsrendezés

Sokszor nehézkesebb, 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 megismerkedni vele.

Nemzetközi neve **Quick sort**. Ha ügyesen írják meg, akkor sokszor győztes a futtatási idők versenyében. Az algoritmus alapeleme a szétválogatás, amit – rekurzívan – ismétel 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 `>=` 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.

```
85.  vector<int> lista { 5, 3, 9, 1, 7, 2, 3, 4, 3 };
86.  …
87.  int main()
88.  {
89.    setlocale(LC_ALL, "Hun");
90.    srand(time(0));
91.    /*adatsorozat nem ismétlődő elemekkel*/
92.    int tomb[5] { 5, 3, 9, 1, 7 };
93.    quicksortH(tomb, 0, 4);
94.    for(int i = 0; i < 5; i++) cout << tomb[i] << " ";
95.    cout << endl;
96.    /*adatsorozat ismétlődő elemekkel*/
97.    quicksortI(0, lista.size() - 1);
98.    for(unsigned i = 0; i < lista.size(); i++) cout << lista[i] << " ";
99.    cout << endl;
100.   return 0;
101.  }
```

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 különbözők.

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 értékként visszaadni.

57. példa: Tömb gyors rendezése (nincs ismétlődő érték)

A rendező algoritmust a mondatszerű leírás alapján írjuk meg:

```
34.  void quicksortH(int tomb[], int eleje, int vege)
35.  {
36.    int hatar = szetoszt(tomb, eleje, vege);
37.    if (hatar - eleje > 1)
38.      quicksortH(tomb, eleje, hatar - 1);
39.    if (vege - hatar > 0)
40.      quicksortH(tomb, hatar, vege);
41.  }
```

A `szetoszt()` függvény mellékhatása a tömb átrendezett új állapota, a visszatérési értéke az elejére válogatott adatok száma lesz. Ez egyben a hátulra 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: `vege - 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.

```
 7.  int szetoszt(int tomb[], int eleje, int vege)
 8.  {
 9.    int e = eleje;
10.    int v = vege;
11.    int valaszt = eleje + rand() % (vege - eleje + 1);
12.    int ertek = tomb[valaszt]; /*egyedi, mert nincs ismétlődés*/
13.    while (e <= v)
```

A helyben szétválogatás algoritmus szerint rossz helyen lévő adatokat keresünk elölről 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 <= 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 problémát okoznak a részsorozat határai. 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 `ertek`, de ez háromféle állapotot jelent. A megoldásban el kell dönteni, hogy hova tartozzon az éppen kiválasztott elem. A szétválogatásnak van olyan algoritmusa, 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).

```
14.   {
15.     /*nagyobbat keres elölről*/
16.     while (e <= vege && tomb[e] <= ertek)
17.       e++;
18.     /*kisebb vagy egyenlőt keres hátulról*/
19.     while (v >= eleje && tomb[v] > ertek)
20.       v--;
21.     if (e < v) /*mindkét irányból talált nem odavalót*/
```

Ha mindkét „térfélen” találunk nem megfelelő adatot, akkor ezeket felcseréljük. Ilyenkor nem lehet a két adat indexe egyenlő, mert akkor önmagát cserélnénk.

```
22.     {/*felcserélés*/
23.       int temp = tomb[e];
24.       tomb[e] = tomb[v];
25.       tomb[v] = temp;
26.       /*jó helyen vannak, (ertek bárhol) következőre lépés*/
```

A csere után az `e` és `v` helyen a „térfélnek” 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.

```
27.       e++; v--;
28.     }
29.   }
30.   /*tomb[i]<=ertek: eleje..e-1; tomb[i]>=ertek: e..vege*/
31.   return e;
32. }
```

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 eljáráson belül hozzuk létre a három listát, akkor minden rekurzív híváskor új listákat gyártana a programunk, ráadásul a másolás során a fokozatos bővítés miatt egy `vector<>` 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 elérhető tömb, de a méretet is ismerni kellene. 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 eljáráson belül hozzuk létre.

Egy részsorozat szétválogatásából két kigyűjtést – az adott értéknél kisebbek és nagyobbak – a globális, előre rögzített méretű listába teszünk, méghozzá a részsorozat kezdetének és végének megfelelő indexek szerint. Az adott értékkel egyenlőket az eljáráson belül létrehozott (átmeneti) listába gyűjtjük ki. Végül a három kigyűjtést visszamásoljuk a részsorozatra.

```
86.  vector<int> qtemp(lista.size(), 0); /*globális hely a kiválogatáshoz*/
45.  void quicksortI(int eleje, int vege)
46.  {
47.    int valaszt = vege; /*vagy: rand() % (vege - eleje + 1) + eleje is*/
48.    int ertek = lista[valaszt];
49.    int e = eleje;
50.    int v = vege;
51.    vector<int> kozepe;
52.    int dbK = 0;
```

A `kozepe` listában lesznek az `ertek`-kel egyenlő elemek, emellett 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, annak egy adattagja szerint végezzük, akkor a teljes adatot el kell tárolni. A `kozepe` eltárolása nagyméretű, összetett adatok, szövegek esetén jelentősen lassíthatja a programot.

```
53.    for (int i = eleje; i <= vege; i++)
54.    {
55.      if (lista[i] < ertek)
56.      {
57.        qtemp[e] = lista[i];
58.        e++;
59.      }
```

Az előző módszer egymásba ágyazott while-ciklusai helyett, itt egyszerű számlálós ciklust használhatunk. Ahogy haladunk a részsorozatban, a kiválasztott értéknél kisebb elemeket kigyűjtjük az átmeneti tároló megfelelő részének elejére: az `e` változó kezdőértéke az `eleje`, és minden új adattal eggyel nő.

```
60.     else if (lista[i] > ertek)
61.     {
62.       qtemp[v] = lista[i];
63.       v--;
64.     }
```

Az értéknél nagyobb adatokat a tároló megfelelő részének végére másoljuk: v kezdőértéke a vege (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.

```
65.     else
66.     {
67.       kozepe.push_back(lista[i]);
68.       dbK++;
69.     }
70.   }
```

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.

```
71.   for (int i = eleje; i < e; i++)
72.     lista[i] = qtemp[i];
73.   for (int i = 0; i < dbK; i++)
74.     lista[e + i] = kozepe[i];
75.   for (int i = v + 1; i <= vege; i++)
76.     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.

```
77.   if (e - eleje > 1)
78.     quicksortI(eleje, e - 1);
79.   if (vege - v > 1)
80.     quicksortI(v + 1, vege);
81. }
```

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”.

Csoportnapló – 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, felfoghatjuk 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) tanulócsoport, 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 vezeték- vagy keresztneveket. A nevek tárolhatók egyetlen szövegként.

Az eddigiek alapján érdemes lesz egy **`diak`** struktúrát létrehozni, egy-egy diák adatainak kezelhető tárolására. A **`diak`** struktúra adattagjai a `nev` és az `osztaly`, valamint a jegyeket ketté válogatva a harmadik adattag lehet a `rendes_jegyek`, ami a „rendes jegyeket” egész számként tartalmazó adatsorozat és negyedik adattag az előzőhöz hasonló `tz_jegyek`.

Eddigi példáinkban még nem kellett egy struktúrán belül adatsorozatot tárolni, most pedig két adatsorozat is lesz a struktúrán belül. A `rendes_jegyek` 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 adattagként a `tzdb` tárolására. (Továbbra is igaz, hogy nem kell tudni mindkét adatsorozat használatát, és itt is bármelyiket helyettesítheti a másik, de egyre több haszna van a célnak megfelelőbb típus választásának.)

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 szükséges csomagok felvételével, fájl hozzáadásával érdemes kezdeni ezt követi program konstansainak megadása. Az adatok – osztály – tárolására használhatunk tömböt és listát is. A kicsit több kódolást igénylő tömbös megoldás lesz itt, ezért meg kell adni az osztály maximálisan lehetséges létszámát. Legyen ez 50. Emellett konkrét adatként szerepel a feladatban a témazáró jegyek maximális értéke: 3. Ezt is tároljuk el a program elején. Mindkét adatról elmondhatjuk, hogy az adott feltételek mellett megfelelőek, de a programunk fejlesztése során lehet, hogy módosítani kell.

```
1.  #include <iostream>
2.  #include <vector>
3.  #include <fsstream>
4.  #include <sstream>
5.
6.  using namespace std;
7.  const int MaxDiak = 50;
8.  const int MaxDiak = 3;
```

Ezt követi a **diak** struktúra elkészítése, körülbelül a 10. sortól kezdődően. A kapott adatokat tekintve, lehet, hogy a konstruktor megírása a teljes feladat legnehezebb része.

```
10.  struct diak
11.  {
12.    string nev;
13.    string osztaly;
14.    vector<int> rendes_jegyek;
15.    int tz_jegyek[Max_TZ_DB];
16.    int tzdb;
```

Az adatokat először csak deklaráljuk, az értékadás a konstruktorban történik:

```
17.    diak(string sor)
18.    {
19.      stringstream s(sor);
20.      getline(s, nev, ',');
21.      s.ignore(1);
22.      getline(s, osztaly, ',');
```

A jegyek feldolgozása külön figyelmet igényel. 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.

```
23.      tzdb = 0; /*reméljük a végén sem lesz több, mint Max_TZ_DB*/
24.      string jegy;
25.      while (s >> jegy)
26.      {
27.        if (jegy.size() == 1)
28.          rendes_jegyek.push_back(jegy[0] - '0');
```

Témazáró jegy értékének megadására többféle módszer létezik, most 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.        else
30.        {
31.          tz_jegyek[tzdb] = jegyek[i][0] - '0';
32.          tzdb++;
33.        } /*else vége*/
34.      } /*while vége*/
35.    } /*konstruktor vége*/
```

Mivel tömbben tároljuk a diákokat, szükség van alapértelmezett adatra is. Írunk kell egy paraméter nélküli konstruktort is, ami jelen esetben nagyon egyszerű, mert az adattagoknak van alapértelmezett értéke, így semmit sem kell megadni benne:

```
36.   diak()
37.   {}
38. };
39.
```

Minden feladat a csoportról szól, ezért globálisan adjuk meg a csoportok tömbjét és a feltöltés mértékét, a csoportlétszámot.

```
40. diak csoport[MaxDiak];
41. int N;
42.
```

A feladatok megoldására eljárásokat és függvényeket fogunk írni, amelyeket a `main()`-ben hívunk meg. Akár már munkánk elején elkészíthetjük a válaszok vázlatát, ehhez a `main()` előtt deklaráljuk az eljárásokat, függvényeket, amiket később, a `main()` után fogunk kifejteni.

```
43. void kiir_zs();
44. void keves_tz();
45. string min_tz();
46. void felelesveszely();
47. void evvege();
```

Körülbelül az ötödik lépés, a **0. feladat** megoldása: az adatok beolvasása és eltárolása a programban. Ezt most a `main()` eljárásban írjuk meg.

```
48. int main()
49. {
50.   setlocale(LC_ALL, "Hun");
51.   N = 0;
52.   string sor;
53.   ifstream fin ("naplo.txt");
54.   while (getline(fin, sor))
55.   {
56.     csoport[N] = diak(sor);
57.     N++;
58.   }
59.   fin.close();
60.   cout << "1. A 11.zs tanulói:" << endl;
61.   kiir_zs();
62.   cout << "2. Kevés témazárót írtak:" << endl;
63.   keves_tz();
64.   cout << "3. 11.x-ből legkevesebb témazárót ";
65.   cout << min_tz() << " írta." << endl;
66.   cout << "4. Felelésre felkészülők:" << endl;
67.   felelesveszely();
68.   cout << endl;
69.   cout << "5. Év végi átlagok és jegyek:" << endl;
70.   evvege();
71. return 0;
72. }
```

A `csoport` tömböt és a csoporttagok számát (N) globális változóként hoztuk létre, ezért a függvényeknek, eljárásoknak most nem lesz paramétere.

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.)

```
73.  void kiir_zs()
74.  {
75.    for (int i = 0; i < N; i++)
76.      if (csoport[i].osztaly == "11")
     /*if (csoport[i].osztaly.substr(0,2)=="11" &&
     csoport[i].osztaly.substr(csoport[i].osztaly.size()-2,2) == "zs")*/
77.        cout <<'\t' << csoport[i].nev << endl;
78.  }
79.
```

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.

```
80.  void keves_tz()
81.  {
82.    for (int i = 0; i < N; i++)
83.      if (csoport[i].tzdb < 3)
84.      {
85.        cout << '\t' << csoport[i].nev + ": ";
86.        for (int j = 0; j < csoport[i].tzdb; j++)
87.          cout << csoport[i].tz_jegyek[j] << ", ";
88.        cout << "\b\b " << endl;
89.      }
90.  }
```

A 87. sorban a `csoport[i].tz_jegyek[j]` jelentése: a `csoport` `i`-edik diákjának `j`-edik témazáró jegye. Ha az adatstruktúra létrehozásakor a témazárókat is listában tároljuk, akkor a `tzdb`-re nem lenne szükség, helyettesítené a `csoport[i].tz_jegyek.size()`.

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.

```
91.  string min_tz()
92.  {
93.    int mini = -1;
94.    int mindb = 4; /*legfeljebb 3 lehet*/
95.    for (int i = 0; i < N; i++)
96.    {
97.      if (csoport[i].osztaly == "11x")
98.        if (csoport[i].tzdb < mindb)
99.        {
100.         mindb = csoport[i].tzdb;
101.         mini = i;
102.       }
103.   }
104.   return mini == -1 ? "nincs 11x-es diák" : csoport[mini].nev;
105. }
106.
```

A 99. és 100. 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 kommentben jelezzük az egyes részmegoldásokat.

```
107. void felelesveszely()
108. {
109.   /*legtöbb rendes jegy*/
110.   unsigned maxe = csoport[0].rendes_jegyek.size();
111.   for (int i = 1; i < N; i++)
112.     if (csoport[i].rendes_jegyek.size() > maxe)
113.       maxe = csoport[i].rendes_jegyek.size();
114.   /*kigyűjtés*/
115.   for (int i = 0; i < N; i++)
116.     if (csoport[i].rendes_jegyek.size() < maxe * 0.8)
117.       cout <<'\t' << csoport[i].nev << endl;
118. }
119.
```

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 118. 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. Kevésbé szembetűnő, de sokkal problémásabb a `size()` használata cikluson belül. Ha ez tényleg függvény, akkor lehet, hogy programunk minden használatkor végignézi a kérdéses listát, hogy megtalálja a végét. Jobb esetben megkeresi az eltárolt változó értékét. egy cikluson belül 2-szer.

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 jegyektől. Ezt kihasználva a függvényünk viszonylag egyszerű formát ölt:

```
120.  double atlag(diak d)
121.  {
122.    double szum = 0;
123.    for (unsigned i = 0; i < d.rendes_jegyek.size(); i++)
124.      szum += d.rendes_jegyek[i];
125.    for (int i = 0; i < d.tzdb; i++)
126.      szum += 2 * d.tz_jegyek[i];
127.    return szum / (d.rendes_jegyek.size() + 2 * d.tzdb);
128.  }
129.
```

A függvényt felhasználva az értékelés már nem nehéz, csak hosszú.

```
130.  void evvege()
131.  {
132.    for (int i = 0; i < N; i++)
133.    {
134.      double a = atlag(csoport[i]);
135.      int jegy;
136.      if (a <= 1.7)
137.        jegy = 1;
138.      else if (a <= 2.5)
139.        jegy = 2;
140.      else if (a <= 3.5)
141.        jegy = 3;
142.      else if (a <= 4.5)
143.        jegy = 4;
144.      else
145.        jegy = 5;
146.      cout << "\t " << left << setw(25) << csoport[i].nev;
147.      cout << " átlaga: " << fixed << setprecision(2) << a;
148.      cout << "\tjegye: " << jegy << "." << endl;
149.    }
150.  }
```

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 felezőpont kerekítésére a felhasználási területtől függően más-más szabályt használnak. Például van, ahol a két egész közötti felező értéket a páros

szám felé kerekítik, így a 3,5 és a 4,5 is 4 lesz. Kellemetlen a jeles osztályzatra vágyóknak, de ez a szakmák körében jellemzőbben elfogadott kerekítési szabvány.

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, az adatstruktúra tervezésére és megalkotására eszköz az objektumorientált programozás. Ebben épp úgy, ahogy a strukturált programozásban a vezérlési struktúráknak és típusalgoritmusoknak, vannak szabványok, betartandó adatszervezési 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 változóinak az értéke 0-ról 1-re változik, ami egy belső eljárás eredményeként másik változókat 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 és 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 objektumosztályok esetén?

C++ nyelvben majdnem semennyire. A programozási nyelv fejlődése során a C-ből átvett **struct** mellett megjelent a **class**, ami többet tudott, mint a **struct**. Nagyon sok korábban – esetleg C-ben – írt programban használták a **struct**-ot, a programok fejlesztése során ezeket mind át kellett volna írni **class**-ra … de ehelyett a **struct** képességeit fejlesztették, ebben is lett művelet végzésére nyelvi lehetőség. Így a kettő szinte szinonimává vált. Az egyetlen különbség, hogy a **struct** nyitott, az adattagjai – ha nem szabályozzuk – publikusak, kifelé minden látható és módosítható. Ezzel szemben a **class** zárt, az adatok és műveletek privátak, belsők. Az interface az, amihez publikus hozzáférést engedélyezünk.

Az alapértelmezést felül lehet írni. Akár **struct**, akár **class**, a definíción belül globálisan megadott változók elé írt **private:** biztosítja, hogy csak belülről lehessen használni. A **public:** részen megadott változó kívülről látható **tulajdonság** (property). Gyakori, hogy a változó privát, de két publikus függvénnyel elérhetővé tesszük. Egy belső, privát változót egy publikus **getter** függvénnyel tehetjük olvashatóvá és egy publikus **setter** függvénnyel tudjuk módosítani kívülről. Ezek a függvények részei az interface-nek. Ha nem vesszük (túl) komolyan az objektum orientáltság elvárásait, akkor minden olyan változót, amihez gettert és settert is írnánk, publikus változónak állítunk be. Ha az objektumunk minden adatát publikusnak adnánk meg, akkor legegyszerűbb **struct**-ként megadni.

A használat szempontjából kényes kérdés, hogy ha egy **class**-ban privát változók vannak, amelyek kívülről nem érhetők el, akkor hogyan lesz bármilyen értékük. Erre jók a – mindig **public** hozzáférésű – konstruktorok, ahol a belső változók értékadásához szükséges paramétereket megadhatjuk.

A konstruktor majdnem olyan, mint egy függvény, de mégsem az. Vagy, ha az, akkor nagyon speciális és nem úgy, speciális, ahogy az eljárás, mert nem csak tesz-vesz, de létre is hoz, van eredménye: az objektum. Ezt fejezzük ki azzal, hogy a konstruktornak nincs visszatérési értéke, void sem … Mivel most már tudjuk, hogy a struct is majdnem class, így gyakorlati tapasztalatunk is volt a különböző konstruktorokra, azok használatára, a példányosítás módjaira.

Az alapértelmezett konstruktor – nincs paramétere – használata: **class-név** objektum_név**;**
Azaz a class-név egy adattípus neve, az objektum_név eddig (és ezután is) változó név is.

```
allat uj;
```

A paraméterezett konstruktort úgy használjuk, hogy a változó név után zárójelbe írjuk a paramétereket. **class-név** objektum_név(paraméterek); Például

```
195.  ifstream fin("fajlnev"); /*ifstream osztály, fin objektum neve*/
196.  ifstream fin = ifstream("fajlnev"); /*u. az hosszabban*/
197.  allatlist.push_back(allat(sor)); /*sor-ból készített allat*/
```

A copy constructor, azaz másoló konstruktor olyan, mint egy értékadás. Egy létező példányból – ez a paramétere – készíti el az új példányt. Ha nem egyértelmű, hogy melyik adattagot hogyan kell átadni (például másolatot vagy hivatkozást), akkor ennek a kivitelezését is meg kell adni. A használatára volt példa:

```
198.  allat temp = kedvencek[i];
199.  allat temp(kedvencek[i]); /*így is lehet*/
```

Az értékadás direkt formája C++ nyelven a konstruktort kiegészítő **inicializáló lista**. Ilyenkor nem a konstruktor törzsében adjuk meg a bemenet-változó kapcsolatot, hanem előtte (képletesen a konstruktor nyakánál). Egy példa:

```
200.  allat():nev(""),faj(""),kor(0)//inicializáló lista
201.  {}                                          //nem a törzsben, hanem
      előtte van*/
```

Az inicializáló lista lényeges tulajdonsága, hogy az adattagokat a megadott értékekkel inicializálva hozza létre, míg a konstruktor törzsében a már létező adattagot módosítjuk. Csak így lehet egy **const** jelzővel kötött privát adat kezdőértékét megadni.

A használat során nem látszik, hogy listával vagy kifejtéssel kapnak-e kezdőértéket az adattagok. Ugyanakkor az objektumok létrehozására és kezdőértékének megadására van egy rövid inicializálási lehetőség: kapcsoszárójelek között adunk meg értékeket.

```
202.  malacok[1] = Malac{"Röfi", "fa" };
```

Ebben csak a kódba írt adatok lehetnek, a létező konstruktorok valamelyikével értelmezhető adatokkal. Ezzel szemben az inicializáló listában az értékadás lehet változó, számított érték vagy konstruktor. Ez a kétféle inicializálás „üti” egymást. Ha nem írunk elő semmit, akkor a két adat „megtalálja a helyét”. Ha írunk konstruktort, akkor annak a paraméterlistájához kapcsolja az adatokat, az inicializáló lista elemeinek a zárójelében is ezek a paraméterek szerepelhetnek.

60. példa: A tanuló osztálya a `tanulo` osztály.

Készítsünk egy objektum osztályt a tanulók vezetéknevének, keresztnevének és évfolyamának tárolására! Próbáljuk ki, hogyan használhatjuk, ha csak a privát és publikus adatokat adunk meg, majd írjunk konstruktort is.

```
 1. #include <iostream>;
 2. using namespace std;
 3.
 4. class tanulo
 5. {
 6.   int evf;
 7. public:
 8.   string vez;
 9.   string ker;
10.   int get_evf(return evf);
11.   void set_evf(int value){evf = value; }
12. };
13.
```

Az `evf` **`private`** adat, az értékét a 11. sorban látható `get_evf()` függvény mutatja meg, módosítani a 12. sorban látható `set_evf()` függvénnyel lehet. Miután megírtuk az adattagokat, az alapértelmezett (default) konstruktor segítségével már használhatjuk is:

```
27. int main()
28. {
29.   tanulo egyik;
30.   tanulo masik;
31. cout << egyik.ker << "*" << masik.get_evf() << endl;
```

Használhatjuk, de a privát `evf` csak a `get_evf()` függvénnyel érhető el, ráadásul bármi lehet. Módosíthatjuk az adatokat, de a privát adathoz csak publikus függvényen keresztül férünk hozzá:

```
32.   egyik.vez = "Lopakodó";
33.   egyik.ker = "Kasztanyetta";
34.   //egyik.evf = 6; //nem megy.
35.   egyik.set_evf(6);
36.   masik.vez = "Surranó";
37.   masik.ker = "Szalicil";
38.   masik.set_evf(6);
39.   cout << egyik.ker << " " << masik.get_evf() << endl;
40.   masik.set_evf(masik.get_evf() + 1);
41.   egyik.ker = egyik.ker + " " + "Harmónia";
42.   cout << egyik.ker << " " << masik.get_evf() << endl;
```

Lehetne másképp is? Igen. Egyrészt megfontolandó, hogy mi legyen privát, mi legyen publikus. Másrészt, lehet az adatokat másképp értelmezni. 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.

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 tanulság, 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 paraméterei 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. Az alábbi példában a kor megadása a lényeg, a többi „magától” is "" lenne.

```
tanulo(int k, string v = "", string k = "")
```

tanulo osztályunkhoz két konstruktort írunk. Most a vezetéknév a meghatározó:

```
14.   tanulo(string vn, string kn = "", int ev = 0)
15.   {
16.     vez = vn;
17.     ker = kn;
18.     evf = ev;
19.   }
```

A fenti konstruktor használatakor meg kell adni a vezetéknevet, ezt követően megadhatjuk a keresztnevet és az évfolyamot is. Ha az évfolyamot meg akarjuk adni, akkor a keresztnevet is meg kell adnunk.

```
20.   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.

```
{
  tanulo egyik
  tanulo masik
  tanulo harmadik()
```

```
tanulo::tanulo(string vn, string kn = "", int ev = 0)
    (2/2)
```

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.   setlocale(LC_ALL, "Hun");
28.   tanulo harmadik("Ó", "Pál");
29.   cout << "Add meg a diák vezetéknevét, keresztnevét és évfolyamát
                szóközökkel elválasztva: ";
30.   string v, k;
31.   int e;
32.   tanulo negyedik = tanulo(v, k, e);
33.   tanulo[] team Tanulo[4] {egyik, masik, harmadik, negyedik };
34.   for (int i = 0; i < 4; i++)
35.     cout << i + 1 << ". " << team[i].vez << " " << team[i].ker << " "
             << team[i].get_evf() << endl;
36. }
```

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, fajját 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.  #include <iostream>
2.  #include <fstream>
3.  #include <sstream>
4.  #include <vector>
5.  using namespace std;
6.
```

A megoldás során olvasunk és írunk konzolra, olvasunk fájlból és a konstruktorban használjuk a stringsteamet. Az állatok listájához `vector<>`-t használunk. Az első feladat az **`allat`** osztály megírása.

```
7.  class allat
8.  {
9.  public:
10.   string nev;
11.   string faj;
12.   int kor;
```

A második feladat megoldásához, a fájlból beolvasott sort egyszerűbb konstruktorban bontani, mint a programban, a másik konstruktor a paraméter nélküli lesz.

```
13.   allat(string sor)
14.   {
15.     stringstream ss(sor);
16.     ss >> nev >> faj >> kor;
17.   }
18.   allat()
19.   {}
20. };
21.
```

A fájlból beolvasást a `main()` programban végezzük, a feladatban kért listába, majd kigyűjtjük a kért adatokat:

```
22. vector<allat> allatok;
23. int main()
24. {
25.   ifstream fin("allatok.txt");
26.   string sor;
27.   while (getline(fin,sor))
28.      allatok.push_back(allat(sor));
29.   fin.close();
```

```
30.   for (allat a : allatok)
31.     if (a.faj == "malac")
32.       cout << a.nev + " ";
33.  }
34.
```

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.  #include <iostream>
 2.  #include <cstdlib>
 3.  #include <ctime>
 4.  using namespace std;
 5.  class macska
 6.  {
 7.  public:
 8.    string nev;
 9.    int szev;
10.    int tomeg;
11.    int tapasztalat;
12.
```

A véletlenszám előállítás érdekében betöltjük a `<cstdlib>` és `<ctime>` csomagokat is, az inicializálás majd a main()-ben lesz, mert csak egyszer kell lefutnia. A konstruktorban pont úgy használjuk az `rand()` függvényt, mint eddig. A többi paraméternél figyeljünk arra, hogy a paraméter neve ne legyen azonos az adattag nevével:

```
13.    macska(string neve, int ev, int tomege)
14.    {
15.      nev = neve;
16.      szev = ev;
17.      tomeg = tomege;
18.      tapasztalat = rand() % 40 + 10;
19.    }
```

A `main()` eljárásban hozzunk is létre két macskát:

```
 1.  int main()
203.  {
204.    setlocale(LC_ALL, "Hun");
205.    srand(time(0));
206.    macska cs = macska("Csillus", 2018, 3652);
207.    macska z("Zokni", 2017, 4003);
208.
```

Az osztály műveleteit függvényekkel (eljárásokkal) valósítjuk meg. Ez a függvény annyiban tér el az eddig megszokott függvényektől, hogy a **macska** osztályon belül van. 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.)

```
20.   void nyavog()
21.   {
22.     cout << "Nyaú! ";
23.   }
```

A második tagfüggvényünk azt szemlélteti, hogy a tagfüggvényeknek is lehet visszatérési értékük. A példában szereplő tagfüggvény egy paramétert is felhasznál a visszatérési érték kiszámításához.

```
25.   int kor(int iden)
26.   {
27.     return iden - szev;
28.   }
```

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:

```
209.   cs.nyavog();
210.   cout << cs.nev <<" kora 2022-ben: "<< cs.kor(2022) <<" év."<< endl;
```

A harmadik tagfüggvény arra mutat példát, hogy miként lehetséges egy jellemző értékének megváltoztatása tagfüggvény hívásával:

```
30.   void eszik()
31.   {
32.     tomeg += 25;
33.   }
```

… és a használata:

```
211.   cout << cs.nev << " tömege evés előtt: " << cs.tomeg << " g";
212.   cs.eszik();  /*nem kell paraméter, mert saját adattal számol*/
213.   cout << "... és evés után: " << cs.tomeg << " g." << endl;
214.
```

Természetesen egy tagfüggvény többször is hívható:

```
215.   cout << z.nev << " tömege zabálás előtt: " << z.tomeg << " g";
216.   for (int i = 0; i < 10; i++)
217.     z.eszik();
218.   cout << "... és zabálás után: " << z.tomeg << " g." << endl;
```

Az osztály utolsó előtti tagfüggvénye pedig azt példázza, hogy egy tagfüggvény egészen bonyolult műveleteket is megvalósíthat. Az egerészéshez kell egy egér, valamilyen ésszel, menekülési képességgel. Ezt a macska tapasztalatához viszonyított véletlen értékkel jellemezzük.

```
35.   void egereszik()
36.   {
37.     int eger_esze = rand() % 100 + 1;
38.     cout << nev << " egy " << eger_esze << " tapasztalatú egérrel
            próbálkozik. => ";
```

```
39.     if (tapasztalat > eger_esze) /*a macska ügyesebb => nyer*/
40.     {
41.       cout << "Megfogta" << endl;
42.       if (tapasztalat < 100)
43.         tapasztalat++;    /*ha lehet, nő a tapasztalata is*/
44.       eszik();                  /*… és megeszi az egeret.*/
45.     }
46.     else                         /*az egér ügyesebb =>*/
47.     {
48.       cout << "Elszaladt :( ";
49.       nyavog();                 /*nem eszik, hanem nyávog.*/
50.       cout << endl;
51.     }
52.   }
```

Természetesen ezt is ki kell próbálni: Kövessük végig valamelyik macskánk egerészéseinek egy hosszabb sorozatát:

```
219.   cout << z.nev << " egy hete" << endl;
220.   for (int i = 0; i < 50; i++)
221.   {
222.     cout << z.nev << " tömege: " << z.tomeg << ", tapasztalata: ";
223.     cout << z.tapasztalat << endl;
224.     z.egereszik();
225.  }
```

Végül, három nagyon haladó kiegészítés:

- Tipikus feladat az objektum összes adatának a kiírása. Az adatok szöveges megjelenítésére írhatunk tagfüggvényt, de C++ nyelven jellemzőbb az extractor kibővítése. Ha van egy objektumosztályunk, ahhoz megadhatjuk, hogy a `<<` hogyan értelmezze. Hasonlóan az insertert is megírhatjuk az objektumhoz, ami a beolvasást segíti. Mivel az ostream nem része az objektumnak, azt, hogy mi történjen vele, nem az új osztálydefinícióban adjuk meg, hanem utána.

```
62.  ostream& operator<<(ostream& os, const macsek m)
63.  {
64.    os<< nev <<" születési éve: "<< szev <<", tömege: "<< tomeg <<" g.";
65.    return os;
66.  }
```

- 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 új objektumokra az új értelmezést: mit jelentsen a programnak a ’+’, a ’<’ stb. A részletekhez az „C++ operátor overloading” kifejezésre érdemes rákeresni. A kivitelezéshez lényeges, hogy ezek publikus, tagfüggvényként vagy az `<<` mintájára a class után írhatók.

```
53.   bool operator< (macska rt)
54.   {
55.     return szev > rt.szev;
56.   }
57.  };/*itt a class vége belül a bal operandus az objektum*/
```

```
58.   bool operator> (macska lt, macska rt)
59.   {
60.     return lt.szev < rt.szev;
61.   }
```

Próbáljuk is ki a kiírást és a relációs jel új használati módját:

```
226.   cout << "Végül: a fiatalabb macskánk:" << endl;
227.   cout << "\t" << (cs < z ? cs : z) << endl;
```

- 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** (többrétegűség). 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**.tapoteszik() 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 nőstényt 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 csomagok elfoglalnak néhány sort, így az osztálydefiníció a 6. sorban kezdődik.

```
 6.  class diak
 7.  {
 8.    string osztaly;
 9.    static const int MaxTZDB = Max_TZ_DB;
10.    int tzdb;
11.  public:
12.    string nev;
13.    int evfolyam;
14.    string tagozat;
15.    vector<int> rendes_jegyek;
16.    int tz_jegyek[MaxTZDB];
17.    int tz_db(){return tzdb;}
18.    void uj_tz_jegy(int jegy){tz_jegyek[tzdb] = jegy; tzdb++;}
```

Programunk új verziójában a **struct** helyett **diak class** szerepel. a korábbi osztály adattag helyett az `evfolyam` és a `tagozat` tulajdonságokat fogjuk használni, de eltároljuk az osztály jelölését is: az `osztaly` privát, belső változó. Fontos jelzés, hogy az `evfolyam` és a `tagozat` látható és módosítható később is, eközben az osztály rejtve van, marad az, 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 íratni, 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 dolgozatok maximális száma lényegét tekintve belső, konstans egész szám, aminek most a programban előírt konstans értéket adjuk meg.

A másik témazáró adat a `tzdb`. Ez minden diák esetén egyedi, attól függ, hány dolgozatjegy van beírva a `tz_jegyek` tömbbe. A program futása során szükséges lehet az adat kiírására, ezért publikus, de ne akarjuk módosítani, mert az a program összeomlásával járhat. Legyen privát és készítsünk hozzá gettert. De ezzel csak félig oldottuk meg a problémát, mert egy jegy beírásakor módosul az értéke … Ezért írunk egy eljárást a tz jegy beírására. Így legalább lehet jól használni, de amíg a `tz_jegyek` publikus, addig megkerülhető a védelem, a teljes tömbnek privátnak kellene lennie és kellene az elemeihez getter és setter …

Amíg a programunk csak néhány száz soros és csak saját használatra készül, a fenti megállapítások irrelevánsok. Minden adathoz rendelhetünk 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 súgó jelzéseit, hogy egy-egy tulajdonságot hogyan használhatunk.

Például – a korábban már használt nyelvi elemek közül a diák neve `string` típusú objektum, a név hossza, a `length()` vagy `size()` tulajdonság, egy privát adatot olvasó függvény, getter. (Közkívánatra ugyanaz két néven.) Vegyük észre, hogy 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 adunk kezdőértéket az egyes tulajdonságoknak.

```
19.     diak(string sor)
20.   {
21.     stringstream s(sor);
22.     getline(s, nev, ',');
23.     s.ignore(1);
24.     getline(s, osztaly, ',');
25.     evfolyam = atoi(osztaly.substr(0,2).c_str());
26.     tagozat = osztaly.substr(osztaly.size()-2,2);
```

Az `evfolyam` és a `tagozat` az osztály adatából lesz megadva. Ezt követően dolgozzuk fel a jegyeket.

```
27.     tzdb = 0;
28.     string jegy;
29.     while (s >> jegy)
30.     {
31.       if (jegy.size() == 1)
32.         rendes_jegyek.push_back(jegy[0] - '0');
33.       else
34.         uj_tz_jegy(jegy[0] - '0');
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.

Míg el nem felejtjük, írjuk meg a paraméter nélküli konstruktort a tömb létrehozásához:

```
38.   diak()
39.   {
40.   }
```

A konstruktorokkal 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 futtatása során szeretnénk használni, ezért a **diak** osztályon belül publikusak lesznek, a **diak** osztály tagfüggvényeiként fognak megjelenni.

A **2. feladat**hoz hasznos, ha a „mindhárom témazárót megírta-e” kérdés helyett arra adunk választ, hogy minden előírt témazárót megírt-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:

```
43.   bool mindenTZmegvan()
44.   {
45.     return tzdb == MaxTZDB;
46.   }
```

A függvényünk szinte semmit sem csinál, de mégis fontos, mert az objektumorientált programozás elve alapján a két változó privát, de a válasz 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.

```
47.   int jegyekszama(string milyen)
48.   {
49.     if (milyen == "tz")
50.       return tzdb;
51.     else if (milyen == "rendes")
52.       return rendes_jegyek.size();
53.     else if (milyen == "mind")
54.       return tzdb + rendes_jegyek.size();
55.     else
56.       return -1;
57.   }
58.
```

Ha ez a függvény létezik, akkor a `tz_db()` nem szükséges.

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.

```
59.   double atlag()
60.   {
61.     double szum = 0;
62.     for (unsigned i = 0; i < rendes_jegyek.size(); i++)
63.       szum += rendes_jegyek[i];
64.     for (int i = 0; i < tzdb; i++)
65.       szum += 2 * tz_jegyek[i];
66.     return szum / (jegyekszama("mind") + jegyekszama("tz"));
67.   }
```

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 azzal, hogy az öttizedet merre kerekíti

```
70.   int osztalyzat() /*év végi*/
71.   {
72.     double atl = atlag();
73.     if (atl <= 1.7)
74.       return 1;
75.       return atl; /*ahogy a C++ kerekít*/
76.   }
77. };                              /*ITT a diak osztály vége!!!*/
```

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 és a függvények deklarációját – 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.  int main()
94.  {
95.    setlocale(LC_ALL, "Hun");
96.    N = 0;
97.    string sor;
98.    ifstream fin ("naplo.txt");
99.    while (getline(fin, sor))
100.   {
101.     csoport[N] = diak(sor);
102.     N++;
103.   }
104.   fin.close();
```

Látható, hogy az objektumok példányosítása ugyanúgy történik, ahogy egy struktúrájával megadott adatot létrehoztunk.

```
105.   cout << "1. A 11.zs tanulói:" << endl;
106.   kiir_zs();
107.   cout << "2. Kevés témazárót írtak:" << endl;
108.   keves_tz();
109.   cout << "3. 11.x-ből legkevesebb témazárót ";
110.   cout << min_tz() << " írta." << endl;
111.   cout << "4. Felelésre felkészülők:" << endl;
112.   felelesveszely();
113.   cout << endl;
114.   cout << "5. Év végi átlagok és jegyek:" << endl;
115.   evvege();
116. return 0;
117. }
```

Az egyes feladatok megoldásában felhasználjuk a **diak** osztály tulajdonságait és függvényeit

```
120. void kiir_zs()
121. {
122.   for (int i = 0; i < N; i++)
123.     if (csoport[i].evfolyam == 11 && csoport[i].tagozat == "zs")
124.       cout <<'\t' << csoport[i].nev << endl;
125. }
```

```
127.  void keves_tz()
128.  {
129.    for (int i = 0; i < N; i++)
130.      if (!csoport[i].mindenTZmegvan())
131.      {
132.        cout << '\t' << csoport[i].nev + ": ";
133.        for (int j = 0; j < csoport[i].tz_db(); j++)
134.          cout << csoport[i].tz_jegyek[j] << ", ";
135.        cout << "\b\b " << endl;
136.      }
137.  }
138.
140.     string min_tz()
141.  {
142.    int mini = -1;
143.    int mindb = Max_TZ_DB + 1;
144.    for (int i = 0; i < N; i++)
145.    {
146.      if (csoport[i].evfolyam == 11 && csoport[i].tagozat == "x")
147.        if (csoport[i].jegyekszama("tz") < mindb)
148.          {
149.          mindb = csoport[i].jegyekszama("tz");
150.          mini = i;
151.        }
152.    }
153.    return mini == -1 ? "nincs 11x-es diák" : csoport[mini].nev;
154.  }
155.
```

Elgondolkodtató, hogy a `jegyekszama()` függvényt itt minden ciklusléptésben kétszer futtatjuk. Ez itt csak egy elágazás és hivatkozás a `rendes_jegyek` méretére, 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.

```
157.  void felelesveszely()
158.  {
159.    /*legtöbb rendes jegy*/
160.    int maxe = csoport[0].jegyekszama("rendes");
161.    for (int i = 1; i < N; i++)
162.      if (csoport[i].jegyekszama("rendes") > maxe)
163.        maxe = csoport[i].jegyekszama("rendes");
164.    /*kigyűjtés*/
165.    for (int i = 0; i < N; i++)
166.      if (csoport[i].jegyekszama("rendes") < maxe * 0.8)
167.        cout <<'\t' << csoport[i].nev << endl;
168.  }
170.  void evvege()
171.  {
172.    for (int i = 0; i < n; i++)
173.    {
174.      cout << "\t " << left << setw(25) << csoport[i].nev;
175.      cout<<" átlaga: "<<fixed<<setprecision(2)<< csoport[i].atlag();
176.      cout << "\tjegye: " << csoport[i].osztalyzat() << "." << endl;
177.    }
178.  }
```

Kiegészítések egyedi alkalmazások készítése elé

Modulok, több fájlból álló programok

A C++ nyelven programozást úgy kezdtük, hogy a Code::Blocks létrehozott egy projektet egy mintakóddal. A kapcsoszárójel-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 kódon kívül előre megírt eszközök is kellenek. Például, ha csak a konzolra akarunk kiírni valamit, akkor a `cin` és `cout` használatához be kell vonni (`include`) programunkba az `<iostream>`-et. Később, szinte minden új eszközhöz újabb csomagokat kellett bevonni.

Egy „igazi” program 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 fájlt nehéz áttekinteni, ezért a programot projektként, több fájlban szokták megírni. A legáltalánosabb fájlokra bontási lehetőség, ha minden osztályt más fájlban írunk meg. A Code::Blocks fájl menüben nem csak új projektet lehet létrehozni, hanem `class`-t is. Ezzel két új fájl jön létre, egy `.h` és egy `.cpp`. Nagy vonalakban a `.h` fájlban vannak deklarációk, azok kódrészek, amelyeket a `main()` elé kell írni, a `.cpp` fájlban találhatók a definíciók, a kifejtések, amit írhatunk a `main()` után is.

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 az include-dal hivatkozott csomagokban több osztály is lehet, de jellemzően összetartozó dolgokat tesznek bele, amelyeket előzetesen nagyon alaposan tesztelnek. A hivatkozott csomagok egy-egy `.h` fájlnak (és rajtuk keresztül a `.cpp` fájlnak) felelnek meg, amelyekből a programban hivatkozott részeket a fordítóprogram felhasználja. Az általunk létrehozott class-t is a programunkba az `#include`-dal vehetjük be. Annyi az eltérés, hogy a kacsacsőrös jelölés helyett idézőjelek között kell megadni a .h fájl nevét, kiterjesztéssel együtt.

Nem `class`, nem `struct` – mi az?

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.  #include <iostream>;
2.  using namespace std;
3.
4.  enum SZINEK { red, blue, green };
5.  int main()
6.  {
7.    setlocale(LC_ALL, "Hun");
8.    string szinek[3] { "sárga", "lila", "fekete" };
9.    string t = szinek[0];
10.   SZINEK e = red;
11.   cout<<t<<" "<<(SZINEK)e<<" "<<(e + 1 == blue?"kék":"nem kék")<<endl;
12.     /*sárga          0                        kék                    */
13.   SZINEK s = (SZINEK)7;
14.   cout <<szinek[e + 1] << " " << s << endl; /*lila zöld*/
15.   return 0;
16. }
```

Az `enum` – teljes nevén enumerátor vagy felsorolás típus – elnevezéseket sorol fel. Ha nem adunk neki máshogyan értéket, akkor 0-tól számlálva nevet ad a számoknak. Lehet következőt kérni (pl. 11. sorban `e + 1`). Használhatjuk a szám helyett, de kiíráskor csak a szám értéke létezik, így „kifelé” nincs hatása, de jól használva a programkódunk olvashatóbb lehet.

63. példa: Mérés és értékelés – (enum és class fájl)

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` osztály fájljait, ebben definiáljuk a `meres` osztályt és a mérést jellemző `FAJTA` felsorolást! A `main()` eljárásban hozzunk létre egy témazáró mérést!**
2. **A `main.cs` fájlban definiáljuk a `jegy` osztályt, amelyet a `meres` osztályból származtatunk. A jegy elnevezéseit `jegyertek` 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.h` és `meres.cs` fájlt a File, New … menüpont választása után, a *Class* … menüre kattintva hozzuk létre. A létrehozás során számos beállításra van lehetőségünk, amelyekkel kódrészleteket készíthetünk elő. A panel alsó felén, a File policy minden jelölőnégyzetét érdemes bejelölni a továbblépés előtt.

A `meres.h` tartalma:

```
1.  #ifndef MERES_H
2.  #define MERES_H
3.  enum FAJTA { proba = 1, rendes, tz, vizsga = 5 };
4.  class meres
5.  {
6.  public:
7.    FAJTA suly;
8.    int szin;
9.    meres();
10.   meres(FAJTA);
11. };
12. #endif // MERES_H
```

Az első két és utolsó sor célja, hogy a fordító észrevegye, ha ugyanazt a kódot többször hivatkoznánk meg a programunkban. A megadott fordítási direktívákkal (feltételekkel) csak az első hivatkozást veszi figyelembe.

A **`meres`** osztálynak két tulajdonsága van, egy enumerátor és egy egész. A FAJTA értékei: 1, 2, 3, 5; az értékadás miatt hiányzik a 0 és a 4.

A `meres.cpp` tartalma:

```
1.  #include "meres.h" /*ide illeszti a fordító*/
2.  meres::meres()
3.  {
4.    suly = rendes;
5.    szin = 1;
6.  }
7.  meres::meres(FAJTA fajta):suly(fajta)
8.  {
9.    szin = fajta * 2 + 1;
10. }
```

A fájlban látható, hogy hogyan kapcsolódik a `.h` a `.cpp`-hez. Mivel egy fájlban akár több osztály is lehetne, a definíciók neveihez az osztály nevét is meg kell adni.

A `main.cp` fájlban megadjuk a `.h` fájlt, ezt követően használhatjuk a fájlokban írt programrészleteket.

```
1.  #include <iostream>
2.  #include "meres.h"
3.  using namespace std;
4.  int main()
5.  {
6.    setlocale(LC_ALL, "Hun");
7.    string tipus[6] {"", "próba", "rendes", "tz", "", "vizsga"};
8.    meres m(tz);
9.    cout << tipus[m.suly] << " szin: " << m.szin;
10.   return 0;
11. }
12.
```

A 8. sorban nem szövegesen adjuk meg a mérés típusát, hanem az egyik **FAJTA** értéket használjuk. Értelemesen, de a háttérben egy egész számot jelölve vele.

A jegyertek felsorolásnál is figyeljünk arra, hogy 0-s jegy nincs. A példától eltérően, nem kell minden értéket megadni, illetve, ha megadnunk minden értéket, akkor nem kell növekvő sorban írni.

A felsorolást és a jegy osztályt is a fájl elején adjuk meg, a `main.cpp` fájlban, kiegészítve az eddigieket.

```
4.  enum JEGYERTEK {elegtelen=1, elegseges=2, kozepes=3, jo=4, jeles=5};
```

A **jegy** „származtatása” a **meres** osztályból azt jelenti, hogy a Jegy egy **meres**, ami további speciális tulajdonságokkal bír.

```
5.  class jegy : public meres /*Kettőspont, a meres publikus rész lesz*/
6.  {
7.  public:
8.    JEGYERTEK ertek;
```

Bár a **jegy**nek csak egy saját tulajdonsága van, de a **meres** tulajdonságaival is rendelkezik. Ezért a konstruktorában megadjuk a **JEGYERTEK** és a **FAJTA** adatot is, és azt is megmondjuk, hogy a fajta az „ősosztály”, **meres** létrehozásához kell.

```
 9.   jegy(JEGYERTEK jegy, FAJTA fajta):meres(fajta)
10.   {
11.     ertek = jegy;
12.   }
```

A származtatott osztály nem adattagként tartalmazza az ősét, 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.

```
13.   jegy(meres p, JEGYERTEK erteke):meres(p.suly)
14.   {
15.     ertek = erteke;
16.   }
17. };
```

Tehát: a `p` egy **meres** típusú adat, ennek a `suly` tulajdonsága **FAJTA** típusú, amivel paraméterezhető az ős, így kap értéket a `suly` és a `szin`. Ezután a **JEGYERTEK** típusú `ertek` tulajdonságot is megadjuk.

A kipróbálás a `main()` függvényen belül:

```
23. jegy a(jo, tz);
24. cout << "A jó tz: " << a.ertek << " súlya: " << a.suly;
25. cout << " színkódja: " << a.szin << endl;
```

A feladatokat megoldottuk. Próbáljunk ki többféle értékadást, változatotatást, figyeljük meg az adathozzáférés jellemzőit!

Modulok és grafikus felhasználói felületű alkalmazások

Konzol alkalmazásainkhoz számos csomagot használtunk. Ha (óvatosan) kutakodunk a Code::Blocks programmappában, megtalálhatjuk ezek `.h` fájlját. Ha valamit nagyon elrontunk a kód írása során, akkor a hiba helyét a megfelelő kódfájlban jelzi az IDE.

Láthatjuk, hogy a saját készítésű **meres** osztály `meres.cpp` és `meres.h` fájlai a programban a **vector**-hoz hasonlóan használhatók: az include után megadjuk a .h fájlt. Ha ez a fájl „saját készítésű”, akkor idézőjelek között, szükség esetén elérési útvonallal adjuk meg. Ha a fájl a nyelv része – a fordítóprogramba integrált –, akkor relációs jelek közé írjuk.

Programjaink megírásához több, mint egy tucat csomagot használtunk. Ezek közül több egy az OOP (objektum orientált paradigma) elveit megvalósító osztály leírását tartalmazza. Ilyenek a ...stream csomagok, amelyeknek közös „ős objektuma” a stream, a vector és a map. Más csomagokban egyes feladatok megoldásához szükséges, többféle objektumhoz megírt függvények találhatók. Jegyzetünkben ezeknek a neve tipikusan c-vel kezdődik, ami arra utal, hogy eredetileg a csomag a C nyelvhez készült .h fájl.

Egy-egy feladat megoldásához csak néhány (1–3) csomagra volt szükségünk, de ahogy nő a feladat, úgy bővül a megoldást segítő előre megírt kódok mennyisége, így a felhasználható csomagok száma is. Ezért a GCC fordítóhoz elkészítették az a csomagot, amelyik tartalmazza az „összes” .h fájlra történő hivatkozást. Ezzel egy csomagban kapunk kb. 80 csomagot

```
bits/stdc++.h
```

Ez, csakúgy, mint az általunk írt osztály, nem szerves része a C++ nyelvnek, ezért kell az elérési út, de a fordítóprogram része, ezért relációs jelek közé kell írni.

```
#include <bits/stdc++.h>
```

A sok csomag helyett könnyű lenne csak ezt az egyet használni, de nagyon kellemetlen, ha egy másik fejlesztőkörnyezetben kell dolgozni (például egy projekt kapcsán), amelyikben nem a GCC fordítja le a programunkat. Ilyen például a Visual Studio, amelyik az MSVC-t használja, vagy más, speciális célra készített fordítóprogramok.

Ha speciális alkalmazást szeretnénk készíteni, akkor telepíteni kell a Code::Blocks-ba a megfelelő csomagot. Ezt követően az új projektünk célja lehet például egy `.dll` fájl készítése, adatbázis-kezelő elérése vagy egérkezelést és grafikus felületet tartalmazó grafikus alkalmazás. Ilyen például az SDL. Elindulhatunk fordítva is. A Qt egy grafikus programok készítésére tervezett IDE, amelyben minden platformra készíthetünk alkalmazást, többek között C++ nyelven is. De ez már egy másik történet …

#INCLUDE PUSKA

iostream	konzolról olvasás, konzolra írás
fstream	fájlból olvasás, fájlba írás
sstream	programon belül szövegfeldolgozáshoz
iomanip	inserter kiegészítése formátummal (setw, setprecision, left)
string	szöveg függvények, pl: substr()
cctype	karakter függvények pl. isalpha(), toupper()
ctime	gépi dátum-idő, véletlen kezdőállapothoz; most = time(0)
cstdlib	véletlenszám készítéséhez inicializálás srand(time(0)), értékhez rand()
vector	dinamikusan átméretezhető tömb (indexelhető)
map	kulcs-érték párok sorozata; pl. szótár, gyakoriság tárolásához
tuple	adatok egymáshoz rendelése; pl: a map egy adata vagy <elemszam, tömb>
algorithm	adatsorozatok tipikus algoritmusainak általános megvalósítása
cmath	matematikai függvények (abszolútérték, trigonometria, hatvány)
numeric	sorozatszámítás függvényei pl: accumulate()
All in One:	
bits/stdc++.h	nem része a C++-nak, de 80 C++ headert tartalmaz (#include … sorokból áll) csak a gcc fordító ismeri (Code::Blocksban működik, Visual Studioban nem) esetenként nagyon nagy méretű lesz a fájl a felesleges headerek miatt.

TÁRGYMUTATÓ