9 Algoritmizálás és a C++ programozási nyelv használata

Előszó

Ez a jegyzet a Digitális kultúra tantárgy 9. évfolyamos tananyagának az Algoritmizálás és programozási nyelv használata témát dolgozza fel. Tematikájában, legtöbb feladatában és megjelenési formájában az állami tankönyv* és online tananyag az alapja, ahol lehet, ott 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.

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

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

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

Sikerekben gazdag tanulást kívánok:

Budapest, 2023.

Szalayné Tahy Zsuzsanna

* Digitális kultúra tankönyv 9. Oktatási Hivatal 2020.

TARTALOM

Mi az a programozás?

Mi a „program”?

A számítógép, a telefon, az összes olyan eszközünk, amiben valamilyen „számítógép” van, önmagában képtelen ellátni azt a feladatot, amire készült. Csak egy darab „vas” – azaz hardver. Csak akkor képes igazán működni, ha fut rajta egy (vagy sok) program, alkalmazás – azaz szoftver. Szoftver, program, alkalmazás – nagyjából ugyanazt jelenti: azt a programozó, a szoftverfejlesztő által megírt valamit, ami elmondja a hardvernek, hogy mikor mit „csináljon”.

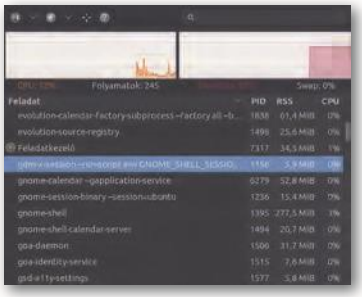
Program mondja meg

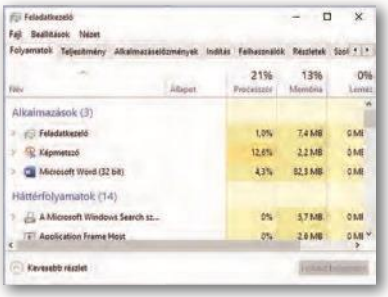
- a kenyérsütő-gépnek, hogy meddig gyúrja a tésztát, meddig hagyja kelni, és mikor kezdje sütni, mennyire legyen meleg a fűtőszál, hányat sípoljon a sütő, amikor kész a kenyér;
- a mosógépnek, hogy mikor és mennyi vizet szívjon be, mennyire melegítse fel, meddig és merre forogjon benne a dob, mikor és meddig kell centrifugálnia.

Ezek a számítógépek egyetlen programot futtatnak. Az informatikaórán bennünket jobban érdekelnek a hagyományos értelemben vett számítógépek (laptopok, asztali gépek, szerverek) és a mobileszközök. Ezek csak bekapcsoláskor futtatnak egyetlen programot, ami azt mondja el nekik, hogy honnan és hogyan kell betölteniük a „fő” programjukat: az operációs rendszert. A többi program (a böngésző, az üzenetküldő, a játék, a képszerkesztő, a szövegszerkesztő, a filmvágó stb.) pedig az operációs rendszerből, annak felügyelete alatt indul el, amikor rákattintunk az egérrel vagy rábökünk az ujjunkkal az indítóikonjára, esetenként automatikusan.

Hol vannak a programok?

Az elindított, pontosabban a futó programok a számítógép memóriájában vannak. Nemcsak a program van itt, hanem az általa éppen használt adatok is: a szövegszerkesztő által szerkesztett szöveg, a képszerkesztőbe betöltött kép. Az operációs rendszerünk feladatkezelőjében megnézhetjük az épp futó programokat. Látjuk, hogy a legtöbbet nem mi indítottuk el, sőt nem is látjuk őket – a háttérben futnak.





Egy Linuxon futó feladatkezelő és a Windows feladatkezelője – Indítsunk el egy programot, keressük meg a feladatkezelőben és állítsuk meg innen! *

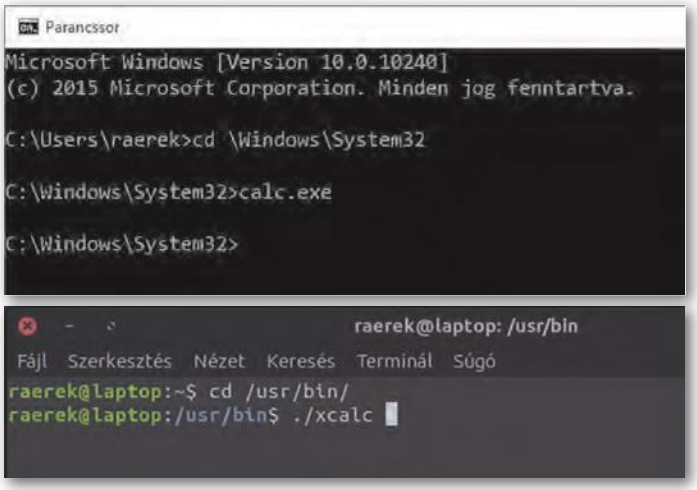
Amíg a programot nem indítjuk el, a számítógép háttértárán, például a laptop SSD-jén, az asztali gép vagy szerver winchesterén, vagy a telefon memóriakártyáján van, ugyanolyan fájlként,

* Digitális kultúra 9. tankönyv 93. oldalán látható kép

mint a képek, a zenék vagy a szövegek. Az egyszerű programok egyetlen fájlból állnak, az összetettek sokszor nagyon sokból.

A grafikus felületű operációs rendszerek elterjedése előtt (az 1990-es évekig) a programokat úgy indítottuk el, hogy a parancssoros felületben beléptünk abba a mappába (könyvtárba), amelyikben a programunk volt és beírtuk a program nevét. A módszer ma is működik, bár többnyire csak a számítógépekhez jobban értő emberek, rendszergazdák, rendszermérnökök és szoftvereket készítők használják. Lévén e fejezet célja épp az, hogy kicsit mi is szoftvert készítsünk, ismerkedjünk meg ezzel a módszerrel!

1. Nyissunk a gépünkön parancssoros felületet: Windowson indítsuk el a Parancssor nevű alkalmazást, macOS-en és Linuxon pedig valamelyik terminált!
2. „`cd`” mappaváltó paranccsal lépkedjünk abba a mappába, ahol a programfájl van (minden sor begépelése után ENTER-t nyomunk)!
3. Írjuk be a program nevét, és nyomjuk meg az ENTER-t!



Program indítása parancssorból Windows 10-en és Ubuntu Linuxon *

A program ilyenkor betöltődik a számítógép memóriájába és a benne lévő utasítások végrehajtódnak, azaz a program futni kezd.

Mi van egy programfájlban?

Ha már úgyis a parancssorban, az imént elindított programunk mappájában vagyunk, adjuk ki

- Windowson a `type`
- macOS-en és Linuxon a `cat`

parancsot, és írjuk utána a programfájl nevét (például `type calc.exe`, `cat xcalc`)!

Rengeteg krikszkrakszot ír a parancssori ablakba a gép. Ha némileg hihetetlen is, a számítógép ezt érti, ebből tudja, hogy mit kell csinálnia. Ez a program egyik alakja, az úgynevezett *gépi kódú* program, amely most a képernyőn karakterek formájában jelenik meg.

Szerencsére a legtöbb szoftverfejlesztőnek nem így kell megfogalmaznia a gép teendőit. Rendelkezésünkre állnak programozási nyelvek, azaz az angol nyelv szavait használó magasabb

* Digitális kultúra 9. tankönyv 94. oldalán látható kép

szintű nyelvek. Ilyen például a C, a C#, a Python, a Pascal, a Ruby, a Go, a Perl, a JavaScript, a Java és még sorolhatnánk. Ilyen az érdeklődésünk fókuszába emelt **C++** is. A szoftverfejlesztő többnyire valamelyik programozási nyelven írja a program **forráskódját**.

Egy egyszerű forráskódot többé-kevésbé már most is tudunk értelmezni. Mit csinál az alábbi, C++ nyelvű program?

```
#include <iostream>
using namespace std;
int main()
{
  setlocale(LC_ALL, "");
  cout << "Üdv néked!" << endl;
  int evek_szama;
  cout << "Hány éves vagy?";
  cin >> evek_szama;
  if (evek_szama < 14)
    cout << "Jé, hogyhogy már középiskolás vagy?";
  } else {
cout << "Egy év múlva" << evek_szama + 1 << "éves leszel.";
}
  return 0;
}
```

Nos, ilyen és ehhez hasonló programokat fogunk mi is írni az elkövetkezendő órákon. Látjuk, hogy az angol szavak mellett van még a programban írásjel, műveleti jel, kerek és kapcsos zárójel – ezek mind a program részei, nem hagyhatók el. C#-ban az utasítások végét pontosvessző jelzi. A programot lehetne egy sorba is írni, de az olvashatóságot, a kód értelmezését segíti az egységek kezdetét és végét jelző kapcsos zárójelek külön sorba írása és a belső tartalom beljebb kezdése.

Természetesen ezt a programkódot a számítógép ebben a formában nem érti, és nem tudja futtatni. A fenti forráskódot egy másik program (C++ esetén a `g++.exe`) előbb elemzi, gépközeli nyelvre fordítja, linkeli – **build**-eli – és létrehozza a futtatható fájlt. Azok a programozási nyelvek, amelyek fordítóprogramja a háttértárra is menti a bináris kódot (`.exe`-t) a **compiler**ek (fordítók). Ettől eltérően, csak a memóriában hozzák létre a bináris kódot (nem készül `.exe` fájl) az **interpreter**ek (értelmezők). Ezeknek a működése hasonló a böngészőkhöz, amelyek a szöveges HTML és CSS kódot értelmezve jelenítik meg a weblapot. A C++ compiler, a Python interpreter nyelv.

Feladatok

1. Az alábbi Python, illetve C#-nyelvű kódok a fenti C++ nyelvűvel megegyező működésű programot eredményeznek. Keressük meg a hasonlóságokat, mutassunk rá a különbségekre!

Python:

```
print('Üdv néked')
évek_száma = input('Hány éves vagy?')
évek_száma = int(évek_száma)
if évek_száma < 14:
  print('Jé, hogyhogy már középiskolás vagy?')
else:
  print('Egy év múlva', évek_száma+1, 'éves leszel')
```

C#:

```
1.  using System;
2.  namespace Valami
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        Console.WriteLine("Üdv néked!");
9.        Console.Write("Hány éves vagy?");
10.       string ev = Console.ReadLine();
11.       int evek_szama = int.Parse(ev);
12.       if (evek_szama < 14)
13.         Console.Write("Jé, hogyhogy már középiskolás vagy?");
14.       else
15.         Console.Write("Egy év múlva {0} éves leszel.", evek_szama + 1);
16.     }
17.   }
18. }
```

2. Rakjunk össze egy másik programot az alábbi részletekből! (Használjunk fel minden darabot! Egy-egy darab többször is használható.)

Csak ennyiből áll egy szoftverfejlesztő munkája?

Nos, igen, a fejlesztői munka legismertebb része az új programok írása.

- Sokkal többen vannak azok, akik meglévő programokat alakítgatnak át az új követelményeknek megfelelően (nekik köszönhetők például a telefonjainkra letöltendő frissítések).
- Van, aki azzal foglalkozik, hogy egy meglévő program fusson másféle gépen is – ez néhány esetben gyorsan megoldható, más programoknál nehéz és kimerítő feladat.
- Vannak, akik azért dolgoznak, hogy egy meglévő program legyen gyorsabb.
- Van, aki programokat tesztel: megnézi, hogy biztosan jól működnek-e minden helyzetben.
- Van, aki azzal foglalkozik, hogy programot, szoftverrendszert tervez. Ő már nem ír kódot, hanem azért felel, hogy a szoftver különböző részei minél jobban tudjanak együttműködni.
- Van, aki biztonsági ellenőrzést végez programokon, például azért, hogy számítógépes bűnözők ne tudják a banki szoftverekkel átutaltatni a pénzünket másik számlára.

A következő leckében mi is megírjuk első programunkat.

Kérdések

1. Mik azok a számítógépes vírusok?
2. Milyen más feladatok merülhetnek fel egy szoftver elkészítésekor? Milyen képzettségű munkatársai vannak a szoftver fejlesztőjének?
3. Milyen kép él benned a programozókról? Milyen előítéletek kapcsolódnak hozzájuk?

4. Milyen világszerte ismert oldalakon foglalkoznak programozási kérdések megválaszolásával? Hány programozással kapcsolatos videó készül naponta?

Első programjaink

A programozási környezet

A programozási (fejlesztői) környezet arra való, hogy benne írjuk meg programjainkat, használatával a programot gépi kódúvá alakítsuk és tesztelhessük, dokumentálhassuk a kész programot. Ezek közül a legfontosabb a gépi kódúvá alakítás, a többit vagy meg tudjuk oldani egy adott környezetben, vagy nem.

A programozási környezetet telepítenünk kell a gépünkre. A C++ nyelvhez legjellemzőbben használt környezet a Code::Blocks, ami a `codeblocks.org/downloads/binaries/` oldalról tölthető le. Válasszuk az operációsrendszernek megfelelő telepítők közül azt, amelyiknek a nevében szerepel a `mingw` és a `setup` is. Az utóbbi egyszerűvé teszi a telepítést; a `mingw` azt jelzi, hogy a fejlesztőkörnyezethez hozzátették a C++ nyelvű csomagot és fordítóprogramot, a `g++.exe`-t.

Másik gyakori fejlesztőkörnyezet a *Visual Studio width C++*, ami a `visualstudio.microsoft.com` webhelyről tölthető le. Válasszuk az operációs rendszernek is megfelelő közösségi (Community) – ingyenes – verziót! Ha már van telepítve Visual Studio (nem Code!), akkor elég, ha bővítjük a programot a *Desktop development with C++* csomaggal. A *Visual Studio Code* szintén alkalmas C++ nyelvű programozásra, csak fel kell telepíteni néhány ajánlott kiegészítést (extension). Ez azoknak ajánlható, akik egyébként is használják – például weblapkészítéshez – a programot.

Az IDE

A Code::Blocks, a Visual Studio (és a fapadosabb Visual Studio Code is) **integrált fejlesztői környezet** (Integrated Development Environment), azaz **IDE**. Egy program elkészítéséhez általában elegendő egy egyszerű szövegszerkesztő és egy fordítóprogram. Azonban ezekkel az eszközökkel nehéz gyorsan és jól programozni, megtalálni az elírásokat, hibákat. Ezért az egyes nyelvekhez programozást támogató eszközöket készítettek, amelyek egy alkalmazásba foglalása az IDE. Ilyen támogató eszközök:

- a kód funkció szerinti színezése;
- a fordító számára nem érthető kódrészlet (hiba) jelzése;
- a program lépésenkénti futtatása, pillanatnyi állapotának a kijelzése;
- a helyérzékeny javaslat a kódra, kódkiegészítésre;
- rövid szóból összetett kódrészletek megjelenítése (snippetek);
- nem használt, valószínűleg felesleges kódrészletek jelzése;
- nem szabványos, esetleg problémás, de működő kódra („code-smell”) figyelmeztetés;
- memóriafoglalás és futási idő figyelése, elemzése;

a fentieket rendszerint több programozási nyelvre, többféle programtípus készítésére tudja alkalmazni.

Néhány további ismertebb fejlesztőkörnyezet: Delphi, Eclipse, IDLE, Lazarus, XCode és egyre inkább a notepad++. A jegyzetben a Code::Blocks 20.03-as verziójának képernyőképei láthatók és a kódszínezésnek is ez az IDE az alapja.

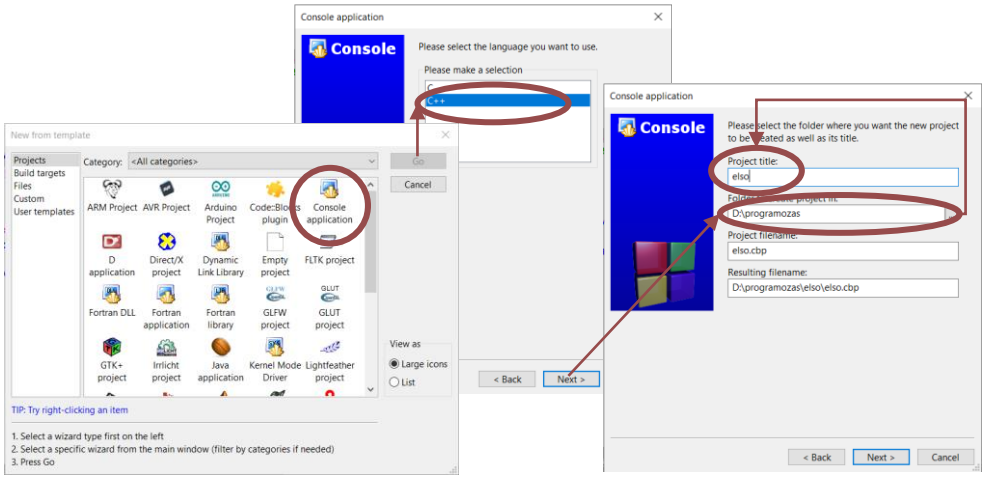
Feladat

1. Keressünk leírást a felsorolt IDE-kről! Nézzünk utána, hogy milyen operációsrendszerre lehet telepíteni, milyen hardverfeltételek szükségesek a futtatásukhoz, illetve melyik programozási nyelvekhez adnak fejlesztői támogatást!

Legelső programunk

Az IDE használatára jellemző, hogy elindítás (és bejelentkezés) után pontosítani kell, hogy mit szeretnénk:

- Létrehozni egy új programot (projektet) vagy megnyitni egy régit, esetleg csatlakozni más projekthez. Elsőre nem sok választásunk van, hozzunk létre és adjuk meg a programunk jellemzőit:



- Ki kell választanunk, hogy az IDE eszköztárából melyik eszközkészletet szeretnénk használni, azaz melyik operációs rendszerre milyen típusú programot szeretnénk készíteni. Jellemzően **Console Application**-t, konzol alkalmazást fogunk készíteni.
- Ezután kell választanunk a két lehetséges nyelv közül. (A C a C++ őse, sok közös részük van.) Válasszuk a C++ nyelvet!
- Végül meg kell adnunk, hogy melyik háttértáron hol legyen a program mappa és mi legyen a programunk neve. Figyeljünk arra, hogy **se a mappa útvonalában se a projekt nevében ne legyen se szóköz se ékezetes betű**, mert ezt a Code::Blocks sokszor nem érti meg. A megadott név lesz többek között a teljes programot, kódot és segédfájlokat tartalmazó mappa neve és – ha megírtuk a kódot és lefordítottuk, akkor – a programfájlunk neve is.
- A sok előkészület után az IDE létrehozza a megfelelő fájl struktúrát és egyben egy minimális programot is.

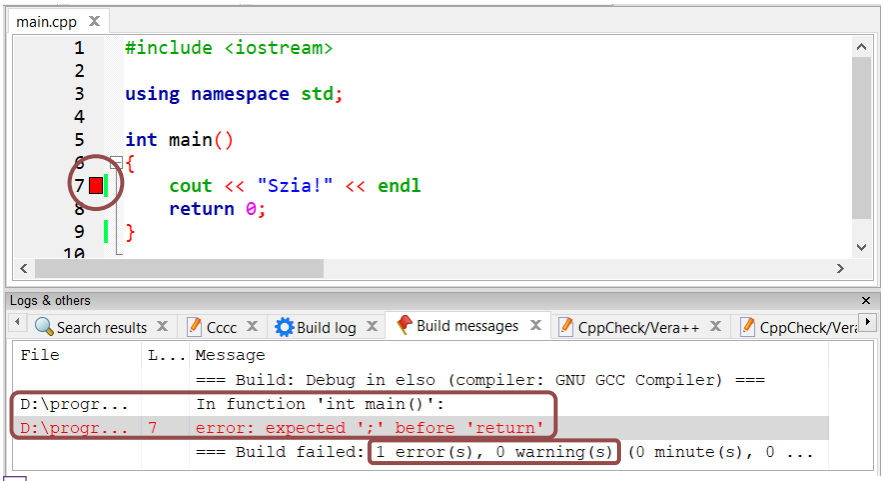
Igaz, hogy az IDE telepítése sok helyet igényel és a használatát is meg kell tanulni (egy kicsit), de cserébe a legelső programunkhoz nem kell semennyit kódolnunk, mert azt előállítja az IDE. Így programunkat rögtön futtathatjuk. 😊

Fejlett IDE a menüsorban és gombon is felkínálja a program fordítását ⚙ és futtatását ▶. Ezt a két lépést lehet külön is indítani, de általában megtalálható a fordítás utáni azonnali futtatás, a „**Build&Run**” lehetősége is: ⚙▶

Pusztán azért, hogy elmondhassuk: programoztunk, módosítsuk az IDE által létrehozott kódot úgy, hogy a „Szia.” szöveget írja ki! Teendőnk: a „Hello World” szöveget kell módosítani.

A fordítás és futtatás egy kattintással ( ) vagy az **F9** billentyűvel indítható. Kis időbe telik, míg az IDE mindent ellenőriz, de utána egy villanás alatt lefut a programunk.

… vagy mégsem, ha valamit elrontottunk.



A fordítás során észlelt hibákat az IDE többféle módon jelzi.

A fordítóprogram csak a **szintaktikának** (nyelvtani és formai szabályoknak) megfelelő helyes kódot tudja binárisra fordítani. Ha elírunk valamit, nem fogja módosítani, javítani. Szerencsére, mert így minimalizálható a félreértés esélye. Az IDE azonban több módon is segíti a hiba helyének és okának a meghatározását.

Hamar észrevehetjük, hogy bár a hibák jelzésének megvan az oka, gyakran nem tudja pontosan megmondani az IDE sem, hogy mit rontottunk el. Ezért mindig – már a kód írása során – figyelni kell a jelzésekre és minden hibás kódsort rögtön a beírást követően ellenőrizni, szükség esetén javítani kell. Kevés módosítás során fellépő hibát a gép pontosabban tud beazonosítani és az ember is könnyebben jön rá a hiba okára.

Ugyanebből a megfontolásból, a programot nem szabad karaktersorozatként írni. Mindig teljes szerkezeti egységeket kódoljunk! A kapcsos zárójelek és a tabulálás is jelzi a kód-blokkok egymáshoz való viszonyát. Mindig írjuk meg a tejes blokkot (nyitó és záró jeleket) és ezután egészítsük ki a jelek között a kódot!

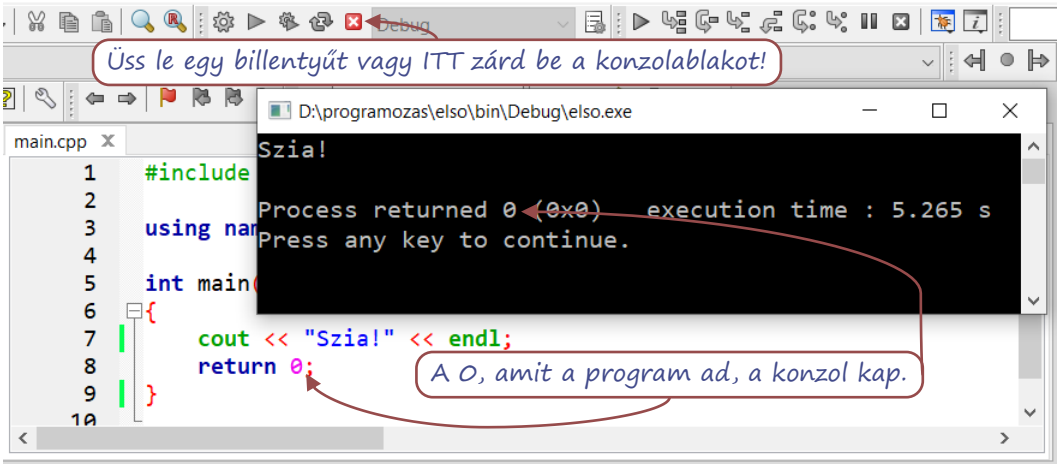
```
D:\progr...      In function 'int main()':
D:\progr... 7    error: expected ';' before 'return'
```

A leggyakoribb hiba a pontosvessző hiánya, amit a „**; expected**” leírás jelez.

A helyes kódolás a helyes gondolkodási móddal is összekapcsolódik. Ezért hibás – és gyakran sikertelen is – a programkód soronkénti másolása, megjegyzése. Megtanulni egy kódot – az egyes részek szerepe és egymáshoz való viszonyának megértése nélkül – szinte lehetetlen.

Ottmarad az ablak

Programunk futtatása konzolablakban történik, ezért a Code::Blocks megnyit egy konzolt. Miután lefutott a program az ablak megnyitva marad, billentyűleütést kér.



Mielőtt bármit tennénk, ellenőrizzük, hogy a programunk helyesen futott-e le, azaz elérte-e a kód végét. Ezt onnan láthatjuk – és az operációsrendszer is onnan tudja –, hogy 0 a programunk visszatérési értéke. A futtatás után ennyi információ marad a program után, ez is csak azért, mert a kódba beírtuk, hogy 0-val térjen vissza.
A konzolablak egy billentyűleütésre bezárul, de a Code::Blocks eszköztárán is bezárhatjuk. Harmadik megoldás nincs. Ha ugyanis az ablak Bezárás ikonjára kattintunk, akkor jó esély van arra, hogy a Code::Blocks ezt nem veszi észre. Így – bár az ablak bezárul, a Code::Blocks „lefagy". Csak a Feladatkezelőben lehet segíteni rajta, ott be tudjuk zárni a programot. Még szerencse, hogy a fordításkor mindig mentjük a kódfájlt, így a munkánk nem vész el, csak az időnk.

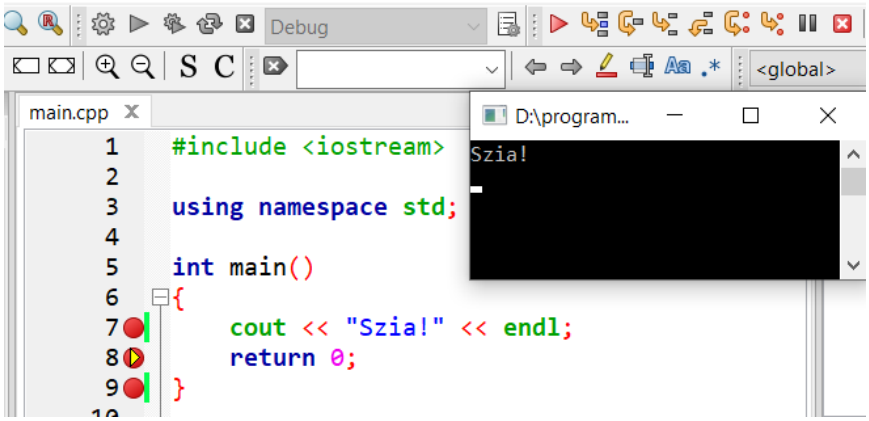
Lépésenkénti futtatás

A futtatás befejezése gomb mellett láthatjuk, hogy **Debug** módban futtatjuk a programot. Ez az indítás előtt átállítható Release módra, ami a véglegesített, optimalizált kódú verzió. Lényegében ilyen szoftvert nem fogunk készíteni, de a Debug módban futtatás speciális lehetőségeit érdemes kitapasztalni.

A *debug* hibakeresést jelent. Arra is van mód, hogy futtatás közben lépésenként megfigyeljük, mi történik a programban, esetleg módosítsunk a kódon.

A programunkba egyes IDE-k esetén kissé eltérő helyen, de mindig a kód sorszámozása környékén lehet kattintással elhelyezni a piros pöttyöt, a **breakpoint**ot. Ha a futtatást nem a ▶ ikonnal (vagy F9) indítjuk, hanem a Debug mód kiválasztó másik oldalán található ▶ ikonnal (vagy **F8** billentyű), akkor a fordítás részeredményét azonnal végrehajtja a Code::Blocks és a breakpointnál megáll.

Minden aktív kódsor elé lehet breakpointot tenni. Ezeket ismételt rákattintással törölni is lehet, jobb gombbal rákattintva még feltételhez is köthetjük a megállítást. Ráadásul futtatás közben is módosíthatjuk a megállítás helyeit.



A program futása minden breakpoint elérésekor, azaz a jelzett utasítás végrehajtása előtt megszakad. A helyet, ahol a végrehajtás éppen tart, sárga nyíl jelzi. A program folytatását a -re kattintva lehet kérni, de a mellette látható ikonok is használhatók továbbhaladáshoz: a soron belül egyes részletekre ugrás vagy következő sorra ugrást lehet ezekkel elérni. A sor végén látható piros négyzettel a futtatás megszakítható.

A hibakeresés (debugolás) talán legnagyobb segítsége, hogy vizsgálhatjuk a program pillanatnyi állapotát. A Watch ablak (jobb oldalon) hasznos segítőnk lesz, mert a megállított program pillanatnyi adatait, állapotát itt láthatjuk majd.

Az elkészült szoftver futtatása

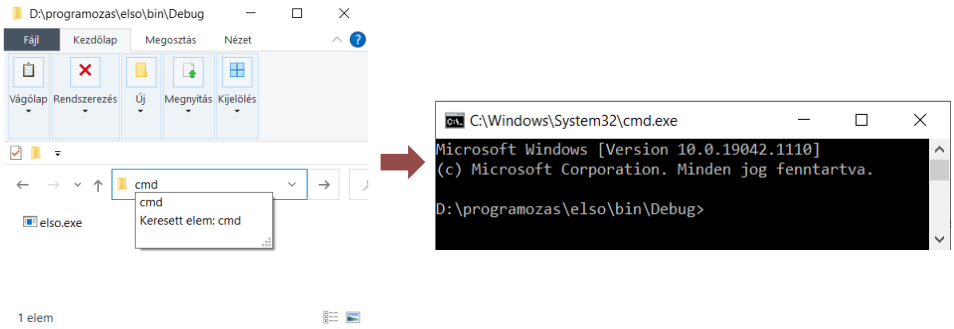
A C++ *compiler* nyelv, ezért a fordítás során elkészíti a futtatható állományt. Ezt az operációs rendszerből indítva is futtathatjuk.

Nyissunk meg egy konzolablakot és lépjünk be a programunk mappájába, azon belül a bin/Debug (vagy, ha Release módban fordítottuk, akkor bin/Release) mappába!

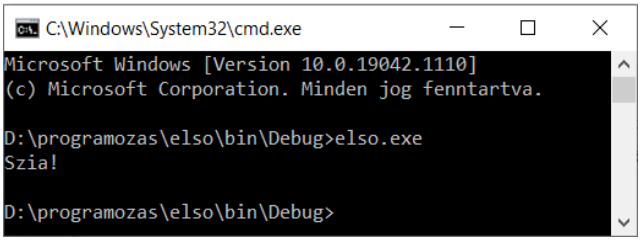
- A konzolablak Windowson a Parancssor indításával vagy a `cmd` parancs futtatásával indítható.
- A program mappájának kiválasztásához először a megfelelő meghajtóra váltsunk át, majd a `cd` parancs után írjuk be az elérési útvonalat.
- Ha a Debug (Release) mappában nem találjuk az `.exe` fájlt, akkor a fejlesztői verziónak megfelelő almappába tovább kell lépni.

Ez a megoldás eléggé nehézkes, ezért álljon itt egy Windows fájlkezelő trükk:

- Keressük meg fájlkezelőben az `.exe` fájlt! Rákattintva elindul a programunk, azonban semmi sem fogja megállítani ezért nem látható a működése.
- 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.



A konzolablakban beírva a program nevét (`elso.exe`) a programunk lefut. Mivel a konzolablakon belül indítjuk a futtatást, a program befejezése után is nyitva marad a konzol, a kiírt szöveg is látható marad.



Ha többször szeretnénk a programunkat konzolablakban futtatni, akkor nem érdemes bezárni az ablakot. A program nevét újra beírva, vagy felfelé kurzormozgató nyíl nyomkodásával visszaírva, a program (esetleg idő közben frissített verziója) újra futtatható. Ha „törölni” szeretnénk a konzolablak tartalmát, ezt a `cls` utasítással tehetjük meg.

A felhasználó mondja meg, mikor legyen vége

Amikor a kész programot futtatjuk – mivel konzolprogramot írunk – egy konzolablakban, akkor az összes kiírás látható marad. Hasonló módon lehetne az eredményt fájlban tárolni vagy átküldeni egy másik programnak helyben vagy interneten keresztül. De programozás közben sem így, konzolban futtatva, sem breakponinttal megállítva nem kényelmes ellenőrizni az eredményt. Ezért jobb lehet, ha kóddal meg tudnánk állítani a program futását egy, a felhasználótól várt billentyűleütésig. Ez felel meg sokszor látott „Press any key to continue…” üzenetnek, amit most nem az IDE ír ki, hanem mi.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.    cout << "Szia!" << endl;
7.    system("Pause");
8.    return 0;
9.  }
```

Mit csináltunk?

Létrehoztunk egy projektet, eközben keletkezett egy mappa a háttértáron. Ebben a mappában 2 + 1 fontos fájl van.

1. A `cbp` kiterjesztésű fájlnak ugyanaz a neve, mint a mappának. Ezzel lehet legközelebb megnyitni a fejlesztőkörnyezetben a projektünket.
2. A `main.cpp` fájlban van a kódunk, amit akár egy szövegszerkesztőben is megnyithatunk. De ez a lényeg! Általában ezt csatoljuk e-mailhez, ennek másolatát(!) adjuk be ellenőrzésre.
3. A projekt nevével azonos nevű `exe` fájl a lefordított programkód. Ez az, ami futtatható, ez maga a program, a szoftver, a termék. Nagyon örülünk neki, ha jól működik, de nem baj, ha töröljük, mert a kódból újra (és újra) elő lehet állítani. Mivel ez egy futtatható állomány, a vírusvédelem érzékenyen reagál a másolására, küldésére, más környezetben futtatására, távoli használatra. Jellemző, hogy e-mailben nem tudjuk elküldeni, ha mégis, akkor a „termékünk” karanténba kerül vagy tisztítás áldozatául esik a címzettnél. Ha tömörítéssel próbáljuk elrejteni, akkor a teljes tömörített mappára vár ugyanez a sors.

… és a többi fájl, ami a fejlesztőkörnyezet, az elemző szoftver és a fordító program működése során keletkezik. Folyamatos munka során nem érdemes ezeket a fájlokat törölni és semmiképp sem érdemes belenyúlni, módosítani.

Programozás IDE nélkül (csak erős idegzetűeknek)

A programozás lényege, hogy beírjuk az utasításokat egy szövegfájlba (hibamentesen), ezt a fájlt átadjuk a fordítóprogramnak, ami az utasításokat gépi kódra fordítja, amit ezután futtathatunk. Ehhez mindössze három dologra van szükség:

- egy parancssoros ablakra,
- egy szöveg készítésre alkalmas programra és
- a fordítóprogramra.

Windows operációs rendszeren a parancssoron belül a `copy` (másolás) parancs alkalmas arra, hogy a beírt szöveget fájlba mentsük. A `copy con elso.cpp` utasítás a konzolról, azaz a billentyűzetről az `elso.cpp` fájlba másolja a tartalmat. A parancs kiadása után ENTER-t ütve kezdhetjük a kód gépelését. A **CTRL+Z** billentyűkombinációval zárjuk le az adatsort (a „fájlt”).

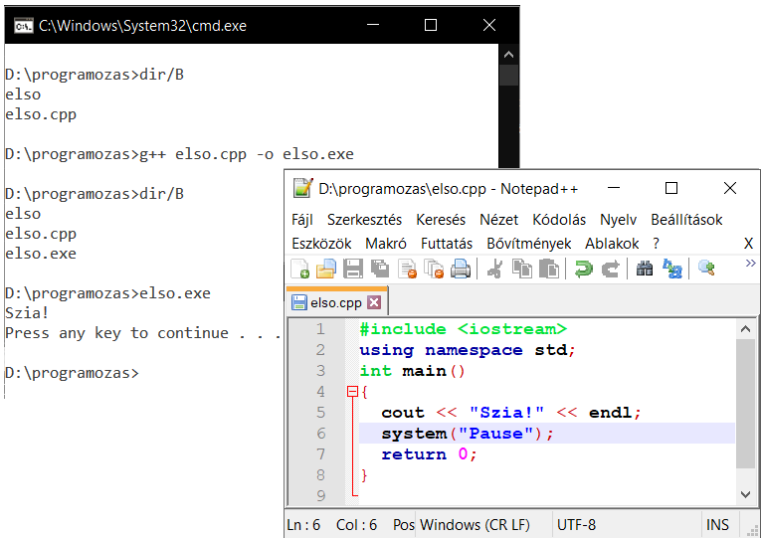
Ezt követően a `type elso.cpp` parancs segítségével ki is írhatjuk a fájl tartalmát, de módosítani csak úgy lehet, ha elölről kezdjük. Ezért szoktak inkább egy szövegszerkesztő alkalmazást használni a kód megírására.

```
Parancssor

D:\programozas>copy con elso.cpp
#include <iostream>
using namespace std;
int main()
{
  cout << "Szia!" << endl;
  return 0;
}
^Z
        1 file(s) copied.

D:\programozas>type elso.cpp
#include <iostream>
using namespace std;
int main()
{
  cout << "Szia!" << endl;
  return 0;
}

D:\programozas>g++ elso.cpp -o elso.exe

D:\programozas>elso.exe
Szia!

D:\programozas>
```

Kódolás, fordítás és futtatás Parancssor ablakban.

A fordításhoz a `g++.exe`-t kell futtatni, ennek lesz az első paramétere a kódfájl, a `-o` (kis o-betű) jelzi, hogy utána az eredmény fájl neve következik; a harmadik paramétere az `exe` fájl neve. A fordító elkészíti a futtatható állományt, amiből semmit sem látunk, majd a futtatás eredménye, hogy a képernyőre (kimeneti konzol) kiíródik a „Szia!” szöveg.

Egyszerű – vagy kicsit többet tudó – szövegszerkesztővel elkészített fájl az előzővel azonos módon fordítható és futtatható konzolablakban.



Kódolás szövegszerkesztőben, fordítás és futtatás Parancssor ablakban.

Talán érdemes egyszer kipróbálni így is a programozást, de csak azért, hogy lássuk, mennyivel jobb egy IDE. Mostantól egy IDE (a Code::Blocks) segítségével fogjuk elkészíteni a kódot, ez fogja meghívni a fordítót, hogy elkészítse a futtatható állományt és általában futtatást is az IDE-ből indítjuk.

VÁLTOZÓK, KIÍRATÁS, ADAT BEKÉRÉSE

Módosítsuk a programunkat úgy, hogy „Szia” helyett írja ki születésünk évét!

Szöveg, karakter és szám – adattípusok

A születésünk éve egy szám. A programozási nyelvekben az a szokás, hogy a szöveges adatokat idézőjelek közé kell írni, ha egyetlen karaktert adunk meg, azt aposztrófok közé írjuk, a számoknak nincs kiegészítő jelölése. A születésünk évéről mi tudjuk, hogy szám, de megadhatjuk szövegként is, sőt – tagolva – karakterenként is esetleg számjegyenként szövegként. Ha több adatot szeretnénk egymásután kiírni, akkor ezeket a `<<` jel ismétlésével megtehetjük.

A `<<` neve *inserter*, az utasítás elejére írt dologba – megfelelően átalakítva – „betáplálja” az adatot. A `cout` – rendes nevén *console output* – a kapu a „konzol” felé azaz a képernyőn megjelenő karakteres ablak felé.

Érdemes már most megjegyezni, hogy a szöveg karakterek sorozatát jelenti, de lehetséges olyan szöveg, amiben nincs egyetlen karakter sem. Ezt egymásmelletti idézőjelekkel adjuk meg: `""`.

Az aposztrófok között mindig pontosan egy karakter lehet, egymás mellé írva nincs értelmük. Például: `'s' 'z'` egy-egy karakter. Speciális esetben jelezhet több karakter egyetlen karaktert. Ilyen például az ENTER `'\n'` karaktere, amit másképp nem tudnánk beírni. Hasonlóan lehet

megadni a TAB billentyűleütésnek megfelelő `'\t'` karakterét és más vezérlő karaktereket, illetve a billentyűzeten nem elérhető betűket. Minden ilyen, speciális karakter a `'\'` backslash jellel kezdődik, ez a C#-ban (és sok más nyelvben is) az „**escape character**”.

Kiírásokban nagyon gyakori, hogy a végére kell egy Enter. A Szia! szöveget lezárhatjuk úgy is az Enter karakterével, hogy beírjuk az idézőjelek közé: `"Szia!\n"`, vagy a szöveg után írással: `"Szia!" << '\n'`, vagy a készen kapott kódban használt `endl` utasítással. Ez utóbbi az „endline”, a `'\n'` helyettesítője, amit azért szeretünk használni, mert se aposztróf se backslash nem kell a beírásához.

A számokat is írhatjuk idézőjelbe, de ebben az esetben a C++ karaktersorozatként (szövegként) kezeli az adatot Mit jelent ez? A legegyszerűbb, ha kipróbáljuk! A `+` jel a számokat összeadja, a `string` típusú szövegeket összefűzi. Nézzük meg, mi lesz az eredménye, ha kiírunk egymásután egy hármas és egy ötöst, azaz a `3`-hoz, a `'3'`-hoz, illetve a `"3"`-hoz hozzáírjuk az `5`, `'5'` és `"5"` adatokat, valamint mi lesz az eredmény, ha ugyanezeket a kiírás előtt a + jellel adjuk/fűzzük össze!

Az alábbi kódot beírva és futtatva látható néhány szabály és néhány talány.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main()
5.  {
6.  cout << 3 << 5 << endl;
7.    cout << '3' << '5' << endl;
8.  cout << "3" << "5" << endl;
9.    cout << 3 + 5 << endl;
10. cout << '3' + '5' << endl;
11.   cout << (string)"3" + "5" << endl;
12. return 0;
13. }
```

```
35
35
35
8
104
35
```

Az eredmény a 6-8. sorok esetén az elvárásunknak megfelelő, a 3-as és 5-ös adatot a `<<` megérti és egymásután írja a kimenetre, majd az 5-ös után „küldött” `endl` hatására új sor kezdődik.

A `+` jel a számok között elvégzi az összeadást (9. sor) és az eredményt a `<<` kiírja. Ha a 3-as és 5-ös karaktert adjuk össze, akkor egy harmadik eredményt kapunk. Ez a két karakter ASCII kódjának az összege, 51 `+` 53. Nagyon sok programozási nyelvben tapasztalhatjuk, hogy a karakter néha a karakterképét mutatja, máskor a bináris szám kódjának értékét. A 7. sorban a `<<` a karakter megfelelőt adja ki, a 10. sorban a `+` a bináris értékként összeadja és a kapott számot küldi tovább a `<<` a konzolra.

Szövegként a `"3"` és az `"5"` egyelemű karaktersorozat. Hiába csak 1 számjegy van benne, a `+` számára így olyan, mintha két számsorozatot kellene összeadnia, ami nem megy. Az összefűzéshez legalább az egyik tagot át kell alakítani C++ típusú `string` szöveggé. Ezt látjuk a 11. sorban: a `(string)` az utána levő `"3"`-at átalakítja, azt követően az `"5"`-öt már ehhez igazítja a fordítóprogram. Ha a példánkban "BK" `+` `5` szerepelne, akkor a BK átalakítása nem lenne elég, mert a `+` jel számot nem tud hozzáfűzni, szöveghez viszont nem tud hozzáadni.

Kísérletünkből az is kiderül, hogy egy számjegyekből álló adat a kódolástól függően lesz szöveg vagy szám, illetve, ha egy számjegyű akkor lehet karakter is. A műveleti jelek hatása és értelmezése függ attól, hogy milyen típusú adattal dolgozunk.

Változók

Íjunk új programot! A kódot mindig az `int main()` kapcsoszárójelei `{}` közé írjuk, ezért a lényeg körülbelül a 7. sortól lesz

```
7.  string allat = "ló";
8.  cout << "allat" << endl;
9.  cout << allat << endl;
```

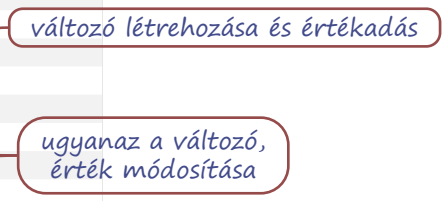
Az első sor új elemet tartalmaz. Az `allat` itt egy *változó*, aminek azt adtuk értékül, hogy „ló”. Legegyszerűbb, ha a változókra olyan dobozként gondolunk, amibe bele tehetünk valamit. Itt egy szöveget tehetünk bele, ezt jelzi a változó neve előtt a `string` szó. A fordítóprogram számára a változó a memóriában lefoglalt területet nevesíti. A változó típusa alapján a fordító program foglalja le a megfelelő méretű memóriaterületet, figyel arra, hogy oda más ne kerüljön, csak az, amit a változóba beleteszünk. Amikor a változó (memóriabéli doboz) nevét írjuk le (idézőjelek nélkül), akkor a doboz *tartalmát* helyettesíti be. Ha a változó nevét idézőjelek közé írjuk, akkor a fordítóprogram szövegként tekint rá, amit meg sem próbál értelmezni.

A programot futtatva láthatjuk, hogy az ékezetes karaktereket nem tudja jól kiírni a konzol. A programunkban be kell állítani, hogy magyar karaktereket is tudjon használni. Erre használjuk a `setlocale()` eljárást. Az ékezetes karakterek használata mellett a tizedesvessző, pénznem és dátumidő formátum is beállítható ezzel, ezért az `LC_ALL` paramétert használjuk, valamint a gépünk beállításait választjuk a `""` beírásával.

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

A változókat azért hívjuk változóknak, mert az értékük változtatható. Ha új értéket adunk nekik, a régi egyszer s mindenkorra nyomtalanul eltűnik. Gondoljuk végig az alábbi program kimenetét, aztán futtassuk a programot, hogy kiderüljön, jól tippeltünk-e.

```
5.  int main()
6.  {
7.    setlocale(LC_ALL, "");
8.    string allat = "ló";
9.    cout << allat << endl;
10.   allat = "nandu";
11.   cout << allat << endl;
12.   allat = "cickány";
13.   cout << allat<< endl;
14.   return 0;
15. }
```



Szabály, hogy a C++változónevei

- betűvel vagy alávonással (_) kezdődhetnek;
- betűvel, számmal vagy alávonással folytatódhatnak (írásjel és szóköz nem lehet bennük), azaz `anyu kora` helyett használjuk az `anyuKora` alakot, vagy aláhúzással írjuk egybe a szavakat: `anyu_kora` (ez szokás C++-ban);
- a kis- és a nagybetű használatára figyelnek, azaz `Majom`, `majom` és `majoM` három külön változó (a C++ `case` sensitive);
- nem egyezhetnek meg az úgynevezett „kulcsszavakkal” – ilyen például az adattípusok megnevezése, a vezérlő szerkezetek megnevezései (`int`, `return`, `for`, `if`, `while`) és a foglalt szavakkal. Például az `std` névtérben foglalt szó a `string`, `cout`, `endl`.

Nem szabály, de érdemes akként tekinteni rá: a programnak mindegy, hogy miként nevezzük el a változókat. Ha a fenti programban az „allat” helyett *mindenhol* „noveny” szerepelne, a program hibátlanul működne. A *programozónak* fontos a jó változónév, hogy ha holnapután előveszi a programját, még mindig el tudjon igazodni rajta. A változónevek választásával a program értelmezését segítjük.

A példán jól látszik, hogy a változó típusát egyszer, a legelső használatkor adjuk meg, pontosabban akkor, amikor a változót létrehozzuk, helyet kérünk számára a memóriában. Ezt később – amíg az adott változó számára a hely le van foglalva, azaz, amíg „él” a változó, – nem tudjuk megváltoztatni és nem lehet újra létrehozni ugyanolyan néven másik változót. A változó tartalmának módosításakor már csak a változó nevét kell beírni, a típust nem.

Arra is lehetőség, hogy a változót első lépésben csak megnevezzük – úgy mondják, hogy deklaráljuk. Programnyelvtől és adattípustól függően ezzel együtt a deklarált adatnak lesz lefoglalt helye a memóriában és abban valami. Lehet, hogy csak az, ami az előző használatkor ottmaradt, de egyes programozási nyelvekben – mint a C++ is – egyes adattípusoknál a memóriafoglaláshoz kezdőérték beállítása is társulhat. Amikor nincs kezdeti értékbeállítás, akkor az adatot konstruktorral tudjuk létrehozni, amivel kezdőértéket is adunk, azaz inicializáljuk.

Így is lehet:

```
 7.  setlocale(LC_ALL, "");
 8.  string allat;
 9.  allat = "ló";
10.  cout << allat << endl;
```

típus és név deklarálása, kezdőérték: ""

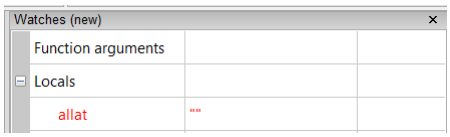
érték módosítása

Így, konstruktorral is lehet:

```
 7.  setlocale(LC_ALL, "");
 8.  string allat{"ló"};
 9.  cout << allat << endl;
```

létrehozás (konstruálás): típus, név, érték

Próbáljunk ki többféle módon adatot deklarálni és kezdőértéket adni neki! A programot állítsuk meg breakpointtal az elején és soronként léptessük. Figyeljük meg, hogyan változik a Watches ablak (jobb oldalon) tartalma!



Adat bekérése a felhasználótól

A legtöbb program kér adatokat a felhasználótól (de legalábbis a környezetéből: egy mérőműszertől, fájlból, hálózatról). A telefonunkba be kell írni az új telefonszámot, vagy egy listából kiválasztani a már rögzítettet. A böngészőnkbe beírjuk, hogy melyik webhelyet nyissa meg. A gépünknek megadjuk a jelszavunkat.

C++-ban a felhasználótól leggyakrabban a `>>` *extractor* fogadja a `cin`-en (*console input*) keresztül a konzolablakban beírt adatot. A `>>` az érkező adatot (karaktersorozat) a nyíl hegyénél megadott változónak megfelelő típusra átalakítja és beleteszi a változóba.

A `>>` nagyon okos, a képességei közül néhány:

- Ha a konzolablakban nem írunk be semmit, akkor vár, egészen addig, amíg a beírást követően leütjük az ENTERt.
- Ha a beírt adatsor szóközt, tabulátort tartalmaz, akkor ott megszakítja a betöltést, a maradékot elraktározza a következő adatbekérésre.

- Egymás után – a `<<`-hez hasonlóan – több adatot betölthetünk a különböző változókba.

Ez a működés akkor nagyon kényelmes, ha egy sorban több adatot szeretnénk megadni. A felhasználó beírhatja előre az adatokat, amit a `>>` a megfelelő időben a megfelelő helyre vesz át. Ugyanakkor egy mondat beírása egyetlen változóba nagyon bonyolult művelet. Ezért van egy másik beolvasási mód, ami többszavas `string` beolvasására jó. Ez a `getline(cin,…)`.

A felhasználó általában nem tudja, hogy a program éppen vár valamilyen adatra, vagy dolgozik valamin. Ezért, ha embertől, humán felhasználótól vár a program adatot, akkor illik a programban egy kiírással tudatni a felhasználóval, hogy az ő aktivitására van igény. Ezért az adatbekérést általában megelőzi egy kiírás.

```
 5.  int main()
 6.  {
 7.    setlocale(LC_ALL, "");
 8.    cout << "Hogy hívnak? ";
 9.    string nev1, nev2;
10.    cin >> nev1 >> nev2;
11.    cout << nev2 << " " << nev1 << endl;
12.    return 0;
13.  }
```

- Mi történik, ha csak egy nevet írunk be és mi lesz akkor, ha három nevet adunk meg?
- Figyeljük meg, hogy a kérdés után nincs `endl`, mert ugyanabba a sorba várjuk a választ. Amikor a választ beírjuk, akkor az ENTER új sorba viszi a kurzort, ezért a két név fordított sorrendű kiírása a következő sorba kerül.

Próbáljuk ki, hogy mi történik, ha több részletben kéri a program a nevünket, de mi egyben adjuk meg:

```
 5.  int main()
 6.  {
 7.    setlocale(LC_ALL, "");
 8.    cout << "Hogy hívnak? ";
 9.    string nev1, nev2;
10.    cin >> nev1;
11.    cout << nev1 << "\nés még? ";
12.    cin >> nev2;
13.    cout << nev2 << " " << nev1 << endl;
14.    return 0;
15.  }
```

A többszavas szöveg beolvasását is teszteljük:

```
14.  int main()
15.  {
16.    setlocale(LC_ALL, "");
17.    cout << "Hogy hívnak? ";
18.    string nev;
19.    getline(cin, nev);
20.    cout << nev << endl;
21.    return 0;
22.  }
```

Kérdések, feladatok

1. Mit írnak ki az alábbi programok? Gondoljuk végig, aztán próbáljuk ki a programokat, nézzük meg, hogy igazunk volt-e!

```
string allat = "kutya";
string kutya = "Bogi";
allat = "macska";
cout << allat << endl;
```

```
1.  string gyumolcs = "alma";
2.  gyumolcs = "szilva";
3.  string alma = "dinnye";
4.  int dinnye = 3;
5.  gyumolcs = alma;
6.  cout << gyumolcs << endl;
```

2. Rajzoljuk ki a jobb oldalon látható ábrát karakterek használatával! Figyeljünk arra, hogy a backslash („\”) különleges szerepű, az utána lévő karakter vezérlőkarakter, aminek a C# speciális értelmet adna, nem írná ki. Egy backslash kiírásához, kettőt kell írnunk – a `'\\'` egy darab ’\’ karakterként jelenik meg a konzolon.*
3. Rajzoljuk ki az alábbi ábrákat karakterek használatával! Próbáljuk meg egyetlen kiíró utasítással megadni a teljes ábrát! Ehhez használhatjuk a `\n` és `\t` vezérlőkaraktereket is.



†


4. Kérdezzük meg a felhasználótól (a program használójától) a vezetéknevét! Kérdezzük meg a keresztnevét is, és köszönjünk neki, a teljes nevén szólítva!

Számok és karakterláncok a programunkban

Eddig még csak karakterláncot (szöveget) tároltunk a változóinkban. Ebben a leckében változtatunk ezen, és számokat is használunk.

Hány éves a felhasználó?

Olyan programot fogunk írni, amely választ ad a fenti kérdésre.

1. Az első részfeladat egy olyan program megírása, amely megkérdi, hogy mikor születtünk, és ezt ki is írja nekünk. Ez eddigi tudásunk alapján megoldható, úgyhogy készítsük el önállóan az adat bekérését!

A változóink eddig **`string`** típusú adatok voltak, a születésünk évét is el tudjuk tárolni **`string`** típusú változóban és ki is tudjuk írni, de most ez nem lesz jó, mert már láttuk, hogy a **`string`**-hez nem tudunk hozzáadni számot, így a kivonás során sem lesz használható. Az egész számok

* Digitális kultúra 9. tankönyv 101. oldalán látható kép
† Digitális kultúra 9. tankönyv 101. oldalán látható képek

tárolásához egy másik adattípusra van szükség, aminek általános neve **integer**. Matematikából azt tanuljuk, hogy az egész számok halmaza végtelen, de a processzor nem tud végtelen nagy számmal dolgozni. Ezért a program írásakor fel kell mérnünk, hogy mekkora számokat kell a változónkban eltárolni és ehhez kell igazítani a választott típust. C++-ban 1, 2, 4 és 8 bájton tárolhatunk egész számokat, minden méretben van olyan adattípus, ami csak nemnegatív értékeket tárol (például az `unsigned char` 0 – 255) és olyan is, amelyiknél a tárolható értékek fele negatív (például `signed char` −128 – 127).

A tízféle integer típus közül legtöbbször a 4 bájtos, előjeles `int` típust használjuk. Ennek korlátjai -2^{31} és $2^{31}-1$, ami nagyjából a $\pm$ 2 milliárd közötti egészeknek felel meg. Ha a negatív értékeket nem engedhetjük meg, akkor az `unsigned int` típust használjuk. Ezt rövidebben is megadhatjuk: `unsigned`. Értéke 0 és körülbelül 4 milliárd közötti lehet. Nem kell használnunk, de nem nehéz elképzelni, hogy a 2 bájtosok nevében a `short`, a 8 bájtosok nevében a `long` jelzi a méretbeli különbséget. Történelmi okai vannak annak, hogy az egybájtos egész típus neve `char`: a ős C nyelvben a karaktereket 1 bájtos számként tárolták, így nem volt értelme ugyanarra két nevet alkalmazni.

A bekért születési év egész számként tárolásához a fentiek alapján létrehozunk egy `int` típusú adatot, de meg kell oldanunk még egy problémát: A beírt karaktersorozatból valahogy `int`-té kell alakítani az adatot. Erre több mód is van, de elsőre használjuk azt, ami kéznél van: a `>>` elvégzi az átalakítást, ha `int` típusú változó várja az adatot.

```
1.  int n;
2.  cin >> n;
```

2. A következő részfeladat a felhasználó korából a születés évének a meghatározása.

Ehhez a programunknak tudnia kell, hogy melyik évben futtatják. A jelenlegi év megkérdezhető az operációs rendszertől, de egyelőre megelégszünk azzal, hogy egy változóba beírjuk.

Azokat az értékeket, amelyek a program léte során nem változnak, **konstans**nak nevezzük, és a C++ nyelvben a változó létrehozáskor jelöljük is a típus elé írt `const` jelzővel. Egyes nyelveknél lehetőség van arra, hogy a konstans értékét a futtatás során első alkalommal a felhasználó adja meg, de a >> létező változóba írja az adatot, így ez már módosításnak számít. A konstans értékét a kódban kell rögzíteni, ezt később nem lehet módosítani. A konstans változókat érdemes a program elején megadni, és sokan csupa nagybetűvel írják a változónevet, hogy később se felejtsék el, hogy fordítási hibát okoz, ha valami meg akarná változtatni az értékét.

Az eddigiek alapján a programunkban az alábbiaknak kell szerepelnie:

- értékadás egy konstansnak;
- a felhasználó korára vonatkozó kérdés kiírása;
- a beírt adat egésszé alakítása és tárolása;
- születés évének kiszámítása;
- eredmény kiírása.

Módosítsuk és egészítsük ki a kódot a fentieknek megfelelően:

```
1.  #include <iostream>
2.  using namespace std;
3.  const int IDEIEV = 2021;
4.  int main()
5.  {
6.    setlocale(LC_ALL, "");
7.    cout << "Mikor születtél? ";
8.    int szEv;
9.    cin >> szEv;
10.   int felhasznaloKora = IDEIEV - szEv;
11.   cout << felhasznaloKora << " éves vagy." << endl;
12.   return 0;
13. }
```

Ha megfigyeljük a mondatainkat, látjuk, hogy kódoláskor előbb foglalkozunk a lejegyzett tennivalók jobb oldalán (végén) lévő dolgokkal és csak utána a bal oldallal. Ezt a kód begépelésekor nyugodtan kövessük. A 3. sort például lehet úgy írni, ahogy gondoljuk: IDEIEV = 2021; legyen egész és végül legyen konstans is. Igaz, hogy így a kurzor oda-vissza cikázik, vagy az egérrel sokszor kell az eddig megírtak elé kattintani, de mégis hatékonyabban tudunk dolgozni, mert a gondolatainkat nem kell fejben előre megformáljuk, hanem a lejegyzés során áll össze. Idővel, sok gyakorlás után egyes sorok kigondolása automatizmus lesz, akkor már természetes, ha egyvégtében, balról jobbra írjuk a kódot.

Ismét látható, hogy a kód másolása miért nem alkalmas a programozás tanulására: az értelmes másoláshoz tudni kellene, hogy hol kezdődik a gondolat, ehhez ki kellene találni a szerző gondolatát. Más fejével gondolkodni még a saját megoldás kitalálásánál is nehezebb.

Nem egész számok

3. Most, hogy már tudjuk a felhasználó korát. Adjuk meg, hogy mennyibe kerülne a gyertya a születésnapjára! Egy gyertya ára Euroban 0,18–0,35 €, a pontos árat kérjük be!

Természetesen tudunk tizedestörtekkel is dolgozni programjainkban. A tizedesjel lehet pont vagy vessző, hogy éppen melyik, az az operációsrendszertől, a fejlesztőkörnyezettől és a programozási nyelvtől is függ. Ha a kódban szerepel a `setlocale()`, akkor az operációsrendszer beállítása miatt tizedesvesszőt fog kiírni, de nekünk tizedespontot kell beírnunk és a programkódba is a programozási nyelv szabványa szerinti tizedespontot lesz a megfelelő.

A tizedestörteket a programozók lebegőpontos számnak is nevezik, ami angolul *floating point number*, innen származik a típus neve: *float*. A lebegőpontos számokat normál alakhoz hasonlóan ($m*2^k$) tárolja a program, ahol a szám pontosságát az határozza meg, hogy a mantissza (m) hány bites. A **`float`** típusnál kétszer pontosabb a **`double`** típus, ezért általában ezt használjuk.

A beolvasáskor ezt is át tudja alakítani az `>>` (extractor), de sokszor a programon belül is át kell alakítani **`int`** és **`double`** (illetve **`float`**) között. Azokban az esetekben, amikor egyértelmű az átalakítás és ebből nem származhat adatvesztés, akkor ez „implicit”, azaz rejtetten végbemegy. Például, egy egész szám bármikor használható **`double`** típusú adatként. Ugyanakkor visszafelé már jelezni kell, hogy át szeretnénk értelmezni az adattípust és néha az egész számról is közölnünk kell, hogy most tizedesjegyes számként szeretnénk használni. Ezt az „explicit”, azaz kijelentett átalakítást úgy adjuk meg, hogy az érték elé zárójelben megmondjuk, hogy

milyen adattípusra akarjuk átalakítani. Például, az (**int**)3.9 értéke 3, egész szám; a (**double**)4 értéke 4.0, dupla pontosságú lebegőpontos szám.

A gyertyák árának meghatározása:

```
12.    cout << "Hány Euro egy gyertya? ";
13.    double egyGyertya;
14.    cin >> egyGyertya;
15.    double gyertyakAra = felhasznaloKora * egyGyertya;
16.    cout << "A gyerták "<< gyertyakAra << " Euróba kerülnek.";
17.    int euro = (int)gyertyakAra;
18.    int cent = (int)((gyertyakAra - euro) * 100);
19.    cout << "A gyertyák ára: " << euro << " Euro " << cent << " cent";
20.    return 0;
21.  }
```

Legyen egyenlő

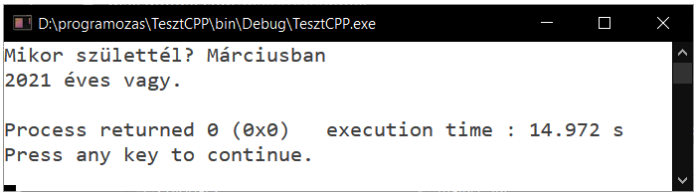
Eddig három műveleti jelünk volt, a „+” a „*” és a „–” jel, de mostanra el kell fogadnunk, hogy programozáskor az egyenlőségjel is műveleti jel. Alapvetően mást jelent ilyenkor az egyenlőségjel, mint matematikaórán. Ott állításokat, kijelentéseket fogalmaztunk meg vele (Például: kettő egyenlő háromból egy; hatszor hat egyenlő harminchattal.) Programozáskor az egyenlőségjel egy művelet elvégzésére való felszólítás: *legyen* a változó értéke…

Programíráskor az egyenlőségjel az értékadás műveletének a jele, olvashatjuk „**legyen egyenlő**” formában. Értékadáskor mindig az egyenlőségjel jobb oldalát értékeli ki a program elsőként, majd a kapott eredményt értékül adja az egyenlőségjel bal oldalán lévő változónak.

Az értékadás egyenlőségjelét egyes programozási nyelvekben emiatt egy kettősponttal kezdik. A jelölés ismertségét mutatja, hogy a – három ponthoz hasonlóan – egyesített formája is létezik a jelölésnek és több szövegszerkesztő írás közben át is alakítja. Külön írva „a := b”, egyben „a ≔ b”. Egyesek szerint még szemléletesebb, ha nyíllal jelöljük az értékadást: a ← b.

Számos problémák

Mi történik, ha a program kér egy számot, de nem azt kap? Ebben az esetben nem a fordításkor dobja fel az IDE a „szokásos” ablakot, hanem a program futása közben kapunk furcsa eredményeket:



Nézzük meg, mi történik a háttérben! Az egyik lehetőség, hogy lépésenként futtatjuk a programot. A másik, hogy beszédessé tesszük, azaz a szEv változása előtt és után kiírjuk konzolra az értékét.

Még sokáig abból indulunk ki programkészítéskor, hogy a felhasználótól érkező bemenetet nem kell ellenőriznünk, *validálnunk*. Egy „igazi” alkalmazás esetében a hibákra való felkészülés (felhasználótól kapott rossz bemenet, elfogyó háttértár, megszakadó hálózati kapcsolat stb.) a programnak igen jelentős része.

```
 8.    int szEv;
 9.    cout << szEv << endl;
10.    cin >> szEv;
11.    cout << szEv << endl;
```

Ha ezzel a kiegészítéssel futtatjuk a programunkat, akkor fontos dolgokat vehetünk észre:

- Az `int` típusú `szEv` deklarációjakor a C++ nem állít be kezdőértéket, azaz nem inicializálja. Lehet, hogy éppen 0 lesz az értéke, de az is lehet, hogy 1428, vagy bármi. A `string`-nek, ha nem adunk kezdőértéket, akkor `""` lesz a tartalma, az `int` (és általában a szám típusok) használata értékadással kezdődik. Erre három nyelvi lehetőség van:
  - `int szam = 1;` értékadó egyenlőségjellel;
  - `int szam(2);` `setlocale()`-hoz hasonlóan a zárójelek közé írva az értéket;
  - `int szam{3};` konstruktorral, azaz kapcsoszárójelek között megadva az értéket.
- Hiába inicializáljuk a `szEv` változónkat, a szöveges adat beolvasása után mindenképp 0 lesz az értéke. Mintha a `>>` számára a szöveg értéke 0 lenne.
- Kérdés, hogy mi lett eközben a hibásan beírt adatunkkal. Sokáig lehet kísérletezni, több adatot beírni és kiírni, de ez mind hatástalan. Ha a `>>` hibát érzékel, akkor „bezárja a kaput”. A beírt adatot csak a kapu kinyitása – `cin.clear();` – után lehet beolvasni.

Még sokáig abból indulunk ki programkészítéskor, hogy a felhasználótól érkező bemenetet nem kell ellenőriznünk, *validálnunk*. Egy „igazi” alkalmazás esetében a hibákra való felkészülés (felhasználótól kapott rossz bemenet, elfogyó háttértár, megszakadó hálózati kapcsolat stb.) a programnak igen jelentős része. Csak azért kell tudnunk, hogy mi történik egy hibás bemenet esetén, mert van, hogy mi magunk rontjuk el a kódot vagy teszteléskor az adat beírását. Ha fut a programunk, de nem az történik, mint amit elvárunk, a hibakeresést a változók értékének ellenőrzésével kezdjük: Van kezdőértéke? Mennyi? Mikor módosul?

Kiegészítés: Szöveggé alakítás

Az extractor (`>>`) és az inserter (`<<`) az esetek jelentős részében megoldja, hogy a kapott szövegből megfelelő típusú adat keletkezzen, illetve, hogy egy adat szövegesen megjelenjen a konzolon. Ebben a fejezetben arról lesz szó, hogy mit tehetünk akkor, ha nem a konzol, hanem egy `string` típusú változó az, ahonnan az adat származik vagy amibe az adatot át szeretnénk tenni szöveges formában.

A C++-ban kétféle karakterlánc van. Az egyszerűbb típus pusztán a karakterek sorozatát jelenti. Amikor idézőjelek között megadunk egy szöveget, az ilyen típusú, de van egy „okosabb” adattípus is, a `string`, amit már szintén használtunk. Láttuk, hogy két karaktersorozat nem fűzhető össze, de ha legalább az egyik `string`, akkor már összefűzhetők. Egy `x` karaktersorozatból például `(string)x` explicit átalakítással lesz `string`. A visszafelé alakításra ritkán lesz szükségünk (esetleg csak évek múlva vagy soha). Csak a teljesség kedvéért az `s` `string`ből az `s.c_str()` készít karaktersorozatot.

Egy-egy számadat szöveggé alakítására általában kiíráskor kerül sor, de néha praktikus előre elkészíteni a számokat is tartalmazó szöveget. Ilyenkor hasznos a `to_string(szám)` függvény.

```
    string szoveg = to_string(3.1417)
```

A függvény zárójelei közé bármilyen egész vagy lebegőpontos számot vagy változót beírhatunk. Visszafelé már egy kicsit bonyolultabb a helyzet, mert minden típushoz külön függvény kell. Az `int` egész számra alakító a `stoi(szöveg)` függvény, a `double` lebegőpontos számra alakító a `stod(szöveg)` függvény.

A double típusú adatok szöveggé alakítása és kiírása további igényeket vet fel. Gyakori igény, hogy az eredmény megadott számú tizedesjeggyel legyen kiírva. A megoldáshoz vezető egyik út során a szorzás, `int`-té alakítás, szöveggé alakítás és szöveg darabolása szükséges. A másik út, hogy használunk egy előregyártott megoldást. Ez a megoldás az `<iomanip>` csomagban van, amit az `iostream`-hez hasonlóan be kell vonni a használatba. Ebben a programozók minden igényhez megtalálják a megoldást, de nekünk bőven elég lesz kettő:

- `<< setw(helyek_száma)` – A következő kiírt adatnak legalább `helyek_száma` helye lesz.
- `<< setprecision(számjegyek_száma)` – ettől kezdve `számjegyek_száma` értékes jegy jelenik meg a számból. Nem csak a tizedesjegyek számítanak, hanem az egész helyiértékek is. Ugyanakkor a 0,5 csak 1 értékes számjegyet tartalmaz.

Kipróbáláshoz minta:

```
double a = 3210.123;
int b = 4346;
double c = 0.2345;
cout << setprecision(3) << a << "|" << b << "|" << c << "|" << endl;
cout << setprecision(12);
cout << setw(10) << a << "|" << setw(10) << b << "|" << c << "|";
```

Kérdések, feladatok

1. Keressük meg és próbáljuk ki, hogyan lehet a 3,14159265 számot két tizedesjeggyel kiírni!
2. Próbáljuk ki a helyőrzőkben a szöveg igazítását és a formátumok megadását! Írjuk ki a szövegeket összefűzéssel, illetve a formátum és szövegigazító függvények használatával is!
3. Kérjünk be egy kilométerben mért távolságadatot a felhasználótól, és írjuk ki tengeri mérföldre átváltva! (Egy tengeri mérföld 1852 méter.) Ha elkészültünk, megírhatjuk a feladat megfordítását.
   a) Kérjünk be a felhasználótól két számot, tároljuk őket egy-egy változóban!
   b) Adjuk össze őket, és írjuk ki az eredményt!
   c) Írjuk ki az eredmény elé, hogy „Az összegük:”!
   d) Írjuk ki magukat a számokat is! Ha például 2-t és 3-at adott meg a felhasználó, akkor a kimenet legyen ilyen: `2 és 3 összege: 5`.
   e) Írjuk át a programot szorzásra! (A szorzás jele a *.)
   f) Írjuk meg a többi matematikai számítást is! (Az osztás jele a /, a kivonásé a –.)
   g) Kihívást jelentő feladat: A hatványozás és a gyökvonás megírásához, a `<cmath>` csomagban található függvényeket lehet használni.
4. Vegyük elő a korábban megírt, nevünket kiíró programot! Ebben a feladatban már megoldottuk a vezeték- és a keresztnév kiírását. Bővítsük úgy a programot, hogy kérdezze meg a születési évünket is, és írja ki a nevünkkel egy sorban: `Kék Blamázs, 2011`. A születésiév-megállapító programunkkal egybegyúrva készíthetünk olyan programot is, amelyik megkérdezi a neveinket, a születési évünket, és a bulvárcikkekben szokásos formában, a korunkkal együtt írja ki a nevünket: `Fehér Karnis (21)`.
5. Adott óra, perc, másodperc hármassal megadott időt váltsunk át másodpercre! (Az adatokat nem kell mindenképp a felhasználótól kérni, beírhatjuk őket a programba is.) Sikeres megoldás után készítsük el a feladat megfordítottját!
6. Kihívást jelentő feladat: Milyen szöget zár be egymással a kis és a nagy mutató adott időpontban? Kérjük be az óra és perc értéket, írjuk ki a mutatók közötti szöget fokban!

Mint a legtöbb programozási nyelvben, a C++ nyelvben is meg tudjuk mondani egy osztás maradékát. Ez az úgynevezett modulo osztás, vagy egyszerűbben csak *mod*, a jele a `%`. Így a `9 % 4` értéke 1, mert kilencet néggyel osztva egy a maradék. Ugyanakkor az egész számok (`int`) körében a `/` jel a bennfoglaló osztás jele, az eredménye egész szám. A `9 / 4` értéke 2, azaz kilencben a négy kétszer van meg (a maradékot a másik, a `%` jellel elvégzett művelet eredményeként adhatjuk meg). A lebegőpontos (`double`) értékek között a `/` jel részekre osztást végez. Emiatt `9.0 / 4.0` értéke 2.25.

Ha két egész (`int`) szám hányadosát lebegőpontos számként szeretnénk megkapni, akkor az egyiket (tipikusan a számlálót) explicit módon át kell alakítani lebegőpontossá. A `(double)9/4` értéke 2.25, mert a `9` átalakult `9.0` értékre, emiatt a '`/`' a részekre osztást el tudja végezni.

Különösen arányok és százalékok számításánál („hányad rész") figyeljünk, mert bennfoglalással, egészek közötti osztással a 9/10 értéke 0, ami 100-zal szorozva: 9/10*100 is nulla. Ha a számításban van szorzás és osztás is, akkor általában érdemes a szorzásokat előre venni: 100*9/10 értéke 90, de ilyenkor is problémát jelenthet, hogy az eredmény egész szám.

Segíts magadon, az IDE is megsegít!

Az integrált fejlesztői környezetek, így a Code::Blocks és a Visual Studio is, többféle eszközzel segíti a programozót a programjának a megírásában. Ezek közül most a kódkiegészítés, a helyérzékeny kódajánlás, a kódszínezés és a helyi súgó használatát nézzük át.

A színek jelentése

Az IDE általában lehetővé teszi, hogy többféle színvilág közül válasszunk. Van, aki a világos, más a sötét színsémát szereti, ez ízléstől, megszokástól és a környezet megvilágításától is függ. A kódok színezése az egyes színsémákban eltérő, de a programozási nyelv elemei alapján a kódon belül egységes jelentéssel bír. A C++ nyelv eredendően procedurális, utasítás orientált, ezért a kód színezése azt mutatja, hogy az adott jel, kifejezés (szó) az algoritmus szempontjából milyen funkciójú. A kódszínezés olyan, mint nyelvtanból a mondatelemzés. Nem a színnek van jelentése, hanem az azonosan megjelenőknek van közös jelentése.

▶

A Code::Blocks C++ kódjának színezésében a legfeltűnőbbek a pirossal kiemelt jelölések. A C++ nyelvnek ezek az operátorai. Az egyes jelölések tipikus elemei a programozási nyelveknek. Amivel eddig találkoztunk:

`+` Műveleti jelek. Ide tartozik a +, −, *, /, %.

`=` Az értékadás és módosítás egyenlőségjele.

`;` Pontosvessző, az utasítások lezárásának jele. Olyan, mint a bevitel végén az Enter, jelzi az utasítás végét.

`,` Vessző a felsorolás elemeinek az elválasztója

`()` Kerek zárójelpár, mindig párban szerepel. A matematikai kifejezések zárójelezésén túl leginkább függvények – például `to_string(3)` –, eljárások – például `setlocale(LC_ALL, "")` – és konstruktorok – például `int szam(2)` – paramétereit fogja egybe. A kód írásakor a nyitó zárójel begépelésével a csukó is megjelenik, de ha több csukó zárójel egymásmellé

kerül, akkor beírás helyett a már ottlévőt tekinti párnak. Ilyenkor a legvégére tudunk új csukó zárójelet tenni.

{} Kapcsos zárójelpár, mindig párban szerepel. Ez a blokk-képző, kijelöli a kód egységeinek az elejét és a végét. Erre is igaz, hogy a nyitó jel beírásakor a záró párja is megjelenik és a kód írását a kétjel között folytathatjuk. Mivel egyértelmű, hogy a záró kapcsoszárójelnél vége van egy utasításnak, ezért ezután jellemzően nem kell pontosvessző. Blokk kijelölő szerepe mellett konstruktorhoz is használható – például az `int szam{2}` létrehoz egy szam nevű egész számot, aminek 2 az értéke.

Nincs könnyű dolga a programozási nyelv fejlesztőinek. A leggyakoribb jelöléseknek célszerű a legrövidebb leírási módot használni, de korlátos a használható jelek száma. Operátor az összes relációs jel, a logikai műveletek jelei, az elem kiválasztás [] jele és a részlet jelölésére a pont is.

▶

Amikor a programot létrehozzuk, rögtön látható néhány kulcsszó: **`using, namespace`**, **`return`**. Azután ez kiegészült az **`int`**, **`double`** egyszerű változók típusaival, a **`const`** jelzéssel. … és a lista tovább fog bővülni.

▶

A programunkban minden „**`#include`**” után egy-egy megnevezés van. Ezek előre megírt programrészleteket tartalmazó csomagok. Az egyértelműség megkívánja, hogy minden elnevezett egységnek más legyen a neve, ezért a csomagokból származó, így már lefoglalt kifejezések (szavak) színezése azonos. Az **`std`** valójában része minden más elnevezésnek. Például a **`cout`** teljes nevén **`std::cout`**, a program elején a névtér (**`namespace`**) megadása olyan, mintha egy családban a vezetéknevet kiemelnénk előre és attól kezdve mindenkit csak a keresztnevén szólítanánk. Figyeljük meg, hogy ha töröljük az **`#include …`** sort (sorokat), akkor a programunk nem ismeri fel a foglalt szavakat, de a kulcsszavakat továbbra is érti.

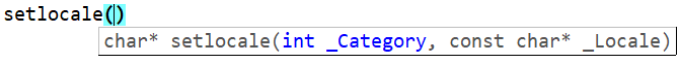
▶▶▶▶▶▶

A kód megértését, hibakeresést segíti a többi szín is. A konkrét adatoknak (literáloknak) típustól függő a színezése. Színpaletta cseréjével vagy egy másik kódszerkesztőben nézve a kódot lehet, hogy a Code::Blocksban különböző színű elemek színe egyforma lesz, miközben az azonos színűekből elkülönülnek a változónevek vagy a függvények neve.

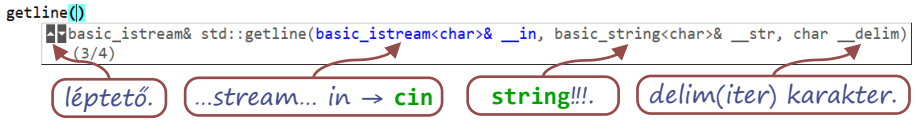
Kódkiegészítés és helyi súgó

Amikor elkezdünk írni valamit, az IDE igyekszik a segítségünkre sietni, kiírja a beírt karakterek alapján lehetséges kifejezések listáját. Célszerű erre – azaz a képernyőre – figyelni a billentyűzet helyett, mert a felajánlott lehetőségek közül nyilakkal választhatunk, a tabulátorral beírhatjuk a kívánt kifejezést, amiben így valószínűleg nem lesz helyesírási, gépelési hiba.

Ha függvény vagy eljárás nevét írjuk be, amit beírás közben felismer a Code::Blocks, akkor a nyitózárójel beírása után a paraméterekről helyi menü tájékoztat. Minden paraméternek láthatjuk a típusát (például **`int)`** és a szerepére vonatkozó kifejezést, változónevet (például: _Category). Bár sok érthetetlen, ismeretlen jelzés is lehet, de egy-egy részlet segíthet az intuitív megértésben.

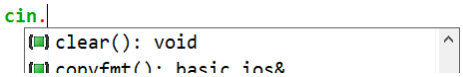


Ha többféleképpen lehet egy függvényt paraméterezni, akkor a lehetséges paraméterezések között a baloldalon látható nyilakra kattintva érjük el.



A függvény neve előtt az eredmény típusát láthatjuk. A név után a zárójelben az elvárt sorrendben láthatjuk a paraméter típusát és belső használatú nevét. Mindig az éppen aktuális paraméter van kiemelve.

Később egyre hasznosabb ismeret lesz, hogy egyes foglalt szavak olyan dolgokat – objektumokat- neveznek meg, amiknek a neve után egy pontot beírva megkapjuk a részeinek a listáját Például a `cin`-nek van `clear()` eljárása.



A `()` miatt (és az előtte levő jelzés alapján) vagy függvényről vagy eljárásról lehet szó. A kettőspont után írt `void` azt jelenti, hogy a `clear()` úgy csinál valamit, hogy a végén nincs semmilyen adat, amit egy változónak át lehetne adni, ezért ez egy eljárás.

Tanulást is támogató eszközök

A programkódot azért írjuk, hogy a fordítóprogram számára érthető módon megadjuk, mit csináljon a programunk. Amellett, hogy a fordítóprogramnak értenie kell a kódokat, ugyanolyan fontos, hogy mi is értsük a kódot. Ne csak abban a percben értsük, amikor megírjuk, hanem másnap is, sőt két hónap múlva is, vagy akár évek múlva is. Egy programkód olvasottságát vizsgálva azt láthatjuk, hogy egy kódrészletet többször olvas és fordít saját nyelvére ember, mint fordítóprogram. Ezért a programkódot a szakmabeli programozók is a kód humán olvasójának szóló jegyzetekkel egészítik ki. Ez annyira természetes, hogy a programozási nyelv kétféle megjegyzésre is lehetőséget ad.

C++ nyelvben a hosszabb megjegyzéseket, jegyzeteket a `/*` és `*/` jelek közé írhatjuk. Azt szokták mondani a jelölés magyarázataként: ami e jelek között van az – a fordítóprogram számára – nem oszt, nem szoroz. Ez a jelölés lehetőséget ad arra, hogy a kódunkba beírjuk a feladat szövegét, esetleg hosszabb magyarázatokat írjunk a szöveghez, emellett arra is jó, hogy a kód szavai között rövid megjegyzéseket írjunk.

A rövidebb, egysoros megjegyzések `//` után írhatók. A hatása a sor végéig tart. Ezt használjuk a változónevek jelentésének kifejtésére, egy-egy összetett képlet szerepének tisztázására. Az IDE abban is segít, hogy ha olyan kódrészletet írunk, ami pillanatnyilag nem szükséges, akkor a kijelölt sorok mindegyikét ezzel a jelöléssel aktív kódból megjegyzéssé tehessük, illetve újra aktiváljuk: `Edit\Comment` illetve `Edit\Uncomment`. Billentyűkombinációval CTRL+SHIFT+C, illetve – a kicsit veszélyesebb visszavonás `Crtl+Shift+X`.

Van egy sokkal látványosabb és máshol is alkalmazható megoldás is: A kurzort tegyük az első módosítandó sor elejére. Ezután a SHIFT+ALT+LENYÍL segítségével a kurzort kibővíthetjük a többi sorra. (Hasonlóan A SHIFT+ALT+FELNYÍL felfelé bővít.) A kibővített kurzornál bármit beírva vagy törölve az műveletet minden sorban egyszerre elvégezhetjük.

```
//int a = 2;
//int b = 3;
//int c = 4;
```

A programkódjaink egyre több sorból fognak állni. A programozás egyik alapelve, hogy a feladatokat bontsuk le kisebb – és még kisebb – egységekre, majd a programunkat ezekből az alapelemekből építsük fel. A kód áttekinthetőségét jelentősen növeli, ha a kód egy részét el tudjuk rejteni. A kódot írhatnánk egy sorba is, de az átláthatóság kedvéért utasításonként sorokra tördeljük és a kapcsoszárójelekkel határolt blokkok elejét és végét is a többitől elkülönült sorba írjuk. Ebben az esetben az IDE felismeri az egységeket és lehetőséget ad a belső sorok elrejtésére, a kódrészlet összecsukására, kibontására.

```
int main()
{
  //1. feladat
  setlocale(LC_ALL, "");
  cout << "Mikor születtél? ";
  int szEv(4);
  cin >> szEv;
  //cin.clear();
  //string marad;
  //cin >> marad;
  int felhasznaloKora = IDEIEV - szEv;
  cout << felhasznaloKora << " éves vagy." << endl;
  /*----------------------------
    innentől a double vizsgálata
    ----------------------------*/
  cout << "Hány Euro egy gyertya? ";
  double egyGyertya;
  cin >> egyGyertya;
  double gyertyakAra = felhasznaloKora * egyGyertya;
  cout << "A gyerták "<< gyertyakAra << " Euróba kerülnek.";
  int euro = (int)gyertyakAra;
  int cent = (int)((gyertyakAra - /*(double)*/euro) * 100);
  cout << "A gyertyák ára: " << euro << " Euro " << cent << " cent" << endl;
  return 0;
}
```

▶ Jegyzetelési módszerek: tagolás, megjegyzések, régi változatok

Elágazások

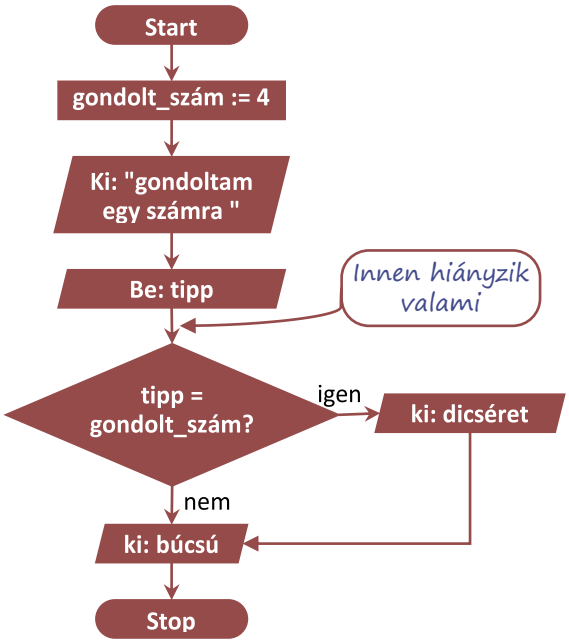
Eddig csupa olyan programot írtunk, ami elkezdődött az elején, sorban egymás után végrehajtott minden utasítást, aztán kilépett. Ebben a fejezetben ezen változtatni fogunk.

Gondoljunk egy számra

A jegyzet elején már láttunk egy olyan programot, amelyikben elágazás van. Az a program mást csinál, ha nem vagyunk még tizennégy évesek, és mást, ha már betöltöttük ezt az életkort. Hasonló működésű az ugyanott megismert „Mi a neve Mátyás királynak?” program.

Az alábbi folyamatábra is egy hasonló programot ír le. Nincsenek benne konkrét utasítások, mert fontosabb, hogy először átgondoljuk, hogy mit csinál a program, ráérünk utána azon elmélkedni, hogy miként kódoljuk.

1. Mit csinál a program?
2. Mi az a lépés, amit nem tüntettünk fel? (Ha most nem jövünk rá, nem probléma: hamarosan úgyis megírjuk a programkódot, akkor szólni fog a C++.)
3. Hogyan olvassuk ki a := jelet? Mi a megfelelője C++-ban?



Miközben a programunkat folyamatábrával ábrázoljuk, algoritmust (receptet) adunk a bennünket érdeklő probléma megoldására. A folyamatábrák elég látványosak, de hamar lelógnának a könyvlapról, így aztán a gyakorlatban gyakrabban használunk egy másik algoritmusleíró eszközt, a mondatszerű leírást. A leírás szabályaira rá fogunk érezni.

```
gondolt_szám := 4
ki: "gondoltam egy számra"
be: tipp
tipp átalakítása egésszé
elágazás
ha tipp = gondolt_szám:
    ki: dicséret
elágazás vége
ki: búcsú
```

1. Vessük össze ezt a leírást a folyamatábrával!
2. Melyik az a művelet, ami a mondatszerű leírásban már megvan, de a folyamatábrában még hiányzik?

Készítsük el a fenti két algoritmusleíró eszközzel megtervezett program kódját!

```
 6.  int gondolt_szam = 4;
 7.  cout << "Gondoltam egy számra. Tippeld meg! ";
 8.  int tipp;
 9.  cin >> tipp;
10.  if (tipp == gondolt_szam)
11.      cout << "Ügyes!";
12.  cout << "Pápá." << endl;
```

Ez szándékosan két egyenlőségjel

Az előző sor folytatása is lehet, de így áttekinthetőbb a kód

Teszteljük a programunkat: adjuk meg a helyes megoldást, de próbáljuk ki helytelennel is!

A program következő változatában – más szóval: verziójában – kicsit bővebben dicsérünk, illetve a hibásan tippelő felhasználókat kicsit ugratjuk. A mondatszerű leírás a következő:

```
gondolt_szám := 4
ki:"gondoltam egy számra"
be: tipp
tipp átalakítása egésszé
elágazás
ha tipp = gondolt_szám:
    ki: kétsoros dicséret
különben
    ki: ugratás
elágazás vége
ki: búcsú
```

Módosítsuk ez alapján a folyamatábrát (segítség: a rombuszból lefelé nem lesz nyíl, de balra igen, és a két nyíl a rombusz alatt összetalálkozik)!

A dicséret második sora újabb kiírás utasítást jelent az előző alatt. Írjuk be az új sort behúzva és behúzás nélkül vagy egysorban az előzővel:

```
12.  cout << "Gratulálok!" << endl;
```

Mindegyik esetben teszteljük a programunkat! Sajnos egyik beírási mód sem jó. A C++ a feltétel teljesülése esetén csak egy utasítást hajt végre. Ezért, ha több utasítást szeretnénk írni, akkor ezeket „egybe kell foglalni”, **kapcsos zárójelek** közé kell zárni. Mivel a C++-nak mindegy, hogy hol van új sor vagy tabulátor, írhatjuk a teljes utasítást egyetlen sorba, de így számunkra olvashatatlanabb lesz. A tördelésre kétféle szokás alakult ki.

- Az egyik szerint a nyitó kapcsos zárójel a feltétel sorának végén marad, utána az utasítást új sorban kezdjük egy tabulálással beljebb.

```
10.  if (tipp == gondolt_szam){
11.    cout << "Ügyes!" << endl;
12.    cout << "Gratulálok!" << endl;
13.  }
```

Nyitó { Utasítások Záró }

- A másik szerint a nyitó kapcsos zárójelet új sorba írjuk és egy sorral lejjebb és beljebb írjuk az első utasítást.

```
10.  if (tipp == gondolt_szam)
11.  {
12.    cout << "Ügyes!" << endl;
13.    cout << "Gratulálok!" << endl;
14.  }
```

Nincs vége, **nincs ;** (pontosvessző)

Nyitó { Utasítások Záró }

Akárhogy is kezdjük, a többi utasítás az első alatt, ugyanannyira behúzva kezdődik, a záró kapcsoszárójel pedig külön sorban, az **if**() i-jével egyvonalban van.

A C++ nyelvre az első tördelés jellemzőbb, de a második is gyakori. Ezért az átláthatóságot figyelembe véve ebben a jegyzetben néha az egyiket, máskor a másikat alkalmazzuk.

Ha elkészültünk a gratulációval, írjuk meg az ugratós részt is. A „különben” szó angolul „**else**” – ezt kell használnunk kódoláskor. Írhatjuk a záró kapcsos zárójel után vagy külön sorba is, utána a kód tördelése az előző esetekhez hasonló.

A kész program:

```
 6.  int gondolt_szam = 4;
 7.  cout << "Gondoltam egy számra. Tippeld meg! ";
 8.  int tipp;
 9.  cin >> tipp;
10.  if (tipp == gondolt_szam) {
11.    cout << "Ügyes vagy!" << endl;
12.    cout << "Gratulálok!" << endl;
13.  } else {
14.    cout << "Hosszan gondolkodtál rajta? :)" << endl;
15.    cout << "Nem érte meg. ;)" << endl;
16.  }
17.  cout << "Pápá." << endl;
```


A programunk tanulságai:

1. Ami az **if** után következik, az az elágazás feltétele.
2. A feltételvizsgálatban két, egymás utáni egyenlőségjel kell.
3. Ami az elágazás ágaiban van (a fenti program 11–12. és 14–15. sora), az kapcsos zárójelek között van. A C++-ban az elágazás egy ágán egy utasítás lehet önállóan, több utasítást kapcsos zárójellel „egyesítenünk” kell.

Az összehasonlítás jelölése

A programozásban általában az egy egyenlőségjel egy felszólítás (ismerjük már, értékadáskor használjuk: „legyen egyenlő”), a két egyenlőségjel kérdés: A tipp egyenlő-e a gondolt számmal?

Nem csak azt szoktuk kérdezni, hogy egyenlő-e két szám. Lehet kisebb (<), nagyobb (>), kisebb vagy egyenlő (<=), nagyobb vagy egyenlő (>=), illetve nem egyenlő (!=). Ezeket a != kivételével matematikából is ismerjük.

Az informatikai jelölések nagyon gyakran a matematikai jelölést veszik át, de az egyenlőségjelnél problémát jelent, hogy a matematika kétféleképpen használja az „=” jelet. Egyrészt két kifejezés egyenlőségét vizsgálva; például egyenletek felírása során. Másrészt definíciókban, például az y = f(x) függvény felírásakor a legyen egyenlő értelemben. Informatikában háromféle megoldás van a félreértések feloldására:

1. A két egyenlőségjel a kifejezésen belüli helye miatt megkülönböztethető. Például táblázatkezelésben =HA(A1=A2;B1;C1) a képletet kezdő egyenlőségjel „legyen” értelmű, az A1=A2 között összehasonlító szerepe van. Ilyen esetekben a „nem egyenlő” jelölése a <> szokott lenni, mert az áthúzott egyenlőségjelet nem tudjuk kódolni.
2. Ahogy C++-ban is használják: Az „=” az értékadó, az „==” az összehasonlító egyenlőségjel. A „nem egyenlő” jelölése ebben az esetben mindig a „!=”.
3. Ahogy a mondatszerű leírásban használják: Az értékadó egyenlőségben a „legyen” kifejezése a „:”, így a jelölése „:=”, az összehasonlításokban használják a „=” jelet. A „nem egyenlő” jelölése „<>”, de nem okoz félreértést „!=” sem.

Az „==” és a „!=”nemcsak számok, hanem szövegek egyezésének vizsgálatára is használhatók, de az ábécé szerinti rendezettséget a relációjeleknél bonyolultabb összehasonlító függvényekkel lehet csak vizsgálni.

Összetett feltétel

Szeretnénk bővíteni a programunkat. Ha csak eggyel tippelt mellé a felhasználó, akkor ezt eláruljuk neki. Elsőként az algoritmus mondatszerű leírását módosítjuk. Az új, „különben ha” ágat megvalósító C++-utasítás az **else if**.

```
gondolt_szám := 4
ki:"gondoltam egy számra"
be: tipp
tipp átalakítása egésszé
elágazás
ha tipp = gondolt_szám:
    ki: kétsoros dicséret
különben ha csak egyet tévedett
    ki: csak egyet tévedtél
különben
    ki: ugratás
elágazás vége
ki: búcsú
```

Akár neki is foghatnánk a kódolásnak, de hogy programozandó a „ha csak egyet tévedett”? Ilyen utasítás nincs, ezért az algoritmusunkat egy-két lépéssel tovább kell finomítanunk.

Vegyük észre, hogy kétféleképp lehet egyet tévedni: eggyel nagyobbat vagy eggyel kisebbet tippelve. Az „eggyel nagyobbat tippelt” így írható le:

```
tipp = gondolt_szam + 1
```

A másik feltétel megfogalmazása nem okozhat gondot, de ezek szerint két feltétel lett az egyből. Megírhatjuk úgy az algoritmust (és a programot), hogy két „különben ha” ágat adunk meg, ugyanazzal a kiírandó üzenettel, de ez nem szerencsés – például azért, mert ha módosítani kell az üzenetet, akkor két helyen is módosítanunk kell, és az egyiket előbb-utóbb elfelejtjük megtenni. Alakítsuk inkább a két feltételünket egy összetett feltétellé:

```
tipp = gondolt_szam + 1 vagy tipp = gondolt_szam - 1
```

A két feltételből így egy lett, hiszen a „vagy” szó kapcsolja össze őket, ami igaz értéket ad, ha az egyik teljesül. Ha „és” kapcsolná őket össze, mindkettőnek teljesülnie kellene, hogy a teljes összetett feltétel igaz legyen. C++ nyelven a `vagy` jelölése **||** (billentyűzeten kétszer AltGr+W); az `és` jelölése **&&** (billentyűzeten kétszer AltGr+C).

A kód most így néz ki (mindegyik zárójelezési szokást 1-1 esetben alkalmazva):

```
 6.  int gondolt_szam = 4;
 7.  cout << "Gondoltam egy számra. Tippeld meg! ";
 8.  int tipp;
 9.  cin >> tipp;
10.  if (tipp == gondolt_szam) {
11.    cout << "Ügyes!" << endl;
12.    cout << "Gratulálok!" << endl;
13.  }
14.  else if (tipp == gondolt_szam + 1 || tipp == gondolt_szam - 1)
15.    cout << "Ó, csak eggyel tévedtél"<< endl;
16.  else {
17.    cout << "Hosszan gondolkodtál rajta? :)" << endl;
18.    cout << "Nem érte meg. ;)";
19.  }
20.  cout << "Pápá." << endl;
```

Kérdések

1. Hány **else if** ág, illetve hány **else** ág szerepelhet egy elágazásban?
2. Melyiket kell utolsóként megadni?
3. Melyiknek nincs feltétele?

4. Létezhet olyan elágazás, amelyikben nincs egyik sem?
5. Kihívást jelentő feladat: Az alábbi sor miért nem jó?

```
9.  else if (tipp == gondolt_szam + 1 || gondolt_szam - 1)
```

Összetett feltétel logikai szabályai

Két feltétel összekapcsolása a „vagy” szóval némiképp hasonlít az összeadáshoz. Ha a hamis állítást 0-val jelöljük, az igaz állítást 1-gyel, akkor a vagy eredménye – ami az igaz érték, ha az egyik teljesül – ugyanolyan értelmű, mit az összeadás eredménye: nem nulla, ha bármelyik tagja nem nulla. Az összeadás során két számot adunk meg és az eredmény egy szám lesz, azaz az összeadás számokon végzett kétoperandusú művelet. Hasonlóan, a „vagy” a logikai kifejezéseken végzett kétoperandusú művelet. A „vagy” műveleti jele C++-ban a **||**, de más nyelvekben az angolból származó **OR** jelöli, mondatszerű leírásban és folyamatábrán magyarul is írhatjuk: **VAGY**. Matematikában jelölhetjük az **V** jellel, illetve – a hasonlóságot kiemelve, a Bool-algebrában a jele **+**.

Hasonlóan, kétoperandusú logikai művelet az „és”, műveleti jele C++-ban az &&, más programozási nyelvekben az angolból származó AND jelöli, mondatszerű leírásban és folyamatábrán magyarul is írhatjuk: **ÉS**. Matematikában jelölhetjük az **Λ** jellel, illetve – a hasonlóságot kiemelve, a Bool-algebrában a jele ***. Ahogy a szorzásnál mondhatjuk, hogy egy szorzat nulla, ha bármelyik tényezője nulla, az ÉS műveletre is jellemző, hogy hamis lesz az értéke, ha az egyik feltétel hamis. (Másképp is mondhatjuk: igaz lesz az értéke, ha mindegyik igaz).

Azt gondolnánk, hogy a VAGY és az ÉS feltételei felcserélhetőek (a műveletek kommutatívak), azonban ez csak a matematikai felhasználások esetén igaz. Az informatika, a programozás erre a két műveletre a „**rövidzár**-kiértékelés” (másképp: lusta kiértékelés) szabályát alkalmazza. Ez azt jelenti, hogy az összetett feltételt csak addig értékeli ki a program, amíg az eredmény (hogy igaz-e vagy hamis) nem biztos. VAGY esetén, ha az első feltétel (operandus) igaz, akkor a másodikat nem vizsgálja. Az ÉS művelet esetén, ha az első feltétel hamis, akkor a másodikat már nem vizsgálja. Természetesen, ahogy összeadásokból vagy szorzásokból is lehet több egymás után, az ÉS és VAGY műveletekből is lehet többet megadni. Ilyenkor az eredményt addig számolja a program, amíg balról olvasva egy VAGY előtt igaz eredmény, illetve egy ÉS előtt hamis eredményt kap. Az ezt követő feltételeket, műveleteket már nem nézik.

Gondoljunk bele, mi is hasonlóképp végeznénk el a műveletet. Ha 10 számot össze kell szoroznunk, de a 3. szám a 0, akkor nem kell figyelnünk, hogy utána milyen értékek vannak…

A rövidzár kiértékelésnek a programozásban nagyon komoly gyakorlati oka van. Tipikus feladat, hogy az egyik feltétel az, hogy egy dolog létezik-e (értelmezhető-e), a másik pedig, hogy jó-e, azaz megfelelő-e. Például, ha egy számot bekérünk és ki akarjuk írni, hogy páros szám-e, akkor az összetett feltétel: a beírt adat egész szám ÉS a beírt adat páros szám. Súlyos programhibát okozhat, ha a kapott adat szöveg és a program megpróbálná kettővel elosztani. Ezért programozási szabály, hogy *összetett feltételben mindig azt a feltételt írjuk előre, amelyik a vizsgált adat létezését, érvényességét, értelmességét ellenőrzi, hogy a rövidzár-kiértékelés megállítsa a kiértékelési folyamatot egy esetleges programhiba előtt*.

Ha a feltételeket nem azonos műveletekkel kapcsoljuk össze, akkor felmerül a kérdés, hogy van-e a szorzáshoz és összeadáshoz hasonló precedencia-szabály, erősebb-e – azaz magasabb rendű-e – az egyik, mint a másik. Általában az ÉS erősebb (matekosan a * erősebb), de ez nem minden programozási nyelvben van így, ezért célszerű zárójelezéssel egyértelművé tenni,

hogy a többszörösen összetett feltételeket hogyan csoportosítjuk. Inkább legyen több zárójel, mint egy félreértett szabály.

A harmadik, informatikában jellemzően használt logikai művelet az egyoperandusú tagadás. A NEM művelet az utána szereplő feltétel eredményének ellenkezőjét adja, így, az igaz-ból hamis, a hamisból igaz értéket állít elő. Jelölése C++-ban és a C-vel rokon nyelvekben a felkiáltójel (!), más nyelvekben az angolból származó **NOT** jelöli, mondatszerű leírásban és folyamatábrán magyarul is írhatjuk: **NEM**. Matematikában jelölhetjük a ¬ jellel, illetve a komplementer képzéssel való hasonlóságot kiemelve, a Bool-algebrában a jele a felülvonás. Például az a != e így írható le az a == e tagadásaként a fenti jelölésekkel: !(a == e), NOT(a == e), NEM(a = e), ¬(a = *e*), $\overline{a=e}$.

Véletlenszám-előállítás

Meglehetősen unalmas lehet, hogy a programunk mindig a négyre gondol. A legtöbb programozási nyelvben van valamilyen módszer véletlenszám előállítására, más szóval *generálására*. A C++-ban erre az előre megírt `rand()` függvényt használjuk. Erre a függvényre nincs minden programban szükség, ezért be kell kérni a megfelelő csomagot. A kód legelejére írjuk be:

```
2.  #include <cstdlib>
```

A `cstdlib` a C++ standard csomagja, a programozáshoz szükséges általános kiegészítéseket tartalmazza. A már bent lévő `iostream` az konzol input és output műveletekért felelős.

Véletlenszám előállítása egy gép számára lehetetlen, mert a gép mindig meghatározott szabályok, utasítások alapján dolgozik. Ezért a véletlenszám előállítása azt jelenti, hogy egy olyan műveletet végez, aminek az eredménye nem jósolható meg, minden lehetséges értéknek egyforma az esélye. Például: vegyünk egy irracionális számot – mondjuk egy prímszám négyzetgyökét -, írjuk egymásmellé valahány tizedesjegyét, majd válasszuk ki az így keletkezett számsorozat középső részét. A következő alkalommal emeljük négyzetre az előző számot (az eredmény kétszer olyan hosszú lesz), és ennek vegyük a közepét… Így megjósolhatatlan, hogy milyen értékeket kapunk. Csak egy aprósár hiányzik: mi legyen az első szám?

A véletlenszám függvényünk használatához először meg kell mondanunk, hogy mi legyen a kiindulási érték. Ez a véletlenszámgenerátor inicializálása, amit C++-ban a srand() eljárás valósít meg.

```
8.  srand(23);
```

Ezután már használhatjuk a rand() randomszám készítő függvényünket, de ha mindig 23 lesz a kezdőérték, akkor a véletlenül kiadott értékek ugyanazt a sorozatot fogják kiadni. Az így nem az igaz. Valami olyan kezdőérték kellene, ami a program minden futtatásakor más. Mi az, aminek mindig más az értéke? Az idő. Ezért kiindulási értéknek általában a számítógép órája szerinti időt szokták megadni. Ez biztosan változik két futtatás között.

A C++ nyelvben az idő kezeléséhez újabb csomagra van szükségünk, a `ctime`-ra. Így programunk elején ezt is fel kell tüntetni:

```
3.  #include <ctime>
```

A pontosidőt egész szám formában a time(0) függvényhívás adja meg. Így a véletlenszám inicializálásakor ezt kell a szám helyére beírni:

```
8.  srand(time(0));
```

Ezzel programunk első sorai a következőképp alakulnak:

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  #include <ctime>
4.  using namespace std;
5.  int main()
6.  {
7.  srand(time(0));
8.  int gondolt_szam = rand() % 5 + 1;
9.  cout << "Súgok: " << gondolt_szam << endl;
10. cout << "Gondoltam egy számra. Tippeld meg! "<< endl;
11. int tipp;
12. cin >> tipp;
```

ez a két csomag kell hozzá.

inicializáljuk a generátort.

a maradékok: 0 1 2 3 4:
az eredmény: 1 2 3 4 5
közül az egyik.

A tesztelés során hasznos kiírni

Figyeljük meg a függvények és eljárások használatát! A `srand()` eljárás, olyan, mint a `setlocale()`. A kódban utasításként szerepel. A `time()` és a `rand()` függvény. A `time()` függvénynek egy paramétere van, amibe most a `0`-t adjuk meg. A `time(0)` eredménye egy egész szám, amit a – szintén egy paraméterrel rendelkező – `srand()` felhasznál. A `rand()` függvénynek nincs paramétere; eredménye egy egész szám. Ez a szám lesz a `%` operátor – a maradékos osztás – első operandusa. A művelet eredményét a `+` operátor használja, végül a keletkezett számot kapja meg a `gondolt_szam` változó.

Elágazások és véletlenek alkalmazása

Feladatok

1. Kérjünk be jelszót a felhasználótól és hasonlítsuk össze a programban tárolttal! Ha a felhasználó eltalálta a jelszót, akkor írjuk ki, hogy „Helyes jelszó.”, különben „Hozzáférés megtagadva.”.

A feltételek megfogalmazásakor használható a „>”, a „<”, a „>=” és a „<=” operátor is.

2. Kérjünk be két számot a felhasználótól! Írjuk ki a nagyobbat!
3. Állítsunk elő két véletlenszámot és kérdezzük meg a felhasználótól az összegüket! Ha helyesen válaszol, dicsérjük meg!

```
1.  srand(time(0));
2.  int egyik = rand() % 10 + 1;
3.  int masik = rand() % 10 + 1;
4.  cout << "Mennyi " << egyik << " és " << masik << " összege? ";
5.  int tipp;
6.  cin >> tipp;
7.  int osszeg = egyik + masik;
8.  if (tipp == osszeg)
9.    cout << "Valóban annyi, ez igen!" << endl;
```

4. Írjunk olyan programot, amelyik bekéri két kosárlabda csapat nevét és a meccsen elért pontszámokat, majd kiírja a mérkőzés eredményét (a felhasználó válaszai vastagabbal szedve):

```
Mi az egyik csapat neve? Tóparti királyok
Hány pontot szerzett? 78
Mi a másik csapat neve? Talpasi csodatévők
Hány pontot szerzett? 54
Az összecsapás eredménye:
Tóparti királyok - Talpasi csodatévők
78 : 54
Tóparti királyok nyert.
```

5. Vegyük elő azt a programunkat, amelyik a felhasználó nevét és korát kezeli. A felhasználónak javasoljunk életkorának megfelelő olvasnivalót!
   - 0–3 év: „Totyogóknak a kettes számrendszerről”
   - 4–6 év: „Hackeljük meg az óvodát!”
   - 7–14 év: „Felhőtechnológia a menzán”
   - 15–18 év: „Big data a középiskolában”
6. Ha bonyolultabb, összetett feltételeket fogalmazunk meg, érdemes lehet zárójelek használatával egyértelműsíteni a szándékunkat. Az alábbiak közül melyik feltételmegfogalmazás biztosítja, hogy „tudjunk rajzolni”?
   a) Ha (van tollunk és van ceruzánk) vagy van egy papírlapunk: tudunk rajzolni valamit.
   b) Ha (van tollunk vagy van ceruzánk) és van egy papírlapunk: tudunk rajzolni valamit.
   c) Ha van tollunk vagy (van ceruzánk és van egy papírlapunk): tudunk rajzolni valamit.
7. Kihívást jelentő feladat: A kistesód szülinapi banános nyaflatyát csinálod. A kistesód szerint a jó banános nyaflaty jellemzője, hogy:
   a) nincs benne egyszerre vaníliás cukor és tortareszelék;
   b) ha van benne fahéj, akkor kell rá tejszínhab is;
   c) nincs benne fahéj, nem kerülhet rá tejszínhab sem.

   Írj programot a nyaflaty jóságának eldöntésére!

Ciklusok

A ciklus eredetileg valamilyen egyforma időközönként, periodikusan ismétlődő dolgot jelent, gondolhatunk holdciklusra, választási ciklusra, árapályciklusra. Mi ebben a könyvben egy olyan programrészletet értünk rajta, amely valahányszor megismétlődik.

- Minden programozási nyelvben van *feltételes* ciklus, ami egy feltétel teljesülése esetén ismétli a programrészletet. A feltételt az ismétlődő rész elé írjuk (megcsinálja-e az ismétlést), ezt nevezzük *elöltesztelős* ciklusnak, vagy *while*-ciklusnak; angol neve: *while-loop*).
- Néhány nyelvben megoldható az is, hogy az ismétlődő rész végén legyen a feltétel (azt dönti el, hogy újra megcsinálja-e). Ez a *hátultesztelős* ciklus vagy *do-while*-ciklus, angolul *do-while-loop*.
- Egyes nyelvekben van *számlálós* ciklus, itt az ismétlések számát egy számlálóval adjuk meg. Például a Scratch-ben olyan számlálós ciklus van, amiben csak az ismétlések számát kell megadni. Általánosabb, hogy megadhatjuk a számláló kezdőértékét és az elérni kívánt végértéket, a program minden ismétlésnél növeli a számlálót, figyeli, hogy az elérte-e a végértéket. Napjainkra jellemző a programozási nyelvekben a számlálós ciklus továbbfejlesztett változata, amelyben a végérték helyett feltételt adhatunk meg, így sokkal általánosabban használható. A számlálás ebben az esetben nem helyettesíti a feltételek megadását,

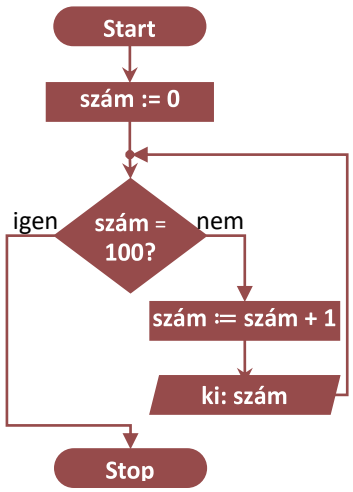
hanem azt kiegészíti. A számlálót léptető ciklus szokásos megnevezése *for*-ciklus (*for-loop)*, ami programozási nyelvekben jellemző angol *for* kulcsszóból ered.

- Néhány nyelvben a számlálós ciklus egyszerűsített változata, a *bejárós* ciklus – közhasználatú nevén *foreach* ciklus, angolul *foreach-loop* – is létezik, ami a gyakran előforduló, tipikus esetekben megkönnyíti a kódolást.

A C++ mindegyik ciklusfajtát ismeri, a négyféle lehetőség a programozó kényelmét szolgálja. Van, aki mindig csak a while-ciklust használja, más mindig a for-ciklust. A lehetőségek ismeretén túl a választást a program könnyű értelmezhetősége és a szokások befolyásolják. Mind a négy változat használatára lesz példa, de a feladatok megoldása során – ha egyébként a program helyesen működik – nem hiba a mintán láthatótól eltérő megoldás használata.

A feltételes ciklus (while-ciklus)

Elsőként megvizsgáljuk az alábbi folyamatábrát. Vajon mit csinál a program?



Mondatszerű leírással így néz ki az algoritmusunk:

```
szám := 0
ciklus amíg szám <> 100:
    szám := szám + 1
    ki: szám
ciklus vége
```

A <> azt jelenti, hogy „nem egyenlő!”
Itt írhatjuk így is: !=

C++ nyelven pedig az alábbi formát ölti programunk:

```
 6.  int szam = 0;
 7.  while (szam != 100)
 8.  {
 9.    szam = szam + 1;
10.    cout << szam << endl;
11.  }
```

A „while” annyit tesz: amíg

„Amíg” a FELTÉTEL teljesül, addig újra meg újra belépünk a ciklusba.

Kapcsos zárójelek mint az if-nél.

Ez a két sor „benne van a ciklusban” – ők a CIKLUSMAG

1. Mi történik, ha a ciklusmag első sorát elhagyjuk? (Ha végtelen ciklusba kerül programunk, akkor a konzolablakot zárjuk be – bezáró gomb vagy CTRL+BREAK vagy CTRL+C – vagy az IDE piros Abort gombjával állítsuk le a futtatást.)
2. A `!=` helyett milyen feltétellel érhetjük el ugyanezt az eredményt?
3. Hogyan változik a program kimenete, ha a ciklusmag két sorát felcseréljük? Ha a felcserélt sorokkal is szeretnénk a számokat 1-től 100-ig kiírni, mit kell még módosítani a programon?

4. Hogyan íratható ki minden második szám 100-ig? Hogyan íratható ki minden harmadik szám 100-ig?

Következő órára leírod százszor, hogy … (for-ciklus)

Bizonyára sokaknak viccekből, régi történetekből ismerős az alcímben jelzett tanári büntetés. Például le kellett írni százszor, hogy „A folyosón sétálunk.”. A fenti program egyetlen sorának módosításával megoldható ez a feladat, mi most mégis több módosítást végzünk, hogy „beszédesebb” legyen a kódunk.

- Azt, hogy hányadik kiírásnál tartunk, számláljuk, ezért a változót nevezzük „szamlalo”-nak.
- Mivel először az elsőt írjuk ki, a számláló kezdőértéke legyen 1 és addig írjuk, amíg ez az érték kisebb vagy egyenlő mint 100. Mivel azzal az értékkel kezdjük, ahányadik kiírás következik, ezért előbb kiírjuk a szöveget és utána módosítjuk a számlálót.
- A számlálót 1-gyel növeljük (nem azt mondjuk, hogy legyen önmagánál eggyel nagyobb). A „növelés”-t kódban is ki tudjuk fejezni: `szamlalo +=1`.

  Hasonló jelölést a többi alapművelettel is tudunk végezni:
  - x `-=` 3 jelentése: x-et 3-mal csökkentjük;
  - y `*=` 2 jelentése: y-t megduplázzuk;
  - z `/=` 4 esetén z-t negyedeljük.

```
14.  int szamlalo = 1;
15.  while (szamlalo <= 100)
16.  {
17.    cout << szamlalo << ". A folyosón sétálunk!" << endl;
18.    szamlalo += 1;
19.  }
```

Kódunk talán jobban érthető, de a szöveg százszor leírása kézzel semmiség, ahhoz képest, hogy egy programozónak hányszor kell számlálót használnia és egyenként növelnie az értékét, azaz számlálnia. Ezért a programozók nem szamlalo-nak hívják a számlálót, hanem az angol *iterator* szóból rövidítve „i”-vel szokták jelölni. Ha az i éppen foglalt, akkor „j” a szokásos jelölés vagy az i-t kiegészítik 1-2 betűvel, például it, ix, ai. Ez a szokás körülbelül annyira általános, mint amennyire a matematikában a változót x-szel – ha már foglalt, akkor y-nal, z-vel … – jelölik. Apró különbség, hogy míg az x általános használatának okát inkább csak találgatjuk, az iterator szó magyarul ismételgetést jelent (ez is i-vel kezdődik), de használhatjuk a magyarított megnevezést is: iterátor.

De még ez sem elég egy programozónak … A programozási nyelvek jelentős részében az egyesével számlálásra külön jelölést használnak:

- `i++`; jelentése: „i értéke ezután eggyel több”, ezt használják leggyakrabban;
- `++i`; jelentése: „i értéke eggyel nő”, az előzőtől alig különbözik, de nehezebb beírni;
- `i--`; jelentése: i értéke ezután eggyel kevesebb” – visszafelé számláláshoz;
- `--i`; jelentése: „i értéke eggyel csökken”, szintén visszaszámláláshoz.

Ezzel a két rövidítéssel így néz ki a kódunk:

```
 6.  int i = 1;
 7.  while (i <= 100)
 8.  {
 9.    cout << i << ". A folyosón sétálunk!" << endl;
10.    i++;
11.  }
```

Hatsoros kódunknak egyetlen hosszabb sora van, a kiírás. Nem lehetne kevesebb sorral megúszni? De, lehet. Ahogy a fejezet elején már láthattuk, van más ciklus is: a számlálós … és mi most itt számlálóval számoljuk, hogy hányadik kiírásnál tartunk. A C++-ban – és több más nyelven – a for-ciklussal ugyanezt a programot sokkal kevesebb sorban, a lényeges dolgokat összefogva írhatjuk meg:

```
 6.  for (int i = 1; i <= 100; i++)
 7.    cout << i  << ". A folyosón sétálunk!" << endl;
```

Első ránézésre brutális, hogy harmadára csökkent a sorok száma, de azért ez egy kicsit csalóka.

A **for** kerekzárójelei között – neve: *ciklusfej* – nemcsak a feltétel szerepel: *három paramétert adhatunk meg pontosvesszőkkel elválasztva*. A feltétel elé írjuk be a korábbi kódban a **while** előtt lévő sort, a feltétel után szerepel a számláló növelése, ami a *ciklusmag utolsó utasítása* volt. A növelés (változtatás) bármelyik megvalósítását beírhatjuk, lehet `i = i + 3` is, csak arra kell figyelni, hogy amit ide írunk, az a ciklusmagba írt utolsó utasítás után lesz végrehajtva.

A kapcsos zárójelek eltűnése újabb két sorral csökkentette a kódunkat, azonban a kapcsos zárójeleket itt is úgy kell használni, mint a **while** vagy az **if** esetén: elhagyható, ha csak egy utasítást kell végrehajtani. Másképp: a ciklusmag itt is egy parancssor vagy kapcsos zárójellel összefogott parancssorok.

A számlálós ciklus különböző nyelvekben

A C++-ban a for-ciklus a számlálós ciklus továbbfejlesztett változata. Gyakorlatilag a for-ciklus nem más, mint while-ciklus összevont megfelelője. C++-ban a két nyelvi eszköz használata között csak egy különbség van:

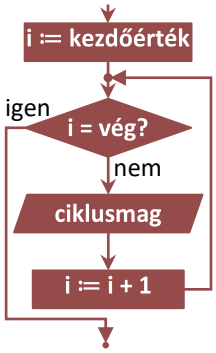
- Ha a számláló létrehozása nem a **while** előtt történik, hanem a **for** első paraméterében, azzal a „láthatóságuk” is különböző lesz.

A gyakorlatban, ha a több while-ciklust írunk egymás után, akkor a számláláshoz elegendő egy számlálót létrehozni, a következő használatkor csak a kezdőértéket állítjuk át és a két ciklus között látható, hogy éppen mennyi a számláló értéke. Ha a számlálót a for-ciklus első paraméterében hozzuk létre, akkor ez a számláló csak a cikluson belül lesz használható. Ilyenkor minden egymásutáni for-ciklushoz használhatjuk ugyanazt a nevet a számláláshoz, de ez nem ugyanaz a változó lesz, mert az egyik megszűnik, mire a következő létrejön. Ha a cikluson kívül is szükségünk van a számláló értékére, akkor a változót a ciklus előtt kell létrehozni – *deklarál*ni – és a for első paraméterében csak kezdőértéket adunk neki, azaz *inicializál*juk.

```
 6.  int i;
 7.  for (i = 1; i <= 100; i++)
 8.  {
 9.    cout << i  << ". A folyosón sétálunk!", << endl;
10.  }
```

Így a feladat első megoldásával teljesen azonos tulajdonságú a kódunk, de csak egy sorral rövidebb. Mondhatnánk, hogy nem a sorok száma, hanem a beírt karakterek száma számít. A „for” rövidebb a „while”-nál. Nézzük meg alaposan, mi hova került: hány karaktert kell begépelni a while-ciklusba és mennyit a for-ciklusba? Az eredmény: a for-ciklusban 1 db tabulátorral van kevesebb. Látható, hogy ezért nem érdemes összevont alakot használni. A for-ciklus népszerűségének (van, aki mindig ezt használja) más oka van.

A „következő órára írd le százszor …” feladatban nemcsak az a nehéz, hogy sokat kell körmölni, hanem az is, hogy fejben tartsuk, éppen hányadiknál tartunk. A szavazatok számlálását is többször ellenőrzik, mert könnyű elrontani. Szerencsére a számítógép sokkal jobban bírja a monotonitást, nem szokta elfelejteni, hogy éppen hol tart, nem ugrik a negyven után a hetvenegyre (kivéve, ha erre utasítjuk). De a programozókról ugyanez nem mondható el. A while-ciklus kódolásakor a leggyakoribb hiba, hogy mire a ciklusmag végére ér a programozó, addigra elfelejtheti, hogy még a ciklusszámlálót is növelni kellene. Az eredmény: végtelen ciklus. Észrevenni is nehéz ezt a hibát, mert a kód a programozó fejében megszületett, csak a kivitelezés maradt el. Ezért a for-ciklus legnagyobb előnye az, hogy a számlálást ott írhatjuk le, ahol gondolunk rá. Valójában egy olyan feltételes ciklus, amely egyes helyzetekben jobban kifejezi a gondolatunkat.



Folyamatábrán a számlálós ciklusnak nincs egyedi formája. Elöltesztelős ciklusként lehet megjeleníteni, ami egyúttal jól mutatja, hogy miből származik.

A számlálós ciklus – több programozási nyelvben – speciálisabb. Azért érdemes megismerkedni ezekkel is, mert mondatszerű leírásban és blokknyelvekben használhatjuk és sok esetben egyszerűbbé teszi az algoritmus leírását.

Számláló változó nélkül:

```
ismételd 100-szor:
    ki: „…séta”
ciklus vége
```

Számláló változóval:

```
ciklus szám = 1-től 100-ig:
    ki: szám
ciklus vége
```

Számlálóval és lépésközzel:

```
ciklus szám = 1-től 100-ig lépésköz 1:
    ki: szám
ciklus vége
```

Túltenni Gausson

A kis Gauss, amikor még nem volt nagy matematikus, az anekdota szerint egész osztályával együtt azt a feladatot kapta a tanárától, hogy adja össze a számokat egytől százig. A tanár közben nekiállt valami más munkának, de a kis Gauss két perc múlva szólt, hogy készen van, és az eredmény 5050. Rájött, hogy 1 + 100 = 101, 2 + 99 = 101 és így tovább. 50-szer 101 pedig 5050. Ha már van számítógépünk, illendő a két percnél jobb eredményt hoznunk, méghozzá Gauss felismerésének kihasználása nélkül.

A megoldás ötlete, hogy a számlálás mellett szükségünk van még egy változóra, amiben az összeget folyamatosan – mindig az újabb értékeket hozzáadva az éppen aktuális értékhez – számítjuk ki a végeredményt. A kivitelezést tekintve, az eddig megismert nyelvi elemekkel, azonos változóneveket használva, nem vizsgálva, a tördelést és a kód igazítását, száznál többféle megoldás létezik. Itt ezek közül kettő látható. Próbáljunk ki több változatot, keressük meg azt a kódot, amely számunkra a legkifejezőbb!

```
 6.  int szamlalo = 0;
 7.  int osszeg = 0;
 8.  while (szamlalo < 100)
 9.  {
10.    szamlalo = szamlalo + 1;
11.    osszeg = osszeg + szamlalo;
12.  }
13.  cout << "Összesen: " << osszeg << endl;
14.
```

```
 6.  int osszeg = 0;
 7.  for (int i = 1; i <= 100; i++)
 8.    osszeg += i;
 9.  cout << "Összesen: " << osszeg << endl;
10.
```

A logikai típus

Vegyük elő a számkitalálós programunkat és csupaszítsuk le annyira, hogy csak a jó válaszra reagáljon! Tervezzük át úgy, hogy kérdezzen addig, amíg ki nem találjuk a számot! A mondatszerű leírásba nem írjuk bele az véletlengenerátort létrehozó utasítást:

```
gondolt_szám := 1 és 6 közötti véletlenszám
kitalálta = hamis
ciklus amíg ki nem találta:
    be: tipp
    ha tipp = gondolt_szám:
       kitalálta = igaz
ciklus vége
```

A mondatszerű leírásban bevezettünk egy változót, amiben igaz vagy hamis érték tárolható. A változó kezdeti értéke hamis, és akkor állítjuk át igazra, ha a tipp jó volt. A változót azért inicializáltuk hamis értékkel, hogy a ciklusba legalább egyszer belépjünk.

Azok a változók, amelyek igaz (**true**) és hamis (**false**) értéket vehetnek fel, úgynevezett logikai típusú változók (a C++-ban a típus neve **bool**, George Boole matematikus után, aki efféle problémákkal való foglalatosságáról vált híressé). Mindez kódként így néz ki:

```
 8.  srand(time(0));
 9.  int gondolt_szam = rand() % 6 + 1;
10.  bool kitalalta = false;
11.  while (!kitalalta)
12.  {
13.    cout << "Szerinted? ";
14.    int tipp;
15.    cin >> tipp;
16.    if (tipp == gondolt_szam)
17.      kitalalta = true;
18.  }
19.  cout << "Végre!" << endl;
```

false és true kis kezdőbetűvel

Vagy: while(kitalalta == false)
A ! olvasata NEM.
Ez a szokásos írásmód.

Beljebb, mert csak az if teljesülése esetén érvényes.

Kódunkban érdemes megfigyelni a változók élettartamát, másnéven hatókörét. A `tipp` változót a ciklusmagban *deklarál*tuk (hoztuk létre) és *inicializál*tuk (megadtuk, hogy mennyi legyen az értéke). Ez azt jelenti, hogy minden ciklusléptésben újra és újra létrehozzuk a `tipp` változót.

Másrészt, a változóknak egyedinek kell lennie, azaz nem lehet két `tipp` nevű változónk. A kód mégis helyes, mert ami a kapcsos zárójelek között keletkezik, az a záró kapcsos zárójelnél megszűnik. A zárójelek nemcsak egybefogják az utasításokat, de programblokkot is képeznek. A belső – lokális – változók a blokk végéig léteznek. Emiatt megfontolandó, hogy hol deklarálunk egy változót, hol adjuk meg a típusát és a nevét. Általában ott célszerű megadni, ahol először használni szeretnénk, de ha – ez egy blokkon belül történne és – a blokk végrehajtása után is használnánk, akkor a deklarálásnak a blokk előtt kell megtörténnie, az értékét a blokkban csak felülírjuk. Ez a teendő akkor, ha a programunkban nemcsak azt írjuk ki, a végén, hogy „Végre!”, hanem a legutolsó beírt értéket is.

```
 8.  srand(time(0));
 9.  int gondolt_szam = rand() % 6 + 1;
10.  bool kitalalta = false;
11.  int tipp;
12.  while (!kitalalta)
13.  {
14.    cout << "Szerinted? " ;
15.    cin >> tipp;
16.    if (tipp == gondolt_szam)
17.      kitalalta = true;
18.  }
19.  cout << "Ez az: " << tipp << "!" << endl;
```

Előre deklarált tipp
Itt nem kell inicializálni, de illene:
int tipp = 0;

Nem deklarálhatjuk újra:

Összetett ciklusfeltétel

A program türelmes, így a felhasználó a végtelenségig próbálkozhat. Írjuk át a programunkat úgy, hogy csak három próbálkozást engedjen meg! Számolnunk kell a próbálkozásokat, de nem elég, ha a ciklus feltételeként csak azt adjuk meg, hogy még nem használtuk el mindegyiket. Arra is figyelnünk kell, ha közben a felhasználó kitalálta a gondolt számot. Figyelni kell a próbálkozások számát **és** a tippelt értéket is. Szerencsére a **`while`**, pont ugyanúgy, mint az **`if`**, képes összetett feltételek kezelésére.

1. Gondoskodjunk az elhasznált lehetőségek nyilvántartásáról (számlálásáról)?
2. Fogalmazzuk meg a **`while`** feltételét!

A teljes kód a következő:

```
 8.  srand(time(0));
 9.  int gondolt_szam = rand() % 6 + 1;
10.  bool kitalalta = false;
11.  int szamlalo = 0;
12.  while (!kitalalta && szamlalo < 3)
13.  {
14.    cout << "Szerinted? ";
15.    int tipp; cin >> tipp;
16.    if (tipp == gondolt_szam)
17.      kitalalta = true;
18.    szamlalo++;
19.  }
20.  cout << "Vége" ;
```

A `szamlalo` változó utal arra, hogy talán számlálós ciklussal is megoldhatjuk a feladatot. Igazi, hagyományos értelemben vett számlálós ciklusban nem tudunk feltételt megadni, de a C++ –

és nagyon sok más nyelv is – a for-ciklus középső részében tetszőleges logikai kifejezést megenged, így rövidíthetjük a kódot:

```
 8.  srand(time(0));
 9.  int gondolt_szam = rand() % 6 + 1;
10.  bool kitalalta = false;
11.  for (int i = 0; !kitalalta && i < 3; i++)
12.  {
13.    cout << "Szerinted? " ;
14.    int tipp; cin >> tipp;
15.    if (tipp == gondolt_szam)
16.      kitalalta = true;
21.  }
17.  cout << "Vége." << endl;
```

3. Helyezzünk el ismét olyan programsorokat a kódban, amelyek a felhasználó dicséretéért, ugratásáért felelnek! Több helyre is elhelyezhetőek, mindnek megvannak az előnyei és hátrányai. Hasonlítsunk össze néhány lehetséges megoldást!
4. Szeretnénk megoldani, ha a felhasználó a harmadik lehetőségre már majdnem kitalálta a megoldást (csak egyet tévedett), akkor kapjon még egy esélyt. Ahogy programozáskor mindig, ismét több helyes megoldás lehetséges – valósítsuk meg a nekünk legjobban tetszőt!

Ciklusok és véletlenek

Feladatok

1. Ciklussal meg tudunk oldani egyenleteket az egész számok halmazán – próbálgatással.
   a) Oldjuk meg a $3x + 2 = 59$ egyenletet a pozitív egészek halmazán!

```
6.  int x = 1;
7.  while (3*x + 2 != 59)
8.    x = x + 1;
9.  cout << "A megoldás: " << x << endl;
```

Érdekesség, hogy for-ciklussal is megoldható a feladat, de a ciklus magnélküli lesz és a sorok száma sem lesz kevesebb, mert az x-et előre létre kell hozni, hogy a kiíráskor is létezzen. Ilyenkor a **for** ciklusfejének első paramétereként az x-nek csak kezdőértéket adunk, nincs **int**.

```
6.  int x; //Itt deklaráljuk, hogy a 9. sorban is létezzen
7.  for (x = 1; 3*x + 2 != 59; x++)
8.  ; //Jelezzük, hogy nincs mag. Másik jelölés: {}
9.  cout << "A megoldás: " << x << endl;
```

   b) Oldjuk meg a $6x^2 + 3x + 8 = 767$ egyenletet az egész számok halmazán! Honnan érdemes indítani a próbálgatást?
   c) Diophantosz ókori görög matematikus verses sírfelirata több fordításban fellelhető az interneten. A felirat alapján fogalmazzuk meg az egyenletet, és írjunk programot, ami megoldja! Hány évesen halt meg Diophantosz?
   d) Ilyen módszerrel csak akkor oldható meg biztosan az egyenlet, ha az eredmény egész szám. Miért?

e) Ha az egyenletnek nincs egész gyöke (például $6x^2 + 3x + 8 = 768$), akkor végtelen ciklusba kerül a programunk. Miként biztosítható, hogy ne lépjen végtelen ciklusba a program, és ha már esélytelen, hogy talál megoldást, ne próbálkozzon tovább? Gondolkozzunk összetett feltételben!

2. Írjunk pénzfeldobás-szimulátort!
   a) A gép írja ki, hogy fejet vagy írást „dobott”! A rand() egy egész számot ad, aminek az értéke legfeljebb RAND_MAX. A randomszám és ennek az aránya [0; 1[közötti szám. Ha nem pontosan felezzük a tartományt, akkor hamis érmét szimulálunk. (A számításnál ügyeljünk arra, hogy részekre osztást végezzünk.)

```
 8.  srand(time(0));
 9.  int dobas = rand() % 2;
10.  if (dobas == 1)
11.    cout << "fej" << endl;
12.  else
13.    cout << "írás"<< endl;
```

```
 8.  srand(time(0));
 9.  double dobas= (double)rand()/RAND_MAX
10.  if (dobas < 0.5)
11.    cout << "fej" << endl;
12.  else
13.    cout << "írás" << endl;
```

   b) Készítsünk statisztikát! Egymillió feldobásból mennyi lesz fej és mennyi írás?
   c) Írjuk át a programot úgy, hogy kockadobásokat számoljon (mennyi lesz 1-es… 6-os)!
   d) Készítsünk cinkelt kockát! A hatos jöjjön ki kétszer akkora eséllyel, mint a többi szám!
3. Írjunk randiszimulátort!
   a) A számítógép kérdezze azt, hogy „Szeretsz?”, amíg azt nem válaszoljuk, hogy „Nagyon!”, vagy azt, hogy „Jobban, mint a kókuszgolyót!”.
   b) A számítógép legyen durcás: legfeljebb három kérdés után zavarjon el bennünket, ha nem adjuk meg a „helyes” válaszok valamelyikét!
   c) A számítógép legyen szeszélyes: zavarjon el bennünket 2–4 „rossz” válasz után véletlenszerűen!

Ciklusok oda-vissza, illetve egymásba ágyazva

Feladatok

1. Írjuk ki a számokat csökkenő sorrendben 100 és –100 között egymás alá! Oldjuk meg úgy is a feladatot, hogy a számokat egymás mellé, szóközzel elválasztva írjuk ki!
2. Írjuk ki (két) ciklussal, hogy „12345678987654321”!
   a) Hány helyen kell módosítanunk a programot, hogy ne kilencig, hanem kilencmillió-kilencszázkilencvenkilencezer-kilencszázkilencvenkilencig írja a számokat?
   b) Hogyan változik a program futásának sebessége, ha nem íratjuk ki a számokat, csak elszámoltatunk oda-vissza? (// jellel tegyük kommentté a kiírást végző sorokat.)
3. Írjunk egymás mellé 10 csillagot („*”) úgy, hogy a programkódban csak egyetlen csillag karakter legyen!
4. Csillagok több sorban
   a) Írjunk egymás alá öt csillagsort! Ötlet: tegyük az előző feladat ciklusát egy másik ciklus belsejébe. Az áttekinthetőség érdekében, ott is írjuk ki a kapcsos zárójelet, ahova nem feltétlenül szükséges.

while-ciklusokkal

```
 6.  int sorsz = 0;
 7.  while (sorsz < 5)
 8.  {
 9.    int csillsz = 0;
10.    while (csillsz < 10)
11.    {
12.      cout << '*';
13.      csillsz++;
14.    }
15.    cout << endl;
16.    sorsz++;
17.  }
```

for-ciklusokkal

```
 6.  for (int si = 0; si < 5); si++)
 7.  {
 8.    for (int csi = 0; csi < 10); csi++)
 9.    {
10.      cout << '*' ;
11.    }
12.    cout << endl;
13.    }
```

Figyeljük meg mindkét megoldásban:

- A két ciklusnak két külön számlálója van.
- Minden sor elején a csillagok számlálását újrakezdjük.

b) Helyezzünk el breakpoint-ot (piros pontot) a csillagot kiíró sor elé! Így Debug/Continue (▶ vagy **F8**) módban futtatva ezen a ponton megáll a programunk, amit ugyanígy (kattintva vagy F8) folytathatunk. Figyeljük meg a Watches ablakban (Debug\Debugging windows\Watches) az egyes változók értékét: hogyan változnak két megállítás között!

c) Az előző program megoldásának egyetlen helyen való megváltoztatásával alakítsuk háromszöggé a program kimenetét! (Ötlet: minden sorba annyi csillagot kell kiírni, ahányadik…)

d) Ha elkészültünk, írjuk át úgy a programot, hogy minden sorban csak az utolsó csillag jelenjen meg!

e) …minden sorban az első és az utolsó csillag jelenjen meg!

5. Írjunk szorzótáblát a kicsiknek!

Minta kimenet a 4.b feladathoz:

```
*
**
***
****
*****
```

Minta kimenet az 5. feladathoz:

```
1 * 1 = 1
1 * 2 = 2
…
6 * 6 = 36
6 * 7 = 42
…
10 * 9 = 90
10 * 10 = 100
```

Összetartozó adatok kezelése

Adatok sorozata

A programjainkban az adatokat változókban tároljuk, és eddig öt változótípust, adattípust ismerünk:

- szöveg típust (**string**);
- az egész szám típusokat (integer, például **int**);
- a valós, tizedestörtként megjelenő számokat (lebegőpontos szám, például **double**)
- karaktereket (**char**)
- és a logikai értékeket (**bool**).

Ezek közül egyik sem jó akkor, amikor több egymáshoz tartozó adatot szeretnénk tárolni, kiírni, műveletet végezni velük. Például tárolni szeretnénk a csoportunkba járók neveit, hogy ki tudjuk sorsolni, ki lesz a következő felelő. Vagy, jó lenne tárolni minden osztály létszámát, hogy meg tudjuk mondani azt is, hogy összesen hány diák jár az iskolába és azt is, hogy melyik osztályokba járnak az átlagosnál többen. A szám-kitalálós programban a végén jó lenne kiírni a tippeket. Esetleg írhatnánk egy szólánc programot, de a gépnek meg kellene jegyeznie a már korábban beírt szavakat.

Az ötletekre minden programozási nyelvben van megoldás, nem csak a szöveg (karaktersorozat) tartalmazhat több adatot. Bármilyen adatokból képezhetünk összetett adattípusokat. A lehetőségek közül most két olyan összetett adattípust ismerünk meg, amelyekben tetszőleges, de azonos típusú adatok sorozatát tárolhatjuk.

Bár a két adattípusnak eltérőek az előnyei és hátrányai, a programozással ismerkedőknek elég az egyiket ismerni. Az, hogy melyik jobb, melyik könnyebb, ízléstől és a konkrét feladattól függ. Az egyiket alaposan meg kell ismerni, meg kell érteni, hogy hogyan tudjuk használni.

A tömb és használata

Több egyforma dolog együttesének sokféle elnevezése van: (szám)sorozat, (karakter)lánc, (fekete) lista és ide tartozik az egydimenziós tömb is. Több egyforma dolog együttesét elképzelhetjük táblázatban, mátrixban, kétdimenziós tömbben. A tömb – akárhány dimenziós – az adatokat elrendezve tartalmazza. A programunkban létrehozandó egydimenziós tömbhöz a memóriában szomszédos memóriaterületeket használunk. Ezt elképzelhetjük úgy, hogy egymás mellé vagy úgy is, hogy egymás alá/fölé tesszük az egyforma adatokat. Ha megadjuk, hogy milyen és mennyi adatból szeretnénk tömböt létrehozni, akkor ennek megfelelő méretű helyet keres a programunk, és – mintha rekeszes szekrény lenne – kijelöli az egyes adatok helyét. Mivel minden adatunknak előkészítjük a helyét, ezután bármelyiknek, akár tetszőleges sorrendben is, megadhatjuk az értékét, később tudjuk módosítani. Problémát csak az jelenthet, ha több adatunk van, mint amennyi helyet kezdetben létrehoztunk. Toldozni, foldozni, bővíteni a már lefoglalt területet nem lehet, ha a lefoglalt részen túl szeretnénk valamit tárolni, akkor – programozási nyelvtől függően – hibajelzéssel elszáll a programunk vagy esélyes egy kékhalál, amikor az adattal egy másik program adatát írja felül a program.

Az egydimenziós tömb a C++ nyelv legegyszerűbb összetett adattípusa. Ezért a létrehozása és használata nem túl bonyolult. Az egyetlen nehézséget az okozhatja, hogy mindig a szögletes zárójelet kell használni, aminek így – a környezetétől függően – kétféle szerepe van:

- `[]` Amikor létrehozzuk a változót és memóriaterületet foglalunk neki, akkor egy speciális konstruktort használunk, [] jel között adjuk meg, hogy hány darab adatra kell a foglalás.
- `[]` Amikor használjuk a változót – ami az összes adat egy néven –, akkor a [] közötti szám azt mutatja meg, hogy a lefoglalt terület kezdetétől hány adatnyira van az éppen kiválasztott hely, azaz – 0-tól számozva – hányadik adatot használjuk.

Feladatok

1. Tároljuk kacatjainkat tömbben! Készüljünk fel arra, hogy akár 1000 db kacatunk is lehet, de csak az első három helyre írjunk be adatot!

```
6.  string kacat[1000];
7.  kacat[0] = "csat";
8.  kacat[1] = "gombolyag";
9.  kacat[2] = "vonatjegy";
```

string-ek tömbje

Hely 1000 db stringnek

A 2-es indexű 3. adat, egy string

2. Tároljuk el a hét napjainak rövidített neveit egy tömbben! Most tudjuk, hogy 7 db stringet kell tárolnunk és pontosan ismerjük a megnevezéseket is. Ilyenkor – a létrehozás és egyenként értékadás helyett, – kapcsos zárójelekkel „egyesítve” – konstruktorral – megadhatjuk a szavak listáját:

```
6.  string napnev[7]{"H", "K", "Sze", "Cs", "P", "Szo", "V"};
7.  cout << napnev[6] << endl; // Ki: V
```

7 string van benne.

3. Tároljuk el az egyjegyű prímszámokat, majd írjuk ki őket szóközökkel elválasztva!

```
6.  int primek1[4] {2, 3, 5, 7};
7.  for (int i = 0; i < 4; i++)
8.    cout << primek1[i] << ' ' ;
```

A tömb NEM ISMERI a méretét.

A ciklusszámlálót használhatjuk indexnek.

Gyakran elő fog fordulni, hogy adatokat egy sorban, szóközzel elválasztva kap a programunk. C++-ban ez éppen olyan jó, mint az ENTERrel lenne elválasztva.

Kiegészítés: Az extractor lelkivilága

A szóközön kívül gyakori soron belüli elválasztójel a vessző, a pontosvessző és a tabulátor. Az extractor (>>) a „nem nyomtatható” (angolul white space) karaktereket egyformán kezeli, így a szóköz, a tabulátor és az enter az éppen olvasott adat végét jelzi, ugyanakkor ezekből bármilyen kombinációban akárhány van, azt a következő adat beolvasásakor eldobja (kiszűri). Emiatt a programozó szempontjából mindegy, hogy ezek közül melyik választja el az adatokat.

Kicsit több odafigyelést igényel a vesszővel, pontosvesszővel vagy egyéb karakterrel elválasztott adat. Ezeket egy string beolvasásakor a >> az adat részének tekinti, de a getline() eljárásban megadható az elválasztó jel (delimitátor), így ezek a szövegek is beolvashatók. Más adattípusnál a beolvasott adat értelmezése megszakad az első nem megfelelő karakternél – a vesszőnél, pontosvesszőnél… – a változóba az addig feldolgozott karakterek alapján kerül be az érték. Problémát a következő adat kiolvasása jelentheti. Ami nem white space és nem is számjegy, az egy szám beolvasását blokkolja. Ezért a következő(!) beolvasások sikertelenek lesznek. A megoldás: az elválasztójelet vegyük ki egy char típusú változóba. Ezzel minden olyan bemenetet be tudunk olvasni, amiben két adat között a white space(ek)en kívül egy (1 db) másmilyen karakter van. Ugyanakkor, ha bárhol hiányzik a két adat között ez a karakter, akkor a második adat első (értékes) karaktere – praktikusan egy számjegy vagy előjel – a kukában végzi.

4. Írjunk programot, ami bekéri a heti lottószámokat egy sorban, szóközökkel elválasztva, majd ezt egész számokként eltárolja! (A feladat folytatása lehet, hogy bekéri a megjátszott számainkat is és megmondja, hány találatot értünk el. De most még csak beolvassuk és eltároljuk a számokat.)

```
1.  cout << "Mik a heti lottószámok? ";
2.  int lotto[5];
3.  for (int i = 0; i < 5; i++)
4.    cin >> lotto[i];
```

A kód egyszerű, de teszteljük, hogy mennyire hibatűrő!

Láthattuk, hogy a tömb elemeit egyenként tudjuk használni, egyenként tudjuk kiírni és beolvasni is. Figyelni kell a tömb elemeinek használatára – és kiíráskor is –, ha a tömb nagyobb méretű, mint ahány adatot éppen tárolunk benne. Ezért, ha a tömböt „üresen” hozzuk létre, akkor célszerű a tényleg felhasznált elemek számának tárolására egy változót mellé tenni és folyamatosan jegyezni, hogy éppen meddig van a tömb feltöltve adatokkal.

Ha az adatokat össze-vissza írjuk be, akkor célszerű a használat előtt az összes tömbelemnek kezdőértéket adni. Programozási nyelvenként eltérő, hogy melyik adattípusnak van automatikusan értelmes kezdőértéke, illetve melyik kapja értékül az adott memóriahelyen talált szemetet. A C++ a `string` típusú adatoknak a `""` kezdőértéket adja, de az `int`, `double`, `bool` (és a többi kék típus) tömbelemek kezdetben memóriaszemetet tartalmaznak.

5. Alakítsuk át a kacatokat tároló programunkat: kezdetben minden kacat legyen `"-"` és számoljuk, hogy épp hány kacatunk van beírva! Végül írjuk ki a tárolt kacatok listáját egymás alatti sorokba!

```
6.  string kacat[1000];
7.  int kacatdb = 0;
8.  for (int i = 0; i < kacat.size(); i++)
9.    kacat[i] = "-"; //Mindegyik "-". Lehet " " vagy "" (üres) is.
10. kacat[0] = "csat"; //Átnevezzük "-"-ről "csatt"-ra
11. kacatdb++; //1 lett
12. kacat[kacatdb] = "gombolyag";
13. kacatdb++; //2 lett
14. kacat[kacatdb] = "vonatjegy";
15. kacatdb++; //3 lett
16. for (int i = 0; i < kacatdb; i++)
17.   cout << kacat[i] << endl;
```

foglalt mennyiség

ahány kacat tényleg van

6. Tegyünk a kód 6., 7., és 10–15. sora elé breakpoint-ot, futtassuk a programot Debug módban! Figyeljük a Watches ablakban a változók és a tömb értékeinek módosulását!

Kiegészítés: 2D

Eddig egydimenziós tömbökről volt szó, de – főleg a legkülönbözőbb helyzetekben kapott – táblázatok kapcsán, a tömböt is „legalább” kétdimenziósnak képzeljük el. C++-ban az egydimenzióhoz képest az adatok két- vagy többdimenziós tárolása nem bonyolult. Persze a használatához – mondjuk a négydimenziós térben tájékozódáshoz – már komoly absztrakciós képesség szükséges.

A kétdimenziós tömb létrehozása és használata az egydimenzióshoz képest annyiban tér el, hogy minden dimenziónak külön [] kell. Így tömbökből képezünk tömböket.

```
int jegyek [13][10];
jegyek[7][5] = 5;
```

A háttérben a programozási nyelvek a többdimenziós tömböket is egydimenziósként tárolják, méghozzá sorfolytonosan, azaz a példában a `jegyek[0][9]` elem után a `jegyek[1][0]` következik. A gyorsabb memóriaműveletek érdekében célszerű az adatok írásánál, sorban olvasásánál ezt a sorrendet betartani.

Hacker feladatok: Adjunk értéket a `jegyek[0][15]` elemnek! Van a 0. sornak 15. eleme? Történt túlindexelés? Mennyi x értéke, ha a `jegyek[7][5]` értékét `jegyek[0][x]` formában szeretnénk megtudni? Ki lehet írni így is a változó értékét?

A lista és használata

Amikor több, valamilyen szempontból egyforma dolgot szeretnénk felsorolni, azt mondjuk, listát készítünk. Ilyen például a bevásárló lista, a hiányzók listája, a „todo list”. Ezeknek a felsorolásoknak a közös jellemzője, hogy nem tudjuk előre, hogy a végén hány elemet teszünk bele. Ez a mindennapok során nem szokott komoly problémát jelenteni: ha betelt a bevásárlólistánk cetlije, akkor: körbe, a szélén még elfér néhány tétel vagy megfordítjuk és a hátoldalán folytatjuk vagy keresünk egy másik cetlit. Amikor egy programozási nyelven szeretnénk listát készíteni, akkor a fordítóprogramnak hasonló problémákkal kell megküzdenie. Olyan programkódot kell készíteni, amelyik akárhány adatot el tud tárolni, eközben nem ír felül más célra lefoglalt területeket és számon tudja tartani, az egyes elemek sorrendjét. Az is előny, ha bármelyik elem gyorsan elérhető, azaz nem kell sorba vennie az elemeket ahhoz, hogy az utolsó előttit megtalálja. A feladat annyira bonyolult, hogy egyes programozási nyelveken többféle adattípust is elkészítettek, mert a „gyorsan végrehajtható algoritmusok írhatók rá”, a „kevés memória is elég neki” és a „könnyen kódolható” hármasfeltételből szinte lehetetlen egyszerre mindet teljesíteni.

Most egy olyan adattípussal fogunk megismerkedni, ami

- egyforma adattípusokból többet tartalmaz,
- tetszőlegesen bővíthető,
- a benne lévő adatok gyorsan, könnyen elérhetők,
- (a többi hasonló típushoz képest) könnyen használható kódjaink írásához.

A második feltétel kivételével, egy fixen lefoglalt adatterület, ahol minden adatnak előre kijelölt helye van, éppen megfelelne. Ez lenne a tömb adattípus. De most többet akarunk … A tetszőleges bővíthetőséget lehet úgy biztosítani, hogy kezdetben lefoglalunk egy adott méretet (bevásárló lista egy cetlije); amikor ez betelik, akkor keresünk egy nagyobb helyet, ahova átvisszük a korábbi adatokat és ott már lehet további adatokat is írni. Ha ez is betelik, akkor újabb költözés …

Az itt leírt adattípus elméleti neve: *dinamikusan bővülő tömb*, C++-ban a **`vector`** adattípus ilyen tulajdonságú. A leírásból következtethetünk az adattípus gyenge pontjaira is: minden átköltözéshez idő kell. A **`vector`** kettőhatványonként bővíti a tárolási kapacitását. Utána számolhatunk: 8 adat beírása a memóriában körülbelül 15 adatírást jelent, a 9. adat beírása a már bennlévő 8 áthelyezése miatt 9 további adatírás, így ekkor már 24-szer írt adatot a programunk. Amikor nem számít a program futási ideje, akkor ez így is jó. De ha nagymennyiségű adatot szeretnénk bevinni, akkor célszerű a konstruktorban megadni a tervezett kapacitást.

Az adattípus figyelemre méltó tulajdonsága, hogy korlátlanul bővíthető futtatás közben a mérete. Vajon ez mindig jó?

A vector<>

A **vector <>** adattípus használata előtt, a programunk elején be kell vennünk a megfelelő – **vector** – csomagot.

```
1.  #include <iostream>
2.  #include <vector>
```

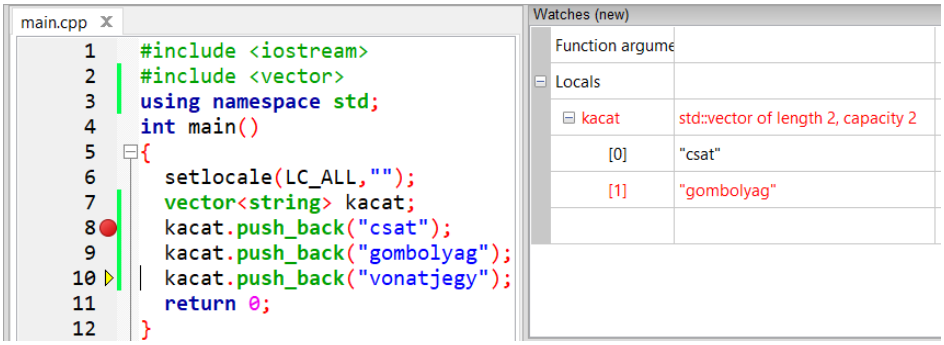
Feladatok

1. Tároljuk a kacatjainkat listában! Tegyünk bele három kacatot! Futtassuk lépésenként (breakpoint F8, F7) a programot, figyeljük meg a kapacitás és az adatok számának a változását!

```
1.  vector<string> kacat;
2.  kacat.push_back("csat");
3.  kacat.push_back("gombolyag");
4.  kacat.push_back("vonatjegy");
```

string-ek listája

A lista végéhez hozzátesz egy adatot



2. Tároljuk el a hét napjainak rövidített neveit egy listában! Most tudjuk, hogy mit szeretnénk tárolni, ezért a lista létrehozása után közvetlenül, kapcsos zárójelekkel „egyesítve” megadhatjuk a szavak tömbjét:

```
7.  vector<string> napnev{"H", "K", "Sze", "Cs", "P", "Szo", "V"};
8.  cout << napnev[6]; // Ki: V
```

3. Ha tudjuk, hogy hány eleme lesz a listánknak, akkor a `reserve()` eljárással le tudunk foglalni ennek megfelelő méretű memóriát, azaz megadhatjuk a kapacitást. Ezzel felgyorsíthatjuk az adatok feltöltését. A módosítás a már meglévő adatokra nincs hatással. Próbáljuk ki a használatát többféle értékkel, 7-nél kisebbel és nagyobbal is. A kiírás sorába tett breakpointtal állítsuk meg programunk futását, figyeljük meg a méret és kapacitás értékeket!

```
9.  vector<string> napnev{"H", "K", "Sze", "Cs", "P", "Szo", "V"};
10. napnev.reserve(7); //7 helyett legyen 10; 5 …!
11. cout << napnev[6]; // Ki: V
```

4. Tároljuk el az egyjegyű prímszámokat, majd írjuk ki őket szóközökkel elválasztva! A végén írjuk ki a lista kapacitását is (`capacity()`)! Ezt követően módosítsuk a programot:
   a) Méretezzük át a listát 7 eleműre a `resize(7)` eljárással, majd írjuk ki ismét az elemeit és a kapacitást!
   b) Bővítsük tovább, 9 eleműre úgy, hogy a kezdőérték 11 legyen! (`resize(9,11)`) Ismételjük meg a kiírást!
   c) Szűkítsük le a méretet 3 elemre és újból írjuk ki az elemeket és a kapacitást!

```
 7.  vector<int> primek1{2, 3, 5, 7};
 8.  for (unsigned i = 0; i < primek1.size(); i++)
 9.    cout << primek1[i] << " ";
10.  cout << "k: " << primek1.capacity() << endl;
11.  primek1.resize(7);
12.  for (unsigned i = 0; i < primek1.size(); i++)
13.    cout << primek1[i] << " ";
14.  cout << "k: " << primek1.capacity() << endl;
15.  primek1.resize(9,11);
16.  for (unsigned i = 0; i < primek1.size(); i++)
17.    cout << primek1[i] << " ";
18.  cout << "k: " << primek1.capacity() << endl;
19.  primek1.resize(3);
20.  for (unsigned i = 0; i < primek1.size(); i++)
21.    cout << primek1[i] << " ";
22.  cout << "k: " << primek1.capacity() << endl;
```

egészek listája

az elemszám (4)

a ciklusszámlálót használhatjuk indexnek

átméretez és feltölt

a kapacitás: ha kell nő, de nem csökken

5. Írjunk programot, ami bekéri a heti lottószámokat egy sorban, szóközökkel elválasztva, majd egész számok listájában eltárolja!

   Mivel tudjuk, hogy 5 db számot fogunk kapni, ekkora méretű listát hozunk létre, 5 db −1-es értékkel. A kapott értékekkel felülírjuk a már meglévő adatokat.

```
 7.  cout << "Mik a heti lottószámok? ";
 8.  vector<int> lotto(5,-1);
 9.  for (int i = 0; i < 5; i++)
10.    cin >> lotto[i];
```

létrehozás és feltöltés

nem hozzáteszi, hanem beírja a változóba

Láthattuk, hogy a lista elemeit egyenként tudjuk használni, egyenként tudjuk kiírni. Ehhez a `[]` szögleteszárójelek között az adat listán belüli sorszámát adjuk meg, ahol az első elem sorszáma 0. A listában nincsenek lyukak, a lista elemszámát a `size()` határozza meg. Mivel a `size()` értéke nem lehet negatív, az eredmény `unsigned` típusú egész szám. Emiatt a ciklusváltozónak is `unsigned` típusúnak kell lennie. Ha a lista létrehozásakor nem adunk meg adatot, akkor a lista elemszáma 0. A lista minden elemének a sorszáma kisebb, mint az elemszám. Ennek ismeretében egy for-ciklussal könnyen kiírhatók a listaelemek. A listák használatakor figyelnünk kell arra, hogy a kapacitás és az elemszám két különböző adat.

6. Egészítsük ki a kacatokat tároló programunkat! Írjuk ki minden lépés után egy-egy sorba a `kacat` lista kapacitását és méretét, ezt követően pedig a tárolt kacatok listáját egymás alatti sorokba!

```
 1.  vector<string> kacat;
 2.  cout << "Hely: " << kacat.capacity() "\tDB: " << kacat.size() << endl;
 3.  kacat.push_back("csat");
 4.  cout << "Hely: " << kacat.capacity() "\tDB: " << kacat.size() << endl;
 5.  kacat.push_back("gombolyag");
 6.  cout << "Hely: " << kacat.capacity() "\tDB: " << kacat.size() << endl;
 7.  kacat.push_back("vonatjegy");
 8.  cout << "Hely: " << kacat.capacity() "\tDB: " << kacat.size() << endl;
 9.  for (unsigned i = 0; i < kacat.size(); i++)
10.    cout << kacat[i] << endl;
```

A szöveg karakterlánc

Miután megismertük több összetartozó adat kezelésének lehetőségét, érdemes egy kicsit átgondolni, hogy mi a helyzet a **string** adattípussal. A **string** karakterek sorozata, karakterlánc, esetleg karakterek tömbje vagy listája. Bár egy adatnak tekintettük, valójában mindvégig tudtuk, hogy több adat együtt alkot egy **string**et, azaz épp úgy összetett, mint a most tanult tömb és lista típus.

Vajon a **string** tömb, vagy lista? Vizsgáljuk meg a tulajdonságait:

- A **string**nek épp úgy **size()** a hossza, mint a **vector<>**-nak.
- A **string**nek használhatjuk a `length()` tulajdonságát is, ami ugyanaz, mint a **size()**, de a **vector<>**-nak nincs ilyen tulajdonsága
- A **string** kapacitása 15 vagy akkora, ahány karakter van benne.
- A **string** karakereit ugyanúgy szögleteszárójelek között *index*eléssel lehet elérni, az első sorszám a 0; ebben mindkét típushoz hasonlít.
- A **string**et nem a **push_back**() függvénnyel bővítjük, hanem a **+** operátorral fűzünk össze vagy a **+=** operátorral fűzünk hozzá. Ráadásul nem csak karaktereket, hanem szövegeket is hozzá (össze) tudunk fűzni.
- A **string** karaktereit egyenként meg tudjuk nézni, ki tudjuk írni, másik szöveghez hozzá tudjuk fűzni és a szövegen belül tudjuk módosítani is, pont úgy, mint egy karaktertömb vagy karakterekből álló lista elemeit.
- A **string**nek más – szövegkezelő – függvényei vannak, ami egy tetszőleges adattípus tárolására alkalmas tömb vagy lista esetén bonyolultabb. Például az s **string** függvénye az `s.substr()` és az `s.insert()`, eljárása az `s.erase()`.
- Két **string**et össze lehet hasonlítani ábécé szerinti rendezés alapján. Lehet az egyik kisebb – az ábécében előrébb – mint a másik. Két szöveg egyenlőségét is vizsgálhatjuk.

Összességében, a **string** is összetett adat, olyan, mint a tömb vagy a lista, de sem nem tömb, sem nem lista.

Mivel az adatsorozatok elemei azonos típusúak, egy **string**-ből előállítható a karaktereinek tömbje és listája is, és ezekből előállítható **string**. Becsapós, hogy a stringek `c_str()` függvényének eredménye karaktertömb, mert elemei nem módosíthatóak.

```
string szo = "ati";
char kartomb[255];
int hossz = 0;
for (unsigned i =0; i < szo.size(); i++)
  kartomb[i] = szo[i];
hossz = szo.size();
kartomb[1] = 'k';
string tszo = "";
for (int i =0; i < hossz; i++)
  tszo += kartomb[i];
vector<char> karlist;
for (unsigned i =0; i < szo.size(); i++)
  karlist.push_back(szo[i]);
karlist[1] = 'l';
string lszo;
for (unsigned i =0; i < karlist.size(); i++)
  lszo += karlist[i];
cout << szo + " " + tszo + " " + lszo << endl;
```

Bekért adatok ellenőrzése (do-while-ciklus)

A kacatos listát a felhasználó saját kacatjaival töltjük fel. Minthogy senkinek sincs csak egy kacatja, ciklust szervezünk a feladatra. A kacatokat egyesével kérdezzük meg és mindegyiket betesszük az adatsorozatunkba. De honnan fogjuk tudni, hogy befejezhetjük-e, hogy nincs több bekérendő kacat? A megoldás kétféle lehet.

1. A kacatok nevének bekérése előtt megkérdezzük a felhasználót, hogy hány kacatot szeretne megadni.
2. A kacatok nevének bekérése előtt megmondjuk a felhasználónak, hogy hogyan jelezze, hogy befejezte a beírást. Például, írja be azt, hogy „elfogyott”.

Az első esetben a megadott számnak megfelelő méretű tömböt hozunk létre, vagy létrehozunk egy, a megadott értéknek megfelelő kapacitású listát. Ezután, mivel ismerjük a beírni kívánt adatok számát, praktikusan for-ciklussal (de a while-ciklus is megfelel) be tudjuk kérni az adatokat.

A második esetben nem tudjuk, hány elemünk lesz. Ha tömböt szeretnénk használni, akkor olyan nagy méretet kell megadni a kódunkban, ami biztosan elég lenne. Mennyi legyen? Becsüljük meg: a beírt adatok a program bezárásával elvesznek, 8 órán át percenként 30 adat … Kerekítsünk felfelé, legyen 20 000. De lehet, hogy valami gép fogja beírni, nem ember … Ezért jobb megoldás, ha beírunk a program elejére egy értelmesnek tűnő értéket (pl. 10), ekkora méretű tömbbel dolgozunk. A programba beleírjuk azt is, hogy ha elérte a határt a kacatok száma, akkor fejezze be az adatok bekérését, továbbá kérjen elnézést a kapacitáshiány miatt.

Ha listával oldjuk meg a feladatot, akkor elvileg nem kell foglalkoznunk az adatok maximális számával. De elképzelhető – ha például a felhasználó egy gép –, hogy addig tölti a listánkba az adatokat, amíg el nem fogy a gépünk memóriája. Erre nem igazán tudunk felkészülni, de programunk jövője szempontjából nem is reális.

Bármelyik megoldást választjuk, minden beírt kacatot meg kell vizsgálnunk mielőtt beillesztjük az adatsorozat végére, mert ha a beírt adat az, hogy „elfogyott”, akkor nem kell beilleszteni az adatsorozatunkba, de be kell fejeznünk az adatok bekérését. Az adatbekérés ciklusának a vége eszerint egy feltételtől függ: a beírt adat elfogyott. Másként: a kacat nevét akkor kérjük, amíg az előzőleg beírt adat nem lesz az „elfogyott”. (Ezt a feltételt kell esetleg kiegészíteni azzal, hogy „és a beírt kacatok száma nem érte el a megadott határértéket”.)

Van ezzel a megoldással egy kis nehézség, rögtön az első adat beírásakor, ugyanis az első adat előtt nincs adat, amit ellenőrizhetnénk. Ezért a megoldásunkban „előre-olvasás” szükséges. Ez azt jelenti, hogy az első adatot beolvassuk és csak ezután jön a feltételes ciklus.

Az adatbekérést követően érdemes kiírni az összes eltárolt adatot, ezzel „visszaigazoljuk” az adatok rögzítését. Az adatok közé vesszőt és szóközt írjunk, a végén legyen pont.

Feladatok

1. Írjuk meg a programunk mondatszerű algoritmusát, foglaljuk össze ebben eddigi gondolatainkat!

```
kacattar létrehozása
ki: "vége: elfogyott"
ki: "Kérek egy kacatot"
be: kacat
ciklus amíg kacat != "elfogyott":
    kacattar.betesz(kacat)
    ki:"Kérek egy kacatot"
    be: kacat
ciklus vége
ki: "feljegyzett kacatok:"
ciklus 0-tól kacattar.elemszama-1-ig:
    ki: kacattar.eleme + ", "
ciklus vége
ki: kacattar.eleme + "."
```

Ha figyelnünk kell az adatok számára, akkor több helyen kiegészül az adatok bekérése:

```
maxkacat ≔ 10
kacattar létrehozása
darab ≔ 0
…
ciklus amíg kacat != "elfogyott" ÉS darab < maxkacat:
    kacattar.betesz(kacat)
    darab++
…
ciklus vége
ki: "feljegyzett kacatok:"
…
ki: kacattar.eleme + "."
ha darab = maxkacat:
    ki: "Sajnos, csak ennyit tudok eltárolni."
```

2. Írjuk meg a programot!

Az ellenőrzéshez egy tömbös megoldás:

```
 6.  const int maxkacat = 10;
 7.  string kacattar[maxkacat];
 8.  int darab = 0;
 9.  cout << "Vége: \"elfogyott\"" << endl; //\" kiírja az "-t
10.  cout << "Kérek egy kacatot " ;
11.  string kacat; cin >> kacat;
12.  while (kacat != "elfogyott" && darab < maxkacat)
13.  {
14.    kacattar[darab] = kacat;
15.    darab++;
16.    cout << "Kérek egy kacatot ";
17.    cin >> kacat;
18.  }
19.  cout << "Feljegyzett kacatok: " ;
20.  for (int i = 0; i < darab - 1; i++)
21.    cout << kacattar[i] << ", ";
22.  if(darab == 0)
23.    cout << kacattar[darab - 1] << "." << endl;
24.  if(darab == maxkacat)
25.    cout << "Sajnos, csak ennyit tudok eltárolni" << endl;
```

Ahhoz, hogy a feladatot a leírásnak megfelelően oldjuk meg – a kiírás végén pont legyen – a minta megoldás 20. sorában a kiírandók számát eggyel csökkenteni kellett, a ciklust követően tudjuk kiírni az utolsó adatot – de csak akkor, ha létezik – a végén a ponttal. Másik megoldás, hogy a cikluson belül feltételhez kötjük az elválasztójel kiírását. Harmadik megoldás a ciklus után: `cout << "\b\b." << "." << endl;`. Melyik jobb és miért?

Bármilyen módon írtuk meg a programunkat, az első kacatot kivételnek kellett tekintenünk. Emiatt egy kódrészlet kétszer szerepel, amit a programozók nem tartanak jó megoldásnak. Ezért van több programozási nyelvben lehetőség olyan ciklus írására, aminek a végén ellenőrizzük, hogy kell-e még ismételni a feladatot, azaz hátul teszteljünk. A C++ – és sok más – nyelvben a do-while-ciklus elején a **do** kulcsszó jelöli azt a pontot, ahova vissza kell lépni. Ezt követi a ciklusmag, majd ennek lezárása után a **while**-nak a paramétereként azt adjuk meg, hogy mikor kell visszatérni a **do**-hoz.

Hátultesztelős ciklust mondatszerű leírásban is használhatunk, a programunk algoritmusa is egyszerűbb lehet így:

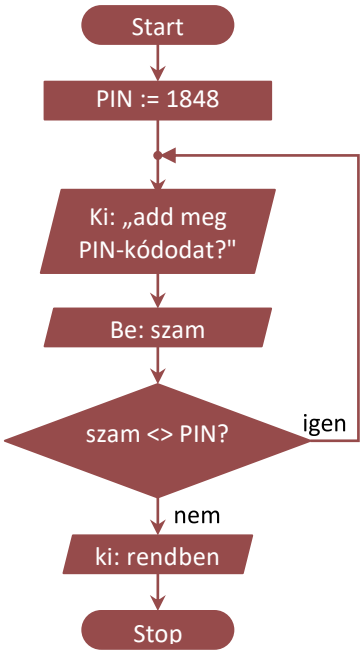
```
kacattar létrehozása
ki: "vége: elfogyott"
ciklus:
    ki:"Kérek egy kacatot"
    be: kacat
    ha kacat != "elfogyott"
     kacattar.betesz(kacat)
amíg kacat != "elfogyott"
ciklus vége
ki: "feljegyzett kacatok:"
ciklus 0-tól kacattar.elemszama-1-ig:
    ki: kacattar.eleme + ", "
ciklus vége
ki: kacattar.eleme + "."
```

A hátultesztelős ciklus alkalmazásával az adatbekérés egységes lett, de a cikluson belül is ellenőrizni kell a választ, vagy a bevitel végén valamit kezdeni kell azzal, hogy az „elfogyott” is bekerült a kacatok közé. A megoldás valami remove-szerű függvénye lehet a `kacattar`-nak.

A hátultesztelős ciklust – ha a programozási nyelvben létezik – jellemzően adatbekérésre szokták használni. Abban az esetben ideális, amikor újra kérnünk egy, a feltételeinknek nem megfelelő adatot.

Például, amikor egy PIN-kódot addig kérdezünk, amíg a felhasználó el nem találja. A folyamatábrán jól látszik, hogy a visszalépés nem közvetlenül a feltétel elé történik, hanem egy korábbi pontra, másrészt ezen az ágon nincs (és nem is lehet) utasítás.

A hátultesztelős ciklusnak egyes programozási nyelvekben kissé más a megvalósítása: a ciklus végén nem azt vizsgálja, hogy vissza kell-e lépnie, hanem azt, hogy mehet-e tovább.



3. Írjuk át az algoritmusnak megfelelően a kódot is!

Az ellenőrzéshez egy listás megoldás:

```
 6.  vector<string> kacattar;
 7.  cout << "Vége: \"elfogyott\"" << endl;
 8.  string kacat;
 9.  do
10.  {
11.    cout << "Kérek egy kacatot ";
12.    cin >> kacat;
13.    if (kacat != "elfogyott")
14.      kacattar.push_back(kacat);
15.  }while (kacat != "elfogyott");
16.  cout << "Feljegyzett kacatok: ";
17.  for (unsigned i = 0; i < kacattar.size() - 1; i++)
18.    cout << kacattar[i] << ", ";
19.  if (kacattar.size() != 0)
20.    cout << kacattar[kacattar.size() - 1] << "." << endl;
```

Informatikus szemmel nézve nem túl szerencsés, hogy a befejezéshez egy szót adunk meg. Mi van, ha egyszer véletlenül a felhasználónak mégis lesz „elfogyott” nevű kacatja? Ezt nem tudja bevinni a programunkba. Ezért a végjeles adatbekérésnél egy megadott szó helyett az „üres karakterlánc” szokott szerepelni. Ha a felhasználó a kérdésünkre csak egy ENTER-rel válaszol, akkor az eredmény a `""` lesz.

Az elképzelés jó, de van egy buktatója: Ha a felhasználó csak egy ENTERt üt, akkor azt a >> elnyeli, tovább fogja várni az adatot. Emiatt szöveg esetén a >> helyett `getline()`-nal lehet beolvasni az adatot.

Kiegészítés: Az adatok beírását cin az ENTER leütésekor kapja meg. Előtte törölhetjük a hibásan beírt adatot, erről a program nem fog tudni.

ADATSOROK KEZELÉSE

Most már tudunk adatsorokat tárolni és ki is tudjuk írni képernyőre. A tárolásnak a célja, hogy mindenféle kérdéseinkre gyorsan tudjon a programunk válaszolni. Persze csak akkor érdekes a feladat, ha sok adattal tudunk dolgozni. Elvileg kacatjainkból is sok van, de beírni mindet …

Hogyan tudunk sok adatot használni egy-egy programban? Az adatokat nem a programok kódja tárolja, hanem mindig kívülről kapja a program: „internetről”, fájlból, mérőműszertől vagy konzolról. Ezekre az „adatcsatornákra” programozási nyelvekben megtalálhatók az adatok írását (kiküldését) és (be)olvasását végző eljárások és függvények. Mi csak a konzolt használjuk, de a programozási nyelv dokumentációjában megtalálhatók a többi csatorna használatára is a megoldások. A megoldások keresésének egyi kulcsszava a „*stream*”, mert a csatornában az adatok egymás után sorban haladnak, a csatornába betöltik az adatot, illetve kiveszik belőle, a byte-ok értelmezés és struktúra nélküli folyamot alkotnak. Minden adatcsatornába az egyik végén lehet betölteni az adatot, a másik végén megnézhetjük, hogy mi van, kivehetjük a szükséges bájtokat (egy integer esetén például 4 bájtot veszünk ki).

Fájlból konzolra másolás

A konzol általában három csatornát tud kezelni: egyet a beolvasáshoz, egyet a kiíráshoz és egy harmadikat a technikai részletek (error vagy log) feljegyzéséhez. Ebből most számunkra a beolvasásra használt csatorna a lényeges, mert azon keresztül lenne jó sok adatot bejuttatni a programunkba. Az első megoldás a működés ismeretéből kézenfekvő: a program nem „látja”, hogy hogyan kerül be az adat a csatorna másik végére. Nem érzékeli a billentyűlenyomásokat … csak a bevitt adatok sorozatáról tud. Ezért a csatorna másik oldalán ügyeskedhetünk: összegyűjthetjük a beírandókat és bemásolhatjuk a bemenetre.

Sok adatot úgy lehet „begépelni”, hogy 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. Ehhez az egérrel jobb-klikk, Beillesztés lehetősége vagy a CTRL+V billentyűkombináció is alkalmas.

Nem kell félni, a programunk nem fog sokkot kapni 1000 adattól még akkor sem, ha csak egyetlen számot vár – főleg, ha a bemásolt adatok első sorában valóban egy szám van –, mert a bájtok kivételét az első enterig végzi. A többi adat a csatornában vár, amíg sorra kerül, vagy amíg befejeződik a program. A program bezárásakor a csatorna is megszűnik, így a benne maradt adatok is törlődnek.

Az adatok beírásának ilyen módja abban az esetben okozhat problémát, ha az ENTER-t nem „érti meg” a programunk. Azt tudjuk, hogy a kódban a `'\n'` escape karakter jelöli ezt. Neve LF (LineFeed), ASCII-kódja 10. Ezt a karaktert Linux operációs rendszerben az ENTER, Windows operációsrendszerben a sortörés, SHIFT+ENTER billentyűleütéssel lehet beírni. A Windowsban az ENTER leütésekor `'\r'` – CR (CarrigeReturn) ASCII kódja 13 – bájt is keletkezik. Emiatt a sorok végén valójában `\r\n` van. Hogy ez ne okozzon gondot, a Windowson használt programozási nyelvekben a `'\n'` együtt értelmezi a két karaktert, beolvasáskor a CR-nél befejezi a beolvasást de még a következő LF-et is beolvassa, kiíráskor mindkettőt kiírja. Nem is lenne gond, ha

mindig ott lenne a sorok végén a CR, de – mivel Linuxon nincs – a sorok végét sokszor csak az LF jelzi. Ilyenkor a program nem veszi észre, hogy hol a sor vége. A problémát bonyolítja, hogy vannak olyan operációs rendszerek, amelyekben csak a CR jelöli a sorvéget és olyanok is, amelyekben fordított sorrendben van a két kód: `\n\r`.

Nem túl bíztató a helyzet, de azért van megoldás. Nagyon sok program fel van készítve arra, hogy bármelyik változatot értse és helyesen értelmezze. Egy jellemző eset, hogy a zip tömörítésből Jegyzettömbbel megnyitott txt fájlokban csak LF van, de ha kitömörítés után nyitjuk meg, akkor már CR LF lesz benne. Ha másképp nem megy, fejlett szövegszerkesztőben van mód arra, hogy a sortörést bekezdésvégjelre cseréljük, ezt visszamásolva egy txt-be, az adott operációsrendszerben használható szöveges fájlt kapunk.

Konzol átirányítása

A konzol csatornáinak egyik végén a program van, a másik vége „szabadon”. Láttuk, hogy a gépelés helyett a „Vágólap”-ról is tudunk adatot bevinni. Az operációs rendszerek nagyon sok olyan programmal dolgoznak, amelyek egymásnak adják át az adatokat, ezért képesek arra, hogy az egyik program konzol kimenetét egy másik program konzol bemenetére irányítsák – összekössék a csatornákat. Ennek kiegészítéseként, egy programba fájlból is be tudjuk irányítani az adatokat. A lehetőséget az operációs rendszer adja, ezért nem az IDE-ből futtatjuk ilyenkor a programunkat, hanem a fordítás után, egy konzolablakban (terminál ablakban) indítjuk el a programunkat. A fájl „becsatornázása” a programba a ’<’ jellel oldható meg. Például így indítjuk a programot:

```
program.exe < adatok.txt
```

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

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

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

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

Lényegében ezt használják ki, amikor egy programot tesztelnek. Az operációs rendszerben lehet írni olyan scriptfájlokat, amelyekben megadják a bemeneteket és a kimeneteket, majd egy ciklussal sorban mindegyikkel lefuttatják a programot. A kimenetet persze át lehet adni egy kiértékelő programnak is, amelyik elemzi az eredményt és a kimenetén megjelenik az értékelés…

Szerencsejáték esélyek

Most már meg tudnánk vizsgálni korábban eltárolt adatokat is, de az adatfájlok elkészítése is idő. Ezért tárolt adatok nélkül, kockadobással gyártunk sok adatot.

Feladatok

1. Szimuláljunk tízmillió kockadobást, és tároljuk az eredményeket!

   A feladatot tömb vagy lista használatával oldjuk meg, aminek a neve `dobasok`. Ebben az esetben – mivel pontosan tudjuk, hogy mennyi adat lesz, egyszerűsége miatt – a tömb a „szebb” megoldás. Azonban a tízmillió túl nagy szám ahhoz, hogy futtatás közben ekkora tömbnek találjon helyet a program, ezért a tömböt a főprogram előtt kell létrehozni. A számok előállításához egy véletlenszám-generátort használunk tízmilliószor.

```
 8.  int dobasok[10000000];
 9.  int main()
10.  {
11.    srand(time(0));
12.    for (int i = 0; i < 10000000; i++)
13.    dobasok[i] = rand() % 6 + 1;
```

2. Programunk számolja meg, hogy hányszor „dobtunk” hatost!

   Itt már az adattípustól független a kódrészlet:

```
14.    int hatosdb = 0;
15.    for (int i = 0; i < 10000000; i++)
16.    {
17.      if (dobasok[i] == 6)
18.      {
19.        hatosdb++;
20.      }
21.    }
22.    cout << "Összesen " << hatosdb << " darab hatost dobtunk." << endl;
23.  }
```

3. Számoljuk össze mind a hat lehetőség előfordulásait!

   A megoldást először egy 1 és 6 között futó külső ciklussal készítjük el:

```
25.  for (int szam = 1; szam <= 6; szam++)
26.  {
27.    int db = 0;
28.    for (int i = 0; i < 10000000; i++)
29.    {
30.      if (dobasok[i] == szam)
31.      {
32.        db++;
33.      }
34.    }
35.    cout << db << " esetben dobtunk " << szam << "-t." << endl;
36.  }
```

Viszonylag kevés módosítással hatszor több eredményünk van. De a megoldási idő is hatszor több, hiszen a tízmillió értéket hatszor nézzük végig. Oldjuk meg úgy a feladatot, hogy csak 1-szer nézünk meg minden dobott értéket és eldöntjük, hogy melyik csoportba tartozik!

Ehhez az dobás értékének vizsgálatát kellene hatszor beírni, természetesen 6 külön változóba. Sem a 6-szor beírni majdnem ugyanazt, sem a 6 külön változót létrehozni nem vidít fel egy programozót. Kezdjük a „6-szor beírni” szépítésével.

Beírhatunk 6 **`if`**-et, de akkor a programunk mindig, mind a hatot ellenőrzi, akkor is, ha már az első helyes volt (és a többi nem lehet az). Ezért, ha így oldjuk meg, akkor az **`else if`** sokkal praktikusabb.

Mivel itt csak néhány konkrét értékeket kell megvizsgálnunk, a feltételek elég unalmasak lesznek: egyenlő az egyikkel, egyenlő a másikkal … Ilyenkor lehet bevetni egy **`switch`**-et. Bár a megoldás nem lesz lényegesen rövidebb, de a kód átláthatóbb lesz.

```
25.  int db6, db5, db4, db3, db2, db1;
26.  db6 = db5 = db4 = db3 = db2 = db1 = 0;
27.  for (int i = 0; i < 10000000; i++)
28.  {
29.    switch (dobasok[i])
30.    {
31.      case 1: db1++; break;
32.      case 2: db2++; break;
33.      case 3: db3++; break;
34.      case 4: db4++; break;
35.      case 5: db5++; break;
36.      case 6: db6++; break;
37.      default: break;
38.    }
39.  }
```

Egész szám vagy karakter... Ez == valamelyik eset

break: kilép a switchből. ha hiányzik, akkor a következő utasítást is végrehajtja

A kód áttekinthetőbb, ezért még jobban látszik, hogy a hat változó mennyire egyforma funkciójú. Ezek bizony egy tömbbe tartoznak! Ha létrehozunk egy tömböt nekik, és az első helyen az 1-es dobások értékét gyűjtjük, akkor könnyen belezavarodhatunk, mert az a 0. index. Ezért úgy készítsünk tömböt, hogy az első adat legyen 0, a második helyen – az 1-es indexű helyen számláljuk az 1-es dobások számát!

A tömb létrehozása és a **switch**-ben alkalmazása:

```
25.  int darab[7];
26.  for (int i = 0; i <= 6; i++)
27.    darab[i] = 0;
28.  for (int i = 0; i < 10000000; i++)
29.  {
30.    switch (dobasok[i])
31.    {
32.      case 1: darab[1]++; break;
33.      case 2: darab[2]++; break;
34.      case 3: darab[3]++; break;
35.      case 4: darab[4]++; break;
36.      case 5: darab[5]++; break;
37.      case 6: darab[6]++; break;
38.      default: break;
39.    }
40.  }
```

Egy változónévben bármilyen szám szerepelhet, a db1 lehetne `dbegy` is. De a jelenlegi formában a `darab`[1] változóban az 1 is szám, a darab tömb egyik elemének az indexe. Ha kiírjuk az adatokat, akkor erre a számra hivatkozva választjuk ki, hogy melyiket szeretnénk kiírni. A kiírás közben az index értéke változik – így lép egyik adatról a másikra. Az index helyére eszerint bármilyen egész számot eredményező kifejezést be lehet írni. A kódot elemezve megállapítható, hogy *a darabszámokok közül mindig azt kell növelni, amelyiknek az indexe megegyezik a* `dobasok[i]` *értékével.* Ezt a gondolatot kódolva, a **switch** egyetlen sorral helyettesíthető:

```
30.  darab[dobasok[i]]++;
```

Azaz vegyük az annyiadik darabot, amennyi az `i`-edik dobás értéke és növeljük meg 1-gyel.

A kód végső állapota az eredmény kiírásával együtt:

```
25.  int darab[7] {0,0,0,0,0,0,0};
26.  for (int i = 0; i < 10000000; i++)
27.    darab[dobasok[i]]++;
28.  for (int szam = 1; szam <= 6; szam++)
29.    cout << szam << " lett " << darab[szam] << " dobás." << endl;
```

A megoldás nem tartalmaz speciális programnyelvi eszközt, a megoldás feltétele, hogy

- a tömb vagy lista minden indexe bármikor elérhető legyen, azaz az egyes elemeket az indexeiken keresztül közvetlenül elérjük.
- értsük, hogy pontosan mit jelent egy adatsorozat elemének az indexe, illetve értéke; értsük, hogy egy adatsorozat egyik elemének az értéke megadhatja egy másik adatsorozatban egy elem indexét, amelynek az értékét így módosíthatjuk.

Programozási tanulmányaink kezdetén a feladatnak mindegyik itt látható megoldása helyes (max. pontos). Nem baj, ha az utolsó még zavaros, az első megoldás is jó. A **switch** használatát sem kell tudni. A programozás egy kicsit olyan, mint a nyelvismeret, ahol 100 szó elég, hogy megértessük magunkat, mégis, évekig tanuljuk, fejlesztjük a szókincsünket, tanuljuk a nyelv szabályait gondolataink minél pontosabb kifejezése érdekében.

Indexek használata

Az előző feladatban láthattuk, hogy az adatsorok elemeit nem csak sorban lehet elérni. A következő feladatokban az lesz a megoldás kulcsa, hogy – bár egymás után nézzük az egyes elemeket, de egy-egy lépésben az adatsor több elemét vizsgáljuk.

4. Hány helyen előzi meg a hatos dobást ötös dobás?

   A feladat nagyon hasonló a 2. feladathoz, amikor a hatos dobások számát vizsgáltuk, de módosult a számlálás – a jó eset – feltétele: hatos előtt ötös. Az egyik dobás értéke 6 és az előtte lévő dobás értéke 5.

   Írjuk át a 2. feladat megoldását ennek megfelelően:

```
33.  int otrehatdb = 0;
34.  for (int i = 1; i < 10000000; i++)
35.  {
36.    if (dobasok[i] == 6 && dobasok[i - 1] == 5)
37.    {
38.      otrehatdb++;
39.    }
40.  }
41.  cout << otrehatdb << " esetben lett 5 után 6." << endl;
```


5. Hány helyen van egymás után két hatos?

   Oldjuk meg a feladatot először úgy, hogy amikor három hatos van egymás után, azt tekintsük két olyan esetnek, amiben két hatos van egymás után. Ezután módosítsuk úgy a programunkat, hogy csak azokat az eseteket számolja, amelyekben pontosan két darab hatos van egymás után. Ennek megoldásában praktikus néhány speciális esetet a cikluson kívül megvizsgálni.

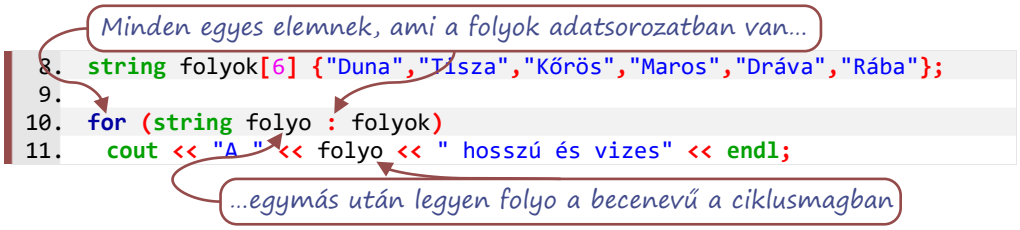
A bejárós ciklus (foreach-ciklus)

Mostanra megismerkedtünk a feltételes ciklusokkal. Először a while-ciklussal, ahol a ciklusmag előtt döntjük el, hogy végrehajtjuk-e vagy kihagyjuk. Tettünk egy rövid kitérőt a hátultesztelős do-while-ciklus felé. Ez abban az esetben könnyíti meg az algoritmus és a kód írását, amikor a ciklusmag végrehajtása után szeretnénk eldönteni, hogy szükséges-e visszamenni és újra (meg újra) végrehajtani a ciklusmagot.

A for-ciklust szinte a while-ciklussal egyidőben kezdtük használni és az adatsorozatok megjelenésével egyre inkább látható a praktikussága: a ciklusfejben adjuk meg a számláló változását, mert sokkal kisebb a valószínűsége a lefelejtésének. Most már az is látszik, hogy az adatsorozatok elemeinek egymás utáni elérését is megkönnyíti, hogy az iterátort (számlálót) fel tudjuk használni az adatsorozat elemeinek indexeléséhez, egymásutáni kiválasztásához. Az iterátor és index olyan gyakran kapcsolódik össze, hogy az i-t nem csak az iterátor szó kezdőbetűje alapján használjuk a ciklusokban. Az adatsorozatok tetszőleges elemének indexét is tipikusan i-vel jelöljük. A matematikai függvényekben – f(x), |x| – az x a függvény értelmezési tartományának egy tetszőleges eleme. Az adatsorozatok esetén hasonló értelemben használjuk az i-t, a feladatról írva: egy *A* adatsorozat tetszőleges eleme *A[i]*, azaz az *A lehetséges indexeinek egyike által meghatározott adat*.

Adatsorok elemzése, statisztikai számítások végzése során gyakori, hogy az adatsor minden elemén végig kell lépkedni és az elemzéshez, az elemen vagy elemmel elvégzendő tevékenységhez az adatsorból csak az éppen kiválasztottra van szükség. Ezeknek a feladatoknak a megoldására írták meg nagyon sok programozási nyelvben a bejárós ciklust.

A bejárós ciklus egyesével végiglépked egy bejárható objektum, például egy lista értékein. Értékein lépked, azaz a ciklusváltozó nem az index lesz, hanem az adat. A ciklus magjában használhatjuk ezt a változót: megnézhetjük az értékét.



A programozói szlengben ezt a fajta ciklust foreach-ciklusnak hívják, mert a kulcsszó általában nem for, hanem foreach. A ciklusszervezés paramétereit itt is a ciklusfejben adjuk meg, valójában erről ismerhető fel a ciklus típusa. Nem három részből áll, itt egyetlen utasítást kell megandunk: kettősponttal elválasztva azt, hogy milyen adat melyik adatsort nézzen végig. Ezt követően egy utasítás a ciklusmag vagy kapcsos zárójelek között adhatunk meg több utasítást.

Ugyanezt mondatszerű leírásban is megadhatjuk:

```
folyok létrehozása
ciklus folyok minden folyo-jára:
    ki: "A " + folyo + " hosszú és vizes."
ciklus vége
```

A foreach-ciklus ciklusváltozójának a típusa mindig megegyezik az adatsorozat típusával, emiatt a példában szereplő folyo egy for-ciklus minden i-jére azonos folyok[i]-vel. Főleg kezdetben, az adattípusok összekeverésének elkerülése miatt érdemes kiírni a ciklusváltozó elé a valódi adattípust. Mivel ez mindig az adatsorozat elemeinek az adattípusa, ezt a fordítóprogram (is) ki tudja találni, ezért – ha már jól tudjuk használni a foreach-ciklust – az adattípus kitalálását rábízhatjuk a fordítóprogramra. A C++-ban a „találd ki, milyen adattípus” jelölése: **auto**.

Kiegészítés: Kicsit módosítanunk kell a kódot akkor, ha az adatokat nem csak megnézni, hanem módosítani is szeretnénk, mert ha csak annyit adunk meg, hogy **string** (vagy **int**, … **auto**), akkor a változónkba az adatsorozat elemének a másolata kerül. Bár ezt módosíthatjuk, de minek, ha egyszer a következő adatra lépéskor a következő elem másolatával felülírjuk. Ha az eredeti elemet szeretnénk elérni (megnézni és módosítani), akkor ezt az adattípust követő **&** jellel kell jeleznünk.

Próbáljuk ki az alábbi megoldást az **&** jelekkel és ezek nélkül is:

```
 8.  string folyok[6] {"Duna","Tisza","Kőrös","Maros","Dráva","Rába"};
 9.
10.  for (string& folyo : folyok)
11.    folyo = "A " +  folyo;
12.  for (string& folyo : folyok)
13.    cout << folyo << " hosszú és vizes" << endl;
```

Az eredeti legyen, ne csak másolat!

Módosítjuk.

A foreach-ciklust nagyon kényelmes használni. Annyira, hogy egyeseknél az addikció gyanúja is felmerül: a feladat megoldásához – akkor is, ha nem praktikus – a foreach-ciklust próbálja használni, ezért a kód – ha egyáltalán sikerül megírni – bonyolult lesz vagy hibás eredményt ad. Ezért nézzük, hogy mely esetekben **nem jó** a foreach-ciklus:

- A foreach minden adatsorozat-típusra használható, de a már megismert tömbökre inkább ne használjuk, mert lehetnek a sorozatban lyukak.
- Ne használjunk foreach ciklust, ha van olyan adat, amelyet ki szeretnénk hagyni. A korábban megoldott feladatok közül ilyen:
  - Adatsorozat elemeinek kiírása úgy, hogy közben vessző választja el az adatokat, de az utolsó után pont van. (Kacatok kiírása.)
  - Olyan adatsor elemeinek kiírása, ahol a 0. elemet – az értelmezés miatt – nem használjuk. (Kockadobás értékeinek számlálása.)
- Olyan feladatokban, ahol egymáshoz viszonyított helyzetet kell vizsgálni az is kellene, hogy elérjük az előző vagy következő elemet is. (Kockadobás szomszédos adatok.)
- Olyan feladatokban, ahol nem az adat értékét kell megadni, hanem azt, hogy melyik adatról van szó, hányadik, hol van az adatsorozaton belül. Ezeknél a feladatoknál külön változó kellene, amiben tároljuk a helyet és esetleg egy másik is, hogy kezeljük, ha több megoldás is lenne. Ha egy feladatban az index és az érték megadása is szükséges, akkor elsősorban az index megadása a cél, mert abból keresés nélkül megadható az érték. Fordítva nem igaz, mert attól, hogy ismerjük az értéket, még nem tudjuk, hogy hol van.

Feladatok

1. Írjunk olyan programot, amely egy futóverseny résztvevőinek célba érkezés szerinti neveit kéri a felhasználótól, majd kiírja a dobogósokat és a sereghajtót! Minden újabb versenyző bekérése előtt írja ki az épp aktuális névsort!

```
 8.  vector<string> versenyzok;
 9.  string versenyzo;
10.  do
26.  {
11.    cout << "A versenyzők névsora:" << endl;
12.    for(string v : versenyzok)
13.      cout << v << endl;
14.    getline(cin, versenyzo);
15.    if (versenyzo != "")
16.      versenyzok.push_back(versenyzo);
17.  } while (versenyzo != "");
18.  for (unsigned i = 0; i< versenyzok.size() && i < 3; i++)
19.    cout << (i + 1) << ". helyezett: " << versenyzok[i] << endl;
20.  if (versenyzok.size() > 1)
27.    cout << "A sereghajtó: " << versenyzok[versenyzok.size() - 1] ;
```

2. A fenti megoldás a négyféle ciklusból hármat tartalmaz, csak a while-ciklust nem használja. Másrészt, a while-ciklus a legáltalánosabb, a többi átírható erre. Írjuk meg a kódot csak while-ciklus használatával!

Adatsorozatok függvényei

Nemcsak a bejárós-ciklus alkalmazása tipikus adatsorok esetén, hanem egyes tulajdonságok, műveletek, statisztikai feladatok is. Ilyen tulajdonság az adatsorozat elemszáma (`size()`), műveletként alapvető a hozzáfűzés `push_back()` eljárása. Ezeken túl az `<algorithm>` csomagban nagyon sok egyéb feladat, számítás elvégzésére létezik kész megoldás.

Feladatok

1. Írjunk programot, amely egy autókölcsönző munkáját szimulálja! A kölcsönző a munkanapot egy listányi autóval kezdi, és addig kölcsönöz, amíg minden autót ki nem ad. A program írja ki az autók listáját és kérdezze meg, melyiket kölcsönzi ki a felhasználó. Írja ki, hogy mik maradtak benn, és kérdezzen újra és így tovább.

   Figyeljük meg az `algorithm` függvényeinek (`find`, `erase`) használatát:

```
 1.  setlocale(LC_ALL,"");
 2.  vector<string> autok{"Trabant", "T-Modell", "Rolls-Royce"};
 3.  while(autok.size() > 0)
 4.  {
 5.    string mind = "";
 6.    for (string ez : autok)
 7.      mind += ez + ", ";
 8.    mind += "\b\b.";
 9.    cout << "Kölcsönözhető:" << mind << endl;
10.    cout << "Melyik autót kölcsönzi ki? ";
11.    string mit;
12.    cin >> mit;
13.    auto itt = find(autok.begin(), autok.end(), mit);
14.    if(itt != autok.end())
15.      autok.erase(itt);
16.    else
17.    cout << "Ilyen autóval nem szolgálhatunk" << endl;
18.  }
```

2. Írjuk meg a programot a `find`() és `erase`() használata nélkül!

3. Írjuk meg a programot úgy, hogy az adatokat (kellően nagy méretű) tömbben tároljuk! törléskor mindig az éppen utolsó adattal írjuk felül a törölt elemet!

Adatsorozatok és ciklusok

Feladatok

A következő feladatok megoldását – amennyire tudjuk – oldjuk meg többféle ciklussal és – esetleg – tömb és lista használatával is!

1. Írjunk olyan programot, amely egy-egy listába bekéri három-három leves, főétel és desszert nevét, majd kiír három menüt, mindegyikben egy levessel, egy főétellel és egy desszerttel!
2. Írjuk ki az első 10 természetes számot és a négyzetüket!
3. Három egymásba ágyazott ciklussal rajzoljuk ki a jobb oldalon láthatóábrát!

```
 6.  for (int tegla = 0; tegla < 3; tegla++)
 7.  {
 8.    for (int sor = 0; sor < 4; sor++)
 9.    {
10.      for (int hely = 0; hely < 5; hely++)
11.        cout << "o";
12.      cout << endl;
13.    }
14.    cout << endl;
15.  }
```

```
ooooo
ooooo
ooooo
ooooo

ooooo
ooooo
ooooo
ooooo

ooooo
ooooo
ooooo
ooooo
```

   a) Mi a szerepe az 7. és 9. sornak?
   b) Hol kell átírni a kódot, hogy három, az itt láthatóval egyező háromszöget rajzoljon?

```
o
oo
ooo
oooo
```

4. Állítsunk elő egy tízelemű, pénzfeldobások eredményeit tartalmazó listát! Hány olyan eset van, amikor az aktuális és az előző dobás is „fej”?
5. Állítsunk elő harmincelemű, nulla és kilenc közötti véletlenszámokat tartalmazó listát! A számok egy útvonal magassági adatait jelentik. Meredek az útszakasz, ha legalább kettővel magasabb az aktuális hely, mint az előző. Hány meredek szakasz van az úton? És visszafelé?
6. Kihívást jelentő feladat: A programunk elején adjunk meg két listát:
   - az első tartalmazzon öt filmcímet,
   - a második a filmek egy-egy főszereplőjét!

   a) Az első filmhez az első szereplő tartozik, a másodikhoz a második, és így tovább.
   b) Írjuk ki a filmcímeket, majd az egyik, véletlenszerűen kiválasztott szereplőt!
   c) Kérdezzük meg a felhasználótól, hogy a kiírt szereplő melyik filmnek a főszereplője!
   d) Értékeljük a válaszát!
7. Kihívást jelentő feladat: Állítsunk elő nyolcvanelemű, –5 és 3 közötti egész számokból álló listát! A számok egy úszó palackorrú delfin magasságát jelentik. A delfin ki-kiugrál a vízből, ilyenkor pozitív a magassága. Nulla a magasság, amikor a felszínen úszik, negatív, amikor a víz alatt. Írjunk programot, ami választ ad a következő kérdésekre!

a) Az idő hányadrészét töltötte a delfin a vízben, illetve a víz alatt? A válaszok megadhatóak törtszámként és százalékként is. (Feltételezhetjük, hogy az adatok egyenlő időközönként érkeztek.)
b) A víz alatt vagy a víz felett volt többet a delfin? A vízfelszínen való utazás egyik esetbe sem számít bele.
c) Milyen hosszú volt a leghosszabb kiugrása? Az út hányadik pontjánál kezdődött?
d) Hányszor törte át a vízfelszínt, azaz hányszor követ a listában negatív számot pozitív, vagy fordítva?
e) Mély merülésnek számít, ha a delfin –4-es vagy –5-ös mélységben van. Az út során hányszor merült mélyre? Figyeljünk arra, hogy például a 4 –2, –4, –5, –5, 3 útvonal csak egy mélyre merülést jelent!

Tárgymutató