Online tananyag

Interdiszciplináris tudományok

Sorozatszerkesztő: Dr. Balázs László

27.

Dr. Zachár András:

Bevezetés a Programozásba

DUNAÚJVÁROSI EGYETEM
UNIVERSITY OF DUNAÚJVÁROS

Tartalomjegyzék

1. fejezet

1. Tárgy célja, algoritmizálási alapok

A Bevezetés a programozásba tárgy alapvető célja, amint azt a neve is hordozza, hogy megismertesse az elsőéves informatikus hallgatókat az algoritmizálás, programozás alapjaival, illetve a hallgatókban az algoritmizálás, algoritmikus gondolkodás képességét kialakítsa. A tárgy alapvetően a strukturált programozás alapjait kívánja az olvasóval ismertetni, ezért az objektum orientált fejlesztésről és programozásról (OOP) csak a legszűkebb értelemben esik szó, ha annak bemutatása valamilyen ok miatt elkerülhetetlen. Mivel a Bevezetés a programozásba tantárgy keretei között a *C#* nyelvet tanítjuk, ezért szükséges néhol említés szintjén az OOP-vel foglalkozni. Mivel a tárgy a programozás alapjaival igyekszik a hallgatókat megismertetni, említés szintjén több programnyelvvel is összehasonlítja a *C#* nyelv egyes részeit, hasonlóságokat, illetve különbségeket bemutatva. Ennek célja az, hogy a hallgatók látóköre már a programozás kurzusok kezdetétől minél szélesebb körű legyen.

1.1 Algoritmus fogalma

Definíció: Az algoritmus egyértelmű előírása egy feladat vagy probléma megoldásának a matematikában és a számítástudományban.

Az algoritmusok számításokat, adatkezelést vagy automatizált következtetési feladatokat hajthatnak végre. Egy algoritmus valamilyen kiinduló állapotot és a bemeneti adatokat felhasználva a megadott utasítások felhasználásával számításokat hajt végre. Az utasítások végrehajtása során az algoritmus véges számú, jól definiált egymást követő állapotokon keresztül jut el a befejező állapotba, miközben a megkívánt kimeneti eredményeket szolgáltatja.

Algoritmusok használata nem a huszadik század során alakult ki a digitális számítógépek megjelenésével párhuzamosan, hanem már évszázadokkal, sőt évezredekkel korábban léteztek különféle algoritmusok, elsősorban matematikai problémák megoldására. A görög matematikusok már időszámításunk előtt használtak algoritmusokat prímszámok keresésére, vagy például ilyen a jól ismert Euklideszi algoritmus két egész szám legnagyobb közös osztójának meghatározására.

1.2 Algoritmus készítés alapeszközei

Az algoritmizálás, mint a programfejlesztés kiinduló lépése feltétlenül meg kell előzze a programírás folyamatát. Egyszerűbb programok esetén, ha a programozó kellő gyakorlattal rendelkezik, az algoritmizálás fejben is elvégezhető és nem szükséges hozzá semmilyen algoritmusleíró eszköz, de ha a kód komplexebb, akkor célszerű valamilyen formában a gondolatainkat valamilyen szabályszerű, strukturált formában összegezni. Fontos megjegyezni, hogy az alábbiakban bemutatott algoritmusleíró eszközöknek is megvannak a maguk hátrányai, korlátai. Alapvetően mindegyikre igaz, hogy a programok komplexitásának növekedésével, amikor egy szoftvert esetleg több tucat programozó fejleszt, már egyik sem igazán alkalmas a teljeskörű és áttekinthető szoftver feladat leírásra.

A következő gyakrabban használt, klasszikusnak is nevezhető alapeszközök állnak rendelkezésre az algoritmus leírásra:

– Folyamatábra
– Struktogram
– Pszeudokód

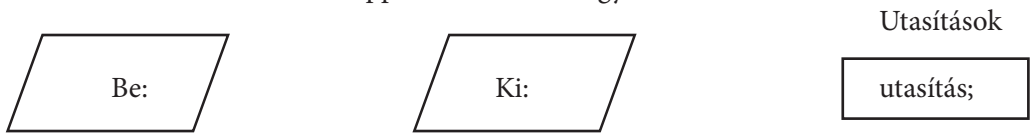
1.3 Folyamatábra

A legszélesebb körben elterjedt algoritmust leíró és szemléltető eszköz a folyamatábra, amely matematikai szempontból nézve tulajdonképpen egyfajta speciális gráfként fogható fel, amely csomópontok és élek halmazából tevődik össze. A folyamatábrát leíró gráf irányított, amelynek kiinduló pontja a program kezdete, végpontja pedig a program leállása. A gráf élei mutatják a futó program folyamatainak irányát. A csomópontok pedig valamilyen a program futása szempontjából fontos lépést ad meg, amiket az alábbiakban részletezünk.

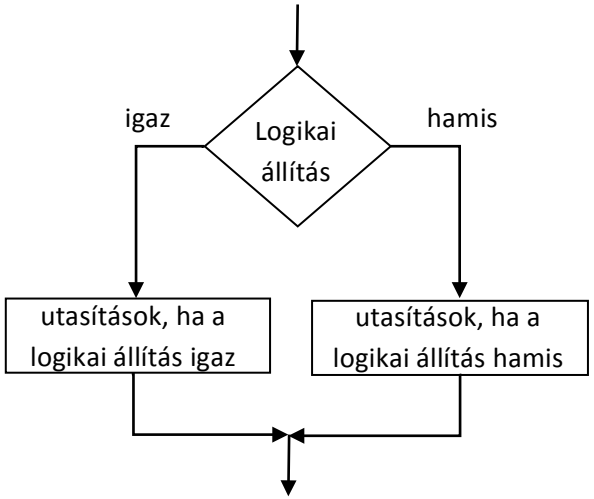
Minden futó programnak van egy kiinduló és egy végpontja. Ezt általában inkább virtuálisnak tekinthetjük abban az értelemben, hogy külön utasítást nem helyezünk el a forráskódban a program indítására vagy annak estleges leállítására. Annyit azért érdemes megjegyezni, hogy minden programnak van valamilyen belépési pontja, amit végső soron felfoghatunk a program „Start” pontjának. A program indulási és végpontját egy ellipszisbe írt start és stop kulcsszavakkal jelöljük a folyamatábrában.

Program kimenet és bemenet (Input, Output)

A programoknak általában van bemente (Input) és kimenete (Output). A program kimenetek általában a képernyőre kerülnek, de a kimenetek kerülhetnek háttértárakra, vagy esetleg egy nyomtatóra. A program bementek többnyire a billentyűzetről, esetleg háttértárról kerülhetnek be a programba. A program be és kimeneteket paralelogrammákkal jelöljük a folyamatábrákban. A paralelogrammákon belül felszokás sorolni azokat a változókat, amiket éppen beolvasunk vagy kiíratunk.



A programok futásuk során különféle feladatokat hajtanak végre. Ezeket a feladatokat a programozó utasításokon keresztül tudja megszabni. Az utasítások általában igen sokfélék lehetnek, egy egyszerű egész szám értékének növeltetésétől, egy bonyolult matematikai összefüggés megadásáig. Az utasításokat a folyamatábrákban téglalapokkal és a téglalapba írt rövid szöveges, illetve esetleg valamilyen matematikai leírással adjuk meg.



Feltételes elágazás

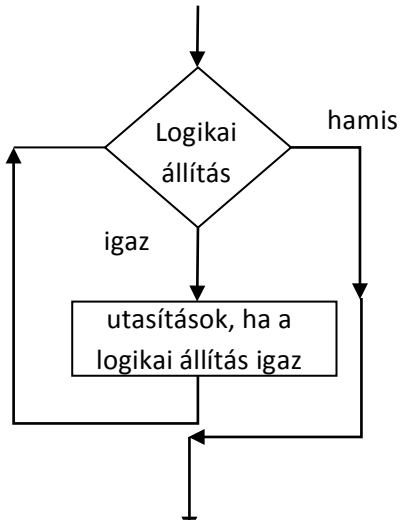
Feltételes elágazást olyan esetekben használunk az algoritmus fejlesztés során, amikor valamilyen eldöntendő helyzetet, vagy állapotot kell vizsgálni és arra a megfelelő módon kell reagálni. Ha logikai állítás igaz, akkor azon az ágon halad tovább a program végrehajtás, amely az „igaz” állapothoz tartozik és végrehajtja az ezen az ágon található utasításokat. Ha a logikai állítás „hamis” akkor a program futása az előzőtől egy eltérő ágon halad az ott található utasításokkal. A logikai állítást egy rombuszban szokás megadni, az egyik oldali csúcsából az igaz a másik oldaliból a hamis állapothoz tartozó folyamat indul ki.

Ciklusszervező utasítások

Ciklusszervező utasítások azt a célt szolgálják az algoritmizálás és a program fejlesztés során, amikor ugyanazokat az utasításokat többször, gyakran akár több ezerszer is végre kell hajtanunk.

Elöltesztelő ciklusszervező utasítás

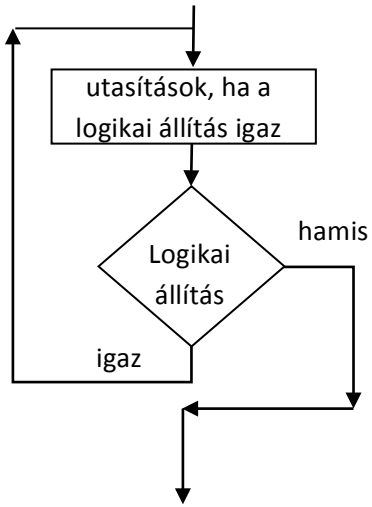
Az elöltesztelő ciklusszervező utasítás lényege, hogy a ciklusban maradás feltételét a ciklus kezdetén vizsgálja. Ha a logikai állítás igaz, akkor a ciklusba foglalt utasításokat végrehajtja, majd visszatér a logikai állítás vizsgálatához és újra ellenőrzi azt. Ha a logikai állítás továbbra is igaz, akkor újra végrehajtja azokat az utasításokat, amiket az előbbiek során már végrehajtott. Ha viszont a logikai állítás hamis, akkor a program a ciklusszervező utasítást követő következő utasításra lép. A folyamatábrában a kialakítása jobbra látható a szöveg mellett.



Hátultesztelő ciklusszervező utasítás

A hátultesztelő ciklusszervező utasítás lényege, hogy a ciklusban maradás feltételét a ciklus végén vizsgálja. A ciklusba foglalt utasításokat először végrehajtja, majd a ciklushoz tartozó logikai állítást vizsgálja. Ha a logikai állítás igaz, akkor visszatér a ciklusba foglalt utasítások elejéhez és megkezdi az utasítások újbóli végrehajtását. Ha viszont a logikai állítás hamis, akkor a program a ciklusszervező utasítást követő következő utasításra lép. A folyamatábrában a kialakítása jobbra látható a szöveg mellett.

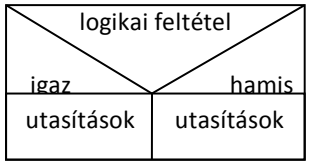
Fontos megjegyezni azt, hogy a hátultesztelő ciklusszervező utasításba foglalt utasítások egyszer mindenképpen végrehajtásra kerülnek.



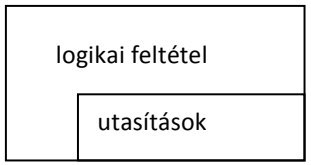
1.4 Struktogram

A struktogram hasonlóan a folyamatábrához grafikusan igyekszik a tervezett program algoritmusát leírni, csak azt jóval tömörebb formában teszi. A struktogram a folyamatábra terjedelmes kialakítását igyekszik csökkenti azzal, hogy az élek eltűnnek ebből a szerkezetből és csak egyszerű síkidomok maradnak meg egymásba ágyazva. A struktogram egymásba ágyazott téglalapokból és esetlegesen rész háromszögekből épül fel. Az alábbiakban a struktogramok legfontosabb részeit mutatjuk be.

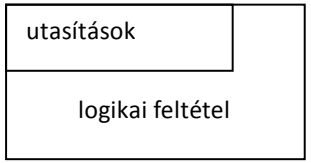
Feltételes elágazás



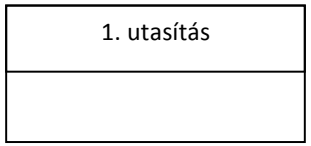
Elöltesztelő ciklus



Hátultesztelő ciklus



Szekvencia



1.5 Pszeudokód (mondatszerű leírás)

Definíció: A pszeudokód egy nem formális, magas szintű leírása egy számítógépes program vagy algoritmus lényegesebb részeinek.

A pszeudokód a programok mondatszerű leírását teszi lehetővé, ahol a különféle programozási eszközöket, például feltételes elágazás vagy ciklusszervezés szavakkal adjuk meg. A legegyszerűbb egy példán keresztül bemutatni ezt az eszközt.

például: Az alábbi program mondatszerű leírását adja annak a rövid programnak, ami egytől tetszőlegesen megadott számig *(n)* összegzi a páros számokat.

Program:

Be: n

összeg=0

Ciklus i=1-től n-ig
ha *i* páros akkor *összeg:=összeg* + i Ciklus vége

Ki *összeg*

Program vége.

1.6. Gyakorló feladatok, kérdések

Mi az algoritmus definíciója?

Mi az algoritmus célja, miért hozunk létre algoritmusokat?

Mondjon példákat matematikai problémák algoritmikus megoldására.

Milyen algoritmust leíró eszközöket ismer?

Mi a folyamatábra?

Milyen matematikai alakzathoz hasonlítható a folyamatábra?

Sorolja fel a folyamatábrák legfontosabb elemeit.

Milyen folyamatábra elemmel adjuk meg az algoritmikus probléma kimeneteit és bemeneteit?

Rajzolja fel a feltételes elágazási utasítás folyamatábráját.

Rajzolja fel az elöltesztelő ciklusszervező utasítás folyamatábráját.

Rajzolja fel a hátultesztelő ciklusszervező utasítás folyamatábráját.

Készítsen folyamatábrát tetszőleges háromszög területének kiszámítására Heron összefüggésével.

Készítsen folyamatábrát tetszőleges másodfokú egyenlet megoldásainak kiszámítására a megoldóképlet felhasználásával.

Hogyan épül fel egy struktogram?

Rajzolja fel a feltételes elágazási utasítás struktogramját.

Rajzolja fel az elöltesztelő ciklusszervező utasítás struktogramját.

Rajzolja fel a hátultesztelő ciklusszervező utasítás struktogramját.

Mi a pszeudokód?

2. fejezet

2. A programfejlesztő rendszer környezet, kezdő lépések *C#* nyelvben

2.1 Rövid történeti összefoglaló a programfejlesztésről

A programjainkat, amikor azokat létrehozzuk, valamilyen programnyelven kell megírnunk, ami mindig valamilyen szöveges állományformátumú file-ként jelenik meg a fejlesztés során. Ezt a szöveges állományt szokás forráskódnak nevezni. A programnyelvtől függően az állomány kiterjesztése lehet *.pas, *.c, *.ccp, *.bas és még számos más megnevezés.

A digitális számítógépek megjelenésének korai szakaszában, ami a huszadik század negyvenes éveinek végét és az ötvenes éveket jelentette, a programozást közvetlenül gépi nyelven kellett elvégezni, ami erősen hardware függő volt és komoly ismereteket feltételezett magáról a programozott számítógép processzoráról, memóriaszervezéséről, valamint a háttértárak és perifériák központi egységhez (CPU) való kapcsolódásáról.

A hatvanas évek közepétől kezdődően számos különféle programnyelvet dolgoztak ki, attól függően, hogy a fejlesztett program milyen célokat szolgált a későbbiekben. Ilyen nyelvek voltak a Fortran, C, Basic, Pascal, Clipper, dBase, stb... Ezek a nyelvek már biztosították azt a lehetőséget, hogy különféle szubrutinokat (eljárásokat és függvényeket) írjunk, amelyek lehetővé tették a fejlesztendő forrás kód strukturálását. Innentől számíthatjuk a strukturált programozás és programfejlesztés kezdeteit. A programok bonyolultságának növekedésével egyre inkább nyilvánvalóvá váltak a strukturált programfejlesztés hiányosságai.

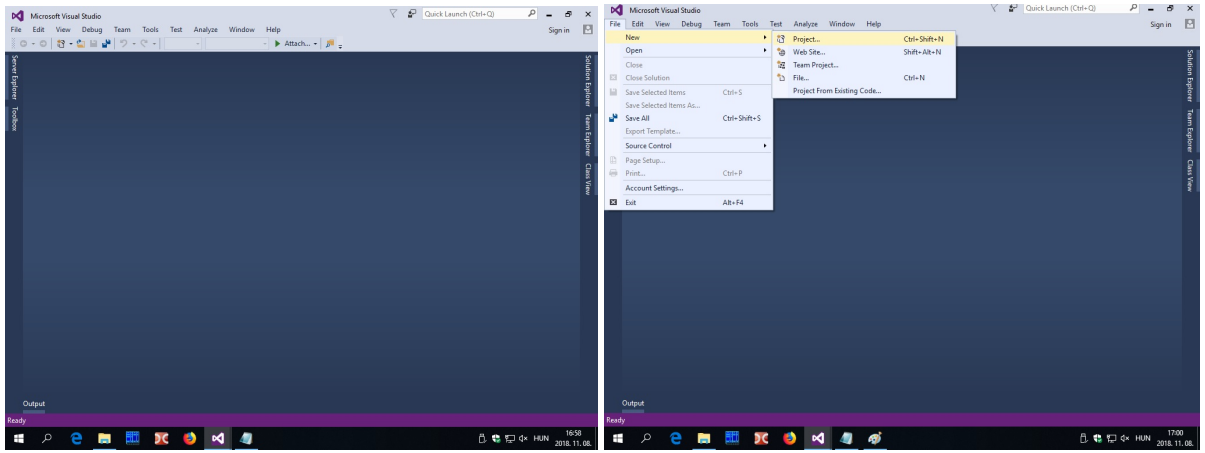
A hetvenes évek végétől kezdődően az objektum orientált programfejlesztés és programozás egyre nagyobb teret nyert. Számos különféle új vagy részben új programnyelv jelenik meg. Ezek közül a legfontosabb időrendi sorrendben is a C programozási nyelv alapján tovább fejlesztett és kidolgozott C++. Ebben a nyelvben jelenik meg az osztály fogalma, amiből az objektum származik, továbbá az osztályok közötti funkciók és tulajdonságok öröklődése. A C++ nem teljes mértékben objektum orientált nyelv, mivel lehetőség van arra, hogy teljes mértékben strukturált módon osztályok létrehozása nélkül fejlesszünk benne programokat. A nyolcvanas évek végétől pedig fokozatosan jelentek meg olyan további programnyelvek, mint például a Java, amely már teljes mértékben objektum orientált nyelv.

2.2 Visual Studio fejlesztői környezet rövid ismertetése

A Visual Studio a Microsoft integrált szoftver fejlesztői környezete, amelynek segítségével különféle programnyelveken tudunk kódot fejleszteni. A programfejlesztés első lépéseként egy projektet kell létrehozni, amely összefogja az összes olyan állományt, amelyre a fejlesztendő programunknak szüksége van. A projekt maga is egy állomány, amit a Visual Studio-val tudunk létrehozni, módosítani, menteni, megnyitni a fejlesztési folyamat során. A Visual Studio-val és a programozással ismerkedők első lépésben a projektet felfoghatják egyfajta tároló, illetve más szóval konténer állománynak, amelyben megtalálhatják azokat a további állományokat, amelyek a fejlesztendő szoftverhez tartoznak.

Az *1. ábra* a Visual Studio szoftverfejlesztő környezetének programablakát mutatja. Új projektet a File menü New almenüpontjával tudunk létrehozni. A New almenüpont első almenüpontja a Project menüpont, amely a 2. ábrán látható párbeszédablakot aktiválja.

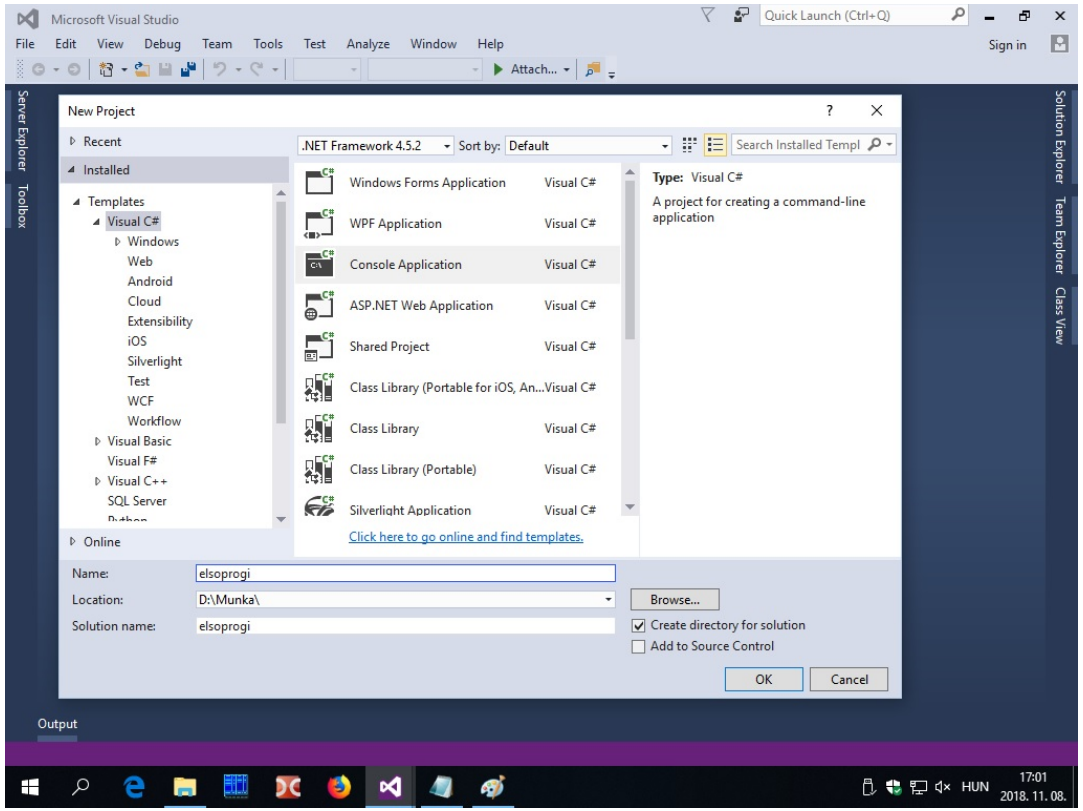
1. ábra



A projekt létrehozása során kell kiválasztani azt a programnyelvet, amelyben a fejlesztendő programot szeretnénk megírni. Számos nyelv közül lehet választani ezek, a C++, C#, F#, JavaScript, Basic. A programnyelv megválasztására a Project almenüpont aktiválásakor megjelenő párbeszédablak bal oldalán látható listadobozban van lehetőség. A *2. ábrán* látható, hogy a sablonok (Templates) közül a Visual *C#* sablont választottuk.

A nyelv kiválasztása után kell megadnunk azt, hogy milyen típusú alkalmazást szeretnénk fejleszteni. Számos lehetőség áll rendelkezésre, amelyeket a középső listadobozban láthatunk.
Most a konzolalkalmazást (Console Application) választjuk. Végül a projekt létrehozása során a Visual Studio egy olyan sablont (template) generál, amivel segíti a programozó munkáját. A generált sablonban olyan előre megírt utasítások láthatók, amelyek minden konzolalkalmazásnak részét kell képezze.

2. ábra


Microsoft Visual Studio
File Edit View Debug Team Tools Test Analyze Window Help
Quick Launch (Ctrl+Q)
Sign in
Attach...
Server Explorer
Toolbox
Solution Explorer
Team Explorer
Class View
New Project
Recent
Installed
Templates
Visual C#
Windows
Web
Android
Cloud
Extensibility
iOS
Silverlight
Test
WCF
Workflow
Visual Basic
Visual F#
Visual C++
SQL Server
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templ
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Application Visual C#
Shared Project Visual C#
Class Library (Portable for iOS, An... Visual C#
Class Library Visual C#
Class Library (Portable) Visual C#
Silverlight Application Visual C#
Click here to go online and find templates.
Type: Visual C#
A project for creating a command-line application
Name: elsoprogi
Location: D:\Munka\
Solution name: elsoprogi
Browse...
Create directory for solution
Add to Source Control
OK
Cancel
Output
HUN
17:01
2018. 11. 08.

2.3 C# NYELV

A konzol alkalmazás létrehozása során a következő sablont hozza létre a Visual Studio. Ebben a rövid kódrészletben több olyan kulcsszó is látható, ami részletesebb magyarázatra szorul. A kódrészletben látható kapcsos zárójelek szerepe, hogy az adott kód blokk, például egy utasítás csoport egy ciklusban vagy feltételes elágazásban, vagy éppen egy függvény kezdetét és végét jelöljék a programozó és a fordítóprogram (compiler) számára. A *C#* ezt a szintaktikai formalizmust és még sok minden mást is a C programozási nyelvből örökölte.

```
using System;
namespace elsoprogi
{ class Program
    { static void Main()
        {}
    }
}
```

A using kulcsszó segítségével lehet különféle modulokat a programunkban felhasználni. A modulokról azt kell tudni, hogy olyan eszközkészletet bocsájt a programozó rendelkezésére, amelyeknek segítségével a programfejlesztést jelentősen megkönnyíti. A modulok különféle osztályok, eljárások, függvények gyűjteménye, amelyeket felhasználhatunk a programjainkban. Ehhez egyetlen dolgot kell tennünk, a using kulcsszó mögött meg kell neveznünk azt a modult amelyből használni szeretnénk valamelyik előre beépített függvényt, vagy osztályt. Tehát a modulokat úgy is felfoghatjuk, mint egy szerszámosládát, amelyből kedvünkre válogathatunk a programjaink megírása során. A using kulcsszó ismerete fontos a további ismeretek elsajátítása szempontjából is.

A következő egyébként rendkívül fontos kulcsszó az **osztály** *(class)*, amire szintén nem lesz szükségünk a strukturált programfejlesztési ismeretek elsajátítása során. Viszont a fontossága miatt feltétlenül szót kell róla itt is ejtenünk. A class kulcsszó tulajdonképpen az OOP talán legfontosabb fogalma, mivel ennek segítségével tudunk osztályokat létrehozni. Egy objektum pedig az osztály egy példánya lesz majd.

A névtér *(namespace)* is az objektum orientált programozás témakörébe tartozó fogalom.

A névtér definíciója: A névtér osztályok egy csoportjának tároló (konténer) elemeként fogható fel, amelyek valamilyen közös jellemző, vagy valamilyen közös felhasználási cél érdekében kerültek az adott névtérbe. A névtér lehetővé teszi a forráskód jobb logikai felépítését azáltal, hogy az osztályok szemantikai felosztását különféle kategóriákba sorolja.

A névtér a *C#* nyelvben osztályok névvel ellátott csoportja, amelyek logikailag kapcsolódnak egymáshoz, mindenféle további speciális követelmény nélkül a file rendszerben való elhelyezkedésüket illetően. Annyi megkötés létezik csak, hogy a tartalmazó könyvtárnak a neve meg kell egyezzen a névtér nevével és a bennük lévő állományok nevei pedig az állományokban definiált osztály nevével kell megegyezzen.

A névtér lényege, hogy a létrehozott függvényeink elnevezése csak az adott névtérben használható és ugyanaz a függvény elnevezés egy másik névtérben újrahasználható. A névtér itt ugyan azt a nevet kapta, amit a projekt létrehozása során az aktuális projektünknek adtunk. A névtérre a továbbiakban nem lesz szükség, mivel az az objektum orientált programozás tématerületéhez tartozik és a továbbiakban ezzel nem foglalkozunk.

Ami viszont nagyon fontos a static void *Main()* függvény. Ez a függvény a program belépési pontja, azaz ennél a pontnál indul a különféle utasítások végrehajtása. A függvény neve Main utal arra, hogy ez a főprogram vagy elsődleges függvény, ahonnan minden más a programmal kapcsolatos feladat végrehajtás kiindul. A void jelentése ebben az esetben az, hogy a függvény semmilyen információt nem ad vissza a meghívása és futásának befejeződése után.

Itt nyomban felmerül a kérdés, hogy ha a *Main* a kiinduló függvény, akkor azt ki vagy mi hívja meg, illetve indítja el. Továbbá az is kérdésként merül fel, hogy a futás befejeződése után hová és kinek adhat vissza bármit is a *Main* függvény. Erre a választ a parancsvezérelt felületű operációs rendszerek (régi DOS, illetve napjainkban is használt Unix, Linux) használata során kapunk. Amikor egy programot indítunk parancsvezérelt felületen, akkor a programállomány nevét kell begépelnünk, aztán az Enter gomb megnyomásával elindítjuk a program végrehajtását. Parancsvezérelt felületen a programok a kimeneti információkat és a függvény végrehajtásának befejezésekor létrejövő eredményeket a képernyő konzol felületén jeleníti meg. Ha a *void* kulcsszót adjuk meg a *Main* függvénynek, akkor nem ad vissza és ír ki eredményeket a képernyőre a programfutás befejeződése után.

A következő program *Main* függvénye egy-egy kimeneti és bementi utasítást tartalmaz. A *Console* kulcsszó a valóságban egy objektumot takar, amelynek tagfüggvényeit (member functions) vagy más szóval metódusait *Console.FüggvényNév()* szintaktikai szerkezetben használhatjuk.

```
using System;
namespace elsoprogi
{ class Program
   { static void Main()
      { Console.WriteLine("Az első C# kódom, ami helyesen le is fut.");
        Console.ReadKey();
      }
   }
}
```

Először feltétlenül meg kell ismerni, hogy egy függvény általánosságban hogyan néz ki. Egy függvénynek neve van, amelynek egyetlen folytonos karakter sorozatból kell állnia. A függvény neve előtt meg kell adni azt az elemi adattípust, ami a függvény visszatérési értéke. Ezután a karaktersorozat után (függvény neve) egy zárójelpár következik, amely lehet üres vagy tartalmazhat valamilyen információt.

visszaadott_paraméter_típusa függvénynév(paraméterlista)

A függvényt magát úgy tudjuk felhasználni, azaz működtetni, hogy begépeljük a nevét, megadjuk a szükséges paraméterlistát a zárójelek között és a bezáró zárójel után egy pontosvesszővel zárjuk az adott utasítás sort a programunkban.

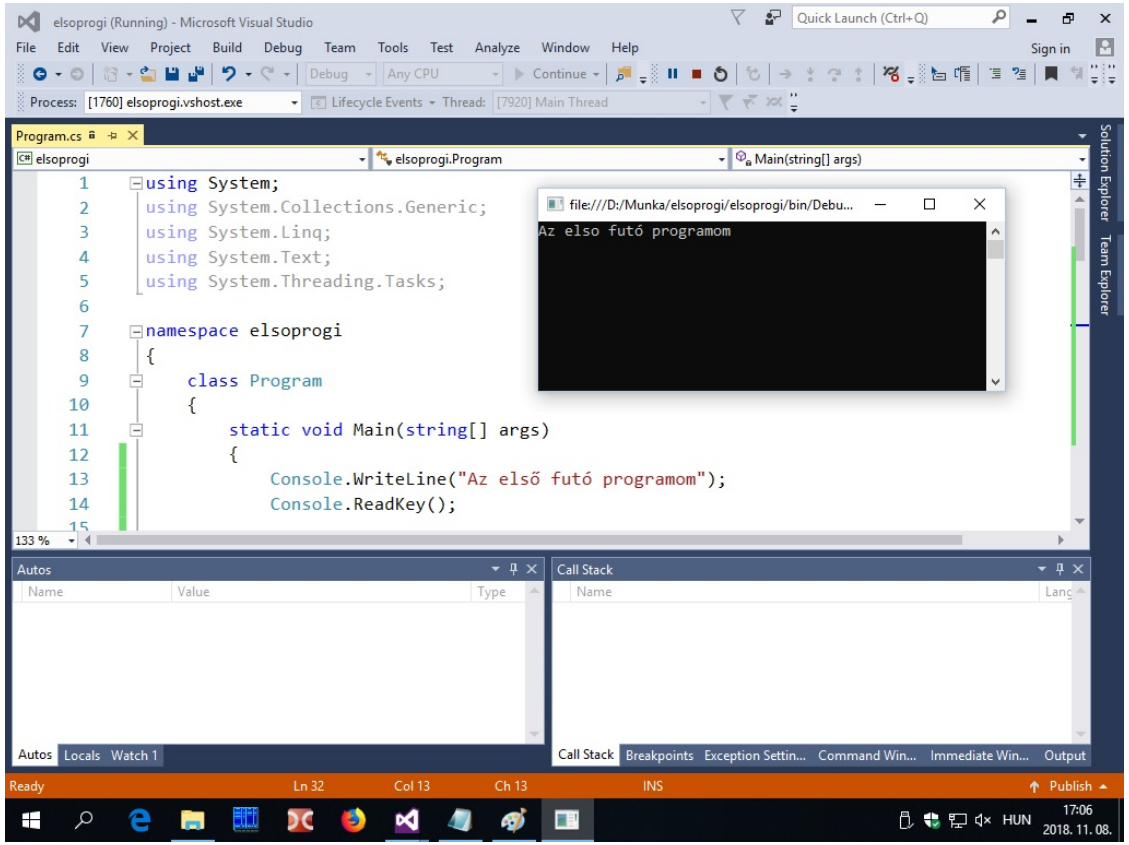
függvénynév(paraméterlista);

A *WriteLine()* függvény segítségével a szöveges képernyőre (konzol, Console) tudunk információkat kiírni. Ha a *WriteLine()* után álló zárójelpárban idézőjelek között valamilyen szöveget adunk meg, akkor azt a képernyőre a *WriteLine()* függvény kiírja. A fenti példában az idéző jelek között a következő szöveg áll. *"Az első C# kódom, ami helyesen le is fut."*

A második sorban egy újabb függvényhívás látható, ami a ReadKey() függvényt tartalmazza. Ez a függvény megállítja a program futását a meghívásának helyén és várakozik egy billentyű lenyomásra. Ha egy tetszőleges billentyűleütés történik, akkor a program végrehajtása folytatódik tovább. Itt most azért használjuk, mivel a program futását megállítjuk, hogy a képernyőn lássuk, hogy mit írt ki az előző sorban meghívott *WriteLine()* függvény. Ha nem használnánk a *ReadKey()* függvényt, akkor a program futása folytatódna és mivel más utasítás vagy függvény nem szerepel a forráskódban így a program futása befejeződne, a programhoz rendelt alkalmazás ablak pedig bezáródna.

A *3. ábrán* egy kódrészlet látható a Visual Studio fejlesztői környezetben, továbbá a program futásának eredménye is látható egy különálló fekete hátterű ablakban. Említést érdemel pár nyomógomb a Visual Studio fejlesztői környezetének menüsorában, amelyek a programfutással kapcsolatosak. Ilyen a program futását felfüggesztő, valamit a futást megállító nyomógombok. Ezek a Help menü pont felirata alatt, attól jobbra láthatók.

3. ábra


elsoprogi (Running) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Sign in
Debug
Any CPU
Continue
Process: [1760] elsoprogi.vshost.exe
Lifecycle Events
Thread: [7920] Main Thread
Program.cs
elsoprogi
elsoprogi.Program
Main(string[] args)
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace elsoprogi
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine("Az első futó programom");
Console.ReadKey();
file:///D:/Munka/elsoprogi/elsoprogi/bin/Debu...
Az elso futó programom
133 %
Autos
Name
Value
Type
Call Stack
Name
Autos Locals Watch 1
Call Stack Breakpoints Exception Settin... Command Win... Immediate Win... Output
Ready
Ln 32
Col 13
Ch 13
INS
Publish
HUN
17:06
2018. 11. 08.
Solution Explorer
Team Explorer

2.4. Gyakorló feladatok, kérdések

Mivel, milyen módon, illetve eszközökkel lehet programokat létrehozni?

Mi a forráskód?

Milyen állománytípusba sorolhatók a forráskódot tartalmazó állományok?

Soroljon fel néhány programnyelvet.

Sorolja fel néhány programnyelv forráskód állomány kiterjesztését.

A számítástechnika fejlődésének korai szakaszában milyen módon lehetett a számítógépeket programozni?

Milyen programnyelvek fejlődtek ki a hatvanas években.

Mi a neve a hatvanas-hetvenes évek programfejlesztési módszertanának?

Foglalja össze mi a strukturált programozás lényege?

Mi a szubrutin?

Mi a célja a szubrutinnak?

Mi okozta strukturált programfejlesztés hiányosságainak felismerését?

Mi a neve a hetvenes évek végén megjelenő a strukturált programfejlesztést felváltó módszertannak?

Melyik az a programnyelv, amelyben az elsők között jelent meg az objektumorientált fejlesztés lehetősége?

Mi az a fogalom, ami a legfontosabbnak tekinthető az objektumorientált fejlesztésben?

Melyik programnyelv az, amelyik teljesen objektumorientált és melyik részben objektumorientált nyelv?

Mi a Visual Studio?

Mi a programfejlesztés első lépése a Visual Studio fejlesztői környezetben?

Mi a projekt a Visual Studio fejlesztői környezetben?

Melyik menü és almenü pont segítségével lehet új projektet létrehozni?

A projekt létrehozása során milyen programnyelveket lehet kiválasztani? Soroljon fel néhányat.

Mi a sablon és mi a célja?

Minek a megnevezése a template?

Mi a kapcsos zárójel szerepe a *C#* programnyelvben?

Hozzon létre egy konzol alkalmazást a megfelelő sablon (template) segítségével.

Mi a using kulcsszó szerepe?

Mi a namespace (névtér) definíciója?

Mi a namespace (névtér) szerepe?

Mi a class (osztály) kulcsszó?

Mi a *C#* programok belépési (kiinduló) pontja?

Mi a void kulcsszó szerepe?

Hogyan néz ki általánosságban egy függvény?

Milyen módon tudjuk hívni (használni) a függvényt?

3. fejezet

3. Elemi típusok, adatszerkezetek, operátorok

A számítógépeken futó programok működése során a legtöbb esetben szükséges van olyan adattároló helyekre, amelyekben átmeneti jelleggel valamilyen információt tárolunk. Ezek az információk a programok működéséhez, azok funkcionális céljaihoz tartoznak. Ezeket az átmeneti adattároló helyeket memória rekeszeknek, illetve memória változóknak nevezzük. A memóriaváltozó célja tehát, hogy átmenetileg, a programjaink futási ideje alatt valamilyen információt, adatot vagy valamilyen numerikus értékeket tároljunk benne a nagysebességű hozzáférés céljából. A nagysebességű hozzáférés, mint feltétel nagyon fontos, mivel nem csak átmeneti, hanem hosszabb időre is tudunk adatokat tárolni például a háttértárakon (merevlemezes, USB, DVD meghajtók), de azok hozzáférési ideje nagyságrendekkel nagyobb valamint a sebessége nagyságrendekkel kisebb, mint az operatív táraké (RAM).

A memóriaváltozók az alábbi fontos jellemzőkkel, tulajdonságokkal rendelkeznek.

– A memóriaváltozónak van neve.

– A memóriaváltozók mindig rendelkeznek valamilyen típussal.

– A memóriaváltozónak mindig van valamilyen értéke, tartalma.

– Továbbá nem utolsó sorban a memóriaváltozóknak címe is van.

Változó név: A névre azért van szükségünk, hogy segítségével tudjunk az adott memóriatárolóra hivatkozni, azaz a név segítségével tudunk az adott memória területre írni, illetve onnan olvasni. A nevet mi adjuk neki, ez elvben akármi lehet, de több megkötés is létezik, mégpedig

– Egyetlen folytonos karakter sorozatból kell állnia, azaz nem állhat több szóból, magyarán a szóközt nem használhatjuk memóriaváltozók neveinek megadására.

– A memóriaváltozó neve csak olyan karaktereket tartalmazhat, amely az angol nyelvű abc-ben megtalálható, például a magyar ékezetes karaktereket nem használhatjuk. De a különféle műveleti jelek, illetve a különféle elválasztó jelek (pont, vessző, pontos vessző) sem használhatók változó nevek megadására.

Változó típusa: A változónak a neve mellett típusa is van, ez egyszerűen szólva azt jelenti, hogy mit tartalmazhat az a memóriaváltozó, milyen típusú információt helyezhetünk el benne. Lehet például a változó típusa egész szám, nem egész (lebegőpontos) szám, karakteres, vagy logikai típusú. Ezeket a típusokat elemi adattípusoknak is nevezik, mivel ezekből összetettebb adattípusok is felépíthetők, mint például az idő, vagy a dátum.

Változó értéke, tartalma: Változókat azért hozunk létre, hogy azok a futó program számára valamilyen információt, adatot átmenetileg tároljanak. Ezek az adatok típusuktól függően lehetnek egyszerű számok, karakterek vagy valamilyen összetettebb szerkezetű adat.

Változó címe: A változó címe az a hely az operatív tárban, ahová a változó tartalma (értéke) elhelyezésre került. A változó címe értelemszerűen mindig csak egész szám lehet.

Változók létrehozásának, más szóval definiálásának szigorúan előírt szintaktikai szabályai vannak. Először a típus azonosítót kell megadni (például **int** vagy *double*), ami meghatározza azt, hogy az adott változó milyen adattípust tartalmazhat a továbbiakban. Ezután egy vagy több szóközzel és vesszővel elválasztva a változók nevei (azonosítói) következnek, betartva a korábban említett a változó nevek megadásával kapcsolatos megkötéseket. Az alábbi kiemelésben a változók létrehozásának általános szintaktikai szabálya látható.

típus azonosító *változó1, változó2, változó3, ...;*

A változó definiálása egyben a megfelelő tárterületet is lefoglalja a memóriában a változó számára. Ezt a tárfoglalási módot statikus tárfoglalásnak is nevezik, mivel a program indulásának pillanatától ez a memória terület lefoglalásra kerül és csak a program futásának végével szabadul fel újra, azaz addig más programok vagy az operációsrendszer nem használhatja azt. A változók definiálása után szintén pontos vesszővel kell lezárni az adott kódsort, hasonlóan a programutasításokhoz.

A *C#* programnyelvben változót bárhol létrehozhatunk a forráskódban a különféle függvényeken belül. Ez azt jelenti, hogy nincsen előírt szabálya annak, hogy a forráskódban hol kell elhelyezni a változó definíciókat. Ez a szabadság talán kellemesnek tűnhet az első pillanatban, de hosszabb, komplexebb programokban nem célszerű ezt követni, mivel erősen csökkenti a későbbi forráskód értelmezést (olvashatóságot). Sokkal jobb az a megközelítés, hogy mindjárt a függvények nevének megadása után a nyitó kapcsos zárójelet követően felsoroljuk az adott függvényben használt összes változót. Például a Pascal nyelvben ez az előbb említett szabadság nem létezett, azaz ott szintaktikailag szigorúan előírt helyen kellett a változókat létrehozni.

3.1 Elemi adattípusok

A *C#* programnyelvben a következő alapvető változó típusok léteznek, az alábbiakban csak a fontosabb változó típusokat sorolom fel. Az alábbi típusazonosítókban szerepel az s és u betűk, mint kezdőbetű, amelyeknek jelentése s=signed és u=unsigned. Tehát az „s” azt jelöli a byte típusban, hogy előjeles, amíg az „u” azt, hogy előjelnélküli az adott típus.

byte, sbyte	1 byte-os (8 bit), előjelnélküli és előjeles egész számoknak
short, ushort	2 byte-os (16 bit), előjeles és előjelnélküli egész számoknak
int, uint	4 byte-os (32 bit), előjeles és előjelnélküli egész számoknak
long, ulong	8 byte-os (64 bit), előjeles és előjelnélküli egész számoknak
float	4 byte-os (32 bit), előjeles valós számoknak
double	8 byte-os (64 bit), előjeles valós számoknak
char	2 byte-os (16 bit), helyfoglalású, karakter típus
string	karakter, vagy szöveges típus

Példák változók definiálására:

int elso;

long a, b, c;

double x,y;

string szoveg;

3.2 Operátorok

Az operátorok célja, hogy különféle műveleteket végezzünk a segítségükkel, például összeadhatunk két számot, esetleg összeadhatunk két szöveget. Ez utóbbi pontosabban azt jelenti, hogy a két szöveget összefűzzük. De arra is lehetőségünk van az operátorok felhasználásával, hogy összehasonlítsunk különféle adatokat. Alapvetően az operátorok két fontosabb csoportra bonthatók.

– egyoperandusú operátorok.

– kétoperandusú (bináris) operátorok.

Elsőként az operandus fogalma szorul magyarázatra. Az operandus a különféle műveletek (operációk) alanya. Az operátorok tehát valamilyen műveletet vagy vizsgálatot végeznek el az operandusokon. Az egy operandusú operátorok tehát egy adaton végeznek el valamilyen műveletet, ilyen lehet például a logikai állapot negálása. A kétoperandusú (bináris) operátorok pedig két adattal hajtanak végre valamilyen műveletet, ilyen például a négy algebrai alapművelet vagy a logikai és valamint a logikai *vagy* művelet.

Az operátorok kapcsán felmerülő másik fontos kérdés a művelet végrehajtási sorrend. Ezt szokás idegenszóval precedenciának is nevezni. Például a következő algebrai műveletekben először a szorzást hajtjuk végre, aztán az összegzést és a legvégén az értékadást. Ez azt jelenti, hogy a szorzási művelet magasabb precedencia szinten áll és csak ennek végrehajtása után lehetséges az alacsonyabb szintű műveletek végrehajtása.

$$x=a+b*c$$

Ellenben ha egy zárójelpárral módosítjuk az előbbi összefüggést az alább következő módon, akkor a művelet végrehajtás sorrendje megváltozik. Ebben az esetben először az összeadási műveletet, majd aztán a szorzási műveletet hajtja végre a program. Ennek oka az, hogy a zárójelpár, mint művelet magasabb precedencia szinten helyezkedik el, mint a szorzási művelet.

$$x=(a+b)*c$$

Fontos kitérni az egyenlőségjel (operátor) kérdésére is, mivel a hétköznapi életben az egyenlőség két egymástól eltérő jelentése könnyen összemosódik. Az egyenlőség kérdése alapvetően két különféle kontextusban merülhet fel.

– értékadó egyenlőség (matematikában, Pascal nyelv :=, C szintaktikájú nyelvekben =)

– logikai állapotot vizsgáló egyenlőség (matematikában, Pascal nyelv =, C szintaktikájú nyelvekben ==)

Létezik viszont olyan programnyelv ilyen például a Basic, ahol szintaktikailag nem különíti el az egyébként különböző egyenlőség kérdését. Természetesen a Basic nyelvben is elkülönül az értékadó és a logikai állapotot vizsgáló egyenlőség, de mindkettőt egy szimpla egyenlőségjellel (=) lehet megadni. Szintaktikailag a nyelv tehát nem különbözteti meg a két különféle egyenlőséget, hanem a felhasználás helye alapján különíti el a Basic, hogy az adott helyen értékadásról vagy logikai állapot vizsgálatáról van-e éppen szó.

Műveleti szint	Operátor megnevezése	Operátor
1.	legmagasabb szintű operátorok	(), [], a++, a--, new, sizeof
2.	egyoperandusú operátorok	+, -, !, ~, ++a, --a
3.	multiplikatív operátorok	*, /, %
4.	additív operátorok	+, -
5.	bináris eltolás	<<, >>
6.	relációs operátorok	<, >, <=, >=
7.	egyenlőség logikai állapot vizsgálatára	==, !=
8.	és (and) operátor	&
9.	kizáró vagy (xor) operátor	^
10.	vagy (or) operátor	|
11.	és (and) op. logikai állapot vizsgálatára	&&
12.	vagy (or) op. logikai állapot vizsgálatára	||
13.	operátor logikai állapot vizsgálatára	? :
14.	értékadás	=

3.3. Példaprogram

Az itt bemutatásra kerülő példaprogramban három darab egész típusú változót használunk néhány operátor használatának, valamint a precedencia jelentőségének a bemutatására. A static void *Main()* függvényben lévő utasításokat egyesével lépésről lépésre nézzük végig. Először egy „eredmeny" nevű előjeles egész típusú (integer) változót hozunk létre, ahol a típusazonosító azt int kulcsszó. A változó létrehozása során az nyomban értéket is kap.

A következő lépésben a *Console.WriteLine()* függvény segítségével azt a mellékelt kísérőszöveggel együtt a képernyőre kiíratjuk, ez látható a 4. ábrán a fekete hátterű program ablak első sorában. A kiíratás során az összegzési operátor a szokásostól eltérő módon nem számokra, hanem karakterekre használtuk. Az "Az eredmény" szöveghez az összeadás operátora hozzáfűzi a kettes (2) karaktert. Mivel az *„eredmeny"* előjeles egész típusú ezért az összeadási művelet végrehajtása előtt, az karakter típussá konvertálódik, majd ezután kerül végrehajtásra az összegző operátor szöveg között.

A *main* függvény harmadik utasítássorában két egész típusú *(int)* változó lett létrehozva. A negyedik sorban három utasítás is látható, az első egy értékadó utasítás, amelyben az *"a"* nevű változó az egyes értéket veszi fel, majd azt követően a ++ operátorral az *"a"* változó értéke eggyel növekszik. A harmadik utasítás ebben a sorban egy újabb értékadó utasítás, ahol a *"b"* nevű változó átveszi az *"a"* nevű változó értékét. Ezután az ötödik sorban a Console.WriteLine() függvény segítségével a két változó értékét a képernyőre kiíratjuk, ami szintén látható a *4. ábrán* a fekete hátterű program ablak második sorában.

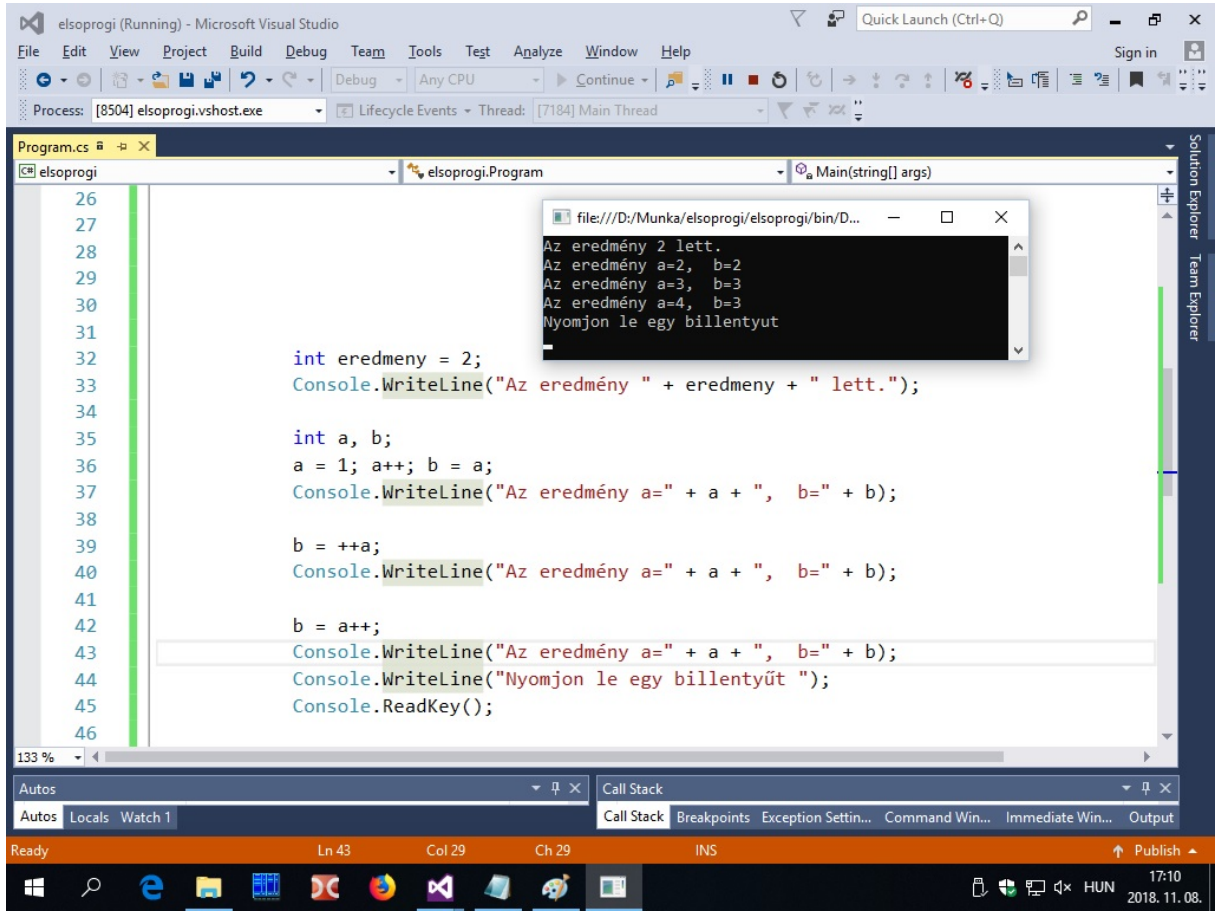
A *main* függvény hatodik és kilencedik sorai között a ++ operátor prefixes és postfixes alakjának használatát és működésükben meglévő különbségüket vizsgáljuk. A hatodik sorban a b=++a; utasítás először megnöveli az *"a"* nevű változó értékét, majd a *"b"* nevű változó átveszi az *"a"* nevű változó értékét az egyenlőség operátor hatására. A hetedik sorban kiírjuk az eredményeket.

A nyolcadik sorban a *b=a++;* utasítás először átadja az *"a"* nevű változó értékét a *"b"* nevű változónak, majd ezután megnöveli az *"a"* nevű változó értékét. Végül a kilencedik sorban újra kiírjuk az eredményeket a képernyőre.

```
using System;
namespace elsoprogi
{ class Program
    { static void Main()
        { int eredmeny = 2;
        Console.WriteLine("Az eredmény "+ eredmeny +" lett.");
        int a, b;
        a=1; a++; b=a;
        Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
        b=++a;
        Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
        b=a++;
        Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
        Console.ReadKey();
        }
    }
}
```

Az utolsó sor a *ReadKey()* függvényt tartalmazza. Ez a függvény, mint az korábban is látható volt megállítja a program futását a meghívásának helyén és várakozik egy billentyű lenyomásra, aminek következményeként a program ablak bezáródása előtt láthatjuk a kiírt eredményeket.

4. ábra


elsoprogi (Running) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Sign in
Debug
Any CPU
Continue
Process: [8504] elsoprogi.vshost.exe
Lifecycle Events
Thread: [7184] Main Thread
Program.cs
elsoprogi
elsoprogi.Program
Main(string[] args)
file:///D:/Munka/elsoprogi/elsoprogi/bin/D...
Az eredmény 2 lett.
Az eredmény a=2, b=2
Az eredmény a=3, b=3
Az eredmény a=4, b=3
Nyomjon le egy billentyut
int eredmeny = 2;
Console.WriteLine("Az eredmény " + eredmeny + " lett.");
int a, b;
a = 1; a++; b = a;
Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
b = ++a;
Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
b = a++;
Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
Console.WriteLine("Nyomjon le egy billentyűt ");
Console.ReadKey();
133 %
Autos
Autos Locals Watch 1
Call Stack
Call Stack Breakpoints Exception Settin... Command Win... Immediate Win... Output
Ready
Ln 43
Col 29
Ch 29
INS
Publish
HUN
17:10
2018. 11. 08.
Solution Explorer
Team Explorer

3.4. Gyakorló feladatok, kérdések

Mi a memória változó?

Melyek a memória változók fontosabb jellemzői, tulajdonságai?

A változó nevek megadásának milyen kötöttségeit ismeri?

Mit értünk egy változó típusa alatt?

Mi a változó értéke?

Mi a változó címe?

Adja meg általánosan, hogyan definiál egy változót.

Milyen elemi adattípusokat ismer?

Sorolja fel az egész típusokat.

Adjon meg egy példát egész típusú változó definiálására.

Adjon meg egy példát lebegőpontos típusú változó definiálására.

Ismertesse, hogy az egyes elemi típusoknak mekkora a helyfoglalása bitekben és byteokban.

Mi az operátor és mi a szerepe a programozási nyelvekben?

Mi az operandus?

Milyen módon csoportosíthatók az operátorok?

Mit értünk egy operandusú operátor alatt?

Soroljon fel legalább három darab egy operandusú operátor.

Mit értünk két operandusú operátor alatt?

Soroljon fel legalább öt darab két operandusú operátor.

Miért fontos a művelet végrehajtási sorrend?

Milyen operátorral tudja a művelet végrehajtás sorrendjét módosítani?

Az egyenlőségjel, mint operátor milyen helyzetekben merülhet fel?

Mi az értékadó egyenlőség operátora a C#, C, C++ nyelvben, mi Pascal nyelvben és mi Basic nyelvben?

Mi az egyenlőség állapotát vizsgáló operátor a C#, C, C++ nyelvben, mi Pascal nyelvben és mi Basic nyelvben?

Írjon programot, ami létrehoz egy egész és egy lebegőpontos változót, adjon értékeket nekik, írassa azokat ki azután, az egész változó értékét adja át a lebegőpontos változónak és újra írassa ki a változók értékeit.

4. fejezet

4. Konzol kimeneti, bemeneti utasításai és adatkezelése

A konzol alkalmazások lényege, hogy a felhasználó a programmal mindenféle interakciót a billentyűzeten keresztül tud kezdeményezni, és a program válaszait szöveges formában jeleníti meg a képernyőn.

4.1. Konzol kimeneti (output) utasításai

Console.Write()
Console.WriteLine()

A *Console.Write()* és a *Console.WriteLine()* függvények segítségével a szöveges képernyőfelületre tudunk kiírni. A két függvény között az a különbség, hogy a *WriteLine()* függvény a képernyőn a kurzort a következő sor elejére pozícionálja, míg a *Write()* függvény esetében a kiírt szöveg utolsó karaktere utáni hely lesz, a kurzor aktuális pozíciója. Ugyanez a hatás kiváltható a ”\n” vezérlőkarakter segítségével is. A *Console.Write()* és a *Console.WriteLine()* függvények utáni zárójelpárban egy *stringet* (karakter sorozatot) kell megadni. Az alábbiakban pár egyszerű példát lehet látni a két függvény használatára.

Példák:

```
Console.Write(”Visual C#”);

Console.WriteLine(” programfejlesztés”);

 int x=3; Console.Write(”x=”);

Console.WriteLine(x);
```

Az első sorban a *Console.Write()* függvény segítségével kiíratjuk a ”Visual C#” szöveget a képernyőre a kurzor pedig a # karakter mögött villog, várva a következő output utasítást. Az pedig a *Console.WriteLine()* függvény, ami a ” programfejlesztés” szöveget írja ki, ami egy szóközzel kezdődik. Ennek hatására végül a ”Visual C# programfejlesztés” szöveg jelenik meg a képernyőn.

A harmadik sorban két utasítás látható, ahol az első egy integer típusú változót hoz létre, amely a hármas éréket veszi fel. Ezután a *Console.Write()* függvény segítségével kiíratjuk a ”a=” szöveget a képernyőre a kurzor pedig az = karakter mögött villog, várva a következő output utasítást. Az pedig az *x* változó numerikus értékének kiírása lesz. A kiíratás során a függvény végrehajt egy típuskonverziót, mivel a *Write()* és a *WriteLine()* függvények paraméterként szöveges információt vesznek át. és ha az nem ilyen, akkor azt szöveges típusúra konvertálja.

4.2. Konzol bemeneti (input) utasításai

Alapvetően kétféle módon lehetséges a billentyűzetről adatokat beolvastatni a futó programokba.

Ezek a következők:

– direkt billentyűzetkezelés

– pufferelt billentyűzetkezelés

A pufferelt billentyűzetkezelés esetén a karaktereket egy átmeneti tárolóba olvassa be az erre a célra kidolgozott függvény mindaddig, amíg a gépelést le nem zárjuk egy *Enter* billentyű lenyomásával. Ezt az átmeneti tárolót szokás puffernek nevezni. A pufferelt billentyűzetkezelés lényege, hogy a beolvasott karaktersorozatot egyszerűbb módokon szerkeszthetjük, azaz a gépelés során nem azonnal kerülnek a karakterek a futó programnak átadásra. Ez a szerkesztés mindössze abból áll, hogy az esetlegesen helytelenül begépelt szöveget a *Backspace* billentyű segítségével törölhetjük. Amint a megfelelő input adatot begépeltük, a szerkesztés befejezését az Enter billentyű lenyomásával jelezzük.

A direkt billentyűzetkezelés alapvetően abban tér el a pufferelt billentyűzetkezeléstől, hogy minden egyes lenyomott karakter kódja azonnal átadódik a futó programnak, ahol azt feldolgozhatjuk. Ebben az esetben viszont a felhasználó részéről nem szerkeszthető a beírt adat, mivel az azonnal bekerül a futó programba.

A *Console.Read()* és a *Console.ReadLine()* függvények segítségével a szöveges képernyőfelületen keresztül, a billentyűzetről tudunk beolvasni tetszőleges karaktersorozatokat. Ezek a függvények pufferelt billentyűzetkezelést biztosítanak a programozó számára. A direkt billentyűzetkezelés *Console.ReadKey()* függvény segítségével valósítható meg.

Console.Read()

Console.ReadLine()

Console.ReadKey()

Nagyon fontos az a tény, hogy ez a három függvény csak karakteres formátumban tud a billentyűzetről adatokat olvasni, azaz típusos adatbevitelre nincsen lehetőség, ami a régebbi C, C++, Pascal, stb... programnyelvekben széles körben elterjed lehetőség volt. Ez a karakteres adatbevitel egyébként a céloknak tökéletesen megfelel, mivel így az általunk írt programnak semmit sem kell feltételezni a billentyűzet felől érkező input adatokról. Tehát nem áll fenn annak veszélye, hogy ha a felhasználó rosszul adja meg a bemeneti adatokat, akkor a programunk futása közben valamilyen hiba következik be.

4.3. Példaprogram

A példaprogram main függvényének első sorában egy *string* típusú változót, a második sorban pedig két integer típusú változót hozunk létre. A harmadik sorban kiírunk egy szöveget, ami mögött elkezdhetjük az adatbevitelt. A negyedik sorban a *Console.ReadLine()* függvény felhasználásával a billentyűzetről beolvassuk a felhasználó által gépelt adatot a beolvas nevű változóba. A *Console.ReadLine()* függvény tehát egy *stringet* ad vissza, amit utána karakterről karakterre fellehet dolgozni annak függvényében, hogy mi a célunk a begépelt adattal. Ebben a példaprogramban ez elmarad, feltételezzük, hogy a program használója helyesen csak számjegyeket gépelt be, így a beolvas nevű változó tartalma azonnal egészszámmá konvertálható.

A konverzióra több lehetőség is rendelkezésre áll, ezek az alábbi kód ötödik és hatodik sorában láthatók. Minden típusazonosító felhasználható olyan módon, mint ha az egy objektum lenne a megfelelő tagfüggvényekkel. Minden típusazonosítóhoz hozzátartozik a *.Parse()* függvény, ami egy *string*ből képes tetszőleges másik típusra konvertálni a *string* tartalmát. Hátránya, hogy csak *string*- ből képes valamelyik másik típusra konvertálni. A Convert segítségével viszont bármely típusról bármely másikra tudunk konverziókat végrehajtani. Az a változó a *.Parse()* a b változó pedig a *Convert.ToInt32()* függvényen keresztül kapja meg azt a numerikus értéket, amit a felhasználó begépelt.

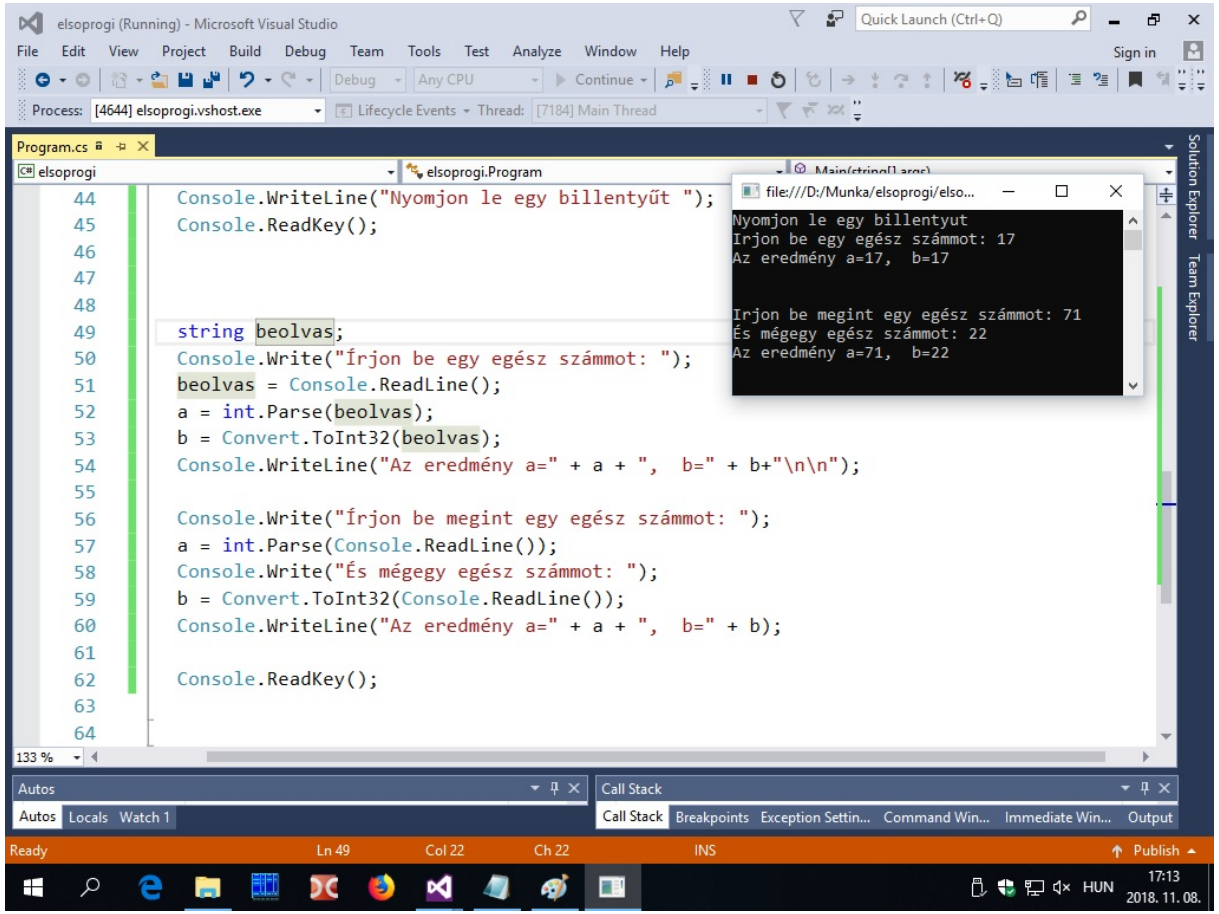
A hetedik sorban *Console.WriteLine()* függvény segítségével a szöveges képernyőfelületre kiírjuk az a és a b változó értékét. A b változó kiíratása után még két ”\n” vezérlőszekvenciát ír ki a program, amelynek hatására két üres sor után folytatódik tovább a konzolablak képernyőjén a ki és bemenetek megjelenítése.

```
using System;
namespace elsoprogi
{ class Program
   { static void Main()
        { string beolvas;
        int a, b;
        Console.Write(“Írjon be egy egész számot: “);
        beolvas= Console.ReadLine();
        a=int.Parse(beolvas);
```

```
            b=Convert.ToInt32(beolvas);
            Console.WriteLine("Az eredmény a=" + a + ", b=" + b+"\n\n");
            Console.Write("Írjon be megint egy egész számot: "); a=int.Parse(Console.ReadLine());
            Console.Write("És még egy egész számot: "); b=Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Az eredmény a=" + a + ", b=" + b);
            Console.ReadKey();
        }
    }
}
```

A kilencedik és a tizenegyedik sorban látható a két konverziós függvény tömörebb használata abban az értelemben, hogy a konverziós függvények argumentumában közvetlenül a Console.*ReadLine()* függvény került megadásra. Ez megtehető, mivel a *Console.ReadLine()* függvény visszatérési értéke egy *string* a két konverziós függvény, pedig *string* típusú paramétert vehet át meghívásukkor.

5. ábra



Az előbbiekben részletezett program futásának eredménye, valamint a forráskód fontosabb részei is láthatók az *5. ábrán*.

4.4. Gyakorló feladatok, kérdések

Ismertesse a képernyő konzol kezelésével kapcsolatos függvényeket.

Mi a különbség a *Console.Write()* és a *Console.WriteLine()* függvények között?

Mi a ”\n” vezérlőkarakter szerepe?

Milyen billentyűzet kezelési megoldásokat ismer?

Ismertesse a pufferelt billentyűzet kezelés fontosabb jellemzőit.

Alapértelmezésben milyen típusú adatokat olvas be a billentyűzetről a *Console.ReadLine()* és a *Console.Read()* függvények?

Miért van szükség általában típus konverzióra a billentyűzetről olvasott adatok kapcsán?

Mi a feladata a *.Parse()* függvénynek?

Mi a feladata a *Convert.ToInt32()* függvénynek?

Miben tér el a *.Parse()* függvény a többi konverziós függvénytől?

Írjon programot, ami beolvas a billentyűzetről két egész számot, kivonja az elsőből a másodikat és kiírja ezt a különbséget a képernyőre.

Írjon programot, ami beolvas a billentyűzetről két lebegőpontos számot, osztja az elsőt a másodikkal és kiírja a hányados értékét a képernyőre.

5. fejezet

5. Feltételes elágazások, logikai állítások és operátorok

5.1. Feltételes elágazásról általánosságban

Feltételes elágazást olyan esetekben használunk a program fejlesztés során, amikor az adott problémát leíró és megoldó algoritmusban a feladat végrehajtás folyamata elágazik, más szóval egy eldöntendő helyzetet kell elemezni. A döntési helyzetet logikai állítások segítségével fogalmazzuk meg, amelyeknek két állapota lehetséges, igaz vagy hamis. Ha logikai állítás igaz, akkor azon az ágon halad tovább a program végrehajtás, amely az „igaz” állapothoz tartozik és végrehajtja az ezen az ágon található utasításokat. Ha a logikai állítás „hamis” akkor a program futása az előzőtől egy eltérő ágon halad és végrehajtja az ott található utasításokat. A feltétel vizsgálatnak a kezelésére minden programnyelvben az if else szerkezetet használják, legfeljebb szintaktikailag létezik némi eltérés az egyes programnyelvek között. Az *if else* szerkezet még az Excel táblázatkezelőben is megtalálható függvény formátumú kivitelben, amit az Excel cellatartományában bárhol használhatunk.

Az alábbiakban néhány a programfejlesztés során előforduló példa látható eldöntendő algoritmikai és programozási helyzetre.

– Másodfokú egyenletek megoldásainak kiszámítása során a diszkrimináns előjelének vizsgálata. A döntési helyzet (pozitív a diszkrimináns, negatív a diszkrimináns). A nullát most a pozitív állapothoz soroljuk.

– Háromszög területének kiszámítása Heron képletével. A döntési helyzet (a megadott három számra fenn áll a háromszög egyenlőtlenség, a megadott három számra nem érvényes a háromszög egyenlőtlenség).

– A felhasználó által a *Console.ReadLine()* függvény segítségével begépelt karakterek vizsgálata abból a szempontból, hogy az átadható-e valamelyik numerikus típusú változónak. A döntési helyzet (csak számjegyeket tartalmaz a begépelt karaktersorozat, nem csak számjegyeket tartalmaz a begépelt karaktersorozat).

A programfejlesztés során előfordul, hogy az eldöntendő helyzet nem két, hanem többállapotú. Erre az esetre kétféle megoldás kínálkozik. Az egyik megoldás az, amikor az *if else* szerkezeteket egymásba ágyazzuk, a másik lehetőség a switch case szerkezet.

5.2. Feltételes elágazás szintaktikai szerkezete

A *C#* nyelv alapvetően C illetve C++ szintaktikájú tehát az if else szerkezet felépítése teljes mértékben megegyezik a C programnyelvben megismertekkel. Az *if* kulcsszó után zárójelek között kell megadni a logikai állítást, ami tetszőlegesen összetett szerkezetű lehet, amit a különféle logikai operátorok segítségével tudunk létrehozni. Az *if()* kulcsszó után kapcsos zárójelek között kell megadnunk azokat az utasításokat, amelyeket akkor szeretnénk végrehajtani, ha a logikai állítás igaz. Ha csak egyetlen utasítást szeretnénk megadni az igaz logikai állapotú ágban, akkor a kapcsos zárójel elhagyható, mivel az *if()* kulcsszó után következő első utasítás az *if()* ághoz tartozik.

```
if (logikai állítás)
{ utasítások;
}
else
{ utasítások;
}
```

Az *if else* szerkezet másik ága az else kulcsszóval kezdődik, ami után szintén kapcsos zárójelek között kell megadnunk azokat az utasításokat, amelyeket akkor szeretnénk végrehajtani, ha a logikai állítás hamis volt. Itt is igaz az, hogy ha csak egyetlen utasítást szeretnénk megadni a hamis logikai állapotú ágban, akkor a kapcsos zárójel elhagyható. A másik fontos megjegyzés az, hogy maga az else ág nem kötelező, abban az értelemben, hogy ha algoritmikailag nincsen rá szükség, azaz semmilyen utasításra sincsen szükség a hamis ágban, akkor az *else* ág egyszerűen elhagyható.

Az alább következő rövid programrészletekben az if else szerkezet használatára látható pár példa. Az elsőben az a nevű integer típusú változó abszolút értékét számítjuk ki, továbbá ha a változónak pozitív az értéke vagy nulla, akkor sárga színnel, ha a szám negatív volt, akkor zöld színnel írjuk ki a képernyőre. Az else ágban látható, hogy ha a változó negatív értékű, akkor az else ág első utasításában az előjelét megfordítjuk azzal, hogy a kivonás operátorát ebben az esetben egyoperandusú operátorként használjuk. Ez tulajdonképpen a mínusz egyel való szorzással ekvivalens.

```
if(a>=0)
{ Console.ForegroundColor = ConsoleColor.DarkYellow; Console.WriteLine(”a=” + a);
}
else
{ a=-a; Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine(”a=” + a);
}
```

Az alább következő rövid programrészletben az a nevű integer típusú változó paritását vizsgáljuk, ami azt jelenti, hogy megvizsgáljuk vajon az a nevű változó páros vagy páratlan szám-e. Erre a célra a maradékos osztás operátorát (%) felhasználva a változó értékét osztjuk kettővel. Ez az osztás pedig nem a hányadost, hanem a maradékot adja vissza. Ha a maradék nulla, azaz a szám páros akkor az if ágban lévő Console.WriteLine() függvény, egyébként az else ágban lévő Console.WriteLine() függvény kerül meghívásra.

```
if((a % 2) == 0)
{ Console.WriteLine(”az a nevű változó páros szám: ” + a);
}
else
{ Console.WriteLine(”az a nevű változó páratlan szám: ” + a);
}
```

5.3. Példaprogramok

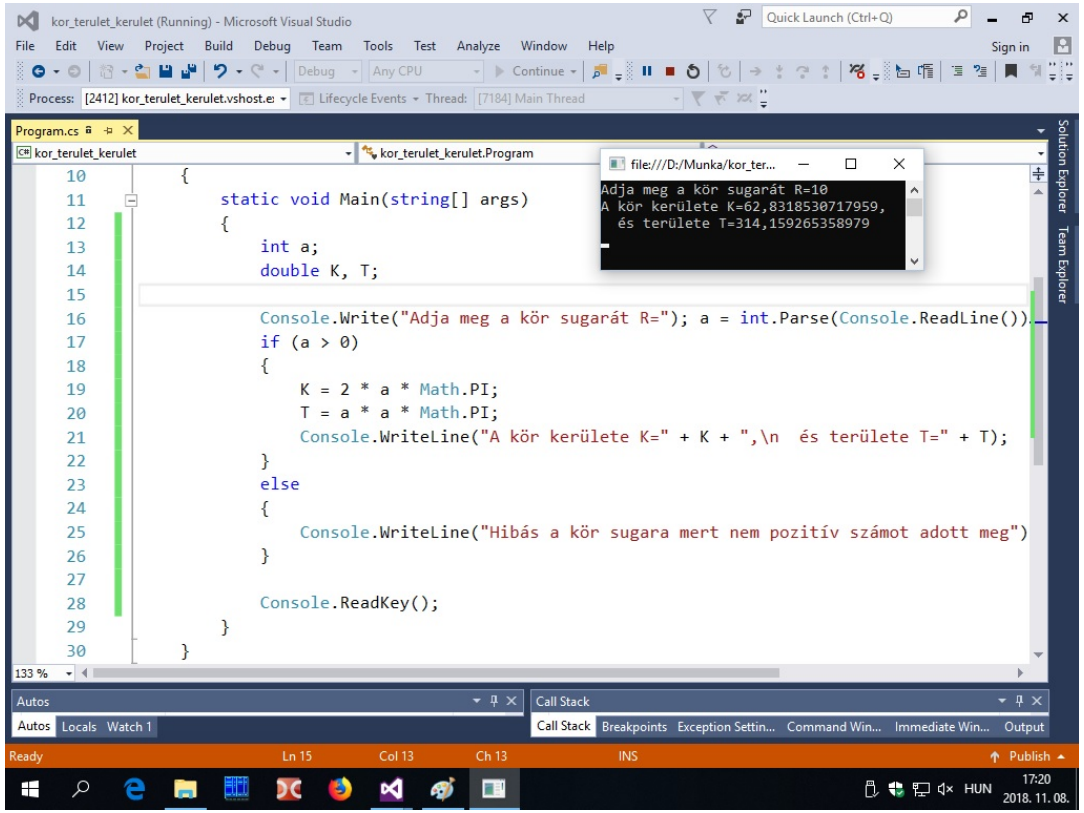
Az első példaprogram egy R sugarú kör kerületét és területét számítja ki, ahol a felhasználónak meg kell adnia a kör sugarát. A döntési helyzet ebben a feladatban az, hogy megállapítsuk, vajon a megadott szám pozitív-e, mivel a kör sugara csak pozitív szám lehet. Ha a szám pozitív, akkor kiszámítjuk ennek alapján a kör területét és kerületét.

A kör sugarának tárolására egy *integer* típusú változót (a), a kör területének és kerületének tárolására pedig két *double* típusú változót hozunk létre. A program harmadik sorában beolvassuk a kör sugarát az a nevű változóba.

A negyedik sorban vizsgáljuk az a nevű változó előjelét, amit az *if (a > 0)* utasítással vizsgáljuk. Ha az állítás igaz, akkor kiszámítjuk a kör területét és kerületét, majd kiírjuk a képernyőre. Ha viszont az állítás logikai értéke hamis, azaz az *else* ágban folytatódik a programvégrehajtás, akkor egy figyelmeztető üzenete írunk ki a képernyőre.

```
using System;
namespace kor
{ class Program
      { static void Main()
          { inta;
              double K, T;
              Console.Write("Adja meg a kör sugarát R="); a = int.Parse(Console.ReadLine());
              if (a > 0)
              { K = 2*a*Math.PI;
              T = a * a * Math.PI;
              Console.WriteLine("A kör kerülete K=" + K + ",\n és területe T="+ T);
          }
          else
          { Console.WriteLine("Hibás a kör sugara mert nem pozitív számot adott meg");
          }
          Console.ReadKey();
      }
  }
}
```

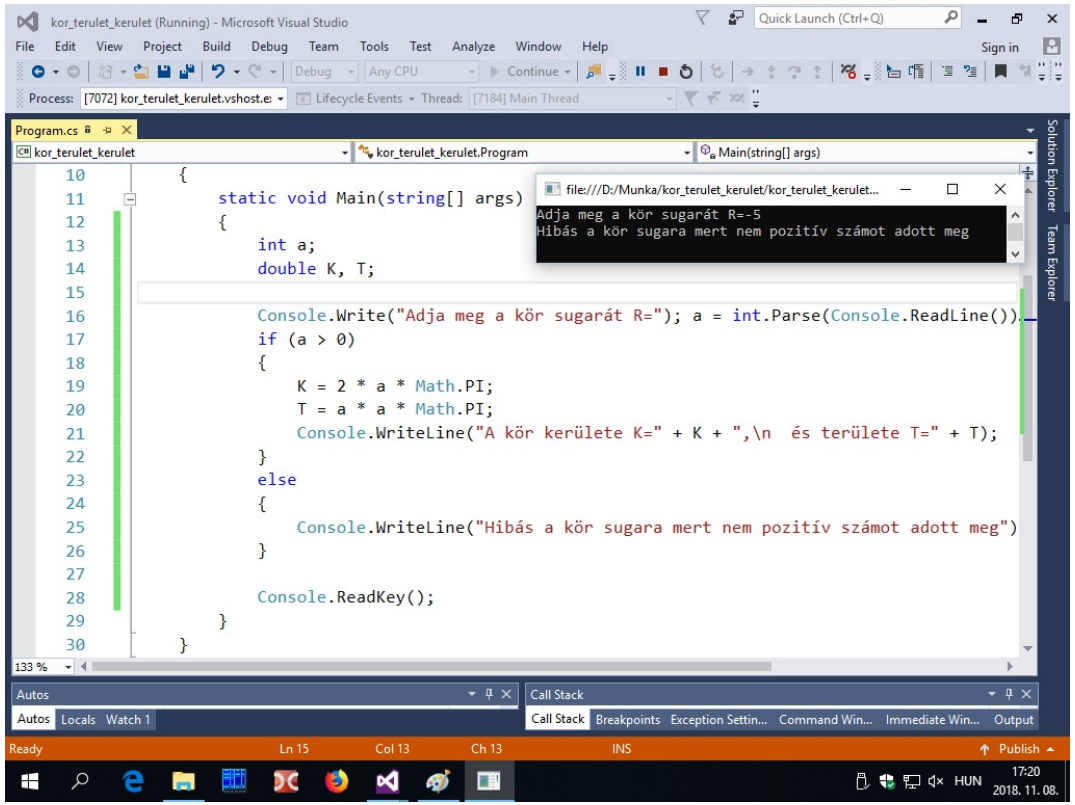
6. ábra



A *6. ábrán* látható a forráskód illetve egy kisebb fekete hátterű ablakban a futás eredménye. A kör sugarának megadott érték a tízes, amihez a program kiszámítja és kiírja a 10-es sugarú kör területét és kerületét.

A *7. ábrán* szintén látható a forráskód, illetve egy kisebb fekete hátterű ablakban a futás eredménye arra az esetre, amikor a kör sugarának negatív számot adunk meg. Ebben az esetben az *if (a > 0)* utasítás logikai állítása hamis, ezért az *else* ágban lévő *Console.WriteLine()* függvény hajtódik végre és a benne látható szöveg *"Hibás a kör sugara mert nem pozitív számot adott meg"* jelenik meg a képernyőn.

7. ábra



A második példa program a Héron képlet alapján számítja ki egy háromszög területét. A háromszög teljesen általános lehet, nem kell semmilyen speciális megkötést figyelembe venni. A Héron féle összefüggésben a háromszög három oldalára van szükségünk a terület nagyságának kiszámításához.

$$T = \sqrt{(s-a)(s-b)(s-c)s}\text{, ahol } s = \frac{a+b+c}{2}.$$

Ebben az esetben a döntési helyzetet a három megadott számnak együtt kell teljesítenie olyan módon, hogy a háromszög egyenlőtlenségnek teljesülnie kell, ami három reláció együttes kiértékelését jelenti.

$$a+b>c, a+c>b, c+b>a.$$

Ebben a programban találkozunk először egy speciális típuskonverzióval, amit a *C#* szintén a C programnyelvből örökölt. Ez egy olyan típuskonverzió, amit nem a programozó ad meg utasítás formájában, hanem maga a fordítóprogram készít olyan bináris kódot a forráskód alapján, amelyben egy újabb típuskonverzió jelenik meg. Ezt a típuskonverziót implicit típuskonverziónak nevezi a szakirodalom, mivel nem a programozó ad utasítást ennek végrehajtására.

Implicit típuskonverzió: Ha egy utasításban kizárólag egy adott típushoz tartoznak az utasításban lévő változók, akkor a változók közötti művelet eredményének típusától függetlenül az eredmény is olyan típusú lesz, mint a műveletben résztvevő változók típusa.

Az alábbi kódrészletben a két integer típusú változót osztunk egymással. Az osztási művelet eredménye általában kivezet az egész számok halmazáról. Az *x* változó lebegőpontos típusú, így azt gondolnánk, hogy az osztás eredménye 3.3333 lesz. A valóságban viszont nem ez a helyzet, mivel az a és a b nevű változó egész szám, ezért a köztük végrehajtott bármelyik algebrai művelet eredménye is egész szám lesz, és ez alól az osztás művelete sem kivétel. Első ránézésre ez a fajta típuskonverzió feleslegesnek tűnik, és inkább gondot okoz, mint hasznot jelent. Ez azonban némi gondolkodás után kiderül, hogy mégsem így van. Az e fajta típuskonverziónak abban az esetekben van komoly haszna, amikor az eredményeknek mindig egész számnak kell lennie, ilyesmire lehet példa a pixelgrafikus képernyő kezelése, ha a képernyőn különféle grafikus objektumok pozícióit akarjuk kiszámítani. Például egy téglalapot akarunk átskálázni (nagyítani, kicsinyíteni) a képernyőn. A téglalap négycsúcsának képernyő koordinátája a transzformáció előtt és után is csak egészszámú pixel koordinátákkal adható meg.

int a=10, b=3; double x;
x=a/b;

Ha az előző példában bemutatott esetben azt szeretnénk, hogy az eredmény ne egész *(int)* hanem lebegőpontos *(double)* típusú legyen, akkor a következő lehetséges megoldásokhoz folyamodhatunk. Az első megoldásban egy olyan állandót helyezünk el az összefüggésben, amit nem egész számként értelmez a fordítóprogram. Erre az egyel való szorzás tökéletesen megfelel, mivel az összefüggés numerikus értékét nem változtatja meg, viszont az egyet meglehet adni olyan formában, hogy a fordító azt nem egész számként értelmezi. Ez a forma a 1.0 érték. A második lehetőséget a *C#* a C programnyelvből örökölte és Castolásnak nevezik. Ennek lényege, hogy explicit módon megadunk adott típusú változónak egy másik típust. Az alábbi példában például az a nevű változó int típusú és ezt módosítjuk a *(double)* típus módosító megoldással.

int a=10, b=3; *double* x;
x=1.0*a/b;
x= *(double)* a/b;
x= Convert.To*Double*(a) /b;

Az utolsó megoldás a *Convert.ToDouble()* függvény, ezzel az *a* nevű változóban lévő egész szám lebegőpontos számként fog az osztási műveletben megjelenni. Ez tulajdonképpen az explicit típuskonverzió, mivel a programozó kezdeményezi a típuskonverziót a forráskódban.

A példa programban három integer típusú változót *(a, b, c)* hozunk létre a háromszög három oldalának tárolására, a háromszög félkerületét, illetve területét egy-egy *double* típusú változóban *(s, T)* tároljuk. A *Main()* függvény 3.–5. soraiban a futó program konzol ablakának előtér és háttérszín beállítását és azok aktiválását lehet látni. A *Main()* függvény 7.–9. soraiban beolvassuk a háromszög oldalait. A típuskonverziós lehetőségek minél szélesebb körű bemutatása miatt különféle módokon van végrehajtva a beolvasott adat *(string)* konverziója egészszámmá *(int)*.

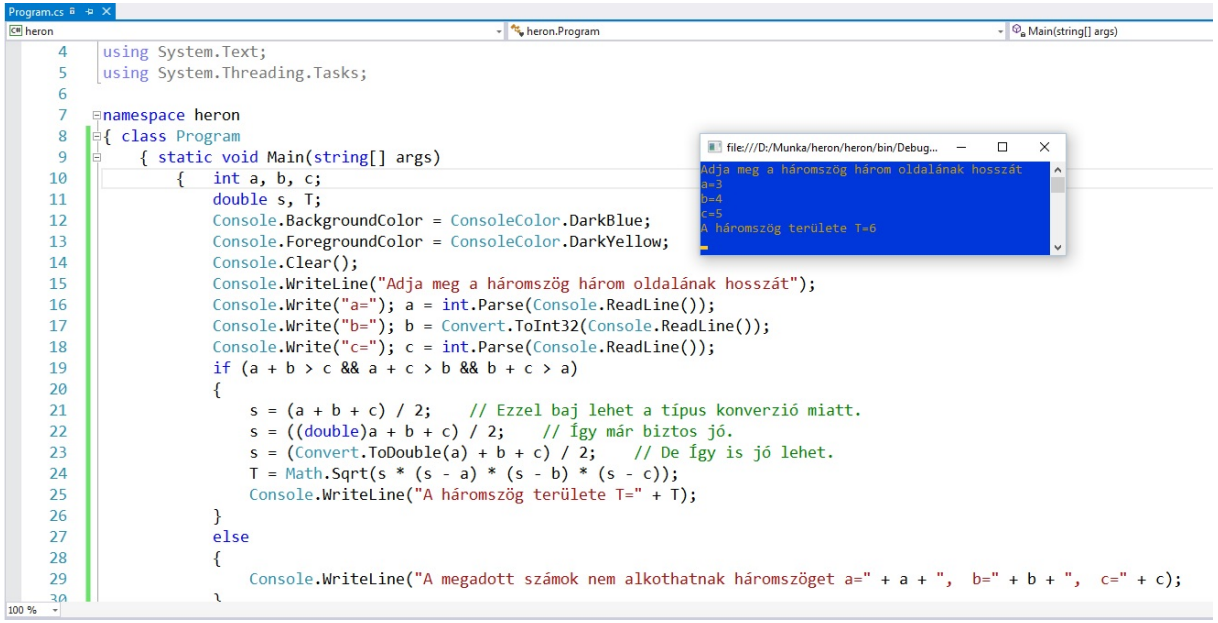
A tizedik sorban vizsgáljuk a háromszög egyenlőtlenség teljesülését, amit az if *(a+b>c && a+c> b && b+c>a)* utasítással vizsgálunk. Mivel a háromszög egyenlőtlenség három különálló relációból áll, ezért azokat a megfelelő logikai operátorral kell összekapcsolni. A háromszög egyenlőtlenség akkor teljesül, ha mind a három reláció egyszerre fenn áll, azaz logikai értékét nézve igaz logikai állapotú. Ezt úgy lehet elérni, ha a logikai állításokat az és && logikai operátorral kapcsoljuk össze. Ennek következtében, ha bármelyik a három feltétel közül sérül, azaz a logikai állítás hamis, akkor az összetett logikai állítás is az *if* utasításban hamis logikai értékű lesz.

```
using System;
namespace heron
{ class Program
{ static void Main(string[] args)
{ int a, b, c;
double s, T;
Console.BackgroundColor = ConsoleColor.DarkBlue;
Console.ForegroundColor = ConsoleColor.DarkYellow;
Console.Clear();
Console.WriteLine("Adja meg a háromszög három oldalának hosszát");
Console.Write("a="); a = int.Parse(Console.ReadLine());
Console.Write("b="); b = Convert.ToInt32(Console.ReadLine());
Console.Write("c="); c = int.Parse(Console.ReadLine());
if (a + b > c && a + c > b && b + c > a)
{s=(a+b+c)/2; //Ezzelbajlehetatípuskonverziómiatt.
      s=((double)a+b+c)/2; //Ígymárbiztosjó.
      s = (Convert.ToDouble(a) + b + c) / 2; // De Így is jó lehet.
      T = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
      Console.WriteLine("A háromszög területe T=" + T);
}
else
{ Console.WriteLine("A megadott számok nem alkothatnak háromszöget a=" + a + ", b=" + b + ", c=" + c);
}
Console.ReadKey(); }
}
}
```

Abban az esetben, amikor az *if* feltételben a logikai állítás igaz a következő lépés a háromszög félkerületének kiszámítása, amit az *s* nevű változóban tároljuk. A félkerület kiszámítására három különféle utasítás is elhelyezésre került a példa programban az esetlegesen felmerülő típuskonverziós probléma bemutatására. Az első megoldás magában hordozza annak lehetőségét, hogy az osztás eredménye nem egész szám, mivel az osztási műveletben csak egész számok szerepelnek.

A második megoldásban az *a* nevű változó castolásával, a harmadik megoldással pedig az a nevű változó értékének típuskonverziójával az osztási műveletben szerepel nem egész szám, így az osztási művelet eredményei is lebegőpontos szám lesz.

8. ábra




```
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace heron
8   { class Program
9       { static void Main(string[] args)
10          {   int a, b, c;
11              double s, T;
12              Console.BackgroundColor = ConsoleColor.DarkBlue;
13              Console.ForegroundColor = ConsoleColor.DarkYellow;
14              Console.Clear();
15              Console.WriteLine("Adja meg a háromszög három oldalának hosszát");
16              Console.Write("a="); a = int.Parse(Console.ReadLine());
17              Console.Write("b="); b = Convert.ToInt32(Console.ReadLine());
18              Console.Write("c="); c = int.Parse(Console.ReadLine());
19              if (a + b > c && a + c > b && b + c > a)
20              {
21                  s = (a + b + c) / 2;    // Ezzel baj lehet a típus konverzió miatt.
22                  s = ((double)a + b + c) / 2;    // Így már biztos jó.
23                  s = (Convert.ToDouble(a) + b + c) / 2;    // De Így is jó lehet.
24                  T = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
25                  Console.WriteLine("A háromszög területe T=" + T);
26              }
27              else
28              {
29                  Console.WriteLine("A megadott számok nem alkothatnak háromszöget a=" + a + ",  b=" + b + ",  c=" + c);
30              }
```

A következő utasítás a háromszög területét számítja ki a Heron képlet segítségével. A képletben szereplő gyökvonás végrehajtására függvényt kell meghívni, mivel hatványozási operátor nem létezik a *C#* programnyelvben. A Math osztály számos matematikai függvényt tartalmaz, amelyek közül most az *Sqrt()* függvényt használtuk a gyökvonás végrehajtására. Az *if* feltételben az utolsó utasítás segítségével a háromszög területét íratjuk ki a képernyőre.

A feltételes elágazás *else* ágában egy *Console.WriteLine()* függvénnyel a képernyőre kiírjuk a felhasználó számára, hogy a begépelt három szám nem alkothatja egy háromszög három oldalának hosszát. Ennek oka, hogy a megadott számok nem teljesítik a háromszög egyenlőtlenséget.

A példa program futása során a háromszög három oldalának a következő értékeket adtuk meg *a=3, b=4, c=5*, ami a 8. ábrán is látható a forráskód fontosabb részleteivel. A három szám Pitagoraszi számhármast alakot, így a háromszög területe, mivel ez egy derékszögű háromszög, könnyen számítható a két befogó (*a=3, b=4*) szorzatának értékéből. Ebben a példában a terület értéke 6 lesz.

5.4. Gyakorló feladatok, kérdések

Mondjon példákat eldöntendő algoritmikai és programozási helyzetekre!

Ha nem két hanem több irányú az elágazási probléma az algoritmusban, akkor milyen lehetőségek, eszközök állnak rendelkezésre a *C#* nyelvben?

Mi a feltételes elágazás szintaktikai szerkezete *C#* nyelvben?

Hozzon létre egy feltételes elágazást, ami megvizsgálja egy numerikus típusú változó (egész, vagy lebegőpontos) előjelét.

Hozzon létre egy feltételes elágazást, ami megvizsgálja egy numerikus típusú változó (egész, vagy lebegőpontos) osztható-e 5-el.

Hozzon létre egy feltételes elágazást, ami megvizsgálja egy numerikus típusú változó (egész, vagy lebegőpontos) értéke 10 és 20 közé esik-e.

Hozzon létre egy feltételes elágazást, ami megvizsgálja egy numerikus típusú változó (egész, vagy lebegőpontos) értéke nem esik a 10 és 20 közé.

Mi az az implicit típuskonverzió?

Miért lehet hasznos az implicit típuskonverzió?

Mutasson egy példát, ahol probléma lehet az implicit típuskonverzió.

Hogyan tudja módosítani az implicit típuskonverziót?

Írjon programot a másodfokú egyenlet megoldására.

Írjon programot, ami eldönti két egész számról, hogy azok oszthatóak-e egymással maradék nélkül.

6. fejezet

6. Ciklusszervező utasítások

6.1. Ciklusszervezésről általánosságban

Ciklusszervező utasításokra akkor van szükség, amikor több alkalommal akarjuk végrehajtani ugyanazokat az utasításokat. Az első fejezetben a folyamatábráknál már röviden ismertetésre került, hogy alapvetően két különféle ciklusszervezési szerkezet létezik, az egyik az elöl- a másik a hátultesztelő ciklusszervezés. A programfejlesztés során többnyire elöltesztelő ciklusszervező utasításokat használunk, de előfordul, hogy a hátultesztelő megoldás választása a célszerűbb.

Fontos hangsúlyozni, hogy szinte bármely probléma, amihez ciklusszervezésre van szükség, megoldható elöl- vagy hátultesztelő ciklusszervezési megoldással is. A lényegi különbség az eszközválasztásban azon múlik, hogy melyik megoldással lehet egyszerűbben kódolni az adott algoritmikai feladatot.

Az alábbiakban néhány a programfejlesztés során előforduló példa látható, ahol mindenképpen célszerű valamilyen ciklusszervezést alkalmazni.

– Az első n darab egész szám összegének kiszámítása, ahol az n értékét a felhasználó adja meg a program futása közben. A ciklusszervezés oka (a felhasználó tetszőleges nagyságú számot adhat meg, emiatt az összegzést el kell végezni egyesével az összes számon a kezdőértéktől kiindulva a megadott felső határértékig).

– Egy tömbben elhelyezkedő adatok elemről elemre történő vizsgálata, valamely előre meghatározott tulajdonság alapján, például egy számokat tartalmazó tömb legnagyobb elemének megkeresése. A ciklusszervezés oka (a tömb elemeinek száma tetszőleges nagyságú, ezért a tömb elemein egyesével el kell végezni a megadott tulajdonság vizsgálatát).

– A felhasználó által a Console.ReadLine() függvény segítségével begépelt karakterek vizsgálata abból a szempontból, hogy az átadható-e valamelyik numerikus típusú változónak. A ciklusszervezés oka (a felhasználó a gépelés során nem egyetlen esetleg maximum két, három karaktert gépel, hanem egy tetszőleges hosszúságú karaktersorozatot, amit egyesével kell aztán vizsgálni a követelményeknek megfelelően).

6.2. Ciklusszervező utasítások típusai

Elöltesztelő ciklusszervező utasítás

Az elöltesztelő ciklusszervező utasítás működésének alapjellemzője, hogy a ciklusba belépés és a ciklusban maradás logikai feltételét a ciklus elején vizsgálja az utasítás, mielőtt a ciklusban elhelyezett utasításokat végrehajtaná. Ha a logikai állítás igaz, akkor a ciklusba foglalt utasításokat végrehajtja, majd visszatér a logikai állítás vizsgálatához és újra ellenőrzi azt. A logikai feltétel újra ellenőrzésére azért van szükség, mivel a ciklus belsejében lévő utasítások valamelyikének mindenképpen hatással kell lennie a logikai állítás állapotára. Ha a ciklusban lévő utasítások sem közvetlenül sem közvetve nem befolyásolják a logikai feltétel állapotát, akkor a ciklus futása soha nem ér véget, azaz egy végtelen ciklus jön létre.

Ha a ciklus futása során a logikai állítás továbbra is igaz, akkor újra végrehajtja azokat az utasításokat, amiket az előbbiek során már végrehajtott. Ha viszont a logikai állítás hamis, akkor a program a ciklusszervező utasítást követő következő utasításra lép.

A ciklusban elhelyezett utasítások végrehajtása egymást követő módon, szekvenciálisan hajtódik végre, de létezik két utasítás, amelyek segítségével ez a lineáris utasítás végrehajtási mód megváltoztatható. Ezek az utasítások a *break* és a *continue*. Fontos azonnal megjegyezni, hogy ennek a két utasításnak a használata nem javasolt, mivel a strukturált programozási módszer ajánlásaival szembemegy a használatuk. Nehezen követhető programvégrehajtást eredményez és az esetlegesen a forráskódban lévő hibák felderítését is rendkívüli módon megnehezíti ezeknek az utasításoknak a használata. Ennek oka, hogy mind a *break* mind pedig a *continue* utasítások használatának csak akkor van gyakorlati értelme, ha azok egy-egy feltételes elágazáshoz *(if)* kapcsolódnak.

A *break* utasítás megszakítja a ciklus végrehajtását és a program végrehajtás menete a ciklusszervező utasítást követő következő utasításra lép. Tehát a *break* utasítás hatása olyan, mintha a ciklus logikai állítása hamis értékűre váltana azon a ponton, ahol a *break* utasítás szerepel a ciklusmagban.

A *continue* utasítás megszakítja a ciklus végrehajtását és a program végrehajtás menete a ciklusszervező utasítás elejére lép és újra vizsgálja ciklusban maradás logikai feltételét. Tehát a *continue* utasítás hatása olyan, hogy azon a ponton, ahol a *continue* utasítás szerepel a ciklusmagban az utasítás végrehajtás sorozata megszakad. A *continue* utasítás mögött álló utasítások pedig nem hajtódnak már végre abban a ciklus lépésben, hanem visszatér a ciklus végrehajtás folyamata a ciklusban maradás logikai feltételének vizsgálatára.

Hátultesztelő ciklusszervező utasítás

A hátultesztelő ciklusszervező utasítás lényege, hogy a ciklusban maradás feltételét a ciklus végén vizsgálja a ciklusszervező utasítás. A ciklusba foglalt utasításokat először végrehajtja, majd a ciklushoz tartozó logikai állítást vizsgálja. Ha a logikai állítás igaz, akkor visszatér a ciklusba foglalt utasítások elejéhez és megkezdi az utasítások újbóli végrehajtását. Ha viszont a logikai állítás hamis, akkor a program a ciklusszervező utasítást követő következő utasításra lép.

A hátultesztelő ciklusszervező utasítás lényegi jellemzője, hogy a ciklusmagban lévő utasításokat egyszer mindenképpen végrehajtja a hátultesztelő utasítás szerkezet, még akkor is ha a logikai állítás már az induláskor hamis értékű.

6.3. *FOR* CIKLUS SZINTAKTIKAI SZERKEZETE

Mivel a *C#* nyelv alapvetően C illetve C++ szintaktikájú tehát a *for* ciklus szerkezeti felépítése teljes mértékben megegyezik a C programnyelvben megismertekkel. A *for(; ;)* kulcsszó utáni zárójelek között három egymástól jól elkülönülő utasításokat kell, illetve lehet megadni, mivel nem feltétlenül szükségszerű mindegyik helyet kitölteni. Ezeket az utasításokat pontosvesszőkkel kell elválasztani egymástól, továbbá akár egynél több utasítás is megadható a három jól elkülönülő helyen.

Az alábbi bekezdésben a *for* ciklus általános szerkezeti felépítése látható.

```
for (inicializáló rész ; logikai állítás ; ciklus indexet módosító illetve általános rész)
{ utasítások;
}
```

A ciklus végrehajtás menete a következő.

– Első lépésben a ciklus első utasítás blokkjában megadott úgynevezett inicializáló utasítások hajtódnak végre.

– A következő lépés a logikai állítás ellenőrzése. Ha az állítás igaz, akkor a kapcsos zárójelpárban (ciklusmag) megadott utasítások végrehajtódnak.

– Miután a ciklusmagban megadott utasítások végrehajtása befejeződött, a *for* ciklus harmadik paraméterében álló utasítások hajtódnak végre.

– A harmadik utasítás csoportban megadottak végrehajtása után a logikai állapot újra ellenőrzése következik és ezzel be is zárul a „kör” azaz a ciklus.

Az első utasítás blokkban a ciklus szervező utasítás indexének kezdőértékét beállító utasítás, valamint vesszőkkel elválasztva további a ciklus inicializálásával kapcsolatos utasításokat lehet megadni. Nagyon fontos tudni, hogy az első pontosvessző előtt álló utasítások (ha megadásra került egyáltalán itt bármi is) csak egyszer hajtódnak végre a *for* ciklusban, a ciklus indulásakor. Az első blokk üresen hagyható a *for(; ;)* ciklusban, mivel az itt megadott utasítások akár a *for* kulcsszó előtt is megadhatók és hatásuk ugyan olyan érvényű.

A logikai állítást, ami tetszőlegesen összetett szerkezetű lehet, amit a különféle logikai operátorok segítségével tudunk létrehozni a *for(; ;)* kulcsszó utáni zárójelek között az első pontos vessző után a második utasítás blokkban kell megadnunk. Ha a logikai állítás igaz, akkor azokat az utasításokat, amiket megadtunk a ciklusmagban, egymás után szekvenciálisan végrehajtódnak. Az utasítások végrehajtása után a vezérlés újra a *for* ciklus logikai állításának ellenőrzésére lép. Ha az ellenőrzés eredménye igaz, akkor újra végrehajtja a ciklusmagban megadott utasításokat, ha pedig hamis, akkor a *for* ciklus után következő utasításra ugrik a program végrehajtás menete.

A harmadik blokkban általában a ciklusváltozó módosításával kapcsolatos utasítások szoktak elhelyezkedni. Ezen a helyen lehet a ciklusváltozó értékét növelni, illetve csökkenteni az adott algoritmikai követelmények szerint.

Szokás szerint, ha csak egyetlen utasítást szeretnénk megadni a *for(; ;)* ciklusban, akkor a kapcsos zárójel elhagyható, mivel a *for(; ;)* kulcsszó után következő első utasítás a ciklusszervező utasításhoz tartozik.

A továbbiakban különféle példák láthatók *for* ciklusok kialakítására. Az első példában a *for* ciklus egy *i* nevű ciklusváltozót tartalmaz, amelynek kezdőértéke egyről indul, és amíg a ciklusváltozó értéke kisebb vagy egyenlő, mint tíz, addig minden egyes ciklus lépésben kiírja az index változó értékét a képernyőre. A ciklusváltozó értékét egyesével növeljük az *i*++ utasítással.

```
for (i=1 ; i<=10 ; i++)
{ Console.WriteLine(“i= “ + i);
}
```

A második példában pontosan ugyan azt az eredményt szolgáltató *for* ciklus kialakítása látható. A különbség, hogy az első és a harmadik paraméter rész a *for* ciklusban üres. A ciklusváltozó kezdőértékét a *for* ciklus előtt külön utasításban, a ciklus változó értékének növeltetését a ciklusmagban oldottuk meg. A *for* ciklus ilyen formátumú felhasználása a később ismertetésre kerülő *while* ciklus kialakításával és használatával egyezik meg.

```
i=1;
for (; i<=10 ;)
{ Console.WriteLine(“i= “ + i);
 i++;
}
```

A következő példában az egy és száz közé eső páratlan számok összegének kiszámítása látható egy *for* ciklus és egy feltételes elágazás segítségével. Természetesen ez a feladat könnyebben is megoldható, ezt a következő példában lehet majd látni.

```
for (i=1, s=0 ; i<=100 ; i++)
{ if ((i % 2)==1)
   { s=s+i;
    }
}
```

A következő példában az előző feladat megoldása látható egyetlen *for* ciklus felhasználásával. A ciklusmagban egyetlen utasítást sem adtunk meg, mivel minden, amire szükség volt a *for* ciklus fejrészében került beállításra.

```
for (i=1, s=0 ; i<=100 ; s=s+i, i+=2)
{
}
```

Egy érdekes, programozás technikai szempontból nem hibás, de nagyon nehezen felderíthető problémát mutat az alábbi példa. A *for(; ;)* kulcsszó után egy pontos vessző áll, ami azonnal le is zárja a *for* ciklust, ami azt jelenti, hogy a kapcsos zárójelpárban megadott utasítások már nem tartoznak a *for* ciklushoz. Ennek következménye az lesz, hogy a kapcsos zárójelpárban megadott utasítások csak egyszer hajtódnak végre azután, hogy ebben a példában a *for* ciklus tíz darab ismétlést hajt végre növelve az i változó értékét 0-ról 10-re.

```
for (i=0 ; i<10 ; i++);
{ utasítások;
}
```

6.4. WHILE CIKLUS SZINTAKTIKAI SZERKEZETE

Mivel a *C#* nyelv alapvetően C illetve C++ szintaktikájú, tehát a *while* ciklus szerkezeti felépítése is teljes mértékben megegyezik a C programnyelvben definiáltakkal. A *while* ciklus általános szerkezeti felépítése az alábbiakban látható, a ciklus szervező utasítás a while kulcsszóval kezdődik, ami után kapcsos zárójelpárban kell megadni azokat az utasításokat, amelyek a ciklushoz tartoznak. A *while* kulcsszó után egy zárójelpáron belül kell megadni azt a logikai kifejezést, aminek igaz *(true)* logikai értéke esetén a ciklusmagban megadott utasítások végrehajtódnak.

```
while (logikai állítás)
{ utasítások;
}
```

Az alábbi példa teljesen megegyezik eredményét tekintve a korábban *for* ciklussal bemutatott példához, azaz egytől tízig az egész számokat kiírja a konzol ablakba. Szintaktikai szempontból viszont teljesen eltérő a kód, mivel a kezdőértéket a ciklus szervező utasítás előtt állítjuk be. A *while* kulcsszó utáni zárójelpárban a ciklusban maradás logikai feltétele *i<=10* látható, azaz amíg a ciklusváltozó i kisebb vagy egyenlő, mint 10 addig a kapcsos zárójelpárba *{ }* foglalt utasítások újra és újra végrehajtódnak. Amint az *i*++ utasítás az 3 értékét 11-re növeli a while ciklus futása befejeződik.

```
i=1;
while (i<=10)
{ Console.WriteLine("i= " + i);
  i++;
}
```

Az is látható, hogy a ciklusváltozót a ciklusmagban módosítja a program.

6.5. DO WHILE CIKLUS SZINTAKTIKAI SZERKEZETE

Mivel a *C#* nyelv alapvetően C illetve C++ szintaktikájú tehát a do while ciklus szerkezeti felépítése is teljes mértékben megegyezik a C programnyelvben megtalálható hátultesztelő ciklus felépítésével. A ciklus szervező utasítás a do kulcsszóval kezdődik, ami után kapcsos zárójelpárban kell megadni azokat az utasításokat, amelyek a ciklushoz tartoznak. Magának a logikai állapotnak a tesztelése pedig a ciklusmagot lezáró kapcsos zárójel után elhelyezkedő while kulcsszót követő zárójelpárban kell megadni. Ha a *while* kulcsszó után megadott logikai kifejezés igaz *(true)* logikai értékű, akkor a ciklusmagban megadott utasításokat újra végrehajtja.

Tehát egyrészről a ciklusmagban álló utasítások addig hajtódnak végre, amíg a ciklusban maradás logikai feltétele igaz, továbbá a ciklusmagban álló utasítások egyszer mindenképpen végrehajtódnak, akkor is ha a logikai feltétel hamis logikai értékű.

```
do
{ utasítások;
} while(logikai állítás);
```

Az alábbi példában a *do while* ciklus egyetlen karaktert olvas be a billentyűzetről a *Console.ReadKey()* függvénnyel, amit a következő függvény a *Console.WriteLine()* a konzol ablakba visszaír ugyan abban a ciklus lépésben. A *while*-ban megadott logikai feltétel úgy lett kialakítva, hogy ha a „*Q*” billentyűt nyomjuk le, aminek eredménye lehet a *”Q”* vagy a *”q”* karakter, akkor a do while ciklus befejezi a ciklusmagban megadott utasítások végrehajtását. Ez azért lehetséges, mivel ha akár a *”Q”* vagy a *”q”* karakter érkezik billentyűzetről, akkor valamelyik logikai állítás az *s!=”Q”* és az *s!=”q”* közül mindenképpen hamis lesz, így az && logikai operátor hatása miatt a teljes logikai állítás is hamis lesz.

```
do
{ s=Console.ReadKey();
    Console.WriteLine(s); }
while(s!=”Q” && s!=”q”);
```

A második rövid példában a do while ciklus egy egész számot olvas be a billentyűzetről a *Console.ReadLine()* függvénnyel, amit egy képernyőtörlés és egy tájékoztató szöveg kiírása előz meg. Itt is elsősorban a logikai feltétel kialakítása az érdekes kérdés. Azt szeretnénk, hogy a begépelt szám nem haladná meg a 100-at, illetve az értéke elérné legalább az egyet. Ha a begépelt szám nem esik az [1, 100] intervallumba, akkor a while mögött álló logikai állítás igaz lesz. Ez azért lehetséges, mivel a két logikai állítást *n<1, n>100* „vagy” logikai operátorral (||) kapcsoljuk össze, ennek következtében, pedig ha a megadott szám nem esik az [1, 100] intervallumba, akkor a két logikai állítás közül valamelyik mindenképpen igaz lesz.

```
do
{ Console.Clear();
  Console.Write(”Adjon meg egy egész számot egytől, százig: ”);
n=int.Parse(Console.ReadLine());
}while(n<1 || n>100);
```

6.6. Példaprogramok

Az alábbi példa program az első n darab természetes szám összegét (1 + 2 + 3 + 4 + ... + *n*), valamint a számok reciprokainak összegét számítja ki. A program egyetlen bemeneti paramétert kap, egy egész számot, a programban megadott *for(; ;)* ciklusban pedig minden egyes lépésben kiírja a természetes számok és reciprokainak részösszegét. A *Main()* függvény első sorában két integer típusú változó definiálunk, amiből az első a ciklus változó *(i)*, a második pedig az a szám *(n)* ameddig a program az összegzést végrehajtja.

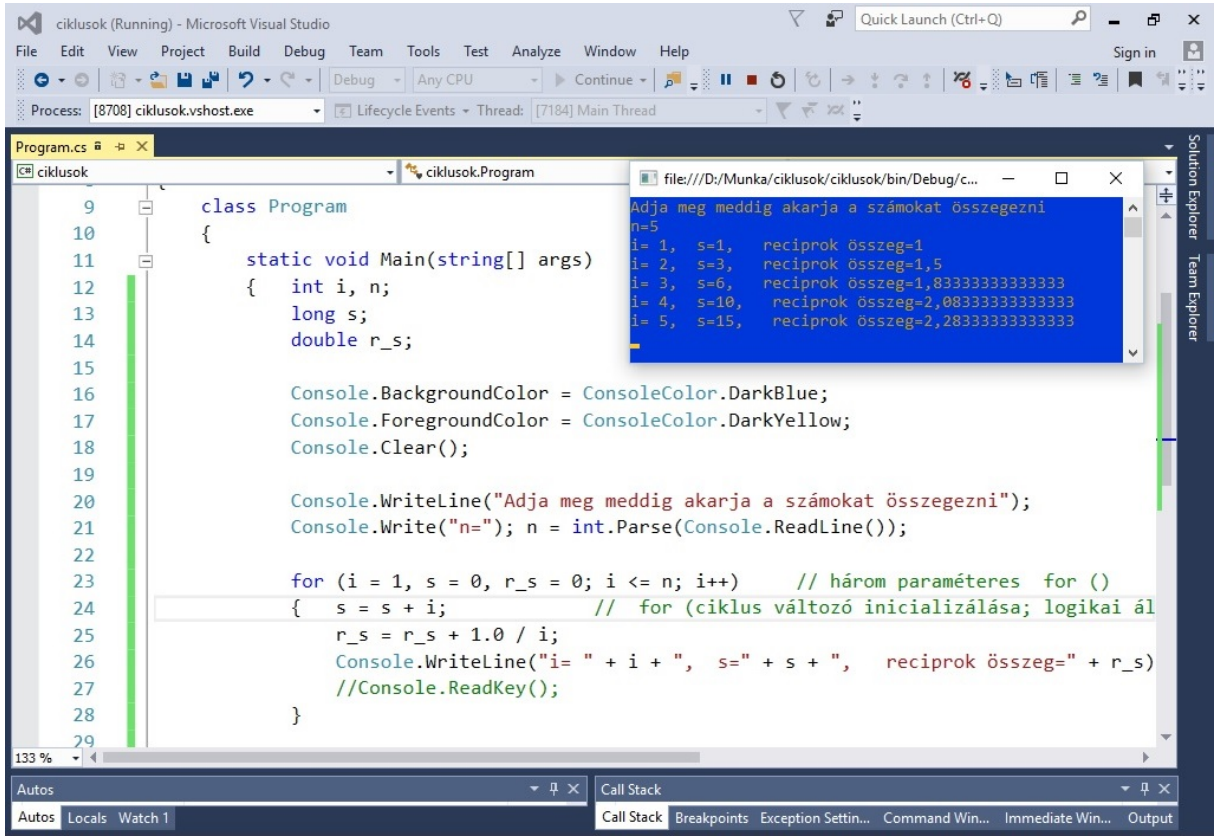
A számok összegét *(s)* egy long típusú változóban, a reciprok összeget pedig egy *double* típusú változóban *(r_s)* tárolja a program. A *Main()* függvény 4–6 sorában a konzol képernyő előtér és háttér színét állítjuk be. A következő két sorban az összegzés felső határát *(n)* olvassa be a program.

A *Main()* függvény kilencedik sorában kerül megadásra a *for(; ;)* ciklus, amelynek első paramétere három utasítást is tartalmaz. Ez a három utasítás sorrendben a következő: a ciklus index *(i)* beállítása, továbbá a két összegző változó *(s, r_s)* értékének nullázása. A *for(; ;)* ciklus második paraméterében a logikai állítás *(i <= n)* került megadásra, aminek hatására a *for* ciklus a korábban begépelt szám *(n)* értékéig fogja növelni a ciklus változó értékét. A *for(; ;)* ciklus harmadik paraméterében pedig a ciklus változó értékének növelése *(i++)* került megadásra. Ha esetleg ez utóbbi utasítás *(i++)* véletlenül kimarad, akkor végtelen ciklust hozunk létre, amely soha nem fog magától leállni, a ciklus leállásához valamilyen külső beavatkozás szükséges.

```
using System;
namespace sorosszeg
{ class Program
    { static void Main(string[] args)
       { inti,n;
         long s;
         double r_s;
         Console.BackgroundColor = ConsoleColor.DarkBlue;
         Console.ForegroundColor = ConsoleColor.DarkYellow;

         Console.Clear();
         Console.WriteLine("Adja meg, hogy meddig akarja a számokat összegezni");
         Console.Write("n="); n = int.Parse(Console.ReadLine());
         for(i=1,s=0,r_s=0;i<=n;i++) //háromparaméteres for(;;)
         { s = s + i; // for (ciklus változó inicializálása; logikai állítása ; ciklus változó módosítása )
           r_s = r_s + 1.0 / i;
           Console.WriteLine("i= " + i + ", s=" + s + ", reciprok összeg=" + r_s);
         }
       }
   }
}
```

A ciklusmagban az első utasítás a ciklusváltozó i értékének hozzáadása az s változó értékéhez. A második utasítás a ciklusmagban a reciprok összeget számítja ki az *r_s* változóban. A ciklusváltozó reciprokát az *1.0 / i* kifejezéssel számítjuk. A számlálóban az 1 értékét az implicit típuskonverzió miatt adtuk meg lebegőpontos alakban.

9. ábra



A ciklusmagban a harmadik utasítással a konzol képernyőre kiírjuk a természetes számok és reciprokaik részösszegét. Ez a *9. ábrán* is látható, ahol a program futásához rendelt ablakban az *n* változó értéke 5, alatta pedig sorról sorra a részösszegek láthatók, az utolsó sor pedig a két végösszeget tartalmazza.

6.7. Gyakorló feladatok, kérdések

Soroljon fel olyan algoritmikai feladatokat, amelyeket csak ciklusszervezéssel lehet vagy célszerű megoldani.

Milyen alapvető ciklusszervezési szerkezeteket ismer?

Ismertesse az elöltesztelő ciklusszervező utasítás működésének alapvető jellemzőit.

Ismertesse a hátultesztelő ciklusszervező utasítás működésének alapvető jellemzőit.

A ciklusban elhelyezett utasításoknak mi az alapértelmezett utasításvégrehajtási sorrendje?

Melyik utasítások használhatók az alapértelmezett utasítás végrehajtási sorrend módosítására?

Mi a *break* utasítás hatása a ciklus végrehajtás menetére?

Mi a *continue* utasítás hatása a ciklus végrehajtás menetére?

Miért nem javasolt a break és a continue utasítások használata?

Írja le a *for* ciklus szintaktikai felépítését.

Hány utasítás blokkja van a *for* ciklusnak?

Milyen karakterrel választjuk el az egyes utasítás blokkokat a *for* ciklusban?

Mit adhatunk meg a *for* ciklus első utasítás blokkjában?

Mit adunk meg a *for* ciklus második utasítás blokkjában?

Mit adhatunk meg a *for* ciklus harmadik utasítás blokkjában?

Mi a ciklusban maradás logikai állapota a *for* ciklusban?

Írja le a *while* ciklus szintaktikai felépítését.

Mi a ciklusban maradás logikai állapota a *while* ciklusban?

Hasonlítsa össze a *for* ciklust és a while ciklust.

Írja le a *do while* ciklus szintaktikai felépítését.

Mi a ciklusban maradás logikai állapota a *do while* ciklusban?

Hasonlítsa össze a *do while* ciklust és a *while* ciklust.

Írjon programot természetes számok reciprok négyzetösszegének kiszámítására egy megadott intervallumban. A program a billentyűzetről kérje be az intervallum határait.

Írjon programot, amely addig fut, amíg le nem nyomunk egy korábban ebben a programban bekért karaktert.

Írjon programot, amely kiszámítja n darab a billentyűzetről gépelt lebegőpontos szám négyzetösszegeinek gyökét. (n komponensű Pithagorasz tétel vagy Euklideszi norma)

Listázza az összes, hárommal maradék nélkül osztható számokat a képernyőre. (több féle módon is megoldható a feladat!)

Írjon programot, amely bekér egy egész számot és addig fut, amíg a programban bekért szám nem osztható öttel.

7. fejezet

7. Összetett adatszerkezetek, egy dimenziós és több dimenziós tömbök, listák

7.1. Összetett adatszerkezetek, tömbök

Ha nagy mennyiségű egymáshoz szorosan kapcsolódó adatokat, például egy adott helyen a hőmérséklet értékeket, vagy egy részvény árfolyamának percenkénti, óránkénti esetleg naponkénti változását, vagy a talajban egy szennyező anyag koncentrációjának értékeit szeretnénk tárolni, akkor célszerű olyan speciális változó létrehozása, ahol egyetlen névvel hivatkozhatunk sok ugyan olyan típusú változóra. Az ilyen szerkezetű összetett változókat tömböknek nevezzük. A tömböt úgy kell elképzelni, mint egy vektort, amelynek véges sok eleme van. Elterjedt megnevezése a programozással foglalkozó tananyagokban az ilyen típusú szerkezeteknek a vektor elnevezés, ami matematikai szempontból nézve erősen pontatlannak tekinthető. Erről a megjegyzésről röviden csak annyit, hogy ha leírunk néhány számot egymás mögé vesszőkkel elválasztva vagy egymás alá az még nem válik vektorrá. Ahhoz, hogy egy ilyen formátumú szerkezet vektornak legyen nevezhető, minimum meg kell adni azt is, hogy milyen műveleteket értelmezhetünk az ilyen alakú, formátumú adatszerkezeteken.

A tömbre a nevével tudunk hivatkozni, hasonlóan, mint egy egyszerű hagyományos változó. Mivel a tömb általában egynél több elemet tartalmaz, ezért az egyes elemek azonosításához nem elegendő a tömb neve, szükség van az elérni kívánt elem sorszámára is. A tömbön belül az egyes elemek sorszámát indexnek nevezzük. A tömb elemeinek elérése a tömb nevének és az indexnek az adott nyelv szintaktikai szabályainak megfelelő módon való felhasználásával van lehetőség.

A tömb indexe értelemszerűen csak egész szám lehet. A tömbök indexelése a programnyelvek többségében nulláról indul, azaz a tömb első elemének indexe nulla. A C, C++, C# nyelvekben a tömbindex nullától kezdődik, de vannak olyan nyelvek, ahol a programozó dönthet arról, hogy nulláról vagy egyről induljon az index, ilyen a Pascal vagy a Basic nyelv.

egyik_tomb_elem = tomb_neve[index];

A tömb tetszőleges elemének az elérése a programnyelvek többségében a fent látható szintaktikai szerkezetben oldható meg, azaz a tömb neve után szögletes zárójelek között kell megadni a keresett elem indexét.

A tömbök lehetnek egy vagy akár több dimenziósak is. Ez azt jelenti, hogy a tömbök nem egy, hanem akár két vagy több indexel is rendelkezhetnek. Például ha egy táblázatos elrendezésű számhalmazt (mátrixszerű) szeretnénk tárolni a programunk futása során, akkor létre kell hozni egy kétdimenziós tömböt, amelynek egyik indexét sor, a másikat oszlop indexként foghatjuk fel. Több dimenziós tömbök használatakor a programozónak nem kell közvetlenül a memóriaszervezés nehézségeivel foglalkozni, a tömb egyes elemeit a tömb nevével és a két index megfelelő kombinációjával elérheti.

Fontos kérdés a tömbök operatívtárbéli (RAM) helyfoglalása. Ezt a kérdést a programozónak mindig szem előtt kell tartania, mivel a RAM korlátos mérete számos esetben komoly problémát jelent mind a program fejlesztés, mind pedig a program későbbi használata során. A tárfoglalás méretét hagyományos tömbök esetén az alábbi összefüggéssel lehet kiszámítani.

*tárfoglalás_nagysága = tömb_elemszáma * elemi_típus_mérete*

példák:

húsz elemű int (2 byte ANSI C-ben) típusú tömb tárfoglalása 40 byte.
húsz elemű int (4 byte Visual Studio C#) típusú tömb tárfoglalása 80 byte.

7.2. Tömbök létrehozása *C#* nyelvben

A *C#* programnyelvben tömböket a **new** operátor felhasználásával tudunk létrehozni. A **new** operátor szintén az objektum orientált programozás része, a C++ nyelvből származik. A használatával kapcsolatban elég annyit tudni a programozással most ismerkedőknek, hogy az alábbi szintaktikai szerkezetben lehet azt tömbök létrehozására használni.

A tömbök létrehozásának általános szerkezete.

típus_azonosító[] tomb_neve = ***new*** *típus_azonosító[elemszám];*

Az alábbiakban pár példa látható tömbök definiálására *C#* nyelven.

```
int[] tomb_nev = new int[10];
char[] karakterek_nev = new char[100];
```

A fenti definíciók közül az elsőben a típusazonosító (**int** kulcsszó) után egy szögletes zárójelpár üresen kell álljon, majd a tömb neve következik **tomb_nev** ezután pedig egy egyenlőségjel majd a new operátort kell alkalmazni aztán megint a típusazonosító következik végül szögletes zárójelben egy szám, ami a tömb elemeinek számát adja meg. Azaz a **tomb_nev** nevű tömb 10 elemből fog állni és az egyes elemekre a következő indexeléssel tudunk hivatkozni. Például a **tomb_nev[2]** a tömb harmadik eleme, mivel a tömb indexelése nulláról indul és a fent megadott szám mínusz egyig, azaz kilencig tart. Tehát az első elem a **tomb_nev[0]**, az utolsó elem, pedig **tomb_nev[9]** hivatkozással érhető el. A **tomb_nev** nevű tömb az operatív tárban 40 byte-ot foglal, mivel az int típus a *C#* nyelvben 4 byte méretű.

A következő példa egy karakter tömb definícióját mutatja, ami hasonlít majd a későbbiekben ismertetésre kerülő *string*-ekre. A **karakterek_nev** nevű tömb 100 elemből áll. Például a **karakterek_nev[1]** a tömb második elemét jelenti, mivel itt is a tömb indexe nulláról indul, az utolsó elem pedig a **karakterek_nev[99]** hivatkozással érhető el. A **karakterek_nev** nevű tömb az operatív tárban 200 byte-ot foglal, mivel a char típus a *C#* nyelvben 2 byte méretű.

A tömbökről röviden még annyit, hogy a tömb elemeit általában valamilyen ciklusszervező utasítással szokás elérni. A tömb bejárására, elemeinek elérésére bármelyik az előző fejezetben (6.) ismertetett ciklusszervező utasítás alkalmas. Többnyire a *for* ciklust szokás alkalmazni, de ez nem kötelező érvényű.

Fontos kérdés még a tömbök elemeinek kezdő (inicializáló) értéke. Például a C programnyelvben a tömbök egyes elemei nem kapnak kezdőértéket, ennek megadása a programozó feladata. A forráskódot úgy kell kialakítani, hogy a program futásának eredményét ne befolyásolják az előkészítetlen elemek. Például egy külön ciklus segítségével a tömb elemeit inicializálni (például nullázni) kell.

A *C#* programnyelvben a tömbök elemeinek értékei előkészítésre kerülnek, a numerikus típusúak a 0 értéket kapnak, a nem elemi típusúak pedig az úgynevezett **null** értéket kapják. Természetesen a *C#* programnyelvben a tömbök elemeinek értékei szintén előkészíthetők, létezik erre speciális szintaktikai szerkezet. A kezdőértékeket kapcsos zárójelek között vesszőkkel elválasztva kell megadni.

típus_azonosító[] tomb_neve = {érték1, érték2, ... , értékn};

Az alábbiakban két példa látható tömbök inicializálására *C#* nyelven.

```
int[] tomb_nev = {1, 2, 3, 4, 5, 6, 7, 8, 9};
string[] het_napjai = {"hétfő", "kedd", "szerda", "csütörtök", "péntek", "szombat", "vasárnap"};
```

7.3. Több dimenziós tömbök a *C#* nyelvben

Gyakran van szükség olyan adatok tárolására, amelyek táblázatos elrendezésűek, ilyenkor van lehetőség arra *C#* nyelven, hogy két vagy akár több dimenziós tömbökben tároljuk az ilyen szerkezeti elrendezésű adatcsoportokat. A két vagy akár több dimenziós tömbök létrehozásának általános szerkezete *C#* nyelven, az alábbi kiemelt részben látható. Az eltérés az egy dimenziós tömbök definíciójához képest csak annyi, hogy a típus azonosítót követő szögletes zárójelekben egy (kétdimenziós esetben) vagy több vesszőt kell megadni. A **new** operátort követő szögletes zárójelek között pedig a sorok, oszlopok, stb... elemszámát kell megadni.

típus_azonosító[,] tomb2D_neve = ***new*** *típus_azonosító[elemszám1, elemszám2];*

Az alábbiakban két példa látható több dimenziós tömbök definiálására *C#* nyelven, amiket mátrixnak is szoktak pontatlanul nevezni.

```
long[,] tomb2D_nev = new long[2,10];
float[, , ] tomb3D_nev = new float[4,10,3];
```

A fenti két példában az első egy long típusú elemeket tartalmazó két dimenziós tömb. Természetesen az ilyen szerkezetekre már két független indexel tudunk hivatkozni. Például a **tomb 2D_nev[1,2]** a tömb második sorának harmadik elemét jelenti, mivel itt is az indexek nulláról indulnak. A **tomb2D_nev** két sorból és tíz oszlopból álló long típusú elemeket tartalmazó tömb. A **tomb2D_nev** nevű tömb az operatív tárban 160 byte-ot foglal, mivel a long típus a *C#* nyelvben 8 byte méretű.

tomb2D_nev

10. ábra

	0	1	2	3	4	5	6	7	8	9
0	-12	44	-3	4	12	1	41	2	-2	11

A *10. ábrán* látható táblázat a **tomb2D_nev** nevű tömb szemléletes reprezentációja. A táblázat első sora és első oszlopa a két független indexet tartalmazza. A második és harmadik sorokban látható a tömb tényleges tartalma. Például a **tomb2D_nev[0,8] = -2 vagy tomb2D_nev[1,4]** = 17.

A fenti két példában a második egy *float* típusú elemeket tartalmazó háromdimenziós tömb. Az ilyen felépítésű szerkezetekre már három független indexel tudunk hivatkozni.
A tomb3D_nev négy sorból és tíz oszlopból és három az előzőkre *„merőleges”* sorból álló float típusú elemeket tartalmazó tömb. A **tomb3D_nev** nevű tömb az operatív tárban 480 byte-ot foglal, mivel a float típus a *C#* nyelvben 4 byte méretű.

7.4. Listák

A listákat a tömbök tovább fejlesztéseként foghatjuk fel, mivel számos olyan lehetőséget biztosít, amire a hagyományos tömbök esetén nincsen lehetőségünk. Például a tömbök mérete a definiálásuk után már nem változtatható meg, ezzel ellentétben a listák mérete, elemeinek száma dinamikusan bővíthető a futó program igényeinek megfelelően. A lista egy generikus típus, ami azt jelenti, hogy nem csak elemi adattípusok lehetnek az elemei, hanem bármi akár tömbök vagy másik listák is lehetnek elemei az éppen létrehozandó listának. A lista egyes elemeire hasonlóan a tömbökhöz indexel tudunk hivatkozni. A listák a különféle ciklusszervező utasításokkal járhatók be.

Listák létrehozásának első kihagyhatatlan lépése, hogy a using kulcsszó felhasználásával a generikus típusokat a *System.Collections.Generic* névtér használatával beemeljük a forráskódunkba. A generikus változók által foglalt erőforrások (RAM) felszabadítását az úgynevezett Garbage Collector, magyarul szemétgyűjtő végzi el.

Listák létrehozásának általános szerkezete.

List <típus_azonosító> lista_neve = new List <típus_azonosító>(elemszám);

Listákat hasonlóan hozhatunk létre, mint tömböket. A **new** operátor használatával lehet a megadott számú és típusú elemekből álló listát létrehozni.

Listák kapcsán fontos megkülönböztetni a lista kapacitás és a méret fogalmakat. A lista kapacitása az, amit a ( ) zárójelpárban megadunk, amit tulajdonképpen egy tárolási lehetőségként foghatunk fel, ami üres terület, ténylegesen nem tartalmaz semmit sem. Ha a zárójelpárban nem adunk meg semmilyen értéket, akkor az alapértelmezett lista kapacitás 4. A lista mérete pedig a ténylegesen a listában tárolt elemek számát adja meg. A lista kapacitását a *lista_nev.Capacity* a lista méretét a *lista_nev.Count* –al tudjuk lekérdezni.

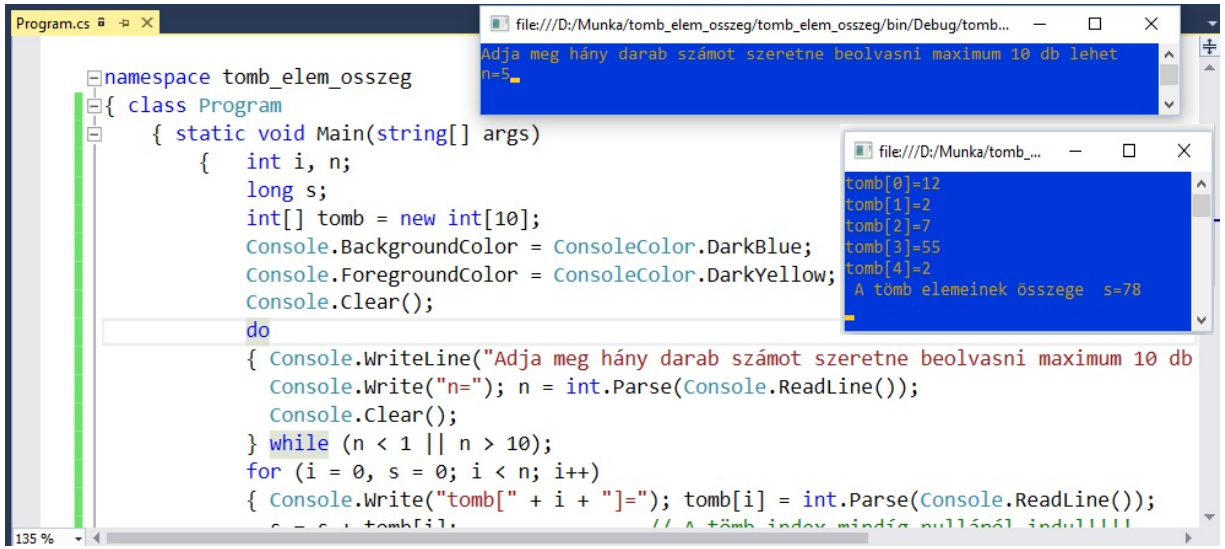
A listákhoz új elemeket az *Add()* függvénnyel (metódussal) lehet hozzáadni. A függvény használata a következő szintaktikai szerkezetben lehetséges *lista_nev.Add(uj_lista_elem).*
A listákhoz használatát számos metódus könnyíti meg, ezekről részletesebben a 8. fejezetben lehet tájékozódni.

7.5. Példaprogramok

Az első példában egy legfeljebb tíz elemből álló integer tömbbe beolvasott számok összegét számítjuk ki. A *Main()* függvény első sorában két egész *(int)* típusú változót definiálunk, az egyik a ciklus változó *(i)*, a másik változó *(n)* pedig a beolvasott számok mennyiségét tartalmazza. A második sorban egy long típusú változó *(s)* definiálása látható, amely a tömbbe begépelt számok összegét tartalmazza. A harmadik sorban egy tíz elemű integer tömb definíciója látható, amelybe a billentyűzetről olvasunk be számokat, majd a következő sorok a háttér és előtér színeket állítja be és valósítja meg *ConsoleClear()* a konzol ablakban.

```
using System;
namespace tomb_elem_osszeg
{ class Program
        { static void Main(string[] args)
                { int i,n;
                  long s;
                  int[] tomb = new int[10];
                  Console.BackgroundColor = ConsoleColor.DarkBlue;
                  Console.ForegroundColor = ConsoleColor.DarkYellow;
                  Console.Clear();
                  do
                  { Console.WriteLine("Adja meg hány darab számot szeretne beolvasni maximum 10 db lehet");
                  Console.Write("n="); n = int.Parse(Console.ReadLine());
                  Console.Clear();
                  } while (n < 1 || n > 10);
                  for (i = 0, s = 0; i < n; i++)
                  { Console.Write("tomb[" + i + "]="); tomb[i] = int.Parse(Console.ReadLine());
                  s = s + tomb[i]; // A tömb index mindíg nulláról indul!!!!
                  }
                  Console.WriteLine("A tömb elemeinek összege s=" + s); Console.ReadKey();
                }
        }
}
```

11. ábra



A program először a begépelendő számok mennyiségét kéri be a billentyűzetről, ami egy *do while()* ciklusba szervezve került megoldásra. Abban az esetben lép ki a hátultesztelő ciklusból, ha a begépelt szám egytől tízig terjedő egész szám. Ha a begépelt szám egynél kisebb vagy tíznél nagyobb, akkor a *while()* logikai feltétele igaz lesz így újra kéri az n változó értékét.

A *do while()* ciklust követő utasítás is egy ciklus szervező utasítás mégpedig egy *for* ciklus. A *for* ciklus feladata a korábban megadott mennyiségű *(n)* egész szám beolvasása a tömbbe. A *for* ciklus fejrészében nullázzuk a ciklus változó *(i)* valamint a részösszeget tartalmazó változó *(s)* értékeit. A ciklusban maradás feltétele, hogy az *i* változó értéke kisebb kell legyen, mint az *(n)* változó értéke. Látható, hogy a tömb első elemének indexelése nulláról indul, emiatt kellett az i változó értékét nullára állítani.

A *for* ciklus első utasítása egy *Console.WriteLine()*, amivel kiírjuk a következő szöveget a képernyőre: *"tomb[2]=".* Ez a szöveg a *for* ciklus harmadik fordulójában íródik ki, amikor az i változó értéke 2-vel egyenlő, egyébként ez a tömb harmadik elemének beolvasását jelenti éppen. A beolvasás után az s változóban elvégezzük az aktuálisan gépelt szám hozzáadását *(s = s + tomb[i];)* a korábban begépelt számok összegének értékéhez.

Itt érthető meg miért szükséges az s változó kezdeti értékének nullázása. Mivel az összegzés olyan módon zajlik, hogy az s mindenkori értékéhez adjuk hozzá az aktuálisan begépelt számot, ezért az s változónak az első begépelt szám előtt a nulla értéket kell adni, különben a begépelt számok összegét hibásan számolná a program.

A *Main()* függvény utolsó előtti utasítása egy *Console.WriteLine(),* amivel kiírjuk a tömbbe begépelt számok összegét, az utolsó utasítás pedig egy *Console.ReadKey(),* aminek hatására a program futása leáll addig, amig egy billentyűt le nem nyomunk a billentyűzeten.

Második példa

A következő példában egy rendező algoritmust láthat az olvasó, amely egy integer tömb elemeit növekvő sorrendbe rendezi. A rendező algoritmus lényege, hogy a számtömbben megkeresi a legkisebb számot és azt kicseréli az első helyen álló számmal a tömbben. Ezt az algoritmikai folyamatot pedig újra és újra megismétli addig, amíg a teljes számtömböt lerendezi. Az algoritmus előnye a könnyű programozhatóság, amely egyébként hatékonyságát tekintve nem a legjobb (gyorsabb) rendező algoritmus.

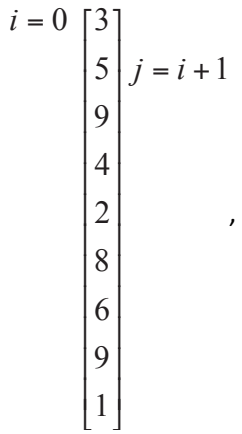
A rendező algoritmus működési menete a *12. ábra* alapján könnyedén megérthető. Alapvető jellemzője, hogy a tömböt két indexel járjuk be, emiatt pedig két ciklus szervező utasítás egymásba ágyazására lesz szükség. A két index, mint két egymástól függetlenül pozícionált mutató lehetővé teszi a szükséges cserék lebonyolítását. Az egyik index *(i)* a tömb elejéről indul és a tömb utolsó előtti eleméig halad. A második index *(j)* a másik indextől indul egyel attól előre léptetve *(j=i+1)*. Ennek egyszerűen az az oka, hogy egyik számot sem akarjuk összehasonlítani önmagával, mivel az teljesen felesleges. Az indexeket egy-egy ciklusszervező utasítással mozgatjuk olyan módon, hogy az egyik ciklus a másikba van ágyazva.

Az *(i)* index az a pozíció a tömbben ahová keressük a sorrendben következő nagyobb számot, ami az *(i)* index aktuális pozíciója előtt áll a tömbben. A kezdeti állapotban az *i* értéke 0, a *j* értéke pedig 1. Ezután a belső ciklus a j indexet elkezdi növelni és minden egyes helyen összeveti az i index aktuális pozícióján álló számmal. Ha a *j* indexen álló szám kisebb, mint az *i* index aktuális pozícióján álló szám, akkor a két számot fel kell cserélni. Ezt a tömb utolsó eleméig ellenőrizni kell, azaz a j index a tömb utolsó indexű eleméig fut. Ezzel zárul az első lépés a rendezés folyamatában.

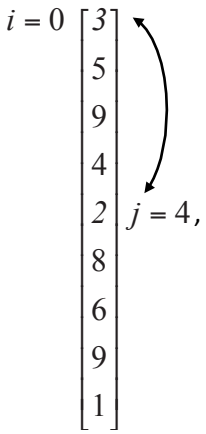
A második lépésben az *i* index pozíciója egyel nő, azaz *i=1, a j=2* lesz. Ezután a belső ciklus újra a j indexet kezdi el növelni és minden egyes helyen újra összeveti az *i* index aktuális pozícióján (2.) álló számmal.

Ezeket a lépéseket mindaddig folytatni kell, amíg az *i* index pozíciója el nem éri az utolsó előtti tömb indexet. Ekkor a *j* index már a tömb utolsó elemén áll, amit az utolsó előttivel összehasonlítva, ha az szükséges végrehajtódik az utolsó csere.

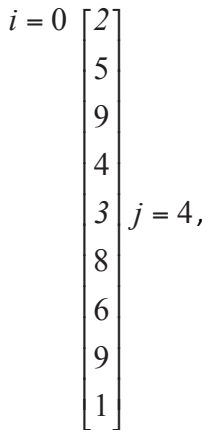
kezdeti állapot



első csere előtt

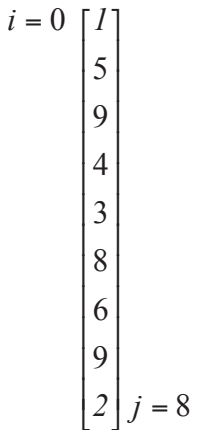


első csere után

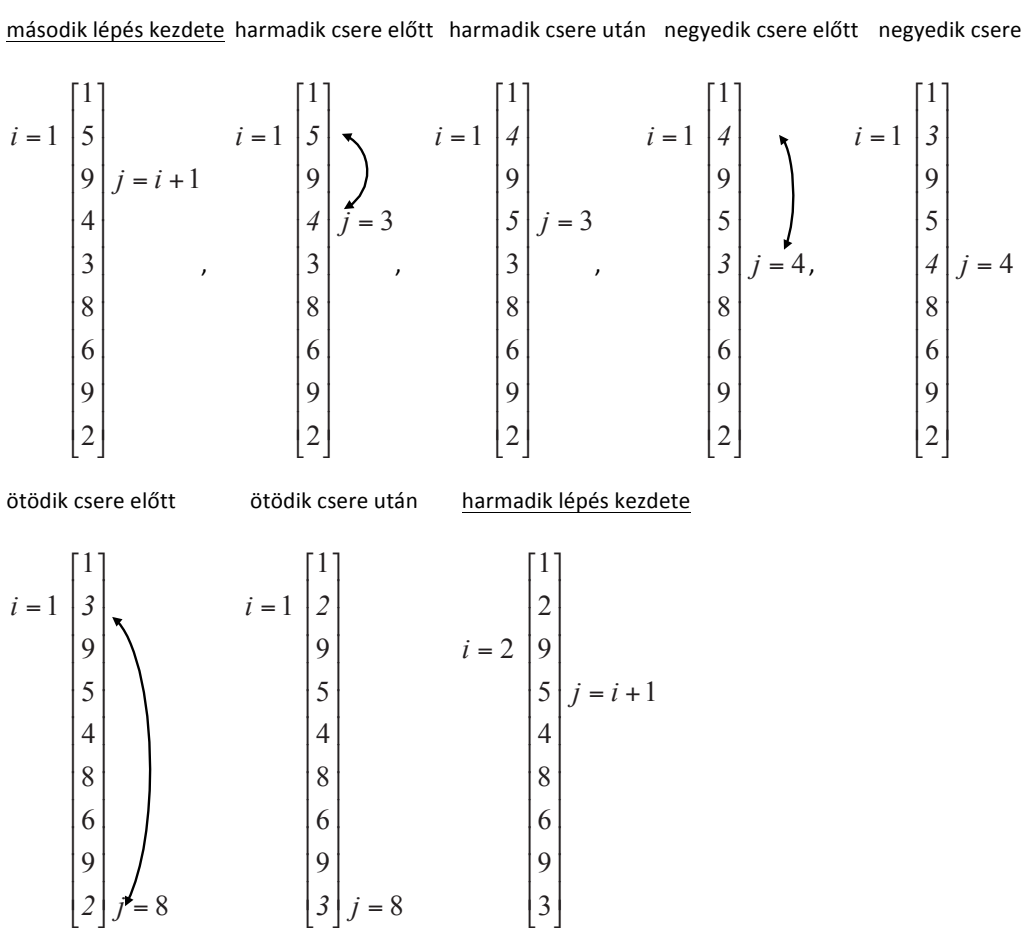


második csere előtt

második csere után



12. ábra


második lépés kezdete
harmadik csere előtt
harmadik csere után
negyedik csere előtt
negyedik csere után
i = 1
j = i + 1
j = 3
j = 4
ötödik csere előtt
ötödik csere után
harmadik lépés kezdete
i = 2
j = 8

Az algoritmust megvalósító *C#* kódot az alábbiakban lehet látni. A forráskód egy egész *(int)* elemű tömb rendezését végzi el. A *Main()* függvény első sorában a kódban használt index változók *i, j*, a rendezendő számok mennyiségét n valamint egy a csere során használt segédváltozót temp definiálunk. A második sor a tömb definícióját tartalmazza. A tömb maximum 20 darab int típusú változót tartalmazhat. A következő három utasítás sor a konzol ablak háttér és előtérszínét állítja be és aktíválja a háttér színét.

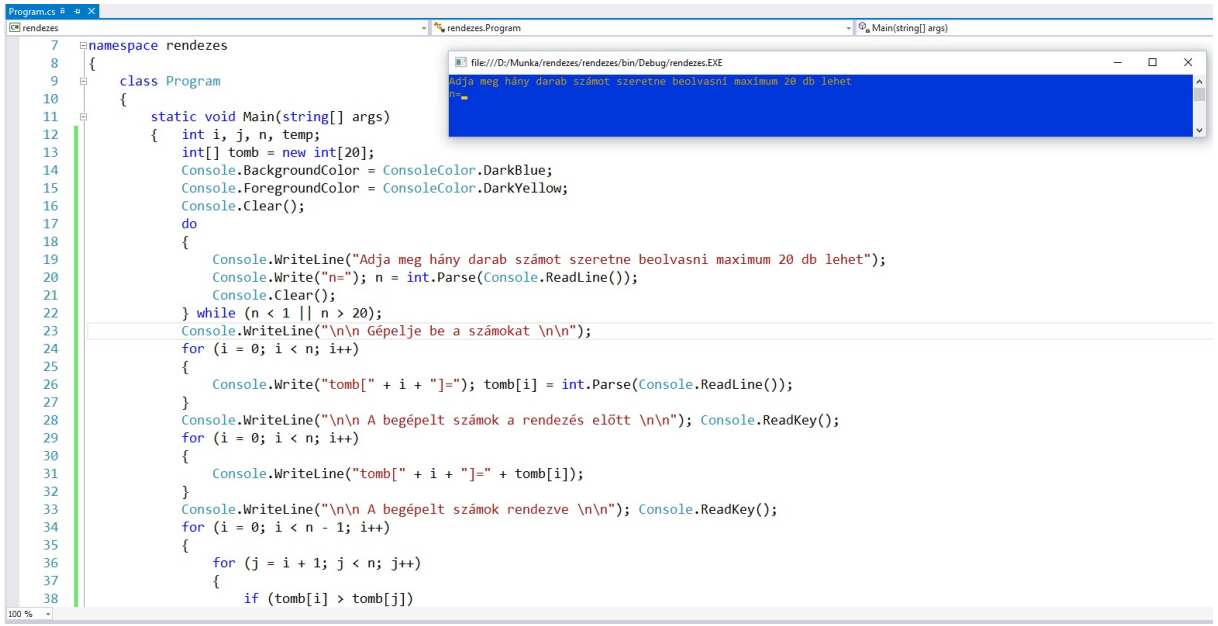
Ezután a program először a begépelendő számok mennyiségét kéri be a billentyűzetről, ami egy d*o while()* ciklusba szervezve került megoldásra. Abban az esetben lép ki a hátultesztelő ciklusból, ha a begépelt szám egytől húszig terjedő egész szám. Ha a begépelt szám egynél kisebb vagy húsznál nagyobb, akkor a *while()* logikai feltétele igaz lesz, így újra kéri az *n* változó értékét. Ez látható a *13. ábra* felső részén, ahol a futó program ablaka kék háttérrel vízszintesen elnyújtva látható a forráskód előtt. A *13. ábra* felső részén éppen az n nevű változó értékének begépelésére vár a program. Az n változó értékének beolvasását a szokásos *Console.Write(), Console.ReadLine()* párossal és a csatolt konverziós függvénnyel *int.Parse()* oldottuk meg.

A *do while()* ciklust követő utasítás is egy ciklus szervező utasítás, mégpedig egy *for* ciklus. A *for* ciklus feladata a korábban megadott mennyiségű *(n)* egész szám beolvasása a tömbbe. A *for* ciklus fejrészében nullázzuk a ciklus változót *(i)*. A ciklusban maradás feltétele, hogy az i változó értéke kisebb kell legyen mint az *(n)* változó értéke. Látható, hogy a tömb első elemének indexelése nulláról indul, emiatt kellett az i változó értékét nullára állítani. A tömb elemeinek értékeit a megszokott *Console.Write(), Console.ReadLine()* párossal és a csatolt konverziós függvénnyel *int.Parse()* olvassuk be a billentyűzetről.

```
using System;
namespace tomb_elemek_rendezese
{ class Program
  { static void Main(string[] args)
    {  int i, j, n, temp;
       int[] tomb = new int[20];
       Console.BackgroundColor = ConsoleColor.DarkBlue;
       Console.ForegroundColor = ConsoleColor.DarkYellow;
       Console.Clear();
       do
       { Console.WriteLine("Adja meg hány darab számot szeretne beolvasni maximum 20 db lehet");
         Console.Write("n="); n = int.Parse(Console.ReadLine());
         Console.Clear();
       } while (n < 1 || n > 20);
       Console.WriteLine("\n\n Gépelje be a számokat \n\n");
```

```
            for (i = 0; i < n; i++)
            { Console.Write("tomb[" + i + "]="); tomb[i] = int.Parse(Console.ReadLine());
            }
            Console.WriteLine("\n\n A begépelt számok a rendezés előtt \n\n"); Console.ReadKey();
            for (i = 0; i < n; i++)
            { Console.WriteLine("tomb[" + i + "]=" + tomb[i]);
            }
            Console.WriteLine("\n\n A begépelt számok rendezve \n\n"); Console.ReadKey();
            for (i = 0; i < n-1; i++)
            {  for (j = i+1; j < n; j++)
               {  if (tomb[i] > tomb[j])
                  { temp= tomb[i]; tomb[i]= tomb[j]; tomb[j]=temp;   // Ez itt a csere!
                  }
               }
            }
            for (i = 0; i < n; i++)
            { Console.WriteLine("tomb[" + i + "]=" + tomb[i]);
            }
            Console.ReadKey();
        }
    }
}
```

13. ábra



Az ezt követő *for* ciklus feladata a begépelt számok kilistázása még a rendezés előtt, így a program használója a rendezés előtt látja a teljes számtömböt. Ez látható a 14. ábra alsó részén, ahol a futó program ablaka kék háttérrel függőlegesen kinagyítva látható a forráskód előtt.

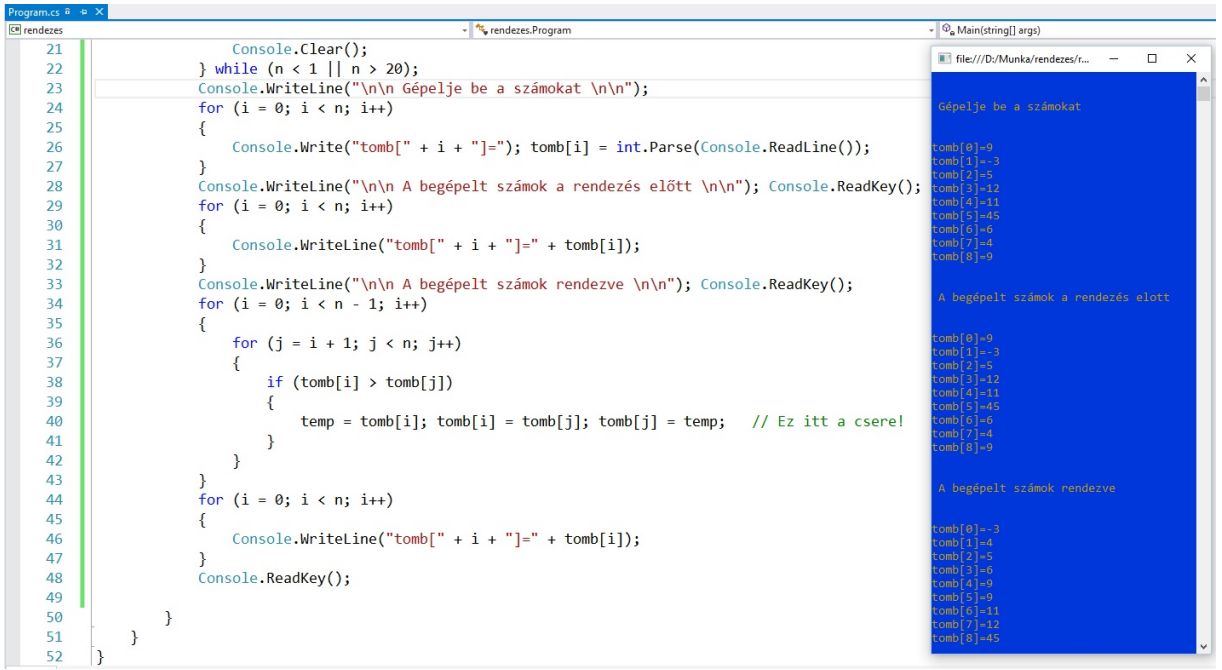
A forráskód következő szakasza a tulajdonképpeni rendező algoritmus. Itt két *for* ciklus került egymásba ágyazva a forráskódba. A külső for*(i=0;i<n-1 ; i++)* ciklus indexe nulláról indul és a tömb utolsó előtti indexű eleméig *(n-2)* növekszik. A belső *for(j=i+1;j<n ; j++)* ciklus indexe az i indexhez képest egyel előre húzva indul és a tömb utolsó indexű eleméig *(n-1)* halad előre. Ez a két ciklus és az indexeik megfelelő beállítása valósítja meg azt a rendező algoritmust, amit a *12. ábrán* láthat az olvasó.

Az algoritmus lényege tulajdonképpen egy folyamatos összehasonlítási folyamat a tömbben szereplő számok között. Ezt az összehasonlítást végzi el a belső *for* ciklusban megadott *if(tomb[i]>tomb[j])* logikai feltétel vizsgálat. Ha talál a j indexel jelölt helyen a tömbben kisebb számot, mint az i indexel jelölt helyen álló szám, akkor a két számot felcseréli.

A szám felcserélése három utasításból áll össze, ami egyetlen sorban került megadásra az *if()* utasításon belül. Itt van szükség a temp nevű int típusú változóra, ami átmeneti tárolóként funkcionál a csere végrehajtásának elősegítésére. A csere során valamelyik indexelt *(i, j)* helyről a tömbből a temp változóba kell átmásolni a cserélendő számok egyikét *temp=tomb[i];* majd a másik indexelt helyről a korábbi indexelt helyre másoljuk a másik számot tomb*[i]= tomb[j]*; és végül a cserét befejezve felül írjuk az utóbbi indexelt hely tartalmát a temp változó tartalmával tomb[j]=temp;.

Az utolsó *for* ciklus feladata a megrendezett számok kilistázása, így a program használója a rendezés után is látja a teljes számtömböt. Ez látható a *14. ábra* alsó részén, ahol a futó program ablaka kék háttérrel függőlegesen kinagyítva látható a forráskód előtt. Az ablak tartalmazza a begépelés során, aztán a rendezés előtt és után is a tömb teljes tartalmát.

14. ábra



Harmadik példa

A harmadik példában egy legfeljebb tíz sorból és oszlopból álló integer elemeket tartalmazó kétdimenziós tömbbe beolvasott számok sor-oszlop cseréjét (transzponált) számítjuk ki. A *Main()* függvény első sorában négy egész *(int)* típusú változót definiálunk, az első kettő ciklus változó *(i, j)*, a másik két változó *(n, m)* pedig a beolvasott számok mennyiségét tartalmazza a sorokban és az oszlopokban. A második sorban két darab kétdimenziós tömböt definiálunk, amelyeknek legfeljebb 10 sora és oszlopa lehet.

A következő sorokban a háttér és előtér színek beállítása és aktiválása után a program a begépelendő sorok és oszlopok számát kéri be a billentyűzetről, ami egy *do while()* ciklusba szervezve került megoldásra. A program abban az esetben lép ki a hátultesztelő ciklusból, ha mind a két begépelt szám egytől tízig terjedő egész szám.

A *do while()* ciklust követő utasítás két egymásba ágyazott *for* ciklus, amelynek feladata a korábban megadott mennyiségű *(n, m)* egész számok beolvasása a kétdimenziós tömbbe. Az első *for* ciklus fejrészében nullázzuk a külső ciklus változót *(i=0)* illetve érdekességképpen itt egy *Console.WriteLine()* utasítás is elhelyezésre került. Mivel a külső *for* ciklus fejrészének első paramétere egyszer hajtódik csak végre, ezért a *Console.WriteLine()* függvényben megadott szöveg egyszer jelenik meg a képernyőn. A ciklusban maradás feltétele a külső ciklusban, hogy az *i* változó értéke kisebb kell legyen, mint az n változó értéke. Látható, hogy a kétdimenziós tömb első sorának indexelése nulláról indul, emiatt kellett az i változó értékét nullára állítani. A belső *for* ciklus hasonlóan épül fel, a különbség csak annyi, hogy a j a ciklusváltozó és az értéke az m változó értékéig növekszik.

```
using System;
namespace matrix_transzp
{  class Program
   { static void Main(string[] args)
     {  int i, j, n, m;
        int[,] matr = new int[10, 10]; int[,] matr_tr = new int[10, 10];
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.ForegroundColor = ConsoleColor.DarkYellow;
        Console.Clear();
        do
        {  Console.WriteLine("Adja meg hány sora legyen a mátrixnak, maximum 10 db lehet");
           Console.Write("n="); n = int.Parse(Console.ReadLine());
           Console.WriteLine("Adja meg hány oszlopa legyen a mátrixnak, maximum 10 db lehet");
           Console.Write("m="); m = int.Parse(Console.ReadLine());
```

```
} while (n < 1 || n > 10 || m < 1 || m > 10);
for (i = 0, Console.WriteLine("mátrix elemeinek beolvasása"); i < n; i++)
{ for (j = 0; j < m; j++)
   { Console.Write("mátrix[" + i + ", " + j + "]="); matr[i, j] = int.Parse(Console.ReadLine());
   }
}
Console.WriteLine(" A begépelt mátrix elemei egy billentyű lenyomása után következik:\n");
Console.ReadKey();
for (i = 0; i < n; i++)
{ for (j = 0; j < m; j++)
   { Console.Write(matr[i, j] +"  ");
   }
  Console.WriteLine("");
}
Console.WriteLine("\n\n A mátrix transzponáltja egy újabb billentyű lenyomása után következik:\n");
Console.ReadKey();
for (i = 0; i < n; i++)
{  for (j = 0; j < m; j++)
   {  matr_tr[j, i]= matr[i, j];
   }
}
for (i = 0; i < m; i++)
{ for (j = 0; j < n; j++)
   { Console.Write(matr_tr[i, j] + "  ");
   }
  Console.WriteLine("");
        }
        Console.ReadKey();
     }
   }
}
```

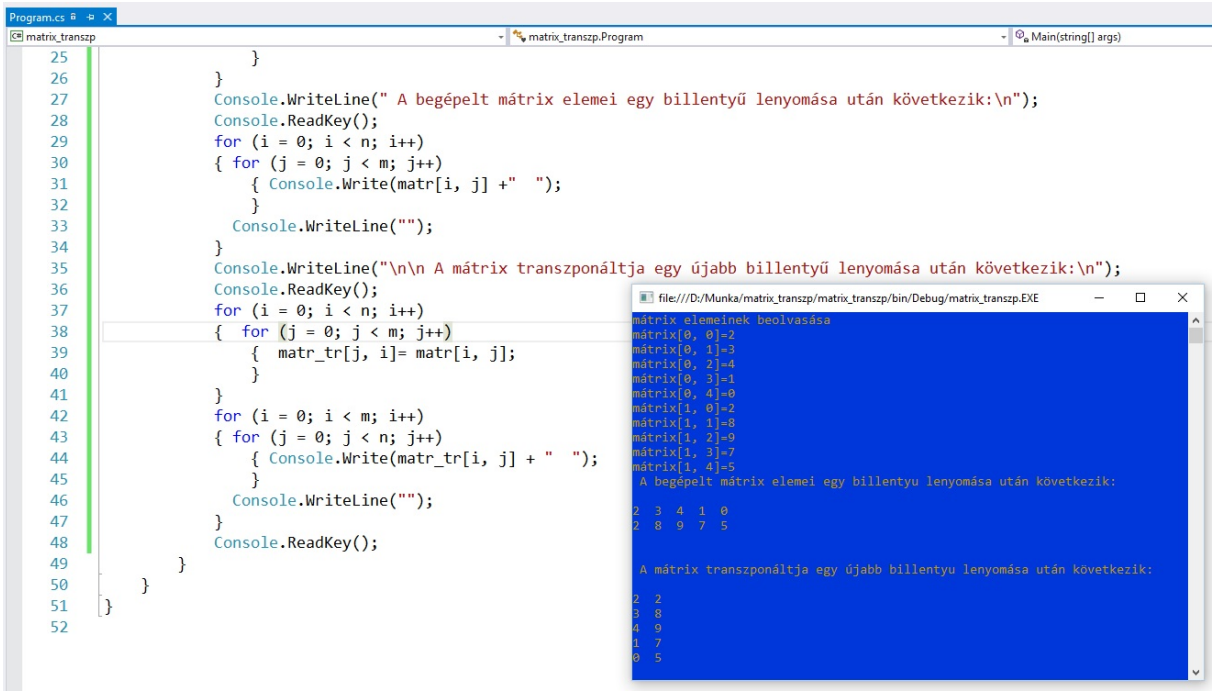
A tömb értékeinek beolvasását a szokásos *Console.Write(), Console.ReadLine()* párossal és a csatolt konverziós függvénnyel int.Parse() oldottuk meg. A *Console.Write("mátrix[" + i + ", " + j + "]=")* utasítás először kiírja a mátrix[szöveget, majd az i változó értékét egy karakterkonverzió után hozzáfűzi a mátrix[szöveghez. Ezt követően egy vessző kerül kiírásra, majd a j változó érték, végül pedig a szögletes zárójel, amit egy egyenlőség jel követ. Például az első sor, harmadik oszlopának kiírása a *mátrix[0, 2]=* formában jelenik meg, ez a *15. ábrán* is jól látható a „mátrix elemeinek beolvasása" szöveg alatt.

A tömb tartalmának feltöltése után a *15. ábrán* is látható módon két sorban és öt oszlopban a kétdimenziós tömb teljes tartalmát kiírja a program. Az ehhez szükséges eszközök hasonlóan a beolvasáshoz, két egymásba ágyazott *for* ciklusból állnak. A két *for* ciklus ciklusváltozóinak beállítása teljesen megegyezik a beolvasás során megadottakkal.

A program következő szakasza végzi el a tulajdonképpeni sor-oszlop cserét. Lényegében a változtatás az előzőkhez képest csak abból áll, hogy a *matr_tr[j,i]* nevű tömbben felcseréltük a két indexet, a *matr[i,j]* nevű tömbhöz képest.

A sor-oszlop csere eredményét szintén két egymásba ágyazott *for* ciklussal listázzuk a képernyőn, de itt van egy lényegi különbség a korábbiakhoz képest. A külső *for* ciklus ciklusváltozója az m változó értékéig növekszik, a belső *for* ciklus ciklusváltozója pedig az *n* változó értékéig.

15. ábra



A sor-oszlop csere eredményét szintén a *15. ábrán* lehet látni a kék hátterű konzol ablak bal alsó sarkában.

7.6. Gyakorló feladatok, kérdések

Soroljon fel olyan algoritmikai feladatokat, amelyeket csak tömbök felhasználásával lehet vagy célszerű megoldani.

Ismertesse általánosságban a tömbök felépítését, jellemzőit.

Hogyan érhetőek el a tömbök egyes elemei?

A tömbök indexei milyen változó típusok lehetnek?

Írja le az egy indexű (egy dimenziós) tömb definíciójának szintaktikai felépítését.

Mutassa be a tömbök operatív tárbeli helyfoglalásával kapcsolatos ismereteket.

Hogyan határozza meg egy tömb operatív tárbeli (RAM) helyfoglalását általánosságban?

Számítsa ki egy 15 elemű *double* típusú elemeket tartalmazó tömb tárfoglalásának nagyságát *C#* nyelvben.

Számítsa ki egy 100 elemű long típusú elemeket tartalmazó tömb tárfoglalásának nagyságát *C#* nyelvben.

Írja le az egy indexű (egy dimenziós) tömb definíciójának szintaktikai felépítését *C#* nyelvben. Hozzon létre egy 15 elemű *double* típusú elemeket tartalmazó tömböt.

Hozzon létre egy 100 elemű long típusú elemeket tartalmazó tömböt.

Adjon meg olyan programnyelvet, ami előkészíti a tömb elemeinek értékeit és olyat, ami nem. Hogyan inicializálja egy tömb elemeit *C#* nyelvben?

Írjon egy példát tömb elemeinek inicializálására *C#* programnyelvben.

Írjon programot, ami a billentyűzetről beolvas tíz darab egész számot egy tömbbe.

Írjon programot, ami a billentyűzetről beolvas öt darab egész számot és ezek reciprokát számítja ki egy tömbbe.

Írjon programot, ami két 20 darab lebegőpontos számot tartalmazó tömb elemeit páronként összeszorozza, majd a szorzatokat összeadja (szorzatösszeg vagy két vektor skaláris szorzata).

Írjon programot, ami csökkenő sorrendbe rendezi egy tetszőleges elemszámú tömb elemeit. 63

Írja le a két indexű (két dimenziós) tömb definíciójának szintaktikai felépítését *C#* nyelvben. Hozzon létre egy 15x5 elemű *double* típusú elemeket tartalmazó kétindexű tömböt. Hozzon létre egy 5x7 elemű long típusú elemeket tartalmazó kétindexű tömböt.

Írja le a listák definíciójának szintaktikai felépítését.

Milyen előnyei vannak a listáknak a tömbökkel szemben?

Hasonlítsa össze a tömbök és a listák használatának fontosabb jellemzőit.

Mi a lista kapacitás?

Melyik tulajdonság tartalmazza a lista kapacitást?

Mi a lista mérete?

Melyik tulajdonság tartalmazza a lista méretét?

Alapértelmezésben mekkora egy lista kapacitása, ha nem adunk meg lista kapacitást?

8. fejezet

8. Fontosabb *C#* függvények

Ebben a fejezetben a fontosabb *C#* függvények, valamint a függvények használatával összefüggő tulajdonságok összefoglaló bemutatására kerül sor.

8.1. Konzolablak kezelésével kapcsolatos függvények és tulajdonságok

Console.Clear() függvény

A *Console.Clear()* függvény letörli a konzolablak teljes tartalmát. A törlés lényegében úgy zajlik, hogy a törlés során a függvény a korábban beállított háttérszínnel tölti ki az ablak karaktereinek pozícióit. Az alapértelmezett hátérszín a fekete, ha a *Console.Clear()* függvényt mindenféle előzetes háttérszín beállítása nélkül hívjuk, akkor a konzolablak háttere egyöntetűen fekete lesz a függvény meghívása után, valamint a kurzor kezdőpozíciója az ablak bal felső sarkába kerül.

ConsoleColor tulajdonság

A *ConsoleColor* tulajdonság felhasználásával tudjuk a konzolablak előtér és háttérszínét módosítani. Széles színválaszték áll rendelkezésre a háttér és a karakterek színének beállítására. Egy nem teljes választékát a rendelkezésre álló színeknek a következő felsorolás tartalmazza: *ConsoleColor.DarkBlue, ConsoleColor.DarkRed, ConsoleColor.DarkGreen, ConsoleColor.DarkYellow, ConsoleColor.LightBlue, ConsoleColor.LightRed, ConsoleColor.LightGreen, ConsoleColor.LightYellow.*

Console.BackgroundColor tulajdonság

A *Console.BackgroundColor* tulajdonság segítségével a konzolablak háttérszínét lehet módosítani, ami alapértelmezésben a fekete. Értéket a következő módon lehet adni ennek a tulajdonságnak. *Console.BackgroundColor= ConsoleColor.DarkBlue*

Console.ForegroundColor tulajdonság

A *Console.ForegroundColor* tulajdonság segítségével a konzolablak előtérszínét lehet módosítani, ami alapértelmezésben a fehér. Értéket a következő módon lehet adni ennek a tulajdonságnak. *Console.ForegroundColor= ConsoleColor.DarkYellow*

Console.ResetColor() függvény

Az alapértelmezett előtér és háttérszíneket állítja be, az aktuálisan futó program számára attól a ponttól ahol meghívásra kerül.

Console.SetWindowSize(width, height) függvény

A konzolablak méretét lehet ezzel a függvénnyel beállítani. Az alapértelmezett ablak méret 80 szélességű és 25 karakterterület magasságú. Ez még a DOS operációsrendszer korából származó méret, amikor a karakteres képernyőt a DOS operációsrendszer ezzel a mérettel kezelte. A *Console.SetWindowSize(int, int)* függvény két egész számot (integer) vár paraméterként, amelyek közül az első az ablak szélessége, a második pedig az ablak magasságát adja meg karakterterület méretben.

Console.SetCursorPosition(x_pos, y_pos) függvény

A konzolablak területén belül lehet ezzel a függvénnyel a kurzor aktuális pozícióját beállítani. A *Console.SetCursorPosition(int, int)* függvény két egész számot (integer) vár paraméterként, amelyek közül az első a kurzor vízszintes (x irányú, oszlop), a második a függőleges (y irányú, sor) koordinátáját adja meg. Az ezt követő *Write() és WriteLine()* függvények ettől a pozíciótól kezdődően írják ki a megadott tartalmat a képernyőre.

Console.Beep(freq, time) függvény

Ez a függvény hangjelzést vált ki abban a programban, ahol meghívásra kerül. A *Console.Beep(int, int)* függvény két egész számot (integer) vár paraméterként, amelyek közül az első a hangjelzés frekvenciáját, a második pedig a hangjelzés időtartamát állítja be mili szekundumokban.

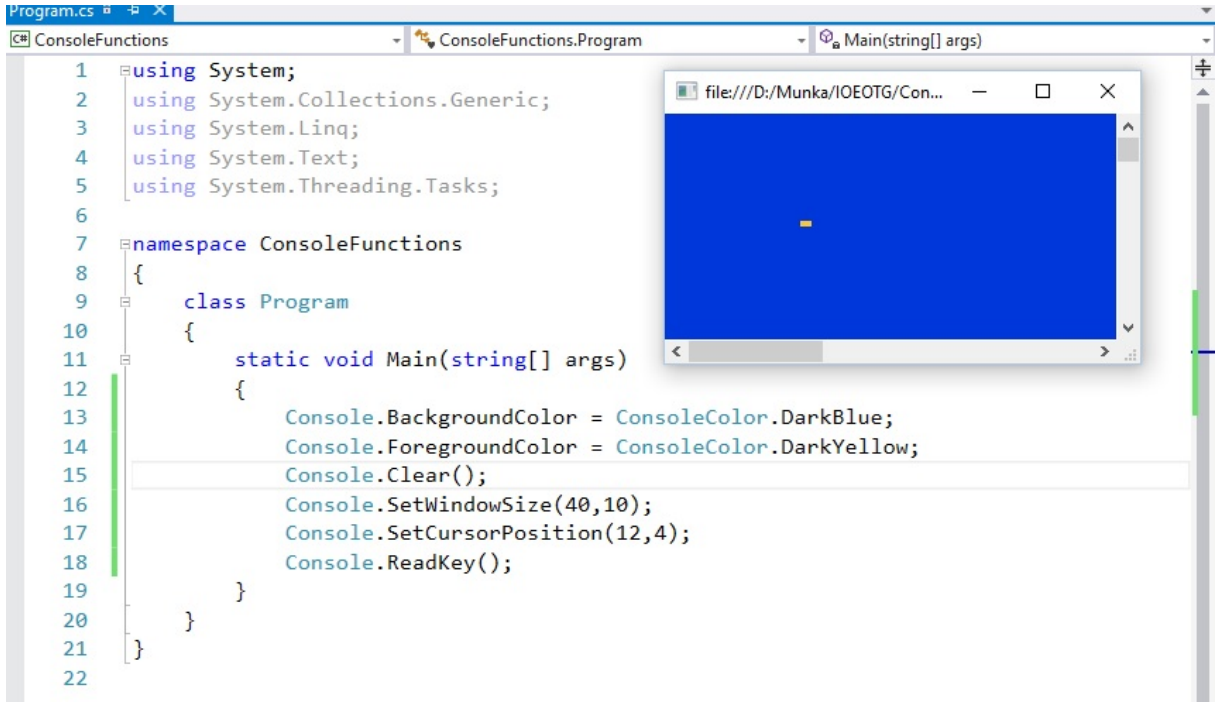
Az itt látható rövid program példa beállítja a konzol ablak méretét, előtér valamit háttér színét, továbbá a kurzor pozíciót is az alapértelmezett helytől (ablak bal felső sarka) eltérő pozícióba állítja.

```
using System;
namespace ConsoleFunctions
{ class Program
   { static void Main(string[] args)
      {  Console.BackgroundColor = ConsoleColor.DarkBlue;
         Console.ForegroundColor = ConsoleColor.DarkYellow;
         Console.Clear();
         Console.SetWindowSize(40,10);|
         Console.SetCursorPosition(12,4);
         Console.ReadKey();
      }
   }
}
```

A *16. ábra* a program futásának eredményét mutatja. Az ablak aktív területének méretét a *.SetWindowsSize()* függvénnyel állítottuk be 40 karakter széles és 10 karakter magasságúra. Az is látható a *16. ábrán,* hogy a kurzor nem az ablak bal felső sarkában, hanem attól jobbra és lefelé került pozícionálásra.

Az ablak kék hátterű, az előtér szín pedig sárga, amit a kurzor színe is igazol. Az ablak a méret beállítások ellenére természetesen futás közben átméretezhető, ha az szükséges, illetve mind vízszintes, mind pedig függőleges görgetősáv is tartozik az ablakhoz.

16. ábra


Program.cs
ConsoleFunctions
ConsoleFunctions.Program
Main(string[] args)
file:///D:/Munka/IOEOTG/Con...
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConsoleFunctions
{
class Program
{
static void Main(string[] args)
{
Console.BackgroundColor = ConsoleColor.DarkBlue;
Console.ForegroundColor = ConsoleColor.DarkYellow;
Console.Clear();
Console.SetWindowSize(40,10);
Console.SetCursorPosition(12,4);
Console.ReadKey();
}
}
}

8.2. Matematikai műveletekkel és feladatokkal kapcsolatos függvények

A *C#* nyelv lehetőséget nyújt a legfontosabb elemi függvények használatára a *Math* osztály felhasználásával. Az egyes függvények használata a következő általános módon zajlik.

Math.függvénynév(paraméterlista)

A fontosabb függvények abc sorrendbe szedve az alábbi felsorolásban láthatók.

Math.Abs(*double* x) függvény

A *Math.Abs(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek abszolút értékét számítja ki a visszatérési értéknek. Példa: *double x=-5, y; y=Math.Abs(x);* Az y értéke 5 lesz.

Math.Ceiling(*double* x) függvény

A *Math.Ceiling(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelyet felfelé kerekít a legkisebb egész értékre és azt adja vissza a hívó függvénynek. Példa: *double x=5.19, y; y=Math.Ceiling(x);* Az y értéke 6 lesz.

Math.Cos(*double* x) függvény

A *Math.Cos(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek koszinuszát számítja ki a visszatérési értéknek. A számot nem fokokban hanem radiánban kell megadni. Példa: *double x=Math.PI, y; y=Math.Cos(x);* Az y értéke -1 lesz. A példában felhasználásra került a Math osztályban definiált π állandó értéke.

Math.Floor(*double* x) függvény

A *Math.Floor(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelyet lefelé kerekít a legnagyobb egész értékre és azt vissza adja a hívó függvénynek. Példa: *double x=5.19, y; y=Math.Abs(x)*; Az y értéke 5 lesz.

Math.Log(*double* x) függvény

A *Math.Log(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek természetes alapú logaritmusát számítja ki visszatérési értéknek. Példa: *double x=Math.E, y; y=Math.Log(x)*; Az y értéke 1 lesz. A példában felhasználásra került egy másik a Math osztályban definiált állandó értéke. Ez az állandó a természetes alapú exponenciális függvény alapja az e.

Math.Log10(*double* x) függvény

A *Math.Log10(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek tízes alapú logaritmusát számítja ki a visszatérési értéknek. Példa: *double x=100, y; y=Math.Log10(x);* Az y értéke 2 lesz.

Math.Max(byte x, byte y) függvény

A *Math.Max(byte x, byte y)* függvény két byte típusú egész számot vár paraméterként, amelyek közül a nagyobbikat adja visszatérési értéknek. Példa: *byte x=-55, y=12, z; z=Math.Max(x, y);* Az z értéke 12 lesz.

Math.Min(byte x, byte y) függvény

A *Math.Min(byte x, byte y)* függvény két byte típusú egész számot vár paraméterként, amelyek közül a kisebbet adja visszatérési értéknek. Példa: *byte x=-55, y=12, z; z=Math.Min(x, y);* Az z értéke -55 lesz.

Math.Pow(*double* a, *double* k) függvény

A *Math.Pow(double alap, double kitevo)* függvény két dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelyek alapján kiszámítja a két szám hatványát és azt adja visszatérési értéknek. Az első paraméter a hatvány alapja, a második paraméter pedig a kitevő. Példa: *double a=1.2, k=2, y; y=Math.Pow(a, k);* Az y értéke 1.44 lesz.

Math.Round(*double* x) függvény

A *Math.Round(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelyet a legközelebbi egész értékre kerekít, és azt adja vissza a hívó függvénynek. Példa: *double x=5.59, y; y=Math.Round(x);* Az y értéke 6 lesz.

Math.Sin(*double* x) függvény

A Math.Sin(*double* x) függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek szinuszát számítja ki a visszatérési értéknek. A számot nem fokokban hanem radiánban kell megadni. Példa: *double x=Math.Pi, y; y=Math.Sin(x/2);* Az y értéke 1 lesz.

Math.Sqrt(*double* x) függvény

A Math.Sqrt(*double* x) függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek négyzetgyökét számítja ki a visszatérési értéknek. Példa: *double x=144, y; y=Math.Sqrt(x);* Az y értéke 12 lesz.

Math.Tan(*double* x) függvény

A *Math.Tan(double x)* függvény dupla pontos lebegőpontos *(double)* számot vár paraméterként, amelynek tangensét számítja ki a visszatérési értéknek. A számot nem fokokban, hanem radiánban kell megadni. Példa: *double x=Math.Pi, y; y=Math.Tan(x/4);* Az y értéke 1 lesz.

Az alábbi rövid példa program a matematikai függvények zömét bemutatja pár numerikus típusú változó felhasználásával.

```
using System;
namespace MatekFunctions
{ class Program
  { static void Main(string[] args)
    {  double x, y, z;
       Console.BackgroundColor = ConsoleColor.DarkBlue;
       Console.ForegroundColor = ConsoleColor.DarkYellow;
       Console.Clear();
       z=-Math.Floor(6.19);
       Console.WriteLine("z=" + z);
       x = Math.Abs(z);
```

```
            Console.WriteLine("x=" + x);
            x = Math.Pow(2, x);
            Console.WriteLine("x=" + x);
            y = Math.Sqrt(x);
            Console.WriteLine("y=" + y);
            x = Math.PI / 6;
            y = Math.Sin(x);
            Console.WriteLine("x=" + x + ", y=" + y);
            x = Math.E * Math.E;
            y = Math.Log(x);
            Console.WriteLine("x=" + x + ", y=" + y);
            Console.ReadKey();
        }
    }
}
```

A példa program a matematikai függvények bemutatására három *double* típusú változót használ. *A .Floor()* függvény kap egy lebegőpontos számot (6.19) aminek értékét lefelé (floor=-padló, földszint) kerekíti, azaz a *.Floor()* függvény által visszaadott érték 6 lesz. De a vissza adott szám nem egész, hanem lebegőpontos típusú. A z nevű változó értéke végül így -6 lesz, amit az *.Abs()* függvény kap meg.

Az *.Abs()* függvény veszi az átvett szám abszolút értékét, amit az *x* nevű változó kap meg. Az *x* nevű változó a *.Pow()* függvény második paramétereként szerepel a következő lépésben. A *.Pow()* függvény első paramétere a hatvány alapja, a második pedig az exponens (kitevő). Az eredmény (64), amit az *x* nevű változó kap kiírjuk a konzol ablakba, ez ott a harmadik sorban látható (x=64).

Az *x* nevű változót ezután a négyzetgyökvonás függvény *.Sqrt()* kapja meg, aminek eredményeként az y nevű változó értéke 8 lesz. Ez az érték is megjelenik a konzol ablak negyedik sorában (y=8).

A *Main()* függvény 13-ik sorában a *Math.PI* tulajdonság értékét kapja meg az *x* nevű változó, ami hordozza a pi állandó közelítő értékét és a következő utasítás sorban alkalmazzuk a .Sin() függvény használatának bemutatására. Fontos észre venni, hogy a *.Sin()* függvény paraméterként radiánban veszi át a keresett ívmértéket és nem fokokban mért értékkel. Mivel a π/6 nevezetes szög, így a *.Sin()* függvény egy racionális számot ad vissza az y nevű változónak. Ezek az értékek megjelennek a konzol ablak ötödik sorában.

17. ábra

Program.cs

MatekFunctions | MatekFunctions.Program | Main(string[] args)

```
5   using System.Threading.Tasks;
6
7   namespace MatekFunctions
8   { class Program
9       { static void Main(string[] args)
10          {   double x, y, z;
11              Console.BackgroundColor = ConsoleColor.DarkBlue;
12              Console.ForegroundColor = ConsoleColor.DarkYellow;
13              Console.Clear();
14              z=-Math.Floor(6.19);
15              Console.WriteLine("z=" + z);
16              x = Math.Abs(z);
17              Console.WriteLine("x=" + x);
18              x = Math.Pow(2, x);
19              Console.WriteLine("x=" + x);
20              y = Math.Sqrt(x);
21              Console.WriteLine("y=" + y);
22              x = Math.PI / 6;
23              y = Math.Sin(x);
24              Console.WriteLine("x=" + x + ", y=" + y);
25              x = Math.E * Math.E;
26              y = Math.Log(x);
27              Console.WriteLine("x=" + x + ", y=" + y);
28              Console.ReadKey();
29          }
30      }
31  }
32
```

file:///D:/Munka/IOEOTG/MatekF...

```
z=-6
x=6
x=64
y=8
x=0,523598775598299, y=0,5
x=7,38905609893065, y=2
```

A *Main()* függvény 16-ik sorában a *Math.E* tulajdonság értékének négyzetét kapja meg az *x* nevű változó. A *Math.E* tulajdonság hordozza a természetes alapú exponenciális függvény *(exp(x))* alapjának mint állandónak a közelítő értékét és a következő utasítás sorban alkalmazzuk a természetesalapú *.Log()* függvény használatának bemutatására. Ennek a műveletnek az eredménye egész szám, amiről a logaritmus azonosságainak felhasználásával meg is győződhetünk. Ennek a műveletnek az eredményei láthatóak a konzol ablak hatodik sorában a *17. ábrán.*

8.3. Listák kezelésével kapcsolatos függvények

A fontosabb listakezelő függvényeket abc sorrendben az alábbiakban lehet látni.

lista_azonosito.Add(tetszőleges adat típus) függvény

Az *.Add()* függvénnyel új elemeket tudunk a listába felvenni. Az új elemet az *.Add()* függvény a lista végén helyezi el. Ha az új elemek hozzáadása során az előre megadott kapacitást meghaladjuk, akkor a lista kapacitása automatikusan bővül.

lista_azonosito.Capacity tulajdonság

A lista kapacitását tartalmazza. Azaz nem a lista tényleges tartalmát jelentő hozzáadott elemek számát adja meg, hanem azt, hogy összesen hány darab elem helyezhető el a listában. A kapacitás automatikusan növekszik az újabb lista elemek hozzáadásával, ha az előre megadott vagy alapértelmezett értéket meghaladjuk.

lista_azonosito.Clear() függvény

A *.Clear()* függvény a lista teljes tartalmát törli. A törlés a lista kapacitását nem befolyásolja, tehát a tárfoglalás nem változik, hiába lett törölve a lista teljes tartalma.

lista_azonosito.Count tulajdonság

A lista méretét tárolja, azaz hány darab elemet tartalmaz a lista, továbbá az *.Add(), .AddRange(), .Insert(), .InsertRange()* valamint *a .Clear()* függvények befolyásolják az értékét.

lista_azonosito.Insert(pozíció, tetszőleges adat típus) függvény

Az *.Insert()* függvény új elemet ad hozzá a listához, a függvény első paraméterében megadott indexű helyen. A megadott pozíción és az utána következő helyeken lévő elemeket egyel tovább (növekvő indexű) helyekre másolja.

lista_azonosito.Remove(tetszőleges adat típus) függvény

Törli a megadott elemet a listából olyan módon, hogy a törölt elem mögött álló elemeket a listában előre húzza.

lista_azonosito.Sort() függvény

Növekvő sorrendbe rendezi a listát az elemek alapján a *.Sort()* függvény.

A következő rövid program egy lista létrehozását és annak használatát mutatja. A *Main()* függvény első sorában egy ciklus változó, a második sorban pedig a lista definíciója látható. A lista *double* típusú elemeket tartalmazhat, ami 10 elemű kapacitással lett létrehozva. Ez az operatívban ténylegesen foglalja a helyet, annak ellenére, hogy a listában itt még egyetlen adat sem lett rögzítve.

A következő három sorban a háttér és az előtérszín beállítása látható, majd a *Main()* függvény hatodik sorában kiíratjuk a képernyőre a lista kapacitását és méretét. Ennek eredménye a *18. ábrán* látható az első kiírt sorban, azaz a listakapacitás értéke 10 a mérete pedig 0, azaz még nem tartalmaz egyetlen elemet sem a lista. Ezután a szöveg után pedig a program használatot segítő információ látható, azaz a felhasználó nyomja le az Enter billentyűt, mivel a program következő sorában egy *Console.ReadKey()* hívás látható.

```
using System;
using System.Collections.Generic;
namespace listak_kezelese
{ class Program
 { static void Main(string[] args)
  { int i;
    List <double> lista = new List <double>(10);
    Console.BackgroundColor = ConsoleColor.DarkBlue;
    Console.ForegroundColor = ConsoleColor.DarkYellow;
    Console.Clear();
    Console.WriteLine("A lista kapacitása: " + lista.Capacity + ", a mérete pedig: " + lista.Count);
    Console.WriteLine("Nyomja meg az Entert, aminek hatására a program három új elemet ad a listához \n\n ");
    Console.ReadKey();
    lista.Add(-3.2);  lista.Add(5);  lista.Add(2);
    Console.WriteLine("A lista kapacitása: " + lista.Capacity + ", a mérete pedig: " + lista.Count);
    for (i = 0; i < lista.Count; i++)
       { Console.WriteLine("lista elemei[ " + i + "]= " + lista[i]);
       }
Console.WriteLine("\nNyomja meg az Entert, aminek hatására a program egy új elemet szúr be a második helyre \n\n
");
    lista.Insert(1,12);
    for (Console.ReadKey(), i = 0; i < lista.Count; i++)
       { Console.WriteLine("lista elemei[ " + i + "]= " +lista[i]);
       }
    Console.ReadKey();
    }
  }
}
```

A *Main()* függvény kilencedik sorában az *.Add()* függvény használata látható, aminek segítségével három számot adunk hozzá a listához. A következő sorban újra kiíratjuk a listakapacitás értékét, ami változatlan maradt, valamint a lista méretét, aminek értéke háromra növekedett. A lista kapacitását a *.Capacity* a méretét pedig a *.Count* tulajdonságok hordozzák, amelyeket a kiíratás során felhasználtunk.

A következő sorban egy *for()* ciklus található, aminek felhasználásával a lista tartalmát jelenítjük meg a képernyőn aminek eredménye a *18. ábrán* látható. A *for()* ciklus logikai állításában a *.Count* tulajdonságot használjuk arra a célra, hogy hány darab ciklusléptet hajtson végre a ciklusszervező utasítás. Fontos megjegyezni, hogy a lista egyes elemei teljesen hasonló módon érhetők el, mint a hagyományos tömbök elemei. Azaz meg kell adni a lista nevét azután pedig szögletes zárójelpárban annak az elemnek az indexét, amit elszeretnénk érni a listából *listanev[index]*. Az indexelés kapcsán itt is tudni kell azt, hogy a lista első elemének indexe nulla értékű.

A *for()* ciklust követő sorban megismétlődik a program használatot segítő információ kiírása, azaz a felhasználó nyomja le az Enter billentyűt. Erre azért van szükség, mivel a következő for() ciklus fejrészének első paraméterében egy *Console.ReadKey()* hívás látható a ciklusváltozó nullázása (i=0) mellett.

Ha az olvasó figyelmesen böngészi a forráskódot, akkor látható, hogy a felhasználót tájékoztató szöveg után azonnal végrehajtódik az *.Insert()* függvény, tehát nem az Enter billentyű lenyomása váltja ki az új adat beillesztését a listába, az még azelőtt megtörténik.

18. ábra

```
using System.Text;
using System.Threading.Tasks;

namespace listak_kezelese
{ class Program
    { static void Main(string[] args)
        { int i;
          List<double> lista = new List<double>(10);
          Console.BackgroundColor = ConsoleColor.DarkBlue;
          Console.ForegroundColor = ConsoleColor.DarkYellow;
          Console.Clear();
          Console.WriteLine("A lista kapacitása: " + lista.Capacity + ", a mérete pedig: " + lista.Count );
          Console.WriteLine("Nyomja meg az entert, aminek hatására a program három új elemet ad a listához\n\n");
          Console.ReadKey();
          lista.Add(-3.2); lista.Add(5); lista.Add(2);
          Console.WriteLine("A lista kapacitása: " + lista.Capacity + ", a mérete pedig: " + lista.Count);
          for (i = 0; i < lista.Count; i++)
            { Console.WriteLine("lista elemei[" + i + "]=" + lista[i]);
            }
          Console.WriteLine("\nNyomja meg az entert, aminek hatására a program egy új elemet szúr be a második helyre\n\n");
          lista.Insert(1, 12);
          for (Console.ReadKey(), i = 0; i < lista.Count; i++)
            { Console.WriteLine("lista elemei[" + i + "]=" + lista[i]);
            }
          Console.ReadKey();
        }
    }
}
```

class System.String
Represents text as a series of Unicode characters.

```
file:///D:/Munka/IOEOTG/listak_kezelese/listak_kezelese/bin/Debug/listak_kezelese.EXE
A lista kapacitása: 10, a mérete pedig: 0
Nyomja meg az entert, aminek hatására a program három új elemet ad a listához


A lista kapacitása: 10, a mérete pedig: 3
lista elemei[0]=-3,2
lista elemei[1]=5
lista elemei[2]=2

Nyomja meg az entert, aminek hatására a program egy új elemet szúr be a második helyre


lista elemei[0]=-3,2
lista elemei[1]=12
lista elemei[2]=5
lista elemei[3]=2
```

8.4. Gyakorló feladatok, kérdések

Sorolja fel a fontosabb a konzol ablakkal kapcsolatos függvényeket.

Hozzon létre egy 50 oszloppal és 10 sorral rendelkező ablakot.

Melyik függvényeket és hogyan használja a feladat megoldásához.

Mi az általános szintaktikai szerkezete a matematikai függvények használatának?

Sorolja fel a fontosabb matematikai feladatokkal kapcsolatos függvényeket.

Milyen ívmértéket kell megadni a trigonometrikus függvényeknek?

Mi a lista a *C#* programnyelvben?

Ismertesse a listák fontosabb tulajdonságait.

Hasonlítsa össze a tömböket és a listákat. Milyen eltéréseket és hasonlóságokat ismer?

Sorolja fel a fontosabb listakezelő függvényeket.

Mi a különbség az *.Add()* és az *.Insert()* függvények között?

9. fejezet

9. Véletlenszám generálás

9.1. Véletlenszám generálásról általánosan

A számítástudomány egyik igen fontos területe azon algoritmusok csoportja, amelyek segítségével véletlenszámokat tudunk előállítani. Ezeket a számokat nem önmagukért hozzuk létre, hanem azokat valamilyen további felhasználás céljából generáljuk. Számos gyakorlati és elméleti alkalmazása létezik a véletlenszámokra alapuló vezérlésnek (Ethernet hálózati szabvány CSMA\CD, CSMA\CA az információátviteli csatornához való hozzáférés megoldására) illetve feladat megoldásnak, mint például a Monte Carlo szimuláció és sztochasztikus folyamatok számítógépes modellezése, szimulációja. Az alábbiakban a véletlenszám generálás legegyszerűbb módszerei kerülnek röviden ismertetésre.

A lineáris kongruens véletlenszám generátorok (Linear Congruential Generator, LCG) igen széles körben elterjedtek, egyszerű alkalmazhatóságuk miatt. Az egyes programfejlesztő rendszerek is többnyire ezt a véletlenszám generálási technikát alkalmazzák egyszerű megvalósíthatóságuk okán. A véletlenszámokat egy egyszerű rekurzív formula segítségével iteratív módon állítják elő.

$$X_n = (aX_{n-1} + b) \bmod m$$

A rekurzív formulában a *mod* a maradékos osztás (moduló osztás) operátorát jelenti. Mivel a formula rekurzív ezért egy inicializáló értékre X_0 van szüksége. Ezt a gyakorlatban a *randomize()* függvény szokta előállítani, amit a véletlenszám generátor inicializálásaként tanítanak az adott programnyelv oktatása során. A randomize() függvény általában az aktuális rendszeridő alapján hozza létre az inicializáló (sheed random number) értéket X_0. Például emiatt van szükség a *C* programfejlesztő rendszerekben a *randomize()* függvény használatakor a *time.h* fejléc állomány használatára is.

Az egyszerű alkalmazhatóságnak viszont komoly ára van, ami abban nyilvánul meg, hogy bizonyos feladatokra, például Monte Carlo szimulációra nem ajánlott a használata, mivel nem kellően hatékony a valódi véletlenszámok előállításában.

A lineáris kongruens véletlenszám generátorok alapvető hátránya a következő. Hajlamos az úgynevezett periódusképződésre. A periódus hossza alkalmasan választott a és b értékek mellett is legfeljebb m-el egyenlő, de rosszul megválasztott *a* és *b* érték esetén egészen kicsi is lehet. Lineáris kongruens véletlenszám generátorok esetén a maximális periódus *(m)* akkor jöhet létre, ha

– b és m relatív prímek.
– a-1 osztható m összes prím tényezőjével
– ha m osztható 4-el, akkor a-1 is osztható 4-el.

A periódusképződést egy példán keresztül a legkönnyebb megérteni.

Legyen a generátor a következő alakú

$$X_n = (7X_{n-1}+1) \bmod 10$$

Nyilvánvalóan az előzőekben ismertetettek alapján a fenti generátor periódus hossza legfeljebb tíz lehet, de amint az majd látható lesz egy kevés számolás után, még sajnos ezt az értéket sem éri el.

Az $X_0 = 1$ induló értékkel a következő számsorozatot adja a fent megadott rekurzív formula, ahol az X_n értékek sorozata 1, 8, 7, 0, 1, 8, 7, 0, 1, ...

$$X_1 = (7*1+1) \bmod 10 = 8$$

$$X_2 = (7*8+1) \bmod 10 = 7$$

$$X_3 = (7*7+1) \bmod 10 = 0$$

...

Ebben az esetben a periódus hossza 4, mivel az inicializáló érték után következő negyedik szám megegyezik az inicializáló értékkel. Ez akkor is igaz, ha az osztónál (10-es) nagyobb egész számmal indítjuk a rekurzív sorozatot. Legyen például $X_0 = 55$ az induló érték, ekkor az X_n értékek sorozata 6, 3, 2, 5, 6, 3, 2, 5, 6, ... Nyilvánvalóan a maradékos osztás osztója értékének növelése a periódus hosszának növekedését is maga után vonja, de ez gyakran nem elég az adott feladat követelményeinek kielégítésére.

A következő táblázat összefoglalja néhány fontosabb és ismertebb fordítóprogram vagy éppen a tématerületet kutató szakember által javasolt, illetve használt LCG generátorok paramétereit.

Forrás	m	a	b
Numerical Recipes	2^{32}	1664525	1013904223
Borland C/C^{++}	2^{32}	22695477	1
glibc (gcc fordítóban)	2^{32}	1103515245	12345
ANSI C: Watcom, ...	2^{32}	1103515245	12345
Borland Delphi	2^{32}	134775813	1
Random Class (Java)	2^{48}	25214903917	11
Donald Knuth	2^{64}	6364136223846793005	1442695040888963407

Az LCG generátorok hiányosságainak kiküszöbölésére számos megoldás született, ezek közül az egyik a Multiple with Carrie (MWC) módszer. Az MWC módszert George Marsaglia dolgozta ki, azzal a célzattal, hogy a közönséges LCG módszer hiányosságait kiküszöbölve, de a számítási módszert nem túlbonyolítva egy a periódusképződésre kevésbé hajlamos véletlenszám generátort alkosson. Az MWC véletlenszám generálási módszer lényege alapvetően megegyezik a LCG módszerrel, azaz rekurzív úton állítja elő az egyes véletlenszámokat. Viszont van két alapvető eltérés az MWC és az LCG módszer között. Az egyik az, hogy nem egy, hanem egyszerre több (akár több ezer) induló értéket (seed values) használ a véletlenszámok előállítására.

A másik fontos különbség pedig az, hogy a lineáris tag eltolási (ofszet) értéke az MWC módszerben nem állandó. Az MWC módszer legnagyobb előnye, hogy egyszerű egész számok közötti műveleteket (integer arithmetic) alkalmazva gyors véletlenszám generálást tesz lehetővé figyelemre méltóan nagy periódus hosszt biztosítva, ami legalább 2^{60}, de akár elérheti a $2^{2000000}$.

Az MWC módszer az X_n , C_n számok egy sorozatát generálja, amit az alábbi rekurzív összefüggésekkel számítunk,

$$X_n = \left(a\, X_{n-r} + C_{n-1}\right) \bmod m,\; C_n = \frac{a\, X_{n-r} + C_{n-1}}{m}, \quad n \geq r,$$

amelynek szüksége van r +1 darab indító értékre $X_0, X_1, X_2,..., X_{r-1}$ és egy Cr_{-1}, amelyek maguk is véletlenszámok. A törtvonallal jelölt osztási művelet egész osztást jelent, azaz a hányados egész részét számítjuk és használjuk fel a további számítások során.

Az MWC algoritmus esetén a periódusképződést egy példán keresztül vetjük össze a közönséges LCG generátorral.

Legyen az LCG generátor a következő alakú $X_n = (7\, x_{n-1} + 1) mod\ 10$, az MWC generátor pedig $X_n = (7X_{n-1} + C_{n-1}) mod 10,$ $C_n = \frac{7\, X_{n-1} + C_{n-1}}{10}$, az r=1 az inicializáló értékek pedig

$X_0 = 1,\ C_0 = 3.$ A rekurzív számsorozat páros első három elemének számítása látható az alábbi összefüggésekben.

$$X_1 = (7 * 1 + 3) \bmod 10 = 0, \qquad C_1 = \frac{7 * 1 + 3}{10} = 1$$

$$X_2 = (7 * 0 + 1) \bmod 10 = 1, \qquad C_2 = \frac{7 * 0 + 1}{10} = 0$$

$$X_3 = (7 * 1 + 0) \bmod 10 = 7, \qquad C_3 = \frac{7 * 1 + 0}{10} = 0$$

Az alábbi táblázat pedig a rekurzív sorozat elemeit tartalmazza addig, amíg a periódusképződés el nem kezdődik. A táblázat első sorában a rekurzív sorozat indexe, a másodikban az ofszet, harmadikban pedig a sorozat értékei, a véletlenszámok találhatóak.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C_i	3	1	0	0	4	6	5	4	0	2	5	6	1	2	1	4	2	3	5
X_i	1	0	1	7	9	7	5	0	4	8	8	1	3	2	6	3	5	7	2

i	19	20	21	22	23	24	25	26	27
C_i	1	6	3	3	1	0	0	4	6
X_i	9	4	4	1	0	1	7	9	7

A generált számsorozatból világosan látható, hogy a periódus hossza ebben az esetben 21 azaz jóval meghaladja az osztó (jelen esetben 10) nagyságát.
pedig a sorozat értékei, a véletlenszámok találhatóak.

9.2. Véletlenszám generálás *C#* programnyelvben

A *C#* programnyelvben a véletlenszámok generálására egy Random nevű osztály és ennek példányosításával egy *Random* típusú objektum létrehozásával van lehetőség. Ez az objektum a *Random* nevű osztály egy példánya. Az objektum a **new** operátor segítségével hozható létre, hasonlóan például a tömbökhöz. Az alábbi rövid példa egy egész és egy lebegőpontos változó létrehozását, valamint egy véletlenszám generátor objektum példányosítását mutatja a *Random* osztályból.

```
int x; double z;
Random rnd= new Random();
x=rnd.Next(1,7);
z=rnd.NextDouble();
```

A kódrészlet első sorában egy egész és egy lebegőpontos számot definiálunk. A második sor mutatja a *Random* osztály példányosítását a **new** operátor felhasználásával. A létrehozott random objektum neve rnd. A harmadik sorban az egész típusú változó *x* kap értéket a .Next() függvény (metódus) felhasználásával. A generált számok értéktartománya 1-től 6-ig terjedhet. Az utolsó sorban a lebegőpontos változó z kap értéket a *.NextDouble()* függvény (metódus) felhasználásával. A *.NextDouble()* függvény a [0, 1) intervallumból generál számokat és adja azt a változónak.

Random.Next(int a), **Random.Next**(int a, int b) függvény

A *Random.Next()* függvény két különféle hívási paraméterlistával is rendelkezik, ahol egész számokat *(int)* vár paraméterként, amelyek a generált számok intervallumát határozzák meg, és a generált számot visszaadja a hívó függvénynek. A visszaadott szám típusa egész szám. Ha a .Next*(n)* függvényt egyetlen paraméterrel hívjuk, akkor [0, n) intervallumból generál számokat, azaz nullától n-1-ig terjedő számot kapjuk vissza a .Next*(n)* függvénytől. Ha a *.Next(a, b)* függvényt két paraméterrel hívjuk, akkor [a, b[intervallumból generál számokat. Példák: *int x; x=Random.Next(10);* Az *x* értéke egy véletlen szám lesz, ami nullától kilencig lehet valamilyen egész szám. int *x; x=Random.Next(10, 20);* Az *x* értéke egy véletlen szám lesz, ami tíztől tizenkilencig lehet valamilyen egész szám.

Random.Next*Double***()** függvény

A *Random.NextDouble()* függvény hívási paraméterlistája üres, mivel a generált számok intervallumát ennél a függvénynél nem tudjuk befolyásolni. A *.NextDouble()* függvény a [0, 1) intervallumból generál számokat és adja azt a hívó félnek illetve változónak. Nagyon fontos, hogy a *Random.NextDouble()* függvény által visszaadott szám típusa nem egész szám, hanem a [0, 1) intervallumban lévő valós szám. Példa: *int x; x=Random.NextDouble();* Az *x* értéke egy véletlen szám lesz, ami nullától egyig lehet valamilyen valós szám, de az egyet mint értéket nem veheti fel.

Annak oka, hogy a *.NextDouble()* függvény miért a [0, 1) intervallumba generál számokat, illetve miért nem érheti el ez a szám az 1 értéket, annak oka a folytonos valószínűségeloszlás függvények matematikai tulajdonságai közt keresendő. A folytonos eloszlásfüggvények értékkészlete a [0, 1] intervallumba esik, mivel a folytonos eloszlásfüggvények csak határértékben ($+\infty$) vehetik fel az 1 értéket.

9.3. Véletlenszám generálás alkalmazásai, példaprogramok

Véletlenszámok generálására három különféle példaprogram látható ebben az alfejezetben, ahol az első egy egyszerű programozási alapfeladat, valamint két további alkalmazás Monte Carlo szimulációs problémák megoldására.

9.3.1. Integer tömb feltöltése véletlenszámokkal

Az alábbi példa program egy maximum harminc elemből álló egész számokat *(int)* tartalmazó tömböt tölt fel véletlenszám generátor által előállított számokkal. A program a véletlenszám generáláson felül a generált számok átlagát is kiszámítja.

A *Main()* függvény első négy sorában a programban használt különféle típusú változók definíciója látható, az ötödik sorban pedig a *Random* osztály pédányosítása. A negyedik sorban hozzuk létre a tömböt, aminek legfeljebb 30 darab egész *(int)* típusú eleme lehet. A 6–8 sorok a konzol ablak előtér és háttérszínének beállítását végzi el.

A *do-while* hátultesztelő ciklust használjuk a generált számok mennyiségének billentyűzetről való bekérésére. Ez látható a *19. ábra* felső részén a futási ablakban, ahol a program az n nevű változó értékének megadására várakozik. Érdemes tanulmányozni a megadott logikai kifejezést a hátultesztelő ciklusban, amely biztosítja, hogy a beolvasandó számok mennyisége ne haladja meg a 30-at és nullánál viszont nagyobb legyen.

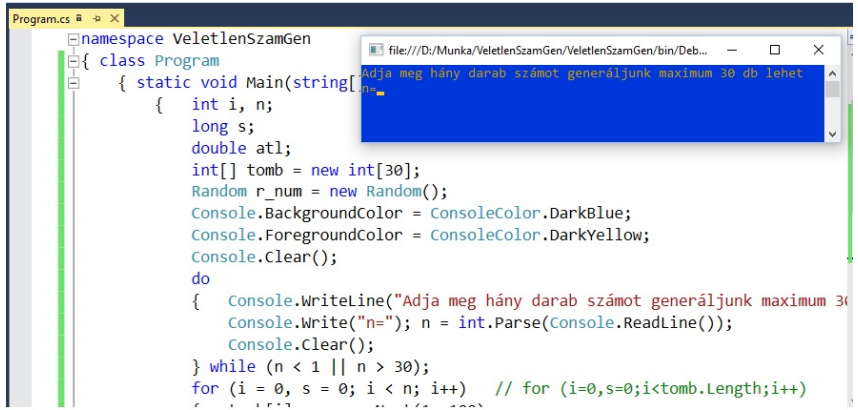
A *for* ciklusban generáljuk egyesével a számokat, tovább Az alábbi példa program egy maximum harminc elemből álló egész számokat *(int)* tartalmazó tömböt tölt fel véletlenszám generátor által előállított számokkal. A program a véletlenszám generáláson felül a generált számok átlagát is kiszámítja továbbá az összegzést is ebben a ciklusban hajtjuk végre. A *.Next(1, 100)* függvény generálja a számokat, amelyek értéke 1 és 100 közé esik. A következő sorban összegezzük is a generált számokat az s nevű változóban, a változó a *for* ciklus fejrészének első blokkjában került kinullázásra. A *for* ciklus utolsó utasítása pedig kiírja a generált számokat a képernyőre, ami a *19. ábra* második felében a futási ablakban látható. A generált számok egymás alatt sorakoznak, a tömbben való elhelyezkedésük sorrendjében.

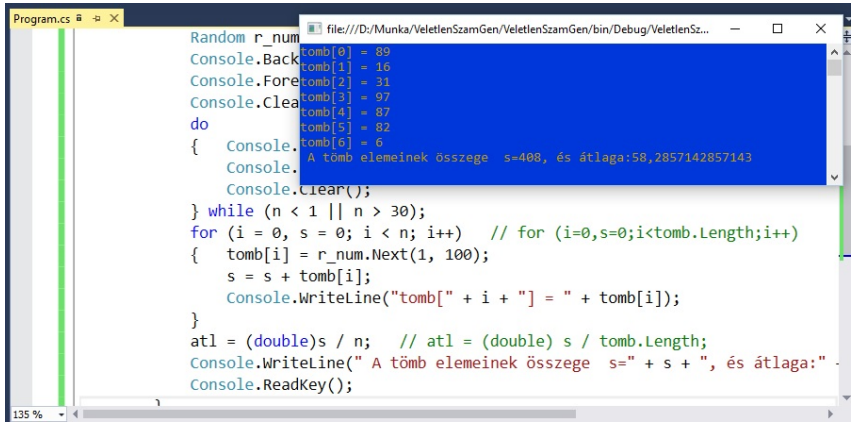
```
using System;
namespace random_gen
{ class Program
        { static void Main(string[] args)
                 { inti,n;
                long s;
                double atl;
                int[] tomb = new int[30];
                Random r_num = new Random(); Console.BackgroundColor = ConsoleColor.DarkBlue;
                Console.ForegroundColor = ConsoleColor.DarkYellow;
                Console.Clear();
                do
```

```
            { Console.WriteLine("Adja meg hány darab számot generáljunk maximum 30 db lehet");
              Console.Write("n=");
              n = int.Parse(Console.ReadLine());
              Console.Clear();
              } while (n < 1 || n > 30);
              for (i=0,s=0;i<n;i++) // for (i=0,s=0;i<tomb.Length;
              i++) { tomb[i]= r_num.Next(1,100);
              s = s + tomb[i];
              Console.WriteLine("tomb["+ i + "] = "+tomb[i]); }
              atl = (double) s / n; // atl = (double) s / tomb.Length;
              Console.WriteLine(" A tömb elemeinek összege s=" + s + ", és átlaga:"+atl);
              Console.ReadKey();
            }
        }
}
```

A *for* ciklus utáni első utasítás számítja ki a generált számok átlagát olyan módon, hogy a *for* ciklusban kiszámított összeget osztjuk a generált számok mennyiségével. Ezután pedig kiírja az összeget és az átlagot a képernyőre, ez látható a *19. ábra* alsó részén.

19. ábra


Program.cs
namespace VeletlenSzamGen
{ class Program
{ static void Main(string[]
{ int i, n;
long s;
double atl;
int[] tomb = new int[30];
Random r_num = new Random();
Console.BackgroundColor = ConsoleColor.DarkBlue;
Console.ForegroundColor = ConsoleColor.DarkYellow;
Console.Clear();
do
{ Console.WriteLine("Adja meg hány darab számot generáljunk maximum 3
Console.Write("n="); n = int.Parse(Console.ReadLine());
Console.Clear();
} while (n < 1 || n > 30);
for (i = 0, s = 0; i < n; i++) // for (i=0,s=0;i<tomb.Length;i++)
file:///D:/Munka/VeletlenSzamGen/VeletlenSzamGen/bin/Deb...
Adja meg hány darab számot generáljunk maximum 30 db lehet
n=


Program.cs
file:///D:/Munka/VeletlenSzamGen/VeletlenSzamGen/bin/Debug/VeletlenSz...
tomb[0] = 89
tomb[1] = 16
tomb[2] = 31
tomb[3] = 97
tomb[4] = 87
tomb[5] = 82
tomb[6] = 6
A tömb elemeinek összege s=408, és átlaga:58,2857142857143
Random r_num
Console.Back
Console.Fore
Console.Clea
do
{ Console.
Console.
Console.Clear();
} while (n < 1 || n > 30);
for (i = 0, s = 0; i < n; i++) // for (i=0,s=0;i<tomb.Length;i++)
{ tomb[i] = r_num.Next(1, 100);
s = s + tomb[i];
Console.WriteLine("tomb[" + i + "] = " + tomb[i]);
}
atl = (double)s / n; // atl = (double) s / tomb.Length;
Console.WriteLine(" A tömb elemeinek összege s=" + s + ", és átlaga:"
Console.ReadKey();
135 %

9.3.2. Pi értékének kiszámítása Monte Carlo módszer segítségével

A Pi értékének kiszámítását az egység sugarú kör területének felhasználásával hajtjuk végre. A számítás alapötlete abból indul ki, hogy ha egy egységoldalú négyzetet és az ebbe beleírt negyedkört nagyon sok homokszemmel vagy hópehellyel lefedünk, akkor a szemek (pelyhek) száma, ami a negyedkörön belülre illetve az egység oldalú négyzetbe esik, arányos a negyedkör illetve a négyzet területével. Ebben a gondolat kísérletben azt feltételezzük, hogy homokszemek egymásra fedésben nem kerülhetnek, azaz egyrétegben fedik le a rendelkezésre álló területet. Ezt az ötletet próbáljuk érzékeltetni a *20. ábrán* elhelyezett pontokkal.

A *20. ábra* egy egység sugarú kört mutat, ami két egység oldalhosszúságú négyzetbe van beágyazva. Ennek a geometria alakzatnak az OABC-vel jelölt négyzet alakú részét használjuk a Pi közelítő értékének kiszámítására. Az OABC négyzet területe 1 egység nagyságú. Ebbe az OABC-vel jelölt négyzet alakú tartományba véletlenszámpárokat (xp, yp) generálunk olyan módon, hogy a generált számok $x_p, y_p \in [0, 1[$ a megadott számhalmazba essenek továbbá vizsgáljuk, hogy a P pont (a generált számpár geometriai szemléltetése) a negyedkörön belülre

$\sqrt{x_p^2 + y_p^2} \leq 1$ vagy kívülre esik. pp

Azokat a „találatokat” amelyek a körön vagy annak belsejében vannak, számláljuk. A sikeres találatok számát a példa programban az *m* változó tartalmazza, az összes esetek számát az *n* változó hordozza.

A Pi közelítő meghatározására a következő összefüggést kell alkalmazni, ami tulajdonképpen a negyedkör (CA ív) területének és az egység oldalú téglalap (OABC) területének hányadosa. A negyedkör (CA ív) területét azon pontok száma reprezentálja (közelíti), amit az *m* változóban tárolunk, az egység oldalú téglalap (OABC) területét pedig az összes generált véletlen szám reprezentálja.

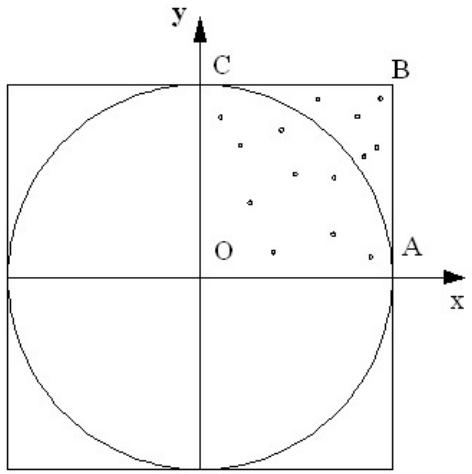
Mivel a kör sugara és a négyzet oldala is egységnyi hosszúságú, így területeik hányadosának értéke $\pi/4$-el egyenlő. Ez az összefüggés pedig már egy egyszerű átrendezéssel lehetővé teszi a π értékének közelítő meghatározását.

$$\frac{\textit{negyedkör területe}}{\textit{négyzet területe}} = \frac{\frac{r^2\pi}{4}}{r^2} = \frac{\pi}{4}$$

Ha a területeket sok-sok ponttal közelítjük, akkor az alábbi összefüggés segítségével számítható a π közelítőértéke.

$$\pi \approx \frac{4\ \textit{CA ív alatti terület}}{\textit{OABC terület}} = \frac{4m}{n}$$

20. ábra



A következő *C#* programnyelven írt kód alkalmas a Pi értékének közelítő kiszámítására. A futtatási tapasztalatok azt mutatják, hogy kb 6 esetleg 7 tizedes jegyig alkalmas a beépített lineáris kongruens generátor a Pi közelítésére, ennél pontosabb értékhez a közönséges lineáris kongruens generátornál jobb véletlenszám generálási technikát kell alkalmazni.

Algoritmikai szempontból a program igen egyszerű felépítésű, mivel egy ciklusból és a ciklusba ágyazott feltételes elágazásból áll. A *Main()* függvény első két sorában a programban használt különféle típusú változók definíciója látható, a harmadik sorban pedig a *Random* osztály pédányosítása. Három egész *(int)* változóra van szükségünk, az első *(i)* egy ciklus változó. Az n változó hordozza a generált pontok mennyiségét, az m változó pedig a negyedkörbe eső pontok mennyiségét. A 4–6 sorok a konzol ablak előtér és háttérszínének beállítását végzi el.

A *do-while* hátultesztelő ciklust használjuk a generált számok mennyiségének billentyűzetről való bekérésére. Az *n* változó itt kap értéket a billentyűzetről, aminek értéke 1-től 100 000-ig terjedhet.

A *for* ciklusban generáljuk párosával a számokat, továbbá a negyedkörön belülre eső pontok számlálását is ebben a ciklusban hajtjuk végre. A két *.NextDouble( )* függvény generálja a számokat, amelyek értéke a [0, 1) intervallumba esik.

Az egy ciklusban lépésben generált két számot egy pont két koordinátájának tekintve az origó és a pont távolsága a Pithagorasz tétel (két pont távolsága derékszögű koordináta rendszerben) segítségével számítható. A *for* ciklusban elhelyezett *if()* utasítás segítségével ellenőrízzük a generált pont helyzetét a negyedkörívhez képest. A *Math.Sqrt()* függvényt használjuk a két pont távolságának
kiszámítására: $\sqrt{x_p^2 + y_p^2} \le 1$.

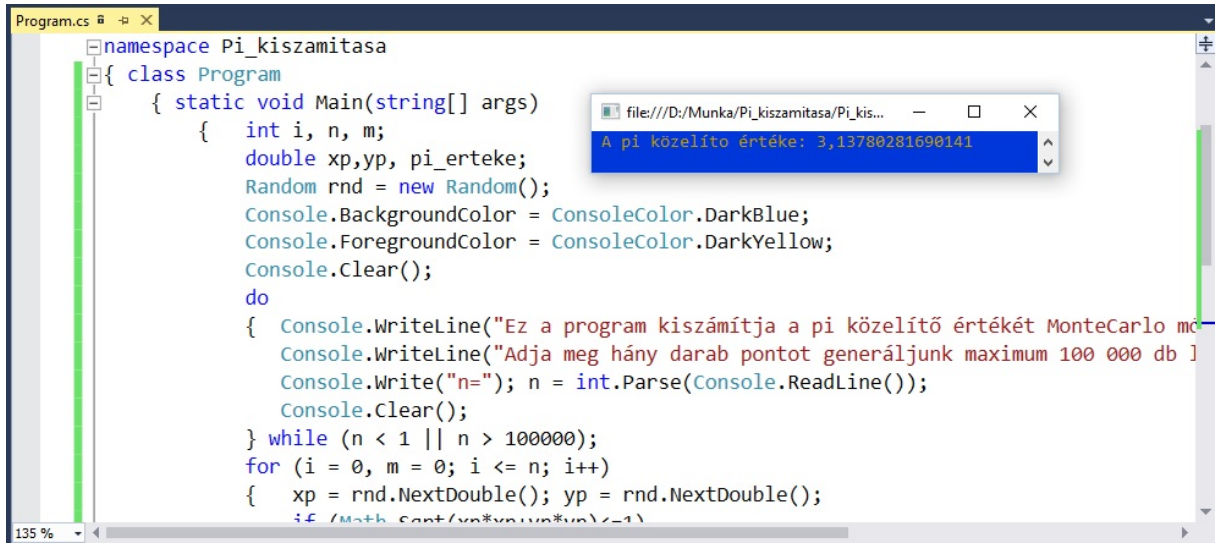
```
using System;
namespace Pi_kiszamitasa
{ class Program
  { static void Main(string[] args)
    {  int i, n, m;
       double xp,yp, pi_erteke;
       Random rnd = new Random();
       Console.BackgroundColor = ConsoleColor.DarkBlue;
       Console.ForegroundColor = ConsoleColor.DarkYellow;
       Console.Clear();
       do
       { Console.WriteLine("Ez a program kiszámítja a pi közelítő értékét MonteCarlo módszerrel");
         Console.WriteLine("Adja meg hány darab pontot generáljunk maximum 100 000 db lehet");
         Console.Write("n="); n = int.Parse(Console.ReadLine());
         Console.Clear();
       } while (n < 1 || n > 100000);
       for (i = 0, m = 0; i <= n; i++)
       {  xp = rnd.NextDouble(); yp = rnd.NextDouble();
          if (Math.Sqrt(xp*xp+yp*yp)<=1)
          { m++;
          }
       }
```

```
            pi_erteke = 4.0*m / n;
            Console.WriteLine(" A pi közelítő értéke: " + pi_erteke);
            Console.ReadKey();
        }
    }
}
```

A *for* ciklus után számítjuk ki a π értékének közelítő értékét a *20. ábra* előtt megadott összefüggés felhasználásával.

A π értékének közelítő értékét a *21. ábrán* látható forráskód részlet előtt látható konzol ablakban írattuk ki. A kiírást elvégző utasítás a *Console.WriteLine()* az utolsó előtti a program forráskódjában, amit egy *Console.ReadKey()* követ.

21. ábra



9.3.3. Integrálás Monte Carlo módszerrel

Számos gyakorlati alkalmazási terület igényli különféle integrálok kiszámítását. Ezek között a feladatok között gyakran előfordulnak olyanok, amelyekhez bonyolult, komplex geometriájú tartományok tartoznak. A másik jellemzően nehezen kezelhető probléma kör, amikor a feladat sokdimenziós, jóval több mint a műszaki alkalmazási területeken általában megszokott két esetleg háromdimenziós problémák. A sokdimenziós komplex geometriájú problémák integrálása a hagyományos rácsgenerálásra alapuló módszerekkel gyakran feloldhatatlanul bonyolult, számításigényes. A Monte Carlo módszer egy rácsnélküli numerikus integrálási lehetőséget kínál sokdimenziós komplex geometriájú problémák megoldására.

Általában egy integrálási feladat a következő n független változóból álló integrál kiszámításából áll, ahol V egy tetszőleges n dimenziós tartomány, amely felett az f függvény korlátos. A tartomány többnyire, főként a különféle gyakorlati alkalmazások során komplex felépítésű, ami alatt azt értjük, hogy nem lehet a tartomány peremeit egyszerű állandókkal megadni, mint például egy téglalap, téglatest, kör, körhenger vagy gömb esetében, sőt gyakran még különféle függvényekkel sem lehet megadni a tartományt határoló peremeket.

$$I = \int_V f(x_1, x_2, \ldots, x_n)\, dV$$

Az integrálokat a Monte Carlo módszert alkalmazva a következő egyszerűsített tartományokon számítjuk ki.

$$I = \int_{a_1}^{b_1}\int_{a_2}^{b_2} \ldots \int_{a_n}^{b_n} f(x_1, x_2, \ldots, x_n)\, dx_1 dx_2 \ldots dx_n$$

A következő példákban nem láthatók komplex geometriájú feladatok, mivel ez a tananyag alapvetően a programozási alapokat igyekszik bemutatni, a Monte Carlo módszer csak egy érdekes alkalmazás.

A fenti határozott integrál az alábbi közelítő formulával számítható

$$I \approx \frac{V_n}{N} \sum_{i=1}^{N} f(x_{1,i}, x_{2,i}, \ldots, x_{n,i}),$$

ahol az $x_{1,i}$, $x_{2,i}$,..., $x_{n,\mathrm{i}}$ szám n-es, véletlenszám generátorral előállított számok sorozata, amelyből $N >> 1$ számút hozunk létre. A V_n egy n dimenziós térfogati tartomány, amin az integrálást végre kívánjuk hajtani.

Egyszerűen szavakkal kifejezve a következőket kell tenni az integrál kiszámítására.

– Generáljunk véletlenszerűen pontokat a feladatként kitűzött integrálási tartományban és határozzuk meg ezeken a helyeken az integrálandó függvény értékét.

– Számítsuk ki a függvény átlagos értékét alapulvéve az összes pontot, amit generáltunk.

– Szorozzuk meg a függvény átlagos értékét annak a „hiper" tartománynak a térfogatával, ami felett a határozott integrált ki akarjuk számítani.

A „hiper" tartomány térfogata alatt egydimenzióban hosszt, két dimenzióban területet, három dimenzióban a hétköznapi értelemben vett térfogatot értjük. Ettől magasabb fokú dimenzió tartományokban nincsen hétköznapi fogalom az általános értelemben vett térfogat megnevezésére, ezért használjuk a „hiper" jelzőt rá.

A különféle példák ellenőrizhetősége miatt olyan integrálási feladatok láthatók a következőkben, amelyek primitív függvényei előállíthatók és az integrálok egzakt módon számíthatók. Az alábbi minta példák láthatók *C#* programnyelven írt kódokkal a következő integrálok kiszámítására

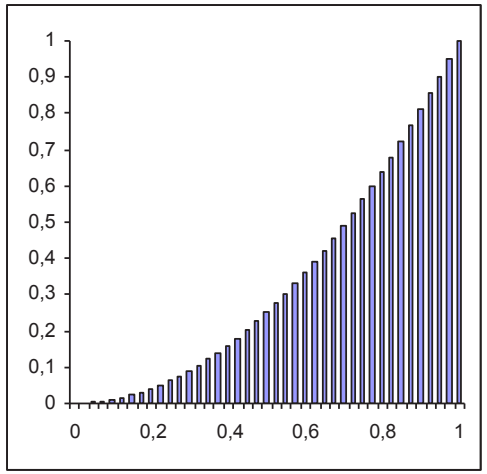
$$\int_a^b x^2\,dx,\ \int_a^b \sin(x)\,dx,\ \int_{xa}^{xb}\int_{ya}^{yb} x^2+y^2\,dxdy.$$

Az alábbi konkrét integrálási határral célszerű tesztelni az x^2 függvény határozott integrálját. A különféle alkalmazásokból is eléggé közismert, hogy a [0, 1] intervallumon az x^2 függvény alatti terület nagysága 1/3.

$$\int_0^1 x^2\,dx = \left[\frac{x^3}{3}\right]_0^1 = \frac{1}{3}$$

A *22. ábra* az x^2 függvényt mutatja a [0, 1] intervallumon, továbbá a határozott integrál geometriai szemléltetését is, a függvény alatti területet is ábrázolja.

22. ábra



A Monte Carlo módszer alkalmazása a numerikus integrál kiszámítására szemléletesen úgy fogható fel, mintha a függvény alatti területet úgy közelítenénk, hogy az integrálási tartományba véletlenszerűen generált számok fölé végtelenül vékony sávokat rajzolunk az *x* tengelytől a függvény értékének megfelelő hosszúsággal. Ha a sávok száma nagyon nagy akkor a határozott integrált kellő pontossággal közelítjük.

Algoritmikai szempontból a program igen egyszerű felépítésű, mivel mindössze egy ciklusból áll. A *Main()* függvény első két sorában a programban használt különféle típusú változók definíciója látható, a harmadik sorban pedig a *Random* osztály pédányosítása. Két egész *(int)* típusú változóra van szükségünk, az első *(i)* egy ciklus változó, a másik az *n* változó hordozza a generált számok mennyiségét.

A második sorban öt darab lebegőpontos változót definiálunk. Az első változó az *x* hordozza a tetszőleges értelmezési tartománybeli elemet, a második változó az *f_x* a függvény értéke az *x* változó értékénél. A harmadik (a) és a negyedik (b) az integrálási határokat tartalmazza, az utolsó pedig a függvényértékek összegét hordozza. A 4–5 kódsorok a konzol ablak előtér és háttérszínének beállítását végzi el.

A *do-while* hátultesztelő ciklust használjuk a generált számok mennyiségének billentyűzetről való bekérésére. Az n változó itt kap értéket a billentyűzetről, aminek értéke 1-től 100 000-ig terjedhet.

A *do-while* ciklus után kérjük be a billentyűzetről az integrálási határokat. Praktikus először a programot olyan tartományon tesztelni, amit könnyedén tudunk ellenőrizni.

A *for* ciklusban generáljuk a véletlen számokat a *.NextDouble( )* függvény felhasználásával, amelyek értéke a [0, 1) intervallumba esik. Viszont az integrált tetszőleges [a, b] intervallumon szeretnénk kiszámítani. Emiatt szükséges a [0, 1) intervallum, a generált véletlenszámok transzformációja az [a, b] intervallumra. Ezt a transzformációt az alábbi összefüggéssel hajtjuk végre.

$$x = a + (b - a)\, r, \quad r \in [0,1)$$

Annyit fontos megjegyezni, hogy a transzformáció az intervallum felső határát (b) nem állítja elő sohasem, de ez a Monte Carlo szimuláció eredményét nem befolyásolja. Az adott ciklus lépésben létrehozott számot *x* felhasználva kiszámítjuk a függvény értékét ezen a helyen. Ezután az s változó értékéhez hozzá adjuk a függvény aktuális értékét.

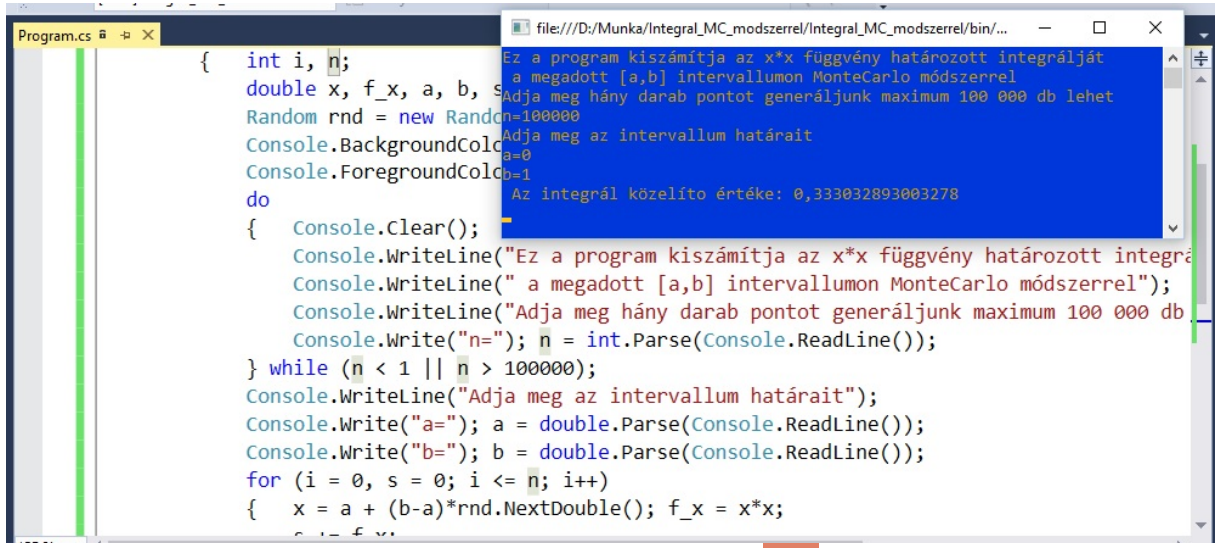
A *for* ciklus utáni utasítás tulajdonképpen az integrálközelítő formula reprezentációja a forráskódban. Ebben az integrálási feladatban a hipertérfogatot most a *(b - a)* mennyiség reprezentálja.

```
using System;
namespace Integral_MC_modszerrel
{ class Program
  { static void Main(string[] args)
    {  int i, n;
       double x, f_x, a, b, s;
       Random rnd = new Random();
       Console.BackgroundColor = ConsoleColor.DarkBlue;
       Console.ForegroundColor = ConsoleColor.DarkYellow;
       do
       {  Console.Clear();
          Console.WriteLine("Ez a program kiszámítja az x*x függvény határozott int
          Console.WriteLine(" a megadott [a,b] intervallumon MonteCarlo módszer
          Console.WriteLine("Adja meg hány darab pontot generáljunk maximum 1(
          Console.Write("n="); n = int.Parse(Console.ReadLine());
       } while (n < 1 || n > 100000);
```

```
            Console.WriteLine("Adja meg az intervallum határait");
            Console.Write("a="); a = double.Parse(Console.ReadLine());
            Console.Write("b="); b = double.Parse(Console.ReadLine());
            for (i = 0, s = 0; i <= n; i++)
            {  x = a + (b-a)*rnd.NextDouble(); f_x = x*x;
               s += f_x;
            }
            s = s * (b-a)/n;
            Console.WriteLine(" Az integrál közelítő értéke: " + s);
            Console.ReadKey();
        }
    }
}
```

A program futása során megjelenő, illetve bekért információk a *23. ábrán* látható futási képernyőn a kék hátterű ablakban kerültek kiírásra. Az eredmény jól mutatja, hogy 100 000 darab generált szám segítségével három tizedes pontossággal tudtuk előállítani az egyébként ismert eredményt, ami 1/3.

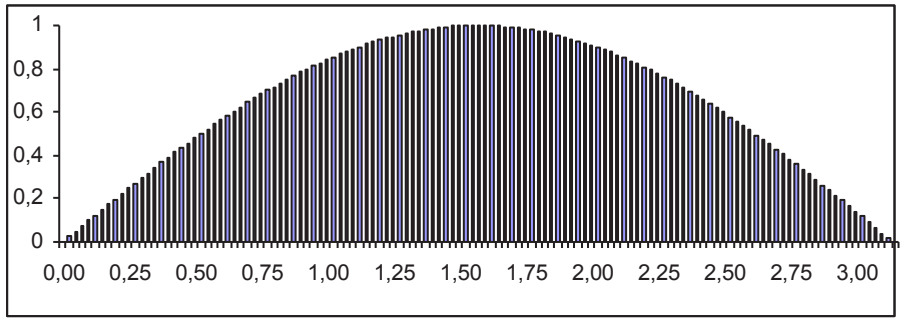
23. ábra



A második példában a *sin(x)* függvény határozott integráljának Monte Carlo módszerrel való kiszámítására készített C# kód látható. Ez a példa is könnyen ellenőrizhető, mivel a szinusz félhullám alatti terület két egység nagyságú a [0, π] intervallumon.

$$\int_0^{\pi} \sin(x)\,dx = \left[-\cos(x)\right]_0^{\pi} = 2$$

24. ábra



A *24. ábrán* látható a függvény az integrálási tartománnyal, valamint a határozott integrál szemléletes jelentés tartalmával. Az ábrázolásban a terület lefedése itt is szándékosan sávosan került ábrázolásra érzékeltetve a diszkrét (nem folytonos) közelítését a határozott integrálnak.

A C# forráskód az alábbiakban látható, ami csak pár ponton tér el az előzőekben az x^2 függvényre írt programtól. Az eltérés egyrészről a kiírt információk tartalmában van, illetve a *for()* ciklusban meghívásra került *Math.Sin(x)* függvény.

```
using System;
namespace Integral_MC_modszerrel
{ class Program
  { static void Main(string[] args)
    {  int i, n;
       double x, f_x, a, b, s;
       Random rnd = new Random();
       Console.BackgroundColor = ConsoleColor.DarkBlue;
       Console.ForegroundColor = ConsoleColor.DarkYellow;
       do
       {  Console.Clear();
          Console.WriteLine("Ez a program kiszámítja az sin(x) függvény határozott integr
          Console.WriteLine(" a megadott [a,b] intervallumon MonteCarlo módszerrel");
          Console.WriteLine("Adja meg hány darab pontot generáljunk maximum 100 000
          Console.Write("n="); n = int.Parse(Console.ReadLine());
       } while (n < 1 || n > 100000);
       Console.WriteLine("Adja meg az intervallum határait");
       Console.Write("a="); a = double.Parse(Console.ReadLine());
       Console.Write("b="); b = double.Parse(Console.ReadLine());
       for (i = 0, s = 0; i <= n; i++)
       {  x = a + (b-a)*rnd.NextDouble(); f_x = Math.Sin(x);
          s += f_x;
       }
       s = s * (b-a)/n;
       Console.WriteLine(" Az integrál közelítő értéke: " + s);
       Console.ReadKey();
    }
  }
}
```

Ebben az integrálási feladatban is a hipertérfogatot a *(b - a)* mennyiség reprezentálja.

Az utolsó példa program pedig egy olyan két független változós függvény határozott integráljának kiszámítását végzi el, amelynek primitív függvénye szintén könnyedén előállítható. Ez a függvény az $f(x,y)= x^2 + y^2$, aminek integrálját analitikus módon is tudjuk számítani. Az alábbiakban az *f(x,y)* függvény határozott integráljának két különböző tartományon való előállítása látható. Ezek az értékek felhasználhatóak a kód tesztelésére, ellenőrzésére.

$$\int_0^1\int_0^1 x^2 + y^2 \; dxdy = \int_0^1 \left[x^2 y + \frac{y^3}{3} \right]_0^1 dx = \int_0^1 x^2 + \frac{1}{3} \, dx = \left[\frac{x^3}{3} + \frac{1}{3} x \right]_0^1 = \frac{2}{3},$$

$$\int_{-1}^1\int_{-1}^1 x^2 + y^2 \; dxdy = \frac{8}{3}$$

A *C#* forráskód az alábbiakban látható, viszont ez a kód már számos ponton eltér a korábbiakban az x^2 és a *sin(x)* függvényekre írt programtól.

Algoritmikai szempontból ez a program is igen egyszerű felépítésű, mivel ugyan úgy mint a korábbi kódok mindössze egyetlen ciklusból áll. A *Main()* függvény első két sorában a programban használt különféle típusú változók definíciója látható, a harmadik sorban pedig a *Random* osztály példányosítása. Ebben a programban is két egész *(int)* típusú változóra van szükségünk, az első *(i)* egy ciklus változó, a másik az n változó hordozza a generált számok mennyiségét.

A második sorban nyolc darab lebegőpontos változót definiálunk. Az első két változó az *x* és az y hordozza a tetszőleges értelmezési tartománybeli elemet, a harmadik változó az *f_xy* a függvény értéke az *x* és y változók értékénél. A negyedik változótól a hetedikig *(xa, xb, ya, yb)* az integrálási határokat tartalmazza, az utolsó pedig a függvényértékek összegét hordozza. A 4–5 kódsorok a konzol ablak előtér és háttérszínének beállítását végzi el.

A *do-while* hátultesztelő ciklust használjuk a generált számok mennyiségének billentyűzetről való bekérésére. Az *n* változó itt kap értéket a billentyűzetről, aminek értéke 1-től 100 000-ig terjedhet.

A *do-while* ciklus után kérjük be a billentyűzetről az integrálási határokat. Praktikus először a programot olyan tartományon tesztelni, amit könnyedén tudunk ellenőrízni, például az előző oldalon kiszámított példákkal.

A *for()* ciklusban generáljuk a véletlen számokat a *.NextDouble( )* függvény felhasználásával, amelyek értéke a [0, 1) intervallumba esik. Viszont az integrált tetszőleges *[xa, xb]* × *[ya, yb]* intervallumon szeretnénk kiszámítani. Emiatt szükséges a [0, 1) intervallum, a generált véletlenszámok transzformációja az *[xa, xb]* × *[ya, yb]* intervallumra. Ezt a transzformációt az alábbi összefüggésekkel hajtjuk végre, ahol az r szimbólum hordozza a véletlenszám generátor által előállított számot.

$$x = xa + (xb - xa)\, r, \quad y = ya + (yb - ya)\, r, \quad r \in [0,1)$$

```
using System;
namespace Integral_MC_modszerrel
{ class Program
  { static void Main(string[] args)
    {  int i, n;
       double x, y, f_xy, xa, xb, ya, yb, s;
       Random rnd = new Random();
       Console.BackgroundColor = ConsoleColor.DarkBlue;
       Console.ForegroundColor = ConsoleColor.DarkYellow;
       do
       {  Console.Clear();
          Console.WriteLine("Ez a program kiszámítja az x*x+y*y függvény határozott integrálját");
          Console.WriteLine(" az [xa, xb] × [ya, yb]  intervallumon MonteCarlo módszerrel");
          Console.WriteLine("Adja meg hány darab pontot generáljunk maximum 100 000 db lehet");
          Console.Write("n="); n = int.Parse(Console.ReadLine());
       } while (n < 1 || n > 100000);
       Console.WriteLine("Adja meg az intervallum határait");
       Console.Write("xa="); xa = double.Parse(Console.ReadLine());
       Console.Write("xb="); xb = double.Parse(Console.ReadLine());
       Console.Write("ya="); ya = double.Parse(Console.ReadLine());
```

```
            Console.Write("yb="); yb = double.Parse(Console.ReadLine());
            for (i = 0, s = 0; i <= n; i++)
            {   x = xa + (xb-xa)*rnd.NextDouble();   y = ya + (yb-ya)*rnd.NextDouble();
                f_xy = x*x + y*y;
               s += f_xy;
            }
            s = s * (xb-xa) * (yb-ya) / n;
            Console.WriteLine(" Az integrál közelítő értéke: " + s);
            Console.ReadKey();
        }
    }
}
```

Ebben az integrálási feladatban a hipertérfogatot a *(xb - xa) (yb - ya)* mennyiség reprezentálja.

9.4. Gyakorló feladatok, kérdések

Ismertessen néhány lehetőséget a véletlenszámok alkalmazásának.

Hogyan tudunk matematikai, algoritmikai úton véletlenszerű (pseudo random) számokat előállítani? Ismertesse a lineáris kongruens véletlenszám (LCG) generátort.

Melyek a korlátai a lineáris kongruens véletlenszám generátoroknak?

Mi a ciklus képződés az LCG generátoroknál?

Melyik paraméter az, amelyik alapvetően behatárolja a ciklus periódus hosszát az LCG generátoroknál?

Hozzon létre egy egyszerű (a paraméterei legyenek egy jegyű számok) lineáris kongruens véletlenszám generátort, majd mutassa meg kézi számolással annak működését.

Mutassa be a Multiply With Carry (MWC-LCG) véletlenszám generátort.

Hozzon létre egy egyszerű (a paraméterei legyenek egy jegyű számok) MWC-LCG véletlenszám generátort, majd mutassa meg kézi számolással annak működését.

Hogyan generálunk véletlenszámokat *C#* programnyelvben?

Ismertesse a Random osztály fontosabb tagfüggvényeit (metódusait).

A Random osztály melyik tagfüggvényét (metódusát) használjuk egész számok generálására?

A Random osztály melyik tagfüggvényét (metódusát) használjuk, valós számok generálására?

A *.NextDouble()* függvény miért a [0, 1) intervallumba generál számokat?

Írjon programot, ami generál 100 darab véletlen számot és kiválasztja közülük a legnagyobb értékűt. Írjon programot, ami generál 50 darab véletlen számot és listázza közülük az 5-tel osztható számokat. Ismertesse a π értékének közelítő meghatározását véletlenszám generátor felhasználásával.

Mi az alapötlet a π értékének közelítő meghatározására véletlenszám generátor használata során?

Írjon programot, ami kiszámítja a cos függvény tetszőleges intervallumon vett határozott integrálját véletlen számgenerátor felhasználásával.

10. fejezet

10. Karaktertömbök *(string)* kezelése

10.1. Karaktertömbök általános leírása

Karaktertömböt létrehozhatunk a szokásos módon a *char* elemi típus felhasználásával, illetve egy speciális a *C#* programfejlesztő nézőpontjából típusnak tekinthető *string* felhasználásával. Ez utóbbi használata számos könnyebbséget jelent a hagyományos karakterekből álló tömb használatával szemben.

Az összefüggő szövegek kezelésére tehát egy speciális, karakterekből álló tömböt definiál a *C#* nyelv, amit *string*-nek nevezünk. A *string* kulcsszó segítségével lehet egy-egy karaktertömböt létrehozni olyan módon, mintha a *string* egy közönséges elemi típus lenne. Ezután magával a létrehozott karaktertömbbel úgy tudunk bánni, mintha az a megszokott módon közönséges elemi típus felhasználásával definiált tömb lenne, sőt mintha az csak egy szimpla változó lenne. Maga a *string*, a valóságban egy osztály, aminek példányosításával (objektum) jön létre egy-egy karaktertömb. Ezután lehetőség nyílik a karaktertömbökkel kapcsolatos összes speciális függvény (metódus), valamint praktikusan felhasználható tulajdonságok használatára. Fontos jellemzője még a *string* típusú karaktertömböknek, hogy a *string*-ek hossza módosítható, azaz lehetőség van újabb karakterek hozzáadására a *string* hosszának kiterjesztésével a már korábban begépelt meglévő karakterek után.

Karaktertömbök létrehozása:

```
string szoveg;
string masikszoveg=”szia”;
char[] karakterek = new char[5];
```

Az első utasítás létrehoz egy szoveg nevű *string* típusú változót (karaktertömböt), aminek tartalma a létrehozás után üres értéket, illetve tartalmat a későbbiek során egy közönséges értékadó utasítás segítségével kaphat.

A második sorban egy *masikszoveg* nevű *string* típusú változót hozunk létre, ami azonnal értéket ”szia” is kap. A második utasításban jól látható, hogy a közönséges értékadó utasítás segítségével lehetséges egyszerre több karaktert is átadni a *string* típusú változónak. Ez egy igen hasznos és kényelmes dolog, mivel ha csak egy hagyományos karaktertömböt hozunk létre, akkor annak a megfelelő tartalommal való feltöltéséhez minimum egy ciklusszervező utasításra van szükség.

Az utolsó sorban pedig egy hagyományos karaktertömb definíciója látható, amelynek hossza karakterekben 5 egység, byte-okban pedig 10 egység, mivel a *C#* nyelvben egy *char* típusú változó két byte-ot foglal az operatív tárban. Ennek a tömbnek a tartalommal való feltöltéséhez külön szükséges egy *for* vagy *while* ciklus.

A *string* típusú karaktertömbök egyes elemei (karakterei), hasonlóan a hagyományos karaktertömbökhöz indexelt módon érhetők el. Az index a *string*-ekben is nulláról indul, azaz a *string*-ben első helyen álló karakter indexe 0, a második pedig 1 és így tovább a *string* végéig.

```
string masikszoveg=”szia”;
char a, b;
a= masikszoveg[0]; b= masikszoveg[1]; // a=”s” és b=”z”
```

A felsorolt függvényeket és tulajdonságokat egy *szoveg_azonosito* nevű *string* típusú „változó” felhasználásával mutatjuk be. A fontosabb *string*-ek használatával kapcsolatos függvények a következők.

– szoveg_azonosito.Insert(pozíció(hely)ahovákerül,abeillesztendőszöveg) .Insert(int , *string*)

A *szoveg_azonosito* nevű *stringben* az első paraméterben megadott pozíciótól kezdődően a második paraméterben megadott karaktereket illeszti be az Insert() függvény, a *szoveg_azonosito* nevű *stringbe*. Az alább látható példában a szoveg nevű *stringben* megadott szöveg elé három darab karaktert a „*C#* ” illesztjük be.

példa:

```
string szoveg=”Programfejlesztő rendszer”, uj_szoveg; uj_szoveg=szoveg.Insert(0, ”C# ”);
// C# Programfejlesztő rendszer
```

– szoveg_azonosito.IndexOf(keresettszöveg,pozíciótól(helytől)keresse) .IndexOf(*string* , int)

A *szoveg_azonosito* nevű *stringben* az első paraméterben megadott szöveget a második paraméterben megadott pozíciótól kezdődően keresi az IndexOf() függvény. Eredményként a megtalált szöveg első karakterének helyét (indexét) adja vissza az *IndexOf()* függvény a *szoveg_azonosito* nevű *stringben*. Ha a megadott szöveg nem található meg akkor az IndexOf() függvény -1 értékkel tér vissza. Ha a keresett szöveg többször is megtalálható az alap szövegben, akkor az elsőnek megtalált szöveg első karakterének helyét (indexét) adja vissza a függvény. Ez utóbbi is látható az alábbi példában, ahol az *IndexOf()* függvény a *„rend”* szöveget kapja meg paraméterként, valamint azt hogy a *„rend”* szöveg kétszer is megtalálható az alapszövegben és a függvény az első előfordulási helyének első karakterpozícióját (17) adja vissza.

példa:

```
string szoveg=”Programfejlesztő rendszer rendben működik”; int pos=szoveg.
IndexOf(”rend”); // 17
```

– szoveg_azonosito.Sub*string*(pozíciótól(helytől)kimásol,másoltkarakterekszáma) .Sub*string*(int , int)

A *Substring()* függvény a szoveg_azonosito nevű *string*ből az első paraméterként megadott pozíciótól kezdődően kimásol karaktereket a második paraméterben megadott számban. A *Substring()* függvény visszatérési értéke az a karaktersorozat *(string)*, amit az alapszövegből kiemelni szándékozunk.

példa:
string szoveg=”Programfejlesztő rendszer”, resz_szoveg; resz_szoveg=szoveg.Sub*string*(17, 4); // rend

– szoveg_azonosito.Replace(”régiszöveg””újszöveg”).Replace(*string*, *string*)

A *Replace()* függvény a szoveg_azonosito nevű *string*ben az első paraméterként megadott szöveget kicseréli a második paraméterben megadott karakterekkel. A *Replace()* függvény visszatérési értéke az a karaktersorozat *(string)*, ami az eredeti alapszöveg módosítása azzal a szöveggel, amit a második paraméterben megadtunk.

példa:
string szoveg=”Programfejlesztő rendszer”, uj_szoveg; uj_szoveg=szoveg.Replace(”rendszer”, ”szoftver”); // Programfejlesztő szoftver

– szoveg_azonosito.Split()

– szoveg_azonosito.Trim()

A fontosabb *string*-ek használatával kapcsolatos tulajdonságok a következők.

– szoveg_azonosito.

Length 10.2. Példaprogramok

A következő példaprogram egy tetszőlegesen begépelt karaktersorozatot (szöveget) fogad a billentyűzetről, ami után egy további karaktert kell megadni. A megadott karaktert megkeresi az előzőleg begépelt szövegben. A program meghatározza, hogy a megadott karakter hányszor szerepel a szövegben és hol, azaz melyik pozíciókban található meg a korábban begépelt karaktersorozatban.

A példa program nem használja fel a beépített különféle *string*-ekkel kapcsolatos metódusokat. Ennek oka az, hogy ezeknek a speciális metódusoknak a segítségével a feladat természetesen egyszerűbben megoldható, kevesebb valós programozói munkával. Viszont ennek a könyvnek, tananyagnak az elsődleges célja, hogy az olvasó minél több algoritmizálási és programozói ismeretet sajátítson el a feladatok megoldása során. Emiatt a *string*-et most algoritmikai szempontból úgy kezeljük, mint egy hagyományos karaktertömb.

A *Main()* függvény első sorában (ez a *25. ábrán* a 12-es sornak felel meg, amit a Visual Studio fejlesztői környezet mutat) két egész típusú *(int)* változót hozunk létre, ahol az i egy ciklusváltozó a db pedig a megadott karakter előfordulásának számát tárolja a begépelt szövegben. Az ezután következő sorban pedig egy külön álló tömböt (poziciok) hozunk létre a későbbiekben keresettkarakter lehetséges pozícióinak tárolására. A *Main()* függvény harmadik sorában két *string* „típusú” változót hozunk létre, az elsőben (szoveg) a begépelendő szöveget tárolhatjuk, a másodikban (str_be) pedig azt a karaktert, amit az előzőekben begépelt szövegben keresünk.

Továbbá szükség van egy char típusú segédváltozóra, amiben elhelyezzük az *str_be string* első karakterét, ez a változó a *Main()* függvény negyedik sorában került definiálásra. Erre azért van szükség mivel, ha esetlegesen a program használója nem egy karaktert gépel be az str_be *string*be, akkor annak első karakterét kiemeljük ebbe a változóba.

Az 5–7 sorokban a programablak előtér és háttérszínének beállítását végezzük el, majd az ezt követő négy programsorban kérjük be az alapszöveget és aztán a keresendő karaktert.

```
using System;
namespace StringKezeles
{  class Program
   {  static void Main(string[] args)
      {   int i, db;
          int[] poziciok = new int[100];
          string szoveg, str_be;
          char ch;
          Console.BackgroundColor = ConsoleColor.White;
          Console.ForegroundColor = ConsoleColor.DarkBlue;
          Console.Clear();
          Console.WriteLine("Írjon be egy tetszőleges szöveget: ");
          Console.Write("szoveg="); szoveg = Console.ReadLine();
          Console.WriteLine("Adjon meg egy betűt");
          Console.Write("betű="); str_be = Console.ReadLine();
          for (i = 0, db = 0, ch=str_be[0]; i < szoveg.Length; i++)
          { if (szoveg[i] == ch)
             {  poziciok[db] =i;
               db++;
             }
          }
          Console.WriteLine("megadott betű ennyiszer szerepel a szövegben: " + db);
          Console.WriteLine("ezeken a helyeken: ");
         for (i = 0; i < db; i++)
          {  Console.WriteLine(poziciok[i]);
          }
         Console.ReadKey();
      }
   }
}
```

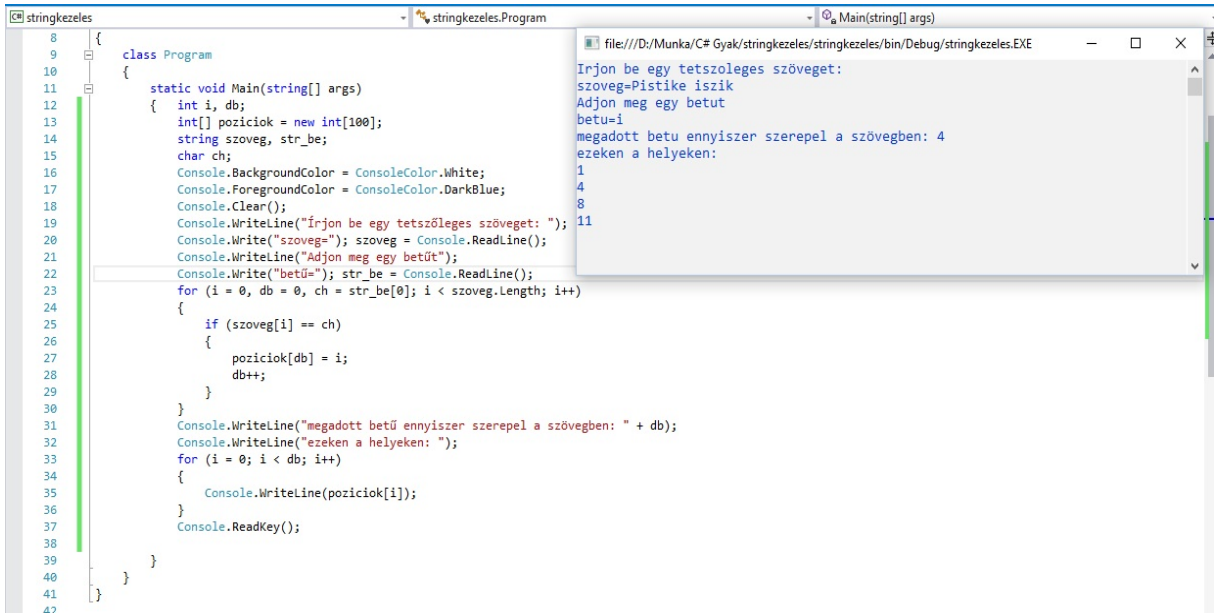
A 12. sorban látható *for()* ciklusban hajtjuk végre az érdemi feladatokat. A *for()* ciklus fejrészének első blokkjában nullázzuk a ciklusváltozót *(i=0)*, a karaktertalálatok számát tartalmazó *(db=0)* változót valamint a karakter típusú változónak *(ch=str_be[0])* átadjuk a keresett karaktert.

A *for()* ciklus ciklusban maradásának feltétele a következő logikai kifejezés i<szoveg.Length, ami lehetővé teszi, hogy pontosan az utolsó begépelt karakterig végig nézzük az alapszöveget. A *.Length* tulajdonság a *string*be gépelt karakterek számát tartalmazza. A for() ciklus fejrészének harmadik blokkjában növeljük egyesével *(i++)* a ciklusváltozó értékét.

A program érdemi és egyben legfontosabb része az a logikai feltétel, ami azonosítja a keresett karaktert *(ch)* az alapszövegben (szoveg). Az *if()* feltételes elágazásban a logikai feltétel szoveg*[i]==ch* teszi lehetővé a begépelt karakter felismerését az alapszövegben. Ha a logikai állítás igaz, akkor találtunk egyet.

A feltételes elágazásban két utasítás áll, az első a poziciok tömbben tárolja, hogy a keresett karakter hol, milyen pozícióban található az alapszövegben. Ehhez felhasználjuk a karaktertalálatok számát tartalmazó db változót, mint indexet a poziciok tömbben *(poziciok[db]=i)*. A második utasítás pedig a db változó értékének növelése.

25. ábra



A *for()* ciklus után kiíratjuk, hogy a keresett karakter hányszor szerepel az alapszövegben. Ez jól látható a 25. ábra jobb felső sarkában, ahol a konzol ablak futási eredményei kerültek kiíratásra.

A program végül egy külön *for()* ciklussal kiírja a poziciok tömb tartalmát, így a program használója láthatja, hogy a keresett karakter hol található meg az alapszövegben.

10.3. Gyakorló feladatok, kérdések

Ismertesse a karaktertömbök fontosabb jellemzőit.

Hogyan hoz létre egy hagyományos *char* típusú tömböt?

Hogyan hoz létre *string* típusú változót?

Sorolja fel a fontosabb *string* kezelő függvényeket.

Mi a feladata a *.Insert()* függvénynek?

Mi a feladata a *.Substring()* függvénynek?

Mi a feladata a *.Replace()* függvénynek?

Melyik tulajdonság tárolja a *string* típusú változóban tárolt karakterek számát?

Hogyan éri el a *string* típusú változó egyik tetszőleges elemét?

11. fejezet

11. Strukturált programozás alapjai

11.1. Strukturált programozásról általánosan

A magas szintű programnyelvek megjelenése előtt a programírás során a forráskód egyetlen folytonos egymást követő utasítások sorozatából épült fel. Ez a fajta programozási, és programfejlesztési stílus nehezen áttekinthető forráskódokat eredményezett, akár már pár száz soros forráskódok esetén is. Ez számos hátrányt vont maga után. Az egyik, hogy a kódot fejlesztő programozó a saját maga által fejlesztett kódot sem tudja egy bizonyos forráskód terjedelemtől kezdve hatékonyan tovább fejleszteni, illetve az esetlegesen jelentkező programozási hibákat feltárni. Ez a probléma fokozottan jelentkezik akkor, ha a kérdéses forráskóddal esetleg huzamosabb ideig nem foglalkozik a fejlesztő. Ez a nehézség még jelentősebb, ha a fejlesztési munkát egy másik programfejlesztőnek kell folytatnia.

A másik hasonlóan jelentős probléma, hogy a programfejlesztési munka megosztása pedig több fejlesztő, programozó között szinte kivitelezhetetlen volt. A programfejlesztési munka megosztása iránti igény viszont a programok bonyolultságának növekedésével párhuzamosan nőtt, ez időben megközelítőleg hatvanas évek kezdeteire datálható. Ennek hatásaként a hatvanas évek végén megjelent magas szintű programnyelvek és fordító programok (C, Basic, Fortran, pár évvel később pedig a Pascal) már biztosították a lehetőséget a programozási feladatok megosztására.

Strukturált programozás: A programtervezési (algoritmizálás) és programozási feladatok, valamint az ebből származó forráskód jól szervezett, hatékony felosztását nevezik strukturált programozási módszernek.

A programozási feladatok szétosztása egyrészt programtervezési (algoritmikai probléma) feladat, erre megfelelő tervezéssel lehetőség van. Viszont a program megírására használt nyelvnek is biztosítania kell azokat az eszközöket, amelyek segítségével a programozó (akár különböző személyek, programozók) a fejlesztendő programot rész programokra tudja bontani a forráskódban és azokat külön-külön tudja megírni. Ezen eszközök összességét nevezik a programozási szakterületen alprogramnak.

Alprogram: Önálló funkcionalitással rendelkező, általában több utasításból álló programrész, amely pontosan definiált bementi paraméterekkel és kimeneti paraméterrel rendelkezik. Fontos jellemzője még az újrahasznosíthatóság, azaz a programunkban nem csak egyetlen helyen tudjuk az alprogramot felhasználni. Ha kellően általánosan és jól tervezzük meg az alprogramot, akkor különféle egymástól teljesen eltérő programok, alkalmazások fejlesztése során is feltudjuk használni azt.

Az alprogramokat szokás a programozás szaknyelvi terminológiában idegenszóval szubrutinnak is nevezni, ez az elnevezés az angolnyelvből származik. A különféle programnyelvek alapvetően két nagyobb csoportra bonthatók az alprogramok létrehozásának és szervezésének szempontjából. Az egyik csoport a Pascal és Basic típusú programnyelvek a másik pedig a C programnyelv.

Pascal, Basic nyelvek és származék programnyelveik:

– eljárások (procedures)

– függvények (functions)

A Pascal és a Basic nyelvekben, amint azt a fenti felsorolás is mutatja, két különféle típusú alprogramot tudunk létrehozni. Az egyik az eljárás, a másik pedig a függvény. A két alprogram típus között lényegében csak egyetlen fontos különbség áll fenn, még pedig az, hogy a meghívott (elindított) eljárások a hívó eljárásnak vagy függvénynek (az, amelyik indítja a meghívott eljárást) nem adnak vissza értéket. A meghívott (elindított) függvények viszont futásuk befejeződése után a hívó számára visszaadnak információt, amelyet futásuk (működésük) közben létrehoztak. Fontos megjegyezni, hogy a megfelelően alkalmazott paraméterlistán keresztül az eljárás is tud a hívó eljárás vagy függvény felé információt tovább adni (lásd paraméterátadási módok). Lényeges különbség még a C típusú nyelvekhez képest, hogy a Pascal és a Basic nyelvekben külön kulcsszó létezik eljárások (procedure, sub) és függvények (function) létrehozására. Továbbá a Basic nyelvben és származék nyelveiben az eljárások hívására (elindítására) szintén külön kulcsszó áll rendelkezésre, amit Call-nak neveznek, ami nem véletlenül, magyarul hívást jelent.

C programnyelv és származék programnyelvei:

– függvények (functions)

A C programnyelv és származék programnyelveiben, ilyen a *C#* nyelv is, kizárólag függvényeket tudunk létrehozni. Természetesen létre lehet hozni olyan függvényeket (void visszatérési típus), amelyek minden szempontból hasonlítanak egy eljárásra, mivel nincsen visszatérési értékük. Fontos továbbá az, hogy itt és ez a *C#* nyelvre is érvényes, hogy nincsen külön kulcsszó a függvények létrehozására, valamint a függvények meghívására sem hoztak létre speciális kulcsszót. A fordítóprogram (compiler) a begépelt forráskód szintaktikai szerkezetéből ismeri fel, hogy egy függvényt definiálunk az adott kódsorban. A függvény hívása pedig nevének és paraméterlistájának pontos megadásával lehetséges.

11.2. Strukturált programozás *C#* nyelven

C# nyelvben alprogramokat tehát kizárólag függvények segítségével tudunk létrehozni. Egy függvény létrehozásának általános szintaktikai szerkezete *C#* nyelvben az alábbi kiemelt szövegrészben látható, az egyes szintaktikai komponensek magyarázata pedig a kiemelt szövegrész után található meg.

```
static visszaadott_paraméter_típusa függvénynév(paraméterlista)
{ lokális_változók_listája;
utasítások;
függvény_hívások(paraméterek);
return változó;
}
```

static: A static kulcsszó az objektum orientált programozás tématerületéhez tartozik és jelenleg elég annyit tudni róla, hogy minden általunk létrehozott függvény neve előtt elkell helyezni ezt.

visszaadott_paraméter_típusa: A visszaadott paraméter típusa adja meg annak a változónak a típusát, amit a függvény működésének befejezése után a hívó függvény számára visszaad. Ez a típusmegadás egyfajta tipizálást (típusrögzítést) is jelent az adott függvény számára, azaz a függvényre tekinthetünk úgy is, mint adott típusú függvényre. A visszaadandó változót a függvényben a return kulcsszó után kell elhelyezni. Nagyon fontos megjegyezni, hogy alapértelmezésben a függvények visszatérési értékének típusa int, azaz egész típusú. Ezért is van szükség a void kulcsszóra abban az esetben, ha nem szeretnénk a függvénynek visszatérési érték típust megadni. A void kulcsszó tulajdonképpen a fordítóprogram számára hordoz információt abból a célból, hogy tudja ez egy olyan függvény, amely a visszatéréskor (amikor a függvény futása befejeződött) nem ad vissza értéket a hívófüggvénynek. A void kulcsszóval definiált függvények tulajdonképpen a korábban ismertetett eljárásoknak felel meg.

lokális_változók_listája: Minden függvénynek lehetnek lokális érvényességi körű változói. A lokális érvényességi kör azt jelenti, hogy a változók csak addig léteznek, amíg a függvény fut, továbbá az itt létrehozott változókat semelyik másik függvényben sem használhatjuk fel. Ez viszont azt is maga után vonja, hogy ezeknek a változóknak az értékeit csak a saját függvénye (a függvény ahol létre lett hozva) tudja megváltoztatni. Amint a függvény futása befejeződik, a lokális változók értékeikkel együtt megszűnnek. A lokális változókat célszerű a függvény elején definiálni, habár erre külön nincsen kötelező előírás. Viszont a függvény működésének könnyebb átláthatósága szempontjából célszerű közvetlenül a függvény definíció kezdetét jelölő nyitózárójel { után megadni a lokális változókat. De ha másképp gondoljuk, akkor ahogy a Main függvényben is, bárhol létrehozhatunk változókat, úgy ezt a magunk által létrehozott függvényekben is megtehetjük.

utasítások, függvény_hívások: Az utasítások és a függvényben meghívott további függvény hívások segítségével adjuk meg a megírandó új függvényünk funkcionalitását. Tulajdonképpen ez a rész határozza meg, hogy a függvény (alprogram) milyen konkrét algoritmikai feladatot lát el.

return változó: A return kulcsszó után kell megadnunk azt a változót, amelynek értékét vissza szeretnénk adni a hívó függvénynek. Az itt megadott változó típusának meg kell egyeznie a függvénynek megadott típussal (visszaadott_paraméter_típusa), amit a függvény definíciós részében adunk meg.

Az alábbi függvény tetszőleges valós szám pozitív egész kitevőjű hatványát képes kiszámítani. A függvény a hívófüggvénytől átveszi a hatvány alapját *(x)* és a kitevőt *(n)*. A függvény képes a nulla kitevőt is helyesen kezelni, amire azáltal képes, hogy a *for* ciklus indexét *(i=1)* egyről indítja és a *for* ciklus első paraméterében a függvény által visszaadott értéket egyre állítjuk *(h=1)*. A *for* ciklus addig fut, amíg a ciklusváltozó értéke kisebb vagy egyenlő *(i<=n)*, mint a függvény paraméterlistájában megadott egész szám, azaz a hatványkitevő *(n)*. A *for* ciklus egyetlen utasítást tartalmaz, amelyben annyiszor szorozzuk meg egyről kiindulva a h változó értékét, amekkora a hatványkitevő *(n)* értéke.

```
static double fv_hatvany(double x, int n)
{ int i; double h;
for(i=1, h=1; i<=n; i++) { h=h*x; }
 return h; }
```

A *for* ciklus befejeződésével a függvény működése is befejeződik, ami a return utasítással zárul. Itt adjuk meg azt a változót, ebben az esetben a h változót, aminek az értékével a függvény a működésének befejeződése után visszatér. Érdemes megfigyelni, hogy a függvény által a return utasításban visszaadott változó típusa *(double)* megegyezik a függvény neve előtt megadott visszatérési érték típussal.

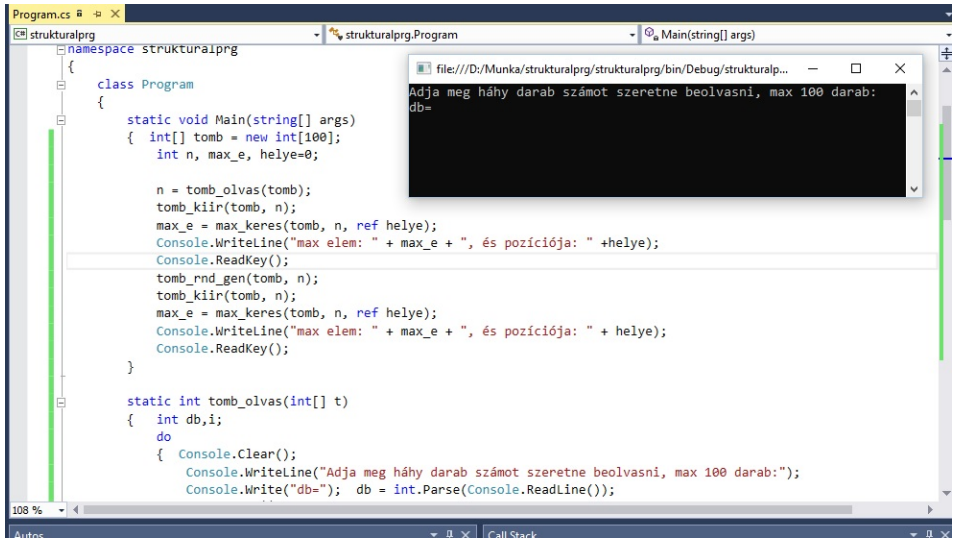
```
int m=3; double y=2.2,
z; z= fv_hatvany(y, m);
z= fv_hatvany(1.7, 5);
```

Az előbb definiált *fv_hatvany()* nevű függvényt a fentebb bemutatott módokon lehet felhasználni. Az első felhasználási esetben különféle változókat definiáltunk és azokon keresztül adtunk a függvény paramétereinek értéket. A második esetben pedig a függvény paramétereinek állandókat adtunk meg és az eredményt egy lebegőpontos típusú változóban (z) vettük át.

11.3. Példaprogramok strukturált programozásra C# nyelven

Az itt bemutatott példa programban több függvényt definiálunk, amelyek aztán a főprogramban, a *Main()* függvényben meghívásra kerülnek. A főprogramban létrehozunk egy 100 elemű egész típusú *(int)* tömböt, továbbá három egész típusú *(int)* változót, amelyekben a beolvasott elemek számát *(n)*, a legnagyobb elem értékét és helyét (*max_e*, helye) tároljuk.

26. ábra



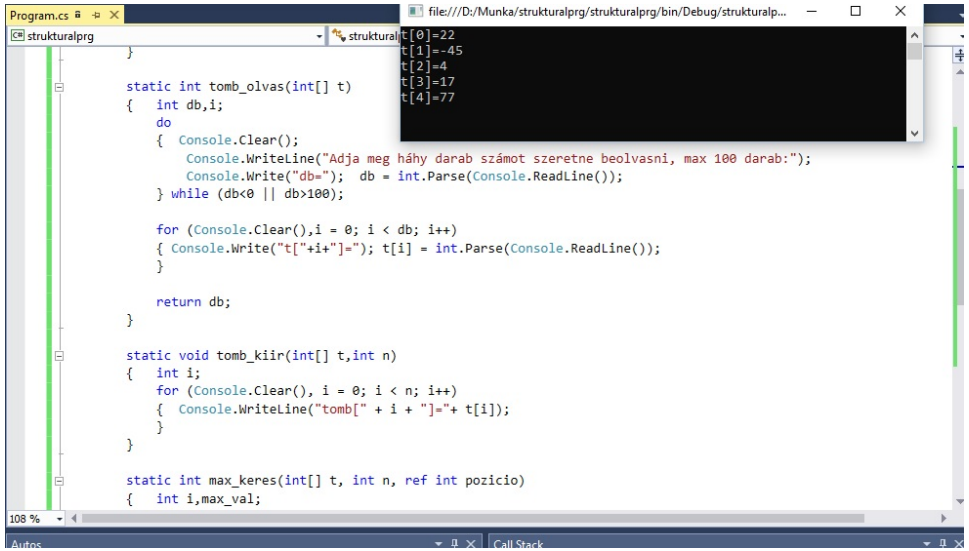
A következő utasítássorban a *tomb_olvas()* függvény kerül meghívásra. A függvény átveszi a százelemű tömb kezdőcímét. *A tomb_olvas()* függvény futásának befejeződése után pedig visszaadja azt az egész számot, ahány elemet a billentyűzetről a tömbbe begépelt a program használója.

A *tomb_olvas()* függvény meghívása után a program végrehajtás a függvényben megadott utasítások végrehajtására ugrik. A *tomb_olvas()* függvény két lokális változót tartalmaz, az egyik *(db)* a felhasználó által begépelni szándékozott egész számok darabszámát tartalmazza, a másik *(i)* pedig egy ciklusváltozó a *for* ciklus számára. A *tomb_olvas()* függvényben először egy do while ciklus hajtódik végre, amelyben a felhasználó megadhatja, hogy hány darab elemet akar az egész elemű tömbbe beolvasni. A ciklus négy utasítást tartalmaz, az első letörli a képernyőt *Console.Clear()* majd egy *Console.WriteLine()* függvénnyel tájékoztatjuk a felhasználót, hogy mit csináljon és maximum hány darab számot gépelhet majd be. Ez látható a 26. ábra jobb felső sarkában a fekete hátterű programablakban. Az utolsó két utasítás pedig a konkrét szám beolvasását végzi el. A változó neve, ami a tömb elemeinek aktuális számát tartalmazza db.

A beolvasás után ellenőrizni kell a felhasználó által megadott számot db. A *do* while ciklus végén kerül megadásra a db változót ellenőrző logikai feltétel while *(db<0 || db>100)*. Mivel a ciklus addig működik, amíg a logikai feltétel igaz ezért a logikai állításban azt kell megfogalmazni, amikor a db változónak megadott érték nem felel meg a megkívánt követelményeknek. Ez az eset akkor áll fenn, ha a felhasználó negatív számot, vagy száznál nagyobb értéket ad meg a db változónak.

Mivel a megadott két számhalmaz *db<0, db>100* diszjunkt azaz nincsen közös részük, ezért a két logikai reláció együtt soha nem teljesülhet. Viszont ahhoz, hogy a *while* után álló összetett logikai állítás igaz legyen egyik vagy a másik eset fenn állása esetén is, a két logikai állítást *vagy* logikai operátorral kell összekapcsolni.

27. ábra



A begépelt számok láthatók a *27. ábra* jobb felső sarkában a fekete hátterű programablakban. A futási példában a db változó értéke öt, aminek hatására ezután öt darab egész számot adott meg a felhasználó, mivel a *do* while ciklus után elhelyezkedő *for* ciklus ciklusváltozója *(i)* addig növekszik az *i++* utasítás hatására, amíg az kisebb, mint a *db* változó értéke *(i < db)*. Itt is megfigyelhető, hogy a beolvasás során kiíratjuk a tömb nevét *(t)* valamint azt is, hogy a felhasználó éppen hányadik elemét adja meg a tömbnek. A *for* ciklusban található Console.Write() és a *Console.ReadLine()* függvényekkel íratjuk ki a szükséges tájékoztatást, illetve olvassuk be a tömbbe egymásután a számokat. Az *int.Parse()* függvénnyel pedig a típus konverziót végezzük el azon a *string* típusú adaton, amit a *Console.ReadLine()* függvény visszaad.

A *for* ciklus befejeződésével a függvényben található utolsó utasításra lép a program végrehajtás menete, ami a *return db.* Ez az utasítás határozza meg, hogy a *tomb_olvas()* függvény milyen értékkel tér vissza a hívó függvényhez, ami ebben az esetben a *Main()* függvény.

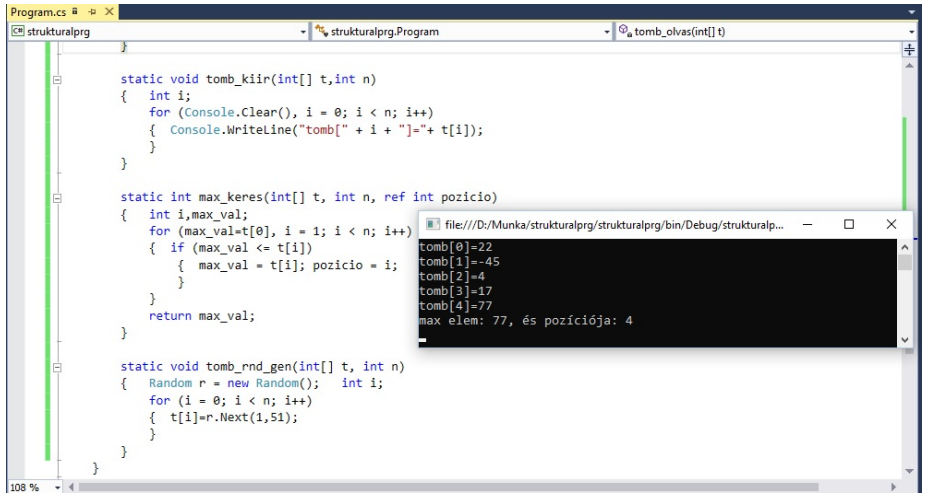
```
using System;
namespace strukturalprg
{ class Program
  { static void Main(string[] args)
    {  int[] tomb = new int[100];
       int n, max_e, helye=0;
       n = tomb_olvas(tomb);
       tomb_kiir(tomb, n);
       max_e = max_keres(tomb, n, ref helye);
       Console.WriteLine("max elem: " + max_e + ", és pozíciója: " +helye);
       Console.ReadKey();
       tomb_rnd_gen(tomb, n);
       tomb_kiir(tomb, n);
       max_e = max_keres(tomb, n, ref helye);
       Console.WriteLine("max elem: " + max_e + ", és pozíciója: " + helye);
       Console.ReadKey();
    }
    static int tomb_olvas(int[] t)
    {   int db,i;
```

```
    do
    {  Console.Clear();
       Console.WriteLine("Adja meg hány darab számot szeretne beolvasni, max 100 darab:'
       Console.Write("db=");  db = int.Parse(Console.ReadLine());
    } while (db<0 || db>100);
    for (Console.Clear(),i = 0; i < db; i++)
    { Console.Write("t["+i+"]="); t[i] = int.Parse(Console.ReadLine());
    }
    return db;
}
static void tomb_kiir(int[] t,int n)
{   int i;
    for (Console.Clear(), i = 0; i < n; i++)
    {  Console.WriteLine("tomb[" + i + "]="+ t[i]);
    }
    }
    static int max_keres(int[] t, int n, ref int pozicio)
    {   int i,max_val;
        for (max_val=t[0], i = 1; i < n; i++)
        {  if (max_val <= t[i])
           {  max_val = t[i]; pozicio = i;
           }
        }
        return max_val;
    }
    static void tomb_rnd_gen(int[] t, int n)
    {   Random r = new Random();   int i;
```

```
            for (i = 0; i < n; i++)
            {  t[i]=r.Next(1,51);
            }
        }
    }
}
```

A *tomb_olvas()* függvény befejeződésével a program végrehajtás menete a *Main()* függvényben folytatódik. Ennek első azonnali lépéseként a tomb_olvas() függvény által visszaadott értéket a *Main()* függvényben definiált n nevű változó átveszi. A következő függvényhívás a *Main()* függvényben a *tomb_kiir()* függvény, ami két paramétert vesz át a *Main()* függvénytől, az első a tömb kezdőcíme, a második az előbbiekben begépelt egész számok darabszáma.

28. ábra



A *tomb_kiir()* függvény egy int típusú lokális változót *(i)* használ ciklus változónak, amit a *for* ciklusban használunk a tömb elemeinek elérésére. A *for* ciklus inicializáló részében letöröljük a képernyőt *Console.Clear()* függvénnyel és beállítjuk a ciklus változó értékét nullára *(i=0)*. A *for* ciklus egyetlen utasítást tartalmaz egy *Console.WriteLine()* függvényt, ami kiírja a tömb tartalmát a képernyőre. Ennek eredménye a *28. ábrán* látható a fekete hátterű programablakban. A *tomb_kiir()* függvénynek nincsen visszatérési érték típusa (void) azaz a *for* ciklus befejeződése után a *tomb_kiir()* függvény befejeződik és a program futás végrehajtása visszatér a *Main()* függvényhez.

A *Main()* függvényben a következő függvényhívás a *max_keres()* függvény, ami három paramétert vesz át a *Main()* függvénytől. Az első a tömb kezdőcíme, a második a korában begépelt egész számok darabszáma, a harmadik pedig az a változó, amiben a legnagyobb elem tömbben elfoglalt helyének indexét tároljuk. Fontos megjegyezni, hogy a harmadik változót címszerinti paraméterátadási móddal adjuk át a *max_keres()* függvénynek, amit a ref kulcsszóval jelöltünk. Erről részletesebben majd a következő fejezetben lehet olvasni, most elég annyit tudni erről, hogy a címszerinti paraméterátadási móddal lehetőségünk van arra, hogy a függvényben értéket kapott vagy módosított változó (ebben az esetben a pozicio nevű) értékét a hívó függvény (*Main()*) felé visszaadjuk.

A függvénynek két lokális int típusú változója van, az első változó *(i)* a ciklus változó, amit a *for* ciklusban használunk a tömb elemeinek elérésére, a másik *(max_val)* pedig a legnagyobb elem átmeneti tárolására szolgál.

A legnagyobb elem keresését egy *for* ciklus felhasználásával és a ciklusba ágyazott feltételes elágazás segítségével oldjuk meg. A *for* ciklus fejrészében a következő dolgokat adjuk meg. A fejrész első blokkjában a legnagyobb elemet tartalmazó változó *(max_val)* értékének a tömb első elemét (nulla indexű) választjuk (max_val=t[0]) a ciklus indulásakor. A ciklus változó *(i)* értékét pedig emiatt 1-ről indítjuk *(i=1)*, mivel az ezt követő összehasonlítások során felesleges önmagával összehasonlítani az első tömb elemet.

Az a megoldás, amelynek során a kiinduló ciklusléptésben a legnagyobb elemnek a tömb első elemét választjuk algoritmikai szempontból rendkívül fontos, mivel a legnagyobb elem keresésekor nem adhatunk meg egy konkrét számot kezdéskor, például a nullát. Ez azért van így, mivel az is elképzelhető, hogy a tömbben csak negatív számok találhatóak és ha induláskor a legnagyobb elemnek a nullát választjuk azt feltételezve, hogy a későbbiekben a tömbben úgy is találunk majd nullánál nagyobbat, akkor ez az algoritmus végrehajtása során teljesen hibás eredményre vezet. A lényeg, hogy kezdéskor a legnagyobb elemnek mindíg az alaphalmazból kell választani egy halmazbéli elemet, jelen esetben például a tömb bármelyik eleme megfelelő lehet, az első, a legutolsó vagy bármelyik másik.

Folytatva a *for* ciklus fejrészének ismertetését a logikai állítás *(i<n)* azt biztosítja, hogy a tömb összes elemét megvizsgáljuk a második elemtől *(i=1)* az utolsóig *(i=n-1)*. A harmadik blokkban pedig a ciklus változót növeltetjük egyesével *(i++)*.

A legfontosabb algoritmikai elem programozása, amelynek segítségével megtaláljuk a legnagyobb elemet és annak helyét, a feltételes elágazás *if(max_val<=t[i])*. Ennek lényege, hogy ha a tömbben találunk a *max*_val nevű változóban tárolt értéknél nagyobb értékű elemet a tömbben *t[i]*, akkor a *max_val* nevű változóban tárolt értéket felül írjuk ezzel az értékkel. Ez az értékadó utasítás *max_val=t[i]* látható a *if()* feltételes elágazásban. A legnagyobb elem helyét pedig a következő utasítás segítségével határozzuk meg pozicio=*i*.

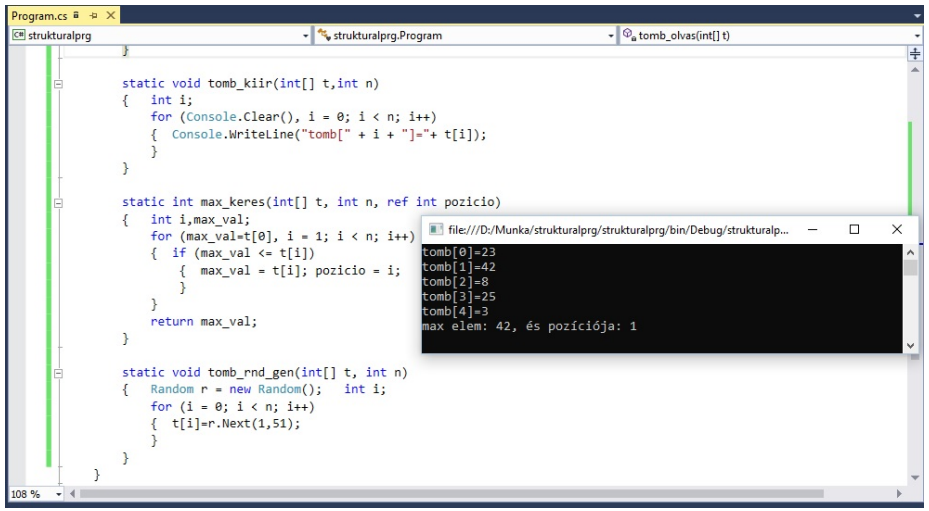
A *max_keres()* függvény a max_val nevű változóban tárolt értékkel tér vissza return max_val. Ez az érték a *Main()* függvényben definiált max_e nevű változóba kerül a pozicio nevű változó pedig a címszerinti paraméter átadás miatt megegyezik a *Main()* függvény helye nevű változójával. Ennek a két változónak a tartalmát a *Main()* függvényben kiíratjuk a konzol képernyőre, majd várakozik a program egy Enter billentyű lenyomására.

A következő lépésben egy *rnd_gen()* nevű függvény hívása következik, amelynek feladata, hogy a *C#* véletlenszámgenerátorával a korábban megadott mennyiségű *(n)* számot létrehozza. Az *rnd_gen()* függvény két paramétert vesz át a *Main()* függvénytől. Az első a tömb kezdőcíme, a második a létrehozandó egész számok mennyisége. Az *rnd_gen()* függvény *n* darab egész számot hoz létre a .Next() függvény felhasználásával. A generált számok értéktartománya 1 és 50 közé esik. A korábbiakban megszokott módon egy *for* ciklus felhasználásával járjuk be a tömböt.

Az *rnd_gen()* függvény nem ad vissza (void) semmilyen értéket sem a *Main()* függvénynek, mindössze a paraméterlistájában megadott címtől (amit *t* nevű változó hordoz) kezdődően n darab int típusú számot hozott létre.

Ezután a korábbiakban már részletezett módon először a *tomb_kiir()* függvényt hívjuk meg és írjuk ki vele a konzol képernyőre a generált számokat, amelyek a *29. ábra* fekete hátterű programablakában láthatók. A következő lépésben a *max_keres()* függvényt is újra felhasználjuk, megkeresve a legnagyobb számot és annak helyét a generált számok között. Végül a *Main()* függvényben kiíratjuk a legnagyobb szám értékét és helyét, ami szintén látható a *29. ábrán*.

29. ábra



11.4. Gyakorló feladatok, kérdések

Miért van szükség alprogramokra?

Milyen alprogramok léteznek a különféle programnyelvekben?

Mi a különbség egy függvény és egy eljárás között?

Sorolja fel egy függvény fontosabb komponenseit.

Adja meg egy függvény általános szintaktikai szerkezetét.

12. fejezet

12. Paraméterátadási módok függvények között

12.1. Paraméterátadási módokról általánosan

A függvények, eljárások közötti információ cserét többnyire a paraméterlistájukon keresztül lehet hatékonyan megoldani, amire alapvetően a *C#* nyelvben két különféle megoldás áll rendelkezésre.

– értékszerinti paraméterátadás

– címszerinti paraméterátadás

Pascal, Basic nyelvek és származék programnyelveikben a fenti két paraméterátadási módszer élesen elkülönülve jelenik meg a programfejlesztő számára, amíg a C nyelvben és a valóságban is csak értékszerinti paraméterátadás létezik. Ez utóbbi megjegyzés arra vonatkozik, hogy a C programnyelvben írt függvények paraméterlistájában mindig egy numerikus értéket (akkor is, ha az egy karakter ASCII kódja) adunk át, az hogy ez az érték egy memória változó tartalma (értéke) vagy a memória változó tárbéli helye (címe) a C programnyelv (és stack memória terület) szemszögéből nézve lényegtelen.

A paraméterátadás működésének megértéséhez szükség van egy speciális memóriaszervezési tárterület működési tulajdonságainak ismeretére. Ez a speciális memória az úgynevezett verem szervezésű memória, angolul röviden a stack memória. Alapvető jellemzője, hogy ennek a tárterületnek nem lehet tetszőleges rekeszét (tárolóhelyét) elérni, azaz nem lehet akárhová a stack memóriába írni vagy onnan olvasni. A stack tartalma csak szekvenciálisan írható, illetve olvasható. A stack memória működési elve lényegében a First In Last Out alapelvre épül, azaz azt az elemet, amit elsőnek ezen a területen elhelyeztünk, azt tudjuk utolsóként újra elérni.

Például ha kiszeretnénk olvasni (pop) a verem (stack) ötödik indexű (5) elemét úgy hogy a verembe korábban tíz darab elemet helyeztünk el (push) akkor először ki kell olvasni sorban a {10, 9, 8, 7, 6} indexű helyeken álló elemeket, hogy hozzáférjünk az ötödik (5) helyen álló elemhez.

A stack memória tárolóhelyeit tehát a programozónak soha nem kell direktben címeznie, azaz nincsen gondunk arra, hogy hova is írjunk a verem szervezésű tárban. Annyi csak a programozó dolga a veremkezelés során, hogy oda írjon (push művelet) vagy onnan olvasson (pop művelet) ki adatokat. A magasabb szintű nyelvekben, mint például a C, C#, Pascal, Basic még ezekkel a műveletekkel (push, pop) sem kell foglalkozni. A különféle nyelveken írt forráskódokban elhelyezett alprogramok (függvények, eljárások) olyan módon kerülnek lefordításra, hogy ezek a műveletek automatikusan elhelyezésre kerülnek a futó programkódban.

Első pillantásra nem különösebben érthető, illetve meggyőző, hogy miért is van szükség valamire, ami így működik. A válasz abban keresendő, hogy olyan adatokat, amelyek elválaszthatatlanul kapcsolódnak valami miatt egymáshoz célszerű így kezelni. A függvény hívások során használt paraméterlisták változóinak értékei, illetve címei pedig pontosan ilyen adatok, amelyekre a függvényhívás során együtt egyszerre van szükség. Ez a szigorúan szekvenciális adatelérés biztosítja azt, hogy semmiképpen nem hagyunk ott adatokat a veremben vagy éppen felejtünk el oda felmásolni adatokat.

A függvények hívása során a függvény paraméterlistát a függvény utasításainak végrehajtása előtt a veremre másolják. Ez azt jelenti a gyakorlatban, hogy egy értéksorozat kerül a verembe, amelyek sorrendben különféle típusú változók értékei, esetleg memória címei lehetnek. A verembe kerülnek a függvény lokális változói is. Ezután a függvény a stack-en található értékeket a megfelelő módon felhasználva végrehajtja a megadott utasításokat. Az utasítások végrehajtása során a paraméterlistában álló változók értékei módosulhatnak.

Nagyon fontos! Amint a függvény hívása befejeződik a veremre másolt paraméterlista értékei elvesznek a futó program (a hívó függvény) számára. Ennek oka, hogy a függvények veremhasználata úgy van felépítve, hogy amint a függvény befejezte a feladatát, takarít maga után, azaz azokat a változókat, amiket a működése során használt visszaolvassa (pop) ezzel szabadítva fel a verem általa használt területét. Ha ezt a takarítást nem tenné meg minden egyes függvény saját maga után, a veremtár rendkívül gyorsan megtelne, ezzel megakadályozva a további függvényhívásokat.

A *C#* programnyelvben a kétféle paraméterátadási mód élesen elkülönül mind a használat, mind pedig a szintaktikai felépítés szemszögéből hasonlóan, mint például a Pascal nyelvben.

12.2. Értékszerinti paraméterátadás

Az értékszerinti paraméterátadás lényege, hogy a függvény a hívó (indító) függvénytől a paraméterlistában megadott típusoknak megfelelő értékeket veszi át. A paraméterlistában felsorolt változók *(var1, var2, ...)* lokális érvényességi körűek abban az értelemben, hogy csak a függvényen belül hivatkozhatunk (használhatjuk) rájuk. Ugyan ez érvényes a függvény lokális változóira is.

Az értékszerinti paraméterátadást bemutató függvény általános szerkezete az alábbiakban látható. A static kulcsszó után azt az elemi *(int, long, double, etc...)* és összetett *(string)* típust kell megadni, amit a függvény a működésének befejeződésével visszaad a hívó függvénynek. A típusazonosító után a függvény neve következik, majd a paraméterlista.

A paraméterlistában minden egyes változónak külön-külön meg kell adni a típusát. **Nagyon fontos!** Ezek a változók kizárólag addig léteznek a veremszervezésű tárban, amíg a függvény működik (fut). Amint a függvény a feladatait befejezte a függvény „takarít” maga után és a változók értékeikkel együtt megszűnnek létezni.

```
static visszaadott_paraméter_típusa fuggvenynev(típus var1, típus var2)
{ lokális_változók_listája;
utasítások;
függvény_hívások(paraméterek);
return valtozo;
}
```

A függvény felépítése egyebekben viszont teljesen megegyezik a korábban ismertetett szintaktikai szerkezettel. A függvény hívása értékszerinti paraméterátadás esetén az alábbiakban látható, ami szintén teljesen megegyezik az előző fejezetben (11.) ismertetett szintaktikai szerkezettel. A függvény hívásakor a paraméterlistában meg kell adni azokat a változókat, vagy explicit értékeket abban a sorrendben ahogy az a függvény definíciós részben szerepel.

valtozo= fuggvenynev(valtozo1, valtozo2);

Nagyon fontos! A függvény működésének semmilyen hatása sincsen a hívásakor a paraméterlistájában felsorolt változók értékeire. Azoktól az értékeket átveszi, felhasználja de azokat a változókat nem befolyásolja, amelyektől a függvény paraméterlista az értékeket kapta. Ennek oka az, hogy a hívás során megadott változó *(valtozo1)* és a függvény definíció során a paraméterlistában megadott változónak *(var1)* nincsen semmi köze egymáshoz azonkívül, hogy azonos típusúaknak kell lenniük és a var1 átveszi a valtozo1 értékét.

Összefoglalva a *var1* és a *valtozo1* két teljesen különböző változó és ha a *var1* változó értéke a függvényben változik az nem fogja a *valtozo1* változó értékét befolyásolni a hívó függvényben.

Az alábbi rövid példa program egy összeadó függvényt mutat, ami átvesz két lebegőpontos szám numerikus értékét, azt összeadja egy a függvényben létrehozott lokális változóban, végül ennek a lokális változónak az értékét visszaadja a hívófél (függvény) számára.

```
static double osszead(double szam1, double szam2)
{ double osszeg;
szam1++; szam2--;
osszeg=szam1+szam2;
return osszeg;
}
```

A bemutatott példa függvényben a lokális paraméterlistában lévő két változó szam1 és szam2 átveszi a meghíváskor az *x* és az y változók értékeit. A szam1 értéke egyel növekszik a szam2 értéke pedig egyel csökken az összegük így természetesen változatlan marad. Az osszead() függvény vissza adja az összeget a hívófüggvénynek és a két változó szam1 és szam2 megváltoztatott értékei pedig elvesznek, mivel azoknak semmi kapcsolatuk sincsen a hívó függvény két változójával x-el és y-nal.

double s, x=1.77, y=2.3; s=osszead(x, y);

12.3. Címszerinti paraméterátadás

A címszerinti paraméterátadás lényege, hogy a függvény a hívó (indító) függvénytől a paraméterlistában megadott változók címeit veszi át. A paraméterlistában felsorolt változók *(var1, var2, ...)* globális érvényességi körűek abban az értelemben, hogy nem csak a függvényen belül hivatkozhatunk (használhatjuk) rájuk. A függvény lokális változói továbbra is csak az adott függvényben használhatók.

static visszaadott_paraméter_típusa fuggvenynev(ref típus var1, ref típus var2) { lokális_változók_listája;
utasítások;
függvény_hívások(paraméterek); return valtozo;
}
valtozo= fuggvenynev(ref valtozo1,ref valtozo2);

Nagyon fontos! A függvény működésének közvetlen hatása van a hívásakor a paraméterlistájában felsorolt változók értékeire. Azoktól nem az értékeket veszi át, hanem a paraméterlistában megadott változók címeit, emiatt a függvény közvetlenül befolyásolja, megváltoztatja a függvény paraméterlistában felsorolt változók értékeit. Ennek oka az, hogy a hívás során megadott változó *(valtozo1)* és a függvény definíció során a paraméterlistában megadott változó *(var1)* ugyanaz, mivel a *var1* a valtozo1 címét kapja meg, ezzel pedig ugyan arra az operatív tárbeli területre hivatkozik a továbbiakban a *var1* nevű változó. Ha tehát a *var1* nevű változó értékét a függvényen belül megváltoztatjuk azzal a *valtozo1* nevű változó értéke is változik.

Az alábbi példa mutatja a címszerinti paraméterátadással megoldott összeadó függvényt, ami minden egyéb szempontból megegyezik az értékszerinti paraméterátadásnál bemutatott összeadó függvénnyel.

static double osszead(ref double szam1, ref double szam2) { double osszeg;
szam1++; szam2--; osszeg=szam1+szam2; return osszeg;
}

A két címszerinti paraméterátadással definiált változó szam1 és szam2 a függvény hívás során az *x* és az y változókkal lesz azonos, azaz a szam1 értéke 1.77 az szam2 értéke 2.3 a hívás pillanatában. Ezt változtatja meg a szam1 értékét növelő, a szam2 értékét csökkentő utasítások. A két szám összege természetesen nem változik, de a függvény futásának befejeződésével az *x* változó értéke 2.77 az y változó értéke pedig 1.3 lesz.

double s, x=1.77,
y=2.3; s=osszead(ref x, ref y);

12.4. Példaprogramok

A *Main()* függvény első sorában két egész típusú egy int valamint egy long és egy lebegőpontos típusú *double* változót definiálunk. A második sorban a képernyő letörlése után a három változó *(a, b, c)* értéket kap. Ezek után az *ertekszerint()* nevű függvényt hívjuk meg, amelynek long típusú a visszatérési értéke.

Az *ertekszerint()* nevű függvény két paraméteres egy *int* és egy *double* típusú változó értékét tudja átvenni és azokat használni. Az egész *(int)* típusú változó értéke *(a=2)* a lebegőpontos *(double)* változó értéke *(c=2.2) ertekszerint()* nevű függvény hívása előtt.

Az *ertekszerint()* nevű függvény lokális paraméter listájában az int típusú változót x-nek a *double* típusú változót y-nak nevezzük, valamint a függvénynek van egy lokális változója is, ami long típusú, a neve z és a definiálása során értéket is kap *(z=5)*. Az értékszerinti paraméterátadás szabálya szerint az *x* értéke 2 az y értéke 2.2 lesz, ez látható a *30. ábra* futási képernyőjén is a fekete hátterű ablakban (x=2, y=2.2, z=5).

A kiíratás után a *x* értékét egyel növeljük *(x++)* az *y* értékéhez 2.2-t hozzáadunk a z értékét pedig egyel csökkentjük. Ezen műveletek hatása a másodikként kiírt érték sorban látszódik (x=3, y=4.4, z=4). A kiíratás után az *ertekszerint()* nevű függvény futása befejeződik és visszakerül a vezérlés a hívó függvényhez, ami a *Main()* volt.

Nagyon fontos! Az *ertekszerint()* nevű függvény lokális paraméter listájában felsorolt változók értékei ertekszerint(a, c) a *Main()* függvényben nem változnak meg. Ez világosan látszódik a 30. ábrán a program futási ablakban is, a „A *Main()* függvényben vagyunk” szöveg alatt *(a=2, c=2.2, b=4)*. Tehát összegezve, az *ertekszerint()* nevű függvény a paraméterként kapott változók (a, c) értékeit felhasználja, de a *Main()* függvénybeli értékeit nem tudja megváltoztatni.

Ezután a *cimszerint()* nevű függvény hívása következik, amely szintén két paraméteres egy int és egy *double* típusú változó értékét tudja átvenni, azokat használni továbbá a paraméterként használt változók értékeit megváltoztatni. Mivel az előző függvény nem változtatta meg a két változó (a, c) értékét az egész *(int)* típusú változó értéke *(a=2)* a lebegőpontos *(double)* változó értéke pedig (c=2.2) a *cimszerint()* nevű függvény hívása előtt. A függvény hívása során a címszerinti paraméterátadással megadott változók előtt a ref kulcsszót kell használni.

A *cimszerint()* nevű függvény lokális paraméter listájában az int típusú változót 3-nek a *double* típusú változót y-nak nevezzük, valamint a függvénynek van egy lokális változója is, ami long típusú, a neve z és a definiálása során értéket is kap *(z=5)*. **Nagyon fontos**! A függvény lokális paraméter listájában a definíció során minden egyes változó típusazonosítója elé a ref kulcsszót el kell helyezni. Ebben az esetben az *x* változó nem egy újabb változó hanem a *Main()* függvényben definiált a nevű változó, amire nem az a nevű szimbólummal hanem *x*-el hivatkozunk a *cimszerint()* nevű függvényben. A címszerinti paraméterátadás esetén is (sőt még inkább) az *x* értéke 2 az y értéke 2.2 lesz, ez látható is a *30. ábra* futási képernyőjén is a fekete hátterű ablakban „A címszerint nevű függvényben vagyunk” szöveg alatt *(x=2, y=2.2, z=5)*.

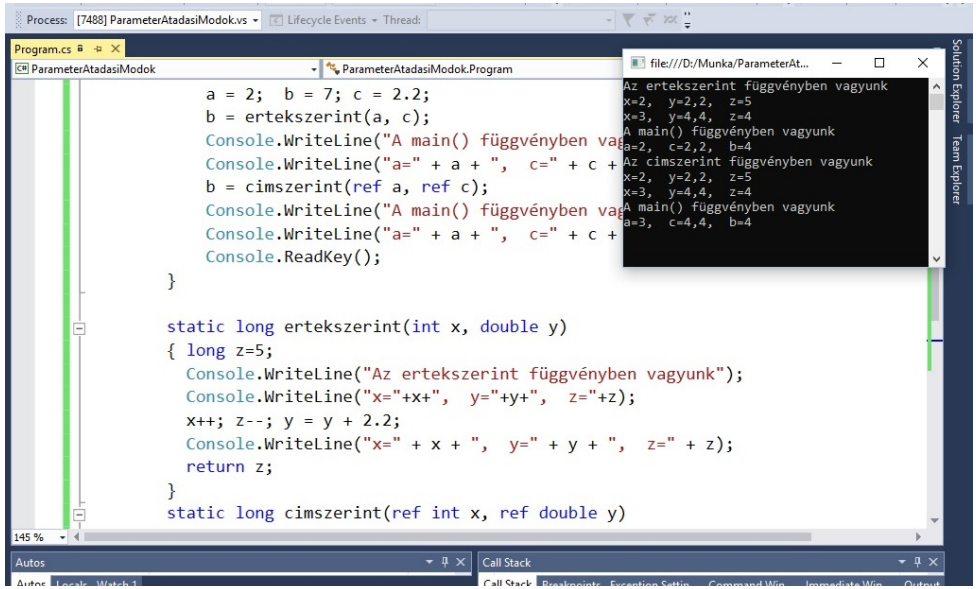
A minél jobb összehasonlíthatóság miatt a *cimszerint()* nevű függvény ugyanazokat a műveleteket hajtja végre a saját változóin, mint az *ertekszerint()* nevű függvény. A kiíratás után a *x* értékét egyel növeljük (x++) az y értékéhez 2.2-t hozzáadunk a z értékét pedig egyel csökkentjük a *cimszerint()* nevű függvényben. Ezen műveletek hatása az ötödikként kiírt érték sorban látszódik *(x=3, y=4.4, z=4)*. A kiíratás után az *ertekszerint()* nevű függvény futása befejeződik és visszakerül a vezérlés a hívó függvényhez, ami a *Main()* volt.

```
using System;
namespace ParameterAtadasiModok
{  class Program
   { static void Main(string[] args)
     {  int a; long b; double c;
        Console.Clear();  a = 2;  b = 7; c = 2.2;
        b = ertekszerint(a, c);
        Console.WriteLine("A main() függvényben vagyunk");
        Console.WriteLine("a=" + a + ",  c=" + c + ",  b=" + b);
        b = cimszerint(ref a, ref c);
        Console.WriteLine("A main() függvényben vagyunk");
        Console.WriteLine("a=" + a + ",  c=" + c + ",  b=" + b);
        Console.ReadKey();
     }
     static long ertekszerint(int x, double y)
     { long z=5;
       Console.WriteLine("Az ertekszerint nevű függvényben vagyunk");
       Console.WriteLine("x="+x+",  y="+y+",  z="+z);
       x++; z--; y = y + 2.2;
       Console.WriteLine("x=" + x + ",  y=" + y + ",  z=" + z);
       return z;
     }
     static long cimszerint(ref int x, ref double y)
     {  long z = 5;
```

```
            Console.WriteLine("Az cimszerint nevű függvényben vagyunk");
            Console.WriteLine("x=" + x + ",  y=" + y + ",  z=" + z);
            x++; z--; y = y + 2.2;
            Console.WriteLine("x=" + x + ",  y=" + y + ",  z=" + z);
            return z;
        }
    }
}
```

Nagyon fontos! Itt jön a lényege a címszerinti paraméterátadásnak. A *cimszerint()* nevű függvény lokális paraméter listájában felsorolt változók értékei cimszerint(ref a,ref c) a *Main()* függvényben a szerint változnak meg, ahogy a *cimszerint()* nevű függvény módosította. Ez világosan látszódik a *30. ábrán* a program futási ablakban is, a „A *Main()* függvényben vagyunk" szöveg alatt *(a=3, c=4.4, b=4)*. Tehát összegezve, a *cimszerint()* nevű függvény a paraméterként kapott változók *(a, c)* értékeit felhasználja, módosítja, majd a módosított értékek a *Main()* függvényben is megjelennek.

30. ábra



12.5. Gyakorló feladatok, kérdések

Milyen paraméter átadási módokat ismer a *C#* nyelvben?

Melyek azok a programnyelvek, amelyekben a paraméterátadási módok elkülönülnek egymástól.

Melyek azok a programnyelvek, amelyekben csak egyféle paraméterátadási mód ismert.

Ismertesse a verem (stack) memóriaszervezési mód működésének lényegét.

Mi a veremtár felhasználásának egyik fontos területe?

Hogyan kezelik az alprogramok veremszervezésű memóriát?

Ismertesse az érték szerinti paraméterátadás fontosabb jellemzőit.

Mutassa be az érték szerinti paraméterátadás szintaktikai szerkezetét függvényeknél.

Mutassa be érték szerinti paraméterátadás esetén egy függvény hívását.

Ismertesse az cím szerinti paraméterátadás fontosabb jellemzőit.

Mutassa be az cím szerinti paraméterátadás szintaktikai szerkezetét függvényeknél.

Mi a ref kulcsszó szerepe?

Mutassa be cím szerinti paraméterátadás esetén egy függvény hívását.

Melyik paraméterátadási mód befolyásolja a hívó függvényben lévő változók (amelyek szerepelnek a függvényhívásban) értékét?

Írjon egy függvényt, ami átvesz egy int típusú változót érték szerint, egy *double* típusú változót szintén érték szerinti paraméterátadással.

Írjon egy függvényt, ami átvesz egy long típusú változót érték szerint és egy float típusú változót pedig cím szerinti paraméterátadással.

Írjon egy függvényt, ami átvesz egy char típusú változót cím szerint és egy float típusú változót szintén cím szerinti paraméterátadással.

Írjon egy olyan függvényt, ami átvesz három int típusú változót, azoknak értéket ad a billentyűzetről és ezeket visszaadja a hívó függvény számára is.

13. fejezet

13. Állománykezelés alapjai

13.1. Állománykezelésről általánosan

A különféle számítógépeken futó programok az operatív tárban tárolt információkat, csak ideiglenesen tudják megőrizni. Legkésőbb a program futásának végeztével az operatív tárban lévő adatokat vagy elmentjük egy olyan speciális tárolóeszközre, amely hosszabb ideig elektromosság felhasználása nélkül például mágneses vagy optikai elven képes azokat tárolni, vagy a memóriában tárolt adatok elvesznek. Ezeket a speciális tárolóeszközöket nevezi a szakirodalom háttértáraknak. A háttértárakon elhelyezett adatok tárolásának szervezése alapvetően tér el az operatív tárakban alkalmazottaktól. Ennek a problémának a feloldására dolgozták ki a különféle információk tárolására szolgáló állományokat. Az állományok kezelésével kapcsolatos feladatokat alapvetően az operációsrendszereknek kell elvégezni.

Az állományokat a különféle háttértárakon logikailag egy speciális struktúrában rendezik el, ami matematikailag egy gráfnak felel meg. Ennek a gráfnak az élei nem irányítottak, a csomópontjait pedig könyvtáraknak (angolul directory) nevezik, szokás az ilyen típusú gráfokat fának is nevezni. Nagyon fontos jellemzője ennek az állomány elhelyezési struktúrának, hogy hierarchikus felépítésű, ahol a legfelső szinten a fa kiinduló pontja áll, aminek szakmai terminológiai elnevezése fő vagy gyökér könyvtár, angolul szokták root-nak is nevezni. Ebből a pontból kiindulva épül fel a logikai struktúrája (fa szerkezet) az egyes háttértárakon az állományok elhelyezésének. Az operációsrendszerek többek között ennek a struktúrának felépítését teszik lehetővé.

Alapvetően kétféle állománytípus létezik:

– szöveges állományok (text file)

– bináris állományok (binary file)

Szöveges állományok: Ez az állománytípus alapvetően ASCII karaktereket tartalmaz, a tartalmát egyszerű szövegszerkesztők (Jegyzettömb-Notepad) segítségével tudjuk megtekinteni. Ez az állománytípus operációsrendszertől függetlenül mindenütt létezik, a Microsoft operációs rendszerei alatt a szöveges állománytípus txt kiterjesztéssel rendelkezik. Fontos megemlíteni, hogy a különféle programnyelvek forráskódjai szintén közönséges ASCII text állományok, a kiterjesztésük (.c, .cpp, .pas, .bas, stb...) pedig utal arra a programnyelvre, amihez tartozó forráskódot tartalmaznak. Szöveges állományok továbbá például a különféle weblapok forrás állományai, .html, .css, .php.

Bináris állományok: A számítógépek háttértárain található állományok többsége ilyen típusú. Ebbe a csoportba tartoznak például a különféle futtatható programállományok, de a különféle adatbázis állományok is bináris file-ok.

13.2. Adatfolyamok C# programnyelvi környezetben

Az adatfolyamok alapötlete még a Unix operációsrendszerben jelent meg, ott általános kimenetnek (stdout), bemenetnek (stdin) és hibakimenetnek (stderr) nevezték.

13.2.1 Adatfolyamokról általánosságban

Adatfolyam (stream): Az adatfolyam byte-ok rendezett szekvenciális sorozata, amelyet egy alkalmazás vagy bementi eszköz küld egy másik alkalmazásnak vagy kimeneti eszköznek. Ezeket a byte-okat egymásután akár olvassuk akár írjuk pontosan abban a sorrendben érkeznek meg, amilyen sorrendben azok küldésre kerültek.

Másszóval az **adatfolyamok** egy adat kommunikációs csatorna absztrakciójának tekinthetők, amely két eszközt vagy alkalmazást köt össze.
A számítástechnikában az adatfolyamokat az elsődleges információ átviteli eszköznek lehet tekinteni. Az adatfolyamok segítségével a különféle alkalmazások elérhetik, használhatják a háttértárakon elhelyezkedő állományokat, továbbá hálózati kommunikációs kapcsolatot létesíthetnek távoli számítógépekkel.

A számítógépek használata során számos művelet fogható fel adatfolyamok írásának és olvasásának. Például a nyomtatás folyamata során a nyomtatást kezdeményező alkalmazás egy byte sorozatot küld egy adatfolyamnak, amely a megfelelő porthoz van hozzárendelve, és a nyomtató pedig ehhez a porthoz kapcsolódik. Egy másik példa lehet egy dokumentum scannelése, amikor egy alkalmazás parancsokat küld a scannernek (kimeneti adatfolyam) amelyeknek hatására a dokumentumot beolvassa (bemeneti adatfolyam) az eszközről. Ezen a módon tehát az alkalmazások bármely perifériával (kamera, egér, billentyűzet, USB dirve, nyomtató, scanner, ...) együttműködhetnek.

A fenti periféria kezelés azért működik kiválóan, mivel a perifériák hardware specifikus kérdéseit, eltéréseit az operációsrendszer és az ott telepített különféle driver (meghajtó) szoftverek végzik. A különféle perifériák kezelésének ez a módja azért előnyös, mivel az alkalmazásnak nem kell semmit sem ismernie az adott hardware perifériáról, elég csak az adatfolyamba a kívánt információkat továbbítani, vagy onnan azokat kiolvasni. A hardware specifikus kérdések, problémák megoldása pedig az operációrendszerre hárul.

Amikor olvasni vagy írni akarunk egy állományba akkor meg kell nyitnunk egy adatfolyamot, amely egy állományhoz kapcsolódik. Az adatfolyam megnyitása után írhatunk vagy olvashatunk az adatfolyamból és ezután az adatfolyamot le kell zárnunk.

Alapvető tudnivalók adatfolyamokkal kapcsolatban.

– Az adatfolyam byte-ok rendezett szekvenciális sorozata. A rendezettség ebben a kontextusban szándékosan hangsúlyozott, mivel fontos tudni, hogy az adatfolyamok erősen rendezettek és szervezettek. Nincs semmilyen lehetőség sem arra, hogy az információáramlás sorrendjét befolyásoljuk, mivel az használhatatlanná tenné az adatfolyamokat. Ha egy byte megelőzi a másikat a küldés során akkor az a byte a beérkezéskor is megfogja előzi azt a bizonyos byte-ot.

– Az adatfolyam szekvenciális adatelérést tesz lehetővé. Ez azt jelenti, hogy az adatfolyamból érkező adatokat csak abban a sorrendben tudjuk manipulálni, amilyen sorrendben azok az adatfolyamból beérkeznek. Az adatfolyamok nem teszik lehetővé tartalmuk tetszőleges elérhetőségét, csak szekvenciális hozzáférést biztosít az adatfolyamhoz, maga neve is erre utal.

– A különféle feladatok megoldásához különféle specifikus adatfolyamok tartoznak, azaz külön adatfolyam létezik állományok kezeléséhez, külön arra, hogy hálózati kommunikációt építsünk fel, de másféle adatfolyamra van szükség, hogy a hardware perifériákat elérjük.

– Az adatfolyamokat mindig meg kell nyitni használat előtt és le kell zárni, ha a használatukra már nincsen tovább szükség. Az adatfolyamok lezárása nagyon fontos lépés, mivel ha a lezárás elmarad akkor kockáztatjuk esetlegesen az adatfolyamba írt adatok egyrészének elvesztését.

– Összességében azt mondhatjuk, hogy az adatfolyam olyanok, mint a közönséges csövek, amelyek két pontot kötnek össze. Az egyik oldalon bejuttatjuk „beöntjük” az adatokat, amelyek a másik oldalon „kifolynak, kiömlenek”. Aki ezt a műveletet végzi nem foglalkozik azzal, hogy miként jut el a kiindulási helyről az adat a végpontig. Emiatt az adatfolyamot egyfajta adat átviteli csatornaként fogható fel.

Alapvető adatfolyamokkal kapcsolatos műveletek.

– Az adatfolyam létrehozása, megnyitása. Ez a lépés kapcsolja az adatfolyamot az adatforráshoz, például egy állományhoz. Amikor létrehozunk egy állományhoz kapcsolt adatfolyamot, az adat folyam megkapja az állomány nevét és elérési útját a könyvtár struktúrában, illetve az állomány megnyitásának módját. A megnyitás módja lehet olvasás, írás, illetve szimultán írás-olvasási mód.
– Adatfolyam olvasása. Az olvasás mindig adatkinyerést jelent az adatfolyamból. Az olvasás mindig szekvenciális hajtódik végre az adatfolyam aktuális pozíciójától.

– Adatfolyam írása. Az írás mindig adatküldést jelent az adatfolyamba. Az írás az adatfolyam aktuális pozíciójától hajtódik végre.

– Pozícionálás az adatfolyamban. Az adatfolyam aktuális pozíciójának módosítását, változtatását, mozgatását (moving) teszi lehetővé. A pozíció vagy másszóval a hely megváltoztatását mindig a valamilyen jól definiált helyhez képest lehet megadni. A pozíció beállítás lehetséges a pillanatnyi aktuális pozíció, vagy az adatfolyam elejéhez vagy végéhez képest.

– Az adatfolyam lezárása. Az adatfolyam lezárása az adott munkafolyamat befejezését jelenti, amelynek hatására a lefoglalt erőforrásokat felszabadítjuk.

Kétféle adatfolyam típus létezik az állományok kezelésével kapcsolatosan ezek a szöveges és a bináris adatfolyamok.

13.2.2 Bináris adatfolyamok

A bináris adatfolyamok, amint azt a neve is sugallja nyers, bináris adatok kezelésével foglalkozik. Ez a jellemzője azért fontos, mivel univerzálissá teszi az adatfolyamot, a kezelt adatok típusától függetlenül bármilyen adattartalmat használni tud. Ez lehetővé teszi, hogy bármilyen típusú állományból (képeket tartalmazó állományok, zenei és multimédiás állományok, szöveges állományok, futtatható program állományok, adatbázis állományok) tudunk olvasni.

A következő osztályok használhatók a bináris állományok kezelésére. FileStream, BinaryReader, BinaryWriter.

13.2.3 Szöveges adatfolyamok

A szöveges adatfolyamok nagyon hasonlóak a bináris adatfolyamokhoz, a lényegi eltérés, hogy kizárólag szöveges adattartalmat tudnak kezelni, ami közönséges karakterek (char) sorozata vagy *string (string)* típusú adatok lehetnek.

13.3. Állománykezelés C# programnyelvi környezetben

Ahhoz, hogy a Visual *C#* programnyelvi környezetben állománykezeléssel kapcsolatos műveleteket valósítsunk meg szükség van az IO elnevezésű névtérre, egészen pontosan a System.IO névtérre. Ezt a using System.IO utasítással tudjuk megvalósítani. Ha használjuk a System.IO névteret, akkor rendelkezésünkre áll két speciális osztály a Stream.Writer és a Stream.Reader. Értelemszerűen a Stream.Writer osztály a szöveges állományok írását, a Stream.Reader osztály pedig a szöveges állományok tartalmának kiolvasását teszi lehetővé.

13.4. Szöveges állományok írása

Szöveges állományok létrehozása és írása a Stream.Writer osztály segítségével lehetséges, ahol az alapértelmezett karakterkódolás az UTF-8 kód. Az állományok létrehozásának és írásának menete a következő alapvető lépésekből tevődik össze.

– Állomány megnyitása írásra (write) vagy adat hozzáfűzésre (append).

– Az állománnyal kapcsolatos írási műveletek végrehajtása.

– Az adatpuffer kiürítése.

– Az adatfolyam és a hozzá kapcsolódó állomány lezárása.

A *Stream.Writer* osztály számos különféle módon példányosítható (különféle konstruktorokkal, OOP, objektum orientált programozás), amelynek hatására különféle módokon tudunk létrehozni egy-egy állományt. Néhány (de nem az összes) lehetőséget az alábbiakban lehet látni. Az alábbi szövegben a path rövidítést fogjuk az állományok nevének és elérési útjának megadására használni. *path = adatállomány_elérési_útja_és_neve*

– *Stream.Writer(string path).* Létrehoz egy új adatfolyamot (objektum), amelyet hozzárendel a megadott elérési úton, a megadott állomány névvel. Ha az állomány nem létezik akkor létrehozza, ha létezik akkor a tartalmát felülírja, azaz minden korábbi adat, ami az állományban korábban tárolva volt elvész.

– *Stream.Writer(string path, bool hozzafuz).* Létrehoz egy új adatfolyamot (objektum), amelyet hozzárendel a megadott elérési úton, a megadott állomány névvel, valamint a hozzafuz paraméter logikai állapotától függően hozzáfűzési vagy újraírási módban nyitja meg. Ha a hozzafuz paraméter logikai állapota igaz, akkor hozzáfűzésre, ha hamis akkor felülírási módban nyitja meg az állományt. Ha az állomány nem létezik akkor azt létrehozza.

– *Stream.Writer(string path, bool hozzafuz, .Encoding text_kod).* Létrehoz egy új adatfolyamot (objektum), amelyet hozzárendel a megadott elérési úton, a megadott állomány névvel. A hozzafuz paraméter hatása ugyan az, mint az előző esetben tárgyalt konstruktor. A harmadik paraméterrel *(.Encoding)* a szöveg kódolásának típusát lehet megadni. Az *(.Encoding)* maga is egy osztály, amelynek tulajdonságaival a következő szöveg kódolási típusokra van lehetőség: *Encoding.ASCII, Encoding.Unicode, Encoding.UTF32, Encoding.UTF7, Encoding.UTF8.*

13.5. Szöveges állományok olvasása

Szöveges állományok olvasása a Stream.Reader osztály segítségével lehetséges ahol az alapértelmezett karakterkódolás az *UTF-8.* Az állományok olvasásának menete a következő alapvető lépésekből tevődik össze.

– Állomány megnyitása olvasásra (read).

– Az állománnyal kapcsolatos olvasási műveletek végrehajtása.

– Az adatfolyam és a hozzá kapcsolódó állomány lezárása.

A *Stream.Reader* osztályt is számos különféle módon lehet példányosítani (különféle konstruktorokkal, OOP, objektum orientált programozás), amelynek hatására különféle módokon tudunk megnyitni egy-egy állományt.

– *Stream.Reader(string path).* Létrehoz egy új adatfolyamot (objektum), amelyet hozzárendel a megadott elérési úton, a megadott állomány névvel. A szöveges állományt olvasási műveletre nyitja meg.

13.6. Példaprogramok

Az első példaprogram egy egyszerű szöveges állomány létrehozását és abba egy számsorozat kiírását végzi el. A program két egész *(int)* típusú változót tartalmaz, az egyik a ciklus változó *(i)*, a másik az állományba kiírt egész számok mennyisége *(n)*. A *Main()* függvény második sorában a StreamWriter osztályt példányosítva egy objektumot hozunk létre, amit az *f* szimbólummal azonosítunk. Az objektum egy állományt fog azonosítani, aminek megadtuk az elérési útját, *"d:\\munka\\adatok.txt" a d:*-vel azonosított háttértáron.

Mivel a *StreamWriter* osztály példányosítása során egyetlen paramétert (az állomány elérési útját) adtuk meg, így ha az adatok.txt állomány nem létezik, akkor az a program futása során létrejön, ha viszont létezett, akkor a tartalma törlődik az újabb használat során felülírással.

```
using System;
using System.IO;
namespace file_iro
{ class Program
    { static void Main()
     { int i, n;
       StreamWriter f = new StreamWriter("d:\\munka\\adatok.txt");
       Console.BackgroundColor = ConsoleColor.White;
       Console.ForegroundColor = ConsoleColor.DarkRed;
       Console.Clear();
       Console.Write("n=");
       n = int.Parse(Console.ReadLine());
       for (i = 0; i <= n; i++)
       { f.WriteLine("i=" + i);
       }
       f.Flush(); f.Close();
     }
    }
}
```

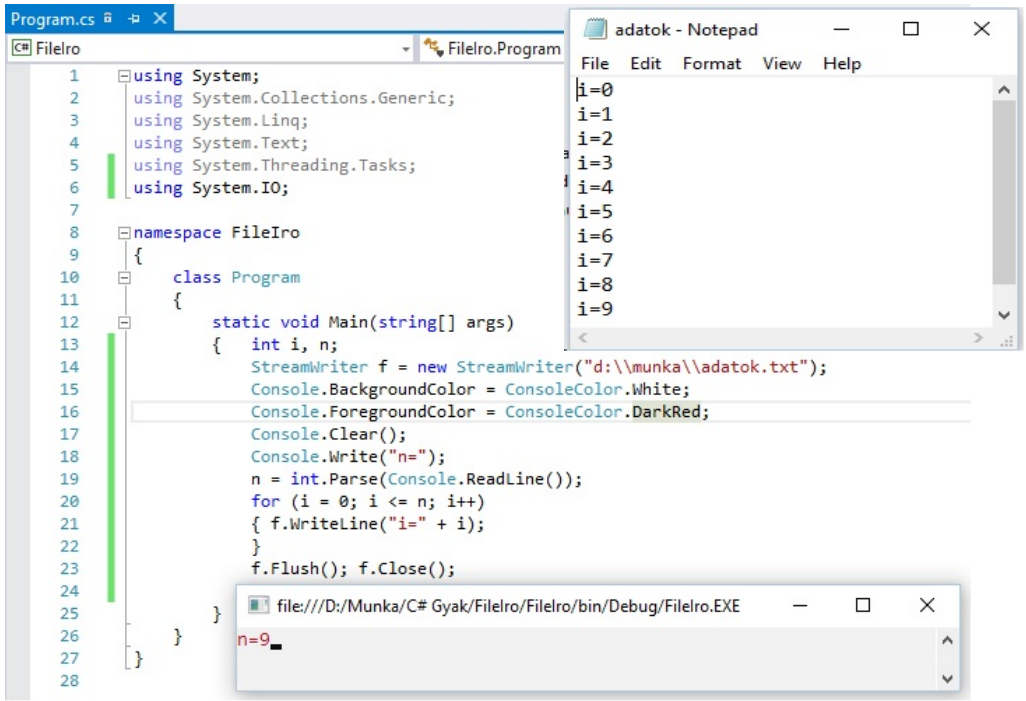
A harmadik, negyedik és az ötödik sorokban a képernyő előtér és háttérszíneit állítjuk be. A következő két sorban pedig az n változó értékét kérjük be a billentyűzetről. Ezután a *for()* ciklusban íratjuk ki az *adatok.txt* nevű szöveges állományba. Jelen esetben a program 1-től *n*-ig kiírja az egész számokat egymás alá a szöveges állományba.

A *31. ábra* mutatja a kódot, továbbá a futó programablakban látszódik az n változó értékének megadása, valamint a szöveges állomány tartalma az állományba kiírt számokkal, a *31. ábra* jobb felső sarkában. A program futása során létrejövő ablak (konzolablak) a *31. ábra* alján látható, éppen amint a felhasználó az *n* változó értékének a 9-et begépelt de az Enter billentyűt még nem nyomta le. A szöveges állományt a Notepad Windows alkalmazással nyitottuk meg, ami látszódik a programablak fejlécén is.

Az állományba a .*WriteLine()* függvénnyel írunk pontosan ugyan olyan formában, mintha azt a konzol ablakba írnánk ki. A .*WriteLine()* függvény hatására minden egyes kiírt szám után egy sorvége karakter kerül kiírásra, emiatt kerülnek a szöveges állományban egymás alá a kiírt számok.

A *for()* ciklus után még két függvényt használunk az állománykezelési folyamat lezárására, az első az adatpuffer ürítése, amit az .*Flush()* függvénnyel hajtunk végre. A következő az utolsó lépés pedig az adatfolyam lezárása, amit a .*Close()* függvénnyel hajtunk végre.

31. ábra



A második példaprogram egy már létező állomány adataihoz ír hozzá további adatokat, amelyeket a billentyűzetről olvasunk be egy *string*-be, majd aztán íratjuk ki a megadott állományba. Az érdekesség kedvéért azt az állományt *(adatok.txt)* használjuk fel, amit az előző programmal létrehoztunk, demonstrálva a korábban felvett adatokhoz való hozzáfűzés működését.

A *Main()* függvény első sorában három *string* típusú változót hozunk létre. Az első *(s)*, amibe a billentyűzetről a felhasználó valamilyen szöveget gépel be, a másik kettő *(q, Q)* tulajdonképpen csak két segéd változó a *do-while* ciklus logikai feltételének kialakításához.

A második sorban létrehozzuk az adatfolyamot, amit összerendelünk a megadott állománnyal. A létrehozás során a StreamWriter osztálynak azt a konstruktorát használtuk, amely lehetőséget biztosít az állománymegnyitási mód beállítására.

A harmadik és negyedik sor az előtér és háttér színeket állítja be. Majd az ötödik sorban kezdődik az a *do-while* ciklus, amiben a billentyűzetről olvasunk és az állományba kiírunk adatokat. Azért választottuk a hátultesztelő ciklusszervező utasítást, mivel mindenféleképpen szükséges a felhasználóval egyszer kommunikálnia a programnak. Az is lehet a program használójának reakciója, hogy nem akar az állományhoz újabb szöveget hozzáfűzni, ezt pedig úgy tudja megtenni, hogy a *q\Q* karakterek valamelyikét adja meg, azaz a *Q*-val jelölt billentyűt nyomja le. Emiatt van szükség még akkor is a ciklusmag legalább egyszeri futására, hogy a minimális felhasználói interakció megvalósuljon a programmal. Fontos megjegyezni, hogy a kilépési feltétel egyetlen q vagy *Q* karakter, azaz például a *„qq”* vagy *„qQ”* karakterpárosra már nem fog kilépni a ciklusból a program végrehajtása.

A *do-while* ciklus első két sorában a képernyő törlés és a szükséges tájékoztató információk kiíratását végzi a program. A következő sorban a *Console.ReadLine()* függvénnyel olvasunk be a billentyűzetről egy karaktersorozatot *(string)* az s nevű változóba. Azután pedig az állományba a *.WriteLine()* függvénnyel írunk pontosan ugyan olyan formában, mintha azt a konzol ablakba írnánk ki. Mindössze annyi a különbség, hogy a *.WriteLine()* függvény elé az állomány azonosítót f helyezzük el a következő módon f.WriteLine().

Fontos megvizsgálni a *do-while* ciklus futási feltételének logikai felépítését. Mivel azt szeretnénk elérni, hogy a ciklus addig fusson amíg a *q\Q* karakterek valamelyike egyedül meg nem jelenik az s változó tartalmaként. Ha tehát egy tetszőleges szöveg van az s-ben akkor az *(s!=q && s!=Q)* közül mind a két logikai állítás igaz, hiszen az s változó értéke nem a q és nem is a *Q*. A két logikai állítást pedig az és (&&) operátorral összekapcsolva csak akkor ad igaz logikai állítást, ha mind a kettő igaz. Ha viszont az s változó csak a q-t vagy csak a *Q*-t tartalmazza akkor a két logikai állítás közül az egyik hamis lesz, mivel a nem egyenlő állítás nem fog fenn állni valamelyiknél. A két logikai állítást összekötő és (&&) operátor hatására pedig a teljes logikai kifejezés is hamis lesz, így a *do-while* ciklus futása ekkor befejeződik. Ha az és (&&) operátor helyett a vagy *(||)* operátort használtuk volna, akkor egy végtelen ciklust hoztunk volna létre a programunkban.

Az előző programhoz hasonlóan a szükséges állományműveletek befejeződése után, az adatfolyam pufferét üríteni *(.Flush())* kell, végül az adatfolyamot le kell zárni *(.Close())*.

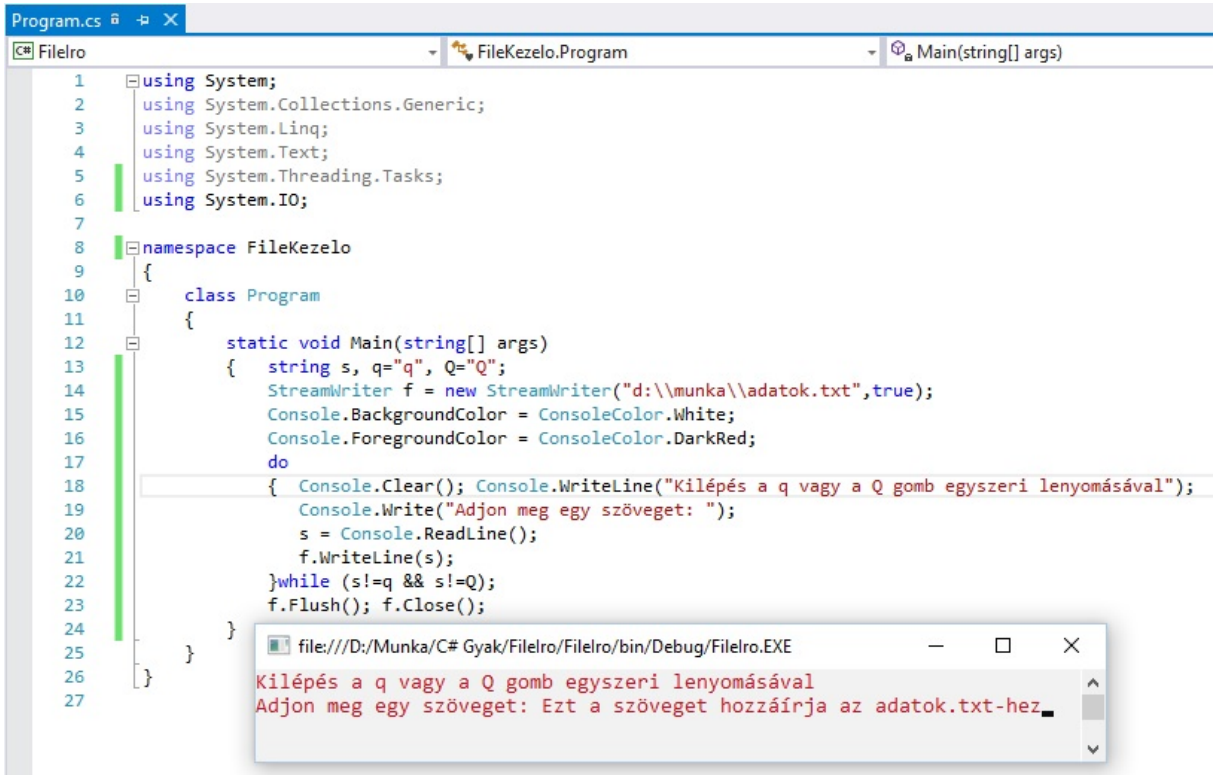
```
using System;
using System.IO;
namespace filekezelo
{ class Program
    { static void Main()
      { string s, q="q", Q="Q";
        StreamWriter f = new StreamWriter("d:\\munka\\adatok.txt",true);
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.DarkRed;
        do
        {  Console.Clear(); Console.WriteLine("Kilpés a q vagy a Q gomb egyszeri lenyomásával");
          Console.Write("Adjon meg egy szöveget: ");
          s = Console.ReadLine();
          f.WriteLine(s);
        }while (s!=q && s!=Q);
        f.Flush(); f.Close();
      }

    }
}
```

A program futásának menete és annak eredménye a *32. ábrán* látható. A *32. ábra* három egymás alatti részből tevődik össze. A felső tartalmazza a háttérben a forráskód egy részét, továbbá a program futásának következtében a konzol ablak tartalma a tájékoztató szöveggel és a felhasználó által begépelt szöveggel „Ezt a szöveget hozzáírja az adatok.txt-hez” együtt látszódik.

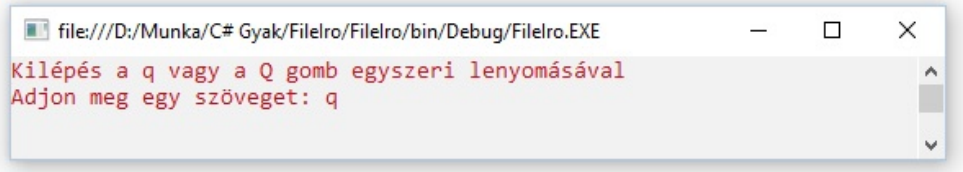
A *32. ábra* közepén a programból való kilépés előtti futási képernyő állapot látható, amikor a felhasználó a q karaktert gépeli be a *do-while* ciklusban. Itt éppen az *s = Console.ReadLine()*; utasításnál várakozik a programvégrehajtás folyamata. Miután a felhasználó az Enter billentyűt lenyomja, a program kilép a *do-while* ciklusból.

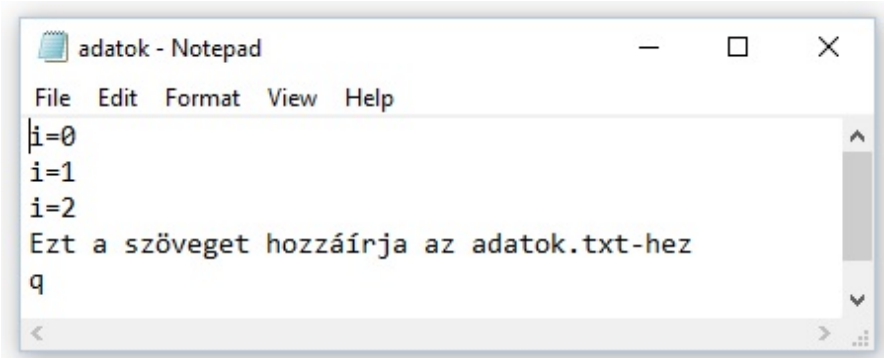
A *32. ábra* alsó részén pedig a szöveges állomány tartalma látható, amit a Notepad Windows alkalmazással nyitottunk meg, ez látszódik is a programablak fejlécén. A 32. ábrán látszódik a szöveges állomány korábbi tartalma (i=0 i=1 i=2) és az újonnan hozzáadott szöveg, valamint a q karakter is, mint utolsónak begépelt szöveg. Ha azt szeretnénk, hogy a q karakter ne jelenjen meg az alkalmazásban akkor a *do-while* ciklusban még egy feltételes elágazásra is szükség van.

32. ábra



```
22              }while (s!=q && s!=Q);
23              f.Flush(); f.Close();
24          }
25      }
26  }
27
```





13.7. Gyakorló feladatok, kérdések

Mik az állományok a számítástechnikában?

Miért van szükség az állományokra?

Milyen alapvető állománytípusokat ismer?

Ismertesse a szöveges állományok jellemzőit.

Ismertesse a bináris állományok jellemzőit.

Mi az adatfolyam?

Mi az adatfolyam célja?

Mi az általános perifériakezelés alapgondolata?

Milyen általános eszközre van szükség az állományok használatához?

Melyek az adatfolyamok kezelésének legfontosabb lépései?

Milyen adatfolyam típusokat ismer állománykezelés kapcsán?

Mi a StreamWriter osztály feladata?

Ismertesse a StreamWriter osztály használatának (konstruktorok) fontosabb eseteit.

Hozzon létre egy szöveges állományt az f:-el jelölt meghajtó munka alkönyvtárában, az állomány neve legyen temp.txt.

Nyisson meg egy szöveges állományt a g:-vel jelölt meghajtó adatok alkönyvtárában új adattartalom hozzáfűzésre, az állomány neve db.txt.

Mi a StreamReader osztály feladata?

Ismertesse a StreamReader osztály használatának (konstruktorok) fontosabb eseteit.

Ajánlott irodalom

Könyvek, jegyzetek:

Magyar nyelvű

1. Reiter István, *C#* programozás lépésről lépésre.

2. Karsa Zoltán, *C#* programozás – jegyzet.

3. Donald Knuth, A számítógép-programozás művészete 1, Alapvető algoritmusok.

4. Donald Knuth, A számítógép-programozás 2, Szeminumerikus algoritmusok.

5. Donald Knuth, A számítógép-programozás művészete 3, Keresés és rendezés.

Angol nyelvű

1. Svetlin Nakov & Co., Fundamentals of Computer Programming with C#, Sofia, 2013.

2. Barry L. Nelson, Stochastic modeling, Analysis & Simulation, McGraw-Hill, Inc., New York, St. Louis, London, 1995.

3. Donald Knuth, The Art of Computer Programming, Volume 1, Fundamental Algorithms.

4. Donald Knuth, The Art of Computer Programming, Volume 2, Semi-numerical algorithms.

5. Donald Knuth, The Art of Computer Programming, Volume 3, Sorting and Searching.

Web linkek

Angol nyelvű

https://en.wikipedia.org/wiki/Algorithm https://en.wikipedia.org/wiki/Flowchart https://en.wikipedia.org/wiki/Pseudocode

3

1. Tárgy célja, algoritmizálási alapok

1.1 Melyek algoritmust leíró eszközök az alábbiak közül?

a. Folyamatábra
b. Struktogram
c. Gráfok

1.2 Mivel jelöljük az utasításokat folyamatábrákban?

a. Paralellogramma
b. Ellipszis
c. Téglalap

1.3 Mivel jelöljük az Input/Output-ot folyamatábrákban?

a. Paralellogramma
b. Ellipszis
c. Téglalap

1.4 Milyen matematikai alakzat a folyamatábra?

a. Geometriai ábra
b. Algebrai egyenlet
c. Irányított gráf

1.5 Milyen algoritmikus eszközzel oldható a másodfokú egyenlet az alábbiak közül?

a. Feltételes elágazás
b. Ciklusszervezés
c. Nem szükséges hozzá sem feltételes elágazás sem ciklusszervezés

2. A programfejlesztő rendszer környezet, kezdő lépések *C#* nyelvben

2.1 Mivel, milyen módon, illetve eszközökkel lehet programokat létrehozni?

a. Hardveres úton például logikai áramkörökkel (FPGA)
b. Valamilyen programfejlesztő rendszer segítségével
c. Nincs szükség semmilyen speciális eszközre

2.2 Milyen állománytípusba sorolhatók a forráskódot tartalmazó állományok?

a. Szöveges
b. Grafikus
c. Bináris

2.3 Melyek programnyelvek az alábbiak közül?

a. C
b. Fortran
c. Function

2.4 Mi a szubrutin?

a. Önálló program
b. Önálló funkcionalitással rendelkező programrész
c. Utasítások egy csoportja

2.5 Mi a névtér?

a. Állományok egy csoportja
b. Osztályok névvel ellátott csoportja
c. Utasítások egy csoportja

2.6 Mi a void szerepe?

a. Csak a főprogram előtt használt formátum vezérlő kulcsszó.
b. A függvény csak egy adatot ad vissza futásának befejeződése után.
c. A függvény semmilyen információt nem ad vissza futásának befejeződése után.

2.7 Mivel zárunk egy utasítás sort?

a. Ponttal
b. Vesszővel
c. Pontosvesszővel

2.8 Melyik a helyes függvényhívás szintaktikailag?

a. függvénynév(paraméterlista).
b. függvénynév(paraméterlista);
c. függvénynév{paraméterlista};

3. Elemi típusok, adatszerkezetek, operátorok

3.1 Melyik a szintaktikailag helyes változó definíció?

a. változó1, változó2 típus azonosító;
b. típus azonosító változó1; változó2; változó3;
c. típus azonosító változó;

3.2 A forráskód melyik részén lehet változót létrehozni?

a. bárhol
b. bárhol a különféle függvényeken belül
c. a függvények elején

3.3 Milyen karakterek közül választhatunk a változó nevek megadása során?

a. nincsen semmilyen megkötés
b. a magyar abc és a számjegyek használhatóak
c. az angol abc és a számjegyek használhatóak

3.4 Hány byte-os az int típusú változó?

a. 2 byte
b. 4 byte
c. 6 byte

3.5 Hány byte-os a short típusú változó?

a. 2 byte
b. 4 byte
c. 3 byte

3.6 Hány byte-os a long típusú változó?

a. 8 byte
b. 4 byte
c. 6 byte

3.7 Hány byte-os a float típusú változó?

a. 2 byte
b. 4 byte
c. 6 byte

3.8 Hány byte-os a *double* típusú változó?

a. 4 byte
b. 8 byte
c. 10 byte

3.9 Melyik egy operandusú operátor?

a. ++
b. *
c. %

3.10 Melyik egy operandusú operátor?

a. /
b. !
c. &&

3.11 Melyik két operandusú operátor?

a. ++
b. !
c. %

3.12 Melyik két operandusú operátor?

a. --
b. !
c. *

4. Konzol kimeneti, bemeneti utasításai és adatkezelése

4.1 Melyik függvény lát el input feladatokat?

a. Console.Write()
b. Console.ReadLine()
c. Console.WriteLine()

4.2 Melyik függvény lát el output feladatokat?

a. Console.Write()
b. Console.ReadLine()
c. Console.WriteLine()

4.3 Mi a ”\n” vezérlőkarakter szerepe?
a. Sor vége és kocsi „kurzor” visszahúzás.
b. Sor vége.
c. Tabulátor karakter.

4.4 Mi a .Parse() függvény?

a. Általános konverziós függvény.
b. Billentyűzet beolvasó függvény.
c. Képernyőre kiíró függvény.

4.5 Mi a feladata a Convert.ToInt32() függvénynek?
a. Tetszőleges típusról konvertál 32 bites egész típusú változóra.
b. *String* típusról konvertál 32 bites egész típusú változóra.
c. 32 bites egész típusú változó tartalmát konvertálja szöveggé.

4.6 Milyen billentyűzet kezelési módokat lehet használni *C#* programnyelvben?

a. direkt billentyűzetkezelés
b. direkt és pufferelt billentyűzetkezelés
c. pufferelt billentyűzetkezelés

5. Feltételes elágazások, logikai állítások és operátorok

5.1 Melyik eldöntendő algoritmikai és programozási helyzet?

a. Egész szám oszthatóságának problémaköre.
b. Egytől százig az egész számok összegének kiszámítása.
c. Egy számhalmaz rendezése valamilyen szempont szerint.

5.2 Melyik eldöntendő algoritmikai és programozási helyzet?
a. Másodfokú egyenlet megoldásának algoritmusa.
b. Egytől százig az egész számok reciprokainak kiszámítása.
c. Egy számhalmaz rendezése valamilyen szempont szerint.

5.3 Melyik eldöntendő algoritmikai és programozási helyzet?
a. Egytől százig az egész számok reciprokösszegének kiszámítása.
b. Háromszög területének kiszámítása Héron összefüggésével.
c. Egy szám előjelének megállapítása.

5.4 Mi a hiba az alábbi utasításokban?
if (a > 10)
{ a++;
}

a. Hibátlan
b. Hiányzik az else ág.
c. A logikai állítás hibás.

5.5 Mi a hiba az alábbi utasításokban?
if (a > 10 and a<20)
{ a++;
}

a. Hibátlan
b. Hiányzik az else ág.
c. A logikai állítás hibás.

5.6 Mi a hiba az alábbi utasításokban?
if ((a % 10) && (a % 3))
[a++;
]

a. Hibátlan
b. Hibás a zárójelpár.
c. A logikai állítás hibás.

6. Ciklusszervező utasítások

6.1 Melyik ciklusszervezési feladat algoritmikai szempontból?

a. Egész szám oszthatóságának problémaköre.
b. Egytől százig az egész számok összegének kiszámítása.
c. Egy szám előjelének megállapítása.

6.2 Melyik ciklusszervezési feladat algoritmikai szempontból?

a. Háromszög területének kiszámítása Héron összefüggésével.
b. Másodfokú egyenlet megoldásának algoritmusa.
c. Egy számhalmaz rendezése valamilyen szempont szerint.

6.3 Melyik ciklusszervezési feladat algoritmikai szempontból?

a. Egytől százig az egész számok reciprokösszegének kiszámítása.
b. Egy szám előjelének megállapítása.
c. Egy tömb (számhalmaz) elemeinek beolvasása.

6.4 Melyik ciklusszervezési feladat algoritmikai szempontból?

a. Két vektor skaláris szorzatának kiszámítása.
b. Egy szám abszolútértékének kiszámítása.
c. Egy tömb (számhalmaz) elemeinek kiíratása a képernyőre.

6.5 Melyik utasítás helyes szintaktikai szempontból?

a. *for* (i=1 ; i<=10 ; i++)
{ Console.WriteLine(“i= “ + i); }
b. *for* (i=1 , i<=10 , i++)
{ Console.WriteLine(“i= “ + i); }
c. *for* (i=1 ; i<=10 i++)
{ Console.WriteLine(“i= “ + i); }

6.6 Melyik paraméter a logikai feltétel a *for* ciklusban?

a. első
b. második
c. harmadik

6.7 Melyik utasítás helyes szintaktikai szempontból?

a. while[i<10] { i++;
}
b. while(i<10) (i++;
)
c. while(i<10) { i++;
}

6.8 Melyik utasítás helyes szintaktikai szempontból?

a. while(i<10 and i>1) { i++;
}
b. while(i<10 ; i>1) { i++;
}
c. while(i<10 && i>1) { i++;
}

6.9 Melyik utasítás helyes szintaktikai szempontból?

a. do
{ i++;
} while(i<10);
b. do
{ i++;
} while(i<10)
c. do
{ i++;
} while{i<10};

6.10 Melyik elöltesztelő ciklusszervező utasítás?

a. for(; ;){ }
b. do {} while(; ;);
c. while(; ;) { }

6.11 Melyik hátultesztelő ciklusszervező utasítás?

a. for(; ;){ }
b. while(; ;) { }
c. do {} while(; ;);

6.12 Melyik ciklusszervező utasításnak hajtódik végre legalább egyszer a ciklusmagja?

a. for(; ;){ }
b. do {} while(; ;);
c. while(; ;) { }

7. Összetett adatszerkezetek, egy dimenziós és több dimenziós tömbök, listák.

7.1 Milyen feladatok esetén célszerű tömböket használni?

a. Tetszőleges egymással kapcsolatban nem lévő adatok tárolására.
b. Nagy mennyiségű egymáshoz szorosan kapcsolódó adatokat szeretnénk tárolni.
c. Nincsen semmilyen szakmailag indokolt javaslat a kérdéssel kapcsolatban.

7.2 A tömb első elemének indexe ...

a. tetszőleges értékű.
b. nulla értékű.
c. egy értékű.

7.3 Melyik helyes szintaktikailag *C#* nyelvben?

a. tomb_neve[index]
b. tomb_neve{index}
c. tomb_neve(index)

7.4 Mekkora helyet foglal az operatív tárban egy 12 elemű int típusú tömb (ANSI C-ben)?

a. 12
b. 24
c. 48

7.5 Mekkora helyet foglal az operatív tárban egy 15 elemű int típusú tömb (Visual Studio C#- ben)?

a. 15
b. 30
c. 60

7.6 Mekkora helyet foglal az operatív tárban egy 10 elemű *double* típusú tömb (Visual Studio C#-ben)?

a. 20
b. 40
c. 80

7.7 Mekkora helyet foglal az operatív tárban egy 10 elemű *double* típusú tömb (ANSI C-ben)?

a. 20
b. 40
c. 80

7.8 Mekkora helyet foglal az operatív tárban egy 10 elemű float típusú tömb (Visual Studio C#-ben)?

a. 20
b. 40
c. 80

7.9 Melyik definíció helyes szintaktikailag?

a. long[] tomb_nev = new long[20];
b. long[] tomb_nev = new int[20];
c. long tomb_nev = new long[20];

7.10 Melyik definíció helyes szintaktikailag?

a. float[] tomb_nev = New float[20];
b. float [] tomb_nev = new float[20];
c. long[] tomb_nev = new float[20];

7.11 Melyik definíció helyes szintaktikailag?

a. long[] tomb_nev = {2, 4, 6, 8, 10};
b. long tomb_nev = {2, 4, 6, 8, 10};
c. long[] tomb_nev = (2, 4, 6, 8, 10);

7.12 Melyik tulajdonság tartalmazza a lista kapacitását?

a. .Length
b. .Capacity
c. .Count

7.13 Melyik tulajdonság tartalmazza a lista méretét?

a. .Size
b. .Capacity
c. .Count

8. Fontosabb *C#* függvények

8.1 A Console.ForegroundColor tulajdonság a konzolablak ...

a. hátterének színét határozza meg.
b. előterének színét határozza meg.
c. előterének és hátterének színét határozza meg.

8.2 Mekkora az ablak mérete a következő függvény hívás hatására Console.SetWindowSize(40, 20)?
a. 40 karakter széles és 20 karakter magas.
b. 39 karakter széles és 19 karakter magas.
c. 20 karakter széles és 40 karakter magas.

8.3 Mi a visszatérési értéke a következő függvényhívásnak? Math.Ceiling(4.1)

a. 4
b. 5
c. 6

8.4 Mi a visszatérési értéke a következő függvényhívásnak? Math.Floor(4.1)

a. 4
b. 5
c. 3

8.5 Mi a visszatérési értéke a következő függvényhívásnak? Math.Sqrt(25)

a. 4
b. 5
c. 15

8.6 Mi a feladata a következő függvénynek? Lista.Add()

a. Törli az első elemet a listából.
b. Új elemet ad a listához, a megadott pozícióban.
c. Új elemet ad a listához, amit a lista végén helyez el.

8.7 Mi a feladata a következő függvénynek? Lista.Insert()

a. Törli az utolsó elemet a listából.
b. Új elemet ad a listához, a megadott pozícióban.
c. Új elemet ad a listához, amit a lista végén helyez el.

8.8 Mi a feladata a következő függvénynek? Lista.Clear()

a. Törli az első elemet a listából.
b. Törli az utolsó elemet a listából.
c. Törli az összes elemet a listából.

9. Véletlenszám generálás

9.1 Melyik a helyes LCG generátor formula?

a. Xn =(aXn +b)mod m
b.Xn =(aXn−1+b)modm
c.Xn =(aXn+1+b)modm

9.2 Melyik a helyes LCG generátor formula?

a.Xn =(aXn−1+b)divm
b.Xn =(aXn−1+b)modm
c.Xn =(aXn+1+b)divm

9.3 Milyen osztási műveletet használunk az LCG generátor formulákban?

a. maradékos osztás
b. egész osztás
c. a hagyományos algebrai osztási alapművelet

9.4 Mekkora lehet maximum a periódus hossza a következő LCG formulának
Xn =(71Xn−1+13)mod100?

a. 71
b. 13
c. 100

9.5 Mi a neve a *C#* nyelvben a véletlenszám generálására használt osztálynak?

a. Rnd
b. Random
c. Rand

9.6 Melyik a szintaktikailag helyes definíció?

a. Random rnd[]= new Random();
b. Random rnd= new Random;
c. Random rnd= new Random();

9.7 Milyen számhalmazba generál számokat a r.Next() függvény?

a. egész számok halmaza
b. racionális számok halmaza
c. valós számok halmaza

9.8 Milyen számhalmazba generál számokat a r.Next*Double*() függvény?

a. egész számok halmaza
b. racionális számok halmaza
c. valós számok halmaza

9.9 Milyen értéktartományba generál számokat a r.Next*Double*() függvény?

a. [0, 1]
b. [0, 1[
c. a teljes valós számhalmaz

9.10 Milyen értéktartományba esnek a generált számok a r.Next(31) függvény hívás hatására?

a. egész számok 0-tól 31-ig.
b. egész számok 0-tól 30-ig.
c. egész számok 1-től 31-ig.

9.11 Milyen értéktartományba esnek a generált számok a r.Next(1,91) függvény hívás hatására?

a. egész számok 0-tól 91-ig.
b. egész számok 0-tól 90-ig.
c. egész számok 1-től 90-ig.

9.12 Milyen értéktartományba esnek a generált számok a r.Next*Double*() * 50 függvény hívás hatására?

a. [0, 50]
b. [0, 50[
c. [1, 50[

10. Karaktertömbök *(string)* kezelése

10.1 Melyik a szintaktikailag helyes utasítás?

a. *string* sz=”szöveg”;
b. *string* sz=’szöveg’;
c. *string* sz={szöveg};

10.2 A *string*-ben található szöveg első karakterének indexe ...

a. nem ismert.
b. egy értékű.
c. nulla értékű.

10.3 Mi a ch változó tartalma? *string* sz=”taxi”; char ch; ch= sz[2];

a. a
b. *x*
c. i

10.4 Mi a feladata a következő függvénynek? s.Insert(,)

a. Az első paraméterben megadott szöveget illeszti be a *string*-be arra a helyre, amit a második paraméterében pozícióban megadtunk.
b. Az első paraméterben megadott pozícióban azt a szöveget illeszti be a *string*-be, amit a második paraméterében megadtunk.
c. Új szöveget ad a *string* végéhez.

10.5 Mi a feladata a következő függvénynek? s.IndexOf(,)

a. *String*-ben egy szöveg keresése.
b. *String* tartalmának rendezése.
c. Szöveg eltávolítása a *String*-ből.

11. Strukturált programozás alapjai

11.1 Milyen alprogramok írására van lehetőség Pascal és Basic típusú programnyelvekben?

a. csak eljárások
b. eljárások és függvények
c. csak függvények

11.2 Milyen alprogramok írására van lehetőség C típusú programnyelvekben?

a. csak eljárások
b. eljárások és függvények
c. csak függvények

11.3 Mi a hiba az alábbi függvényben?
static double fv_teszt() { int i=2; i++; return i; }
a. A függvényben nincsen hiba.
b. A függvény által visszaadott változó típusa nem egyezik meg a definícióban megadott típussal.
c. A függvényben nem definiálhatunk lokális változókat.

11.4 Mi a hiba az alábbi függvényben?
static double fv_hatvany(double x, int n) { int i; double h;
*for(i=1, h=1; i<=n; i++) { h=h*x; }*
return h; }

a. A függvényben nincsen hiba.
b. A függvényben nem definiálhatunk lokális változókat.
c. A függvény paraméterlista hibás.

11.5 Mi a hiba az alábbi függvényben?
static void fv_teszt()
{ int i=2; i++; return i; }

a. A függvényben nincsen hiba.
b. A függvény definícióban void kulcsszó került megadásra ezért nincsen szükség a return kulcsszóra.
c. A függvényben nem definiálhatunk lokális változókat.

12. Paraméterátadási módok függvények között

12.1 Mi a Stack?

a. Veremszervezésű memória modell.
b. Halomszervezésű memória modell.
c. Háttértárak pufferelt gyorsítótára.

12.2 Milyen módon lehet a Stack memória tartalmát elérni?

a. Tetszőlegesen írható és olvasható memóriaterület.
b. FIFO (first in first out) rendszerben írható és olvasható memóriaterület.
c. FILO (first in last out) rendszerben írható és olvasható memóriaterület.

12.3 *C#* nyelvben milyen paraméterátadási módok ismertek?

a. érték és címszerinti paraméterátadás
b. értékszerinti paraméterátadás
c. címszerinti paraméterátadás

12.4 Melyik paraméterátadási mód esetén használjuk a ref kulcsszót?

a. érték és címszerinti paraméterátadásnál
b. csak értékszerinti paraméterátadásnál
c. csak címszerinti paraméterátadásnál

12.5 Értékszerinti paraméterátadás esetén a hívó függvénybeli változó értékét a hívott függvényben megadott a változót a lokális változón keresztül érintő utasítások ...

a. teljes mértékben meghatározzák.
b. nem befolyásolják.
c. a függvényben alkalmazott utasításoktól függően befolyásolják.

12.6 Címszerinti paraméterátadás esetén a hívó függvénybeli változó értékét a hívott függvényben megadott a változót a lokális változón keresztül érintő utasítások ...

a. teljes mértékben meghatározzák.
b. nem befolyásolják.
c. a függvényben alkalmazott utasításoktól függően befolyásolják.

13. Állománykezelés alapjai

13.1 Milyen alapvető állománytípusok léteznek?

a. Bináris
b. Szöveges és bináris.
c. Számos különféle alaptípus létezik.

13.2 Milyen típusú adatelérést tesz lehetővé az adatfolyam?

a. szekvenciális
b. párhuzamos
c. véletlen (random)

13.3 Milyen feladatot hajt végre az alábbi utasítás (konstruktor)?
StreamWriter f = new StreamWriter("d:\\munka\\adatok.txt");

a. Megnyitja az adatok.txt állományt olvasásra.
b. Létrehozza az adatok.txt állományt írásra.
c. Megnyitja az adatok.txt állományt írásra és olvasásra.

13.4 Milyen feladatot hajt végre az alábbi utasítás (konstruktor)?
StreamWriter f = new StreamReader("d:\\munka\\adatok.txt");

a. Megnyitja az adatok.txt állományt olvasásra.
b. Létrehozza az adatok.txt állományt írásra.
c. Megnyitja az adatok.txt állományt írásra és olvasásra.

13.5 Milyen feladatot hajt végre az alábbi utasítás (konstruktor)?
StreamWriter f = new StreamWriter("d:\\munka\\adatok.txt");

a. Megnyitja az adatok.txt állományt írásra és olvasásra.
b. Létrehozza az adatok.txt állományt írásra, ha az állomány már létezik akkor annak tartalmát törli.
c. Létrehozza az adatok.txt állományt írásra, ha az állomány már létezik akkor annak tartalmát megőrzi, az új adatokat az állomány végéhez fűzi hozzá.

13.6 Milyen feladatot hajt végre az alábbi utasítás (konstruktor)?
StreamWriter f = new StreamWriter("d:\\munka\\adatok.txt",true);

a. Megnyitja az adatok.txt állományt írásra és olvasásra.
b. Létrehozza az adatok.txt állományt írásra, ha az állomány már létezik akkor annak tartalmát törli.
c. Létrehozza az adatok.txt állományt írásra, ha az állomány már létezik akkor annak tartalmát megőrzi, az új adatokat az állomány végéhez fűzi hozzá.

13.7 Mi a feladata az alábbi függvénynek?
File_ID.Flush()

a. Megnyitja az állományt (File_ID) írásra.
b. Lezárja az állományt (File_ID).
c. Kiüríti az állományhoz (File_ID) kapcsolt puffert.

13.8 Mi a feladata az alábbi függvénynek?
File_ID.Close()

a. Megnyitja az állományt (File_ID) írásra.
b. Lezárja az állományt (File_ID).
c. Kiüríti az állományhoz (File_ID) kapcsolt puffert.