

ILLÉS ZOLTÁN

Programozás C# nyelven

JEDLIK OKTATÁSI STÚDIÓ
Budapest, 2004



Előszó

e nélkül bármilyen

el. Az esetleges

Az infokommunikációs technológiák fejlődése a hardver eszközök folyamatos fejlődése mellett nagyban függ attól, hogy a fejlesztők milyen szoftvereket írnak a platform lehetőségeinek kihasználására. Fontos, hogy hatékony eszközöket használjanak, melyek emellett a platform széles lehetőségeit biztosítják a fejlesztéshez.

A Microsoft.net technológiai családja a fejlesztői hatékonyság jegyében fogant és a ma elérhető egyik legkomolyabb úgynevezett menedzselt környezet, melyben a programok a .net keretrendszer felügyelete mellett futnak. Ez a felügyelet teszi lehetővé, hogy például a fejlesztőknek ne kelljen törődniük a memóriakezeléssel és számos gyakran használt funkciót (titkosítás, XML adatkezelés, web stb.) már készen kapjanak. Emellett a közös típusrendszer és a közös köztes nyelv nyomán lehetővé teszi, hogy egy alkalmazásban akár több nyelvet (sok más mellett C#, Visual Basic.NET, J#, vagy akár Cobol.NET és Haskell) is használhasson a fejlesztő.

A .net számos nyelve közül a C# (ejtsd: szí-sárp) első az egyenlők között. A C++ nyelvre alapozva azért fejlesztették ki, hogy a .net keretrendszer lehetőségeit kihasználhassa és egy könnyen használható, hatékony fejlesztési utat mutasson azoknak a fejlesztőknek, akik a .net „világban” szeretnének boldogulni.

A C# hasznosságát mi sem bizonyítja jobban, hogy a .net keretrendszer osztálykönyvtárainak egy jelentős részét épp C#-ban írták a Microsoft fejlesztői.

A C# nyelvet a .net keretrendszer bizonyos elemeivel együtt a Microsoft szabványosította az ECMA és az ISO nemzetközi szabványügyi testületeknél. Erre alapozva több független implementáció is született már. Az internetről letölthető a szabványos részek többplatformos, osztott forráskódú változata is, mely a Rotor kódnevet kapta. Ebben található egy C# fordító is, melynek így a forrását is megvizsgálhatják az érdeklődők.

A C# nyelvet bárki könnyen kipróbálhatja ugyanis a legegyszerűbb formájában, parancssori fordítóprogramként ingyenesen letölthető a Microsoft honlapjáról a .net Framework SDK-ban. A Visual Studio.net vizuális fejlesztői környezet az oktatásban elérhető a Microsoft Campus szerződésben, illetve az MSDN Academic Alliance keretében is.

nagy és szövevé-
fejlesztők számára
éhez, melyhez ez a
y dönt, hogy még
Microsoft fejlesztői
r nyelven), és a
leg.

urzusain Microsoft
matikus hallgatók
az oktatói tapaszt-
t nyújtson nekünk.

gy Levente
Programmenedzser
si Platform Üzletág
oft Magyarország

Tartalomjegyzék

BEVEZETŐ.....	11
I. ALAPISMERETEK	13
I.1. A nyelv története.....	13
I.2. A nyelv jellemzői.....	13
I.3. A .NET környezet áttekintése.....	15
I.4. A program szerkezete	16
I.5. Parancssori környezet használata.....	20
I.6. A környezet használata	21
I.7. Windows alkalmazások készítése.....	25
I.8. Feladatok.....	27
II. A C# NYELV ALAPTÍPUSAI.....	28
II.1. Változók definiálása	28
II.2. Állandók definiálása	29
II.3. Változók inicializálása.....	29
II.4. Elemi típusok	30
II.5. Felsorolás típus	35
II.6. Feladatok.....	37
III. KIFEJEZÉSEK, MŰVELETEK	38
III.1. Egyoperandusú operátorok	38
III.2. Kétooperandusú operátorok	39
III.3. Háromoperandusú operátor.....	42
III.4. Kétooperandusú értékadó operátorok	43
III.5. Feladatok.....	46
IV. ÖSSZETETT ADATTÍPUSOK	47
IV.1. Tömbök.....	47
IV.2. Struktúra.....	51
IV.3. Feladatok.....	54
V. UTASÍTÁSOK	55
V.1. Összetett utasítás	55
V.2. Kifejezés utasítás	55
V.3. Elágazás utasítás	56

Tartalomjegyzék

V.4.	Ciklus utasítás	58
V.5.	Ugró utasítások	62
V.6.	Feladatok.....	66
VI.	FÜGGVÉNYEK	67
VI.1.	A függvények paraméterátadása	67
VI.2.	A Main függvény paraméterei	71
VI.3.	Függvények változó számú paraméterrel	73
VI.4.	Függvénynevek átdefiniálása.....	74
VI.5.	Delegáltak, események	76
VI.6.	Feladatok.....	79
VII.	OSZTÁLYOK.....	80
VII.1.	Osztályok definiálása	81
VII.2.	Konstruktor- és destruktor függvények	83
VII.3.	Konstans, csak olvasható mezők	91
VII.4.	Tulajdonság, index függvény.....	93
VII.5.	Osztályok függvényparaméterként	96
VII.6.	Operátorok újradefiniálása.....	97
VII.7.	Interface definiálása	103
VII.8.	Osztályok öröklődése.....	106
VII.9.	Végleges és absztrakt osztályok	109
VII.10.	Virtuális tagfüggvények, függvényelfedés	111
VII.11.	Feladatok.....	116
VIII.	KIVÉTELKEZELÉS.....	117
VIII.1.	Kivételkezelés használata	117
VIII.2.	Saját hibatípus definiálása	120
VIII.3.	Feladatok.....	124
IX.	INPUT-OUTPUT	125
IX.1.	Standard input-output	125
IX.2.	Fájl input – output.....	128
IX.3.	Feladatok.....	132
X.	PÁRHUZAMOS PROGRAMVÉGREHAJTÁS	133
X.1.	Szálak definiálása	133
X.2.	Szálak vezérlése	134
X.3.	Szálak szinkronizálása	137
X.4.	Feladatok.....	142

XI.	ATTRIBÚTUMOK
XI.1.	Attribútumok
XI.2.	Attribútumok
XI.3.	Könyvtárak
XI.4.	Feladatok.....

XII.	SZABVÁZSOK
XII.1.	Könyvtárak
XII.2.	Saját típusok
XII.3.	Feladatok.....

XIII.	KÖNYVTÁRAK
XIII.1.	Helyi könyvtárak
XIII.2.	Távolsági könyvtárak
XIII.3.	Webkönyvtárak
XIII.4.	Feladatok.....

XIV.	AZ ELŐZŐ FEJEZETEK
XIV.1.	Szimulációk
XIV.2.	Területi adatok
XIV.3.	Feltételek
XIV.4.	Hibaelhárítás
XIV.5.	Feladatok.....

XV.	NEM FOLYAMATOS
XV.1.	Nem folyamatos
XV.2.	Mutatók
XV.3.	Feladatok.....

XVI.	GRAFIKA
XVI.1.	Windows
XVI.2.	Webes
XVI.3.	Feladatok.....

IRODALOMJEGYZÉK

.....	58
.....	62
.....	66
.....	67
.....	67
.....	71
.....	73
.....	74
.....	76
.....	79
.....	80
.....	81
.....	83
.....	91
.....	93
.....	96
.....	97
.....	103
.....	106
.....	109
.....	111
.....	116
.....	117
.....	117
.....	120
.....	124
.....	125
.....	125
.....	128
.....	132
.....	133
.....	133
.....	134
.....	137
.....	142

XI. ATTRIBÚTUMOK	143
XI.1. Attribútumok definiálása	144
XI.2. Attribútumok használata	145
XI.3. Könyvtári attribútumok.....	150
XI.4. Feladatok.....	151
XII. SZABVÁNYOS ADATMENTÉS	152
XII.1. Könyvtári típusok szabványos mentése	152
XII.2. Saját típusok szabványos mentése	154
XII.3. Feladatok.....	157
XIII. KÖNYVTÁRAK, TÁVOLI ÉS WEB KÖNYVTÁRAK.....	158
XIII.1. Helyi könyvtárak használata.....	158
XIII.2. Távoli könyvtárhívás	161
XIII.3. Webkönyvtárak használata	167
XIII.4. Feladatok.....	172
XIV. AZ ELŐFELDOLGOZÓ	173
XIV.1. Szimbólumdefiníció használata	173
XIV.2. Területi jelölés	174
XIV.3. Feltételes fordítás	175
XIV.4. Hibaiüzenet	175
XIV.5. Feladatok.....	175
XV. NEM FELÜGYELT KÓD HASZNÁLATA	176
XV.1. Nem felügyelt könyvtár elérése	176
XV.2. Mutatók használata	177
XV.3. Feladatok.....	178
XVI. GRAFIKUS ALKALMAZÁSOK ALAPJAI	179
XVI.1. Windows alkalmazások alapjai.....	179
XVI.2. Webes alkalmazások alapjai	184
XVI.3. Feladatok.....	188
IRODALOMJEGYZÉK	189

Bevezető

Valamilyen programot megírni nem nehéz, de jó programot írni bizony nem egyszerű feladat.

Nehéz lenne megmondani, hogy melyik az aranyút, amely a jó program-készítés iskolájának tekinthető, de az bizonyosan állítható, hogy jó és hatékony programot csakis megfelelő fejlesztési környezetben tudunk készíteni.

Mai rohanó világunkban a gyors gazdasági, társadalmi változások mellett is szembeötlő, mennyire gyors, milyen dinamikus a változás az informatikában.

Nem telik el úgy hónap, hogy az informatika valamelyik területén be ne jelenjenek valamilyen újdonságot, legyen az hardver, vagy szoftver.

A szoftver fejlesztési környezeteket tekintve az utóbbi idők egyik legnagyobb és legtöbb újdonságot hozó egységcsomagját kapták meg a fejlesztők az új Microsoft .NET rendszerrel. Ebben a környezetben, bármilyen területre gondolunk, új technológiák, új lehetőségek segítik a fejlesztők munkáját (Common Language Runtime (clr), ASP.NET, stb.).

Ismerjük már: „Új műsorhoz új férfi kell...”.

Ezt az elvet alkalmazva a már klasszikusnak tekinthető Basic és C++ nyelvek mellett megjelent az új C# (ejtsd: angolosan „szí sárp”, magyarosan C kettőskereszt, Cisz...) programozási nyelv a .NET rendszer részeként.

A nyelv mottójának talán a következő képletet tekinthetjük:

A Basic egyszerűsége + a C++ hatékonysága = C#!

Ebben a könyvben a fejlesztési eszköztárnak ezt az alapját, a C# programozási nyelvet, annak lehetőségeit ismerhetik meg.

Terveim szerint egy következő könyvben a nyelv legfontosabb környezetbeli alkalmazási lehetőségeit részletesen tárgyalni fogjuk.

Remélem, hogy ez a könyv széles olvasótábornak fog hasznos információkat nyújtani. A programozással most ismerkedőknek egy új világ új eszközét mutatja meg, míg a programozásban jártas Olvasók talán ennek a könyvnek a segítségével megérik azt, hogy ez az a nyelv, amit kerestek.

Befejezésül remélem, hogy a magyarázatokhoz mellékelte példaprogramok jól szolgálják a tanulási folyamatot, és az anyag szerkesztése folytán nem kerültek bele „nemkívánatos elemek”.

Végül, de nem utolsósorban meg kell köszönnöm feleségemnek e könyv olvashatóságát segítő munkáját.

Illés Zoltán

I. Alapismeretek

I.1. A nyelv története

A C# programozási nyelv a Microsoft új fejlesztési környezetével, a 2002-ben megjelent Visual Studio.NET programcsomaggal, annak részeként jelent meg.

Bár a nyelv hosszú múlttal nem rendelkezik, mindenképpen elődjének tekinthetjük a C++ nyelvet, a nyelv szintaktikáját, szerkezeti felépítését.

A C, C++ nyelvekben készült alkalmazások elkészítéséhez gyakran hosszabb fejlesztési időre volt szükség, mint más nyelvekben, például a MS Visual Basic esetén. A C, C++ nyelv komplexitása, a fejlesztések hosszabb időciklusa azt eredményezte, hogy a C, C++ programozók olyan nyelvet keressenek, amelyik jobb produktivitást eredményez, ugyanakkor megtartja a C, C++ hatékonyságát.

Erre a problémára az ideális megoldás a C# programozási nyelv. A C# egy modern objektumorientált nyelv, kényelmes és gyors lehetőséget biztosítva ahhoz, hogy .NET keretrendszer alá alkalmazásokat készítsünk, legyen az akár számolás, akár kommunikációs alkalmazás. A C# és a .NET keretrendszer alapja a *Common Language Infrastructure (CLI)*.

I.2. A nyelv jellemzői

A C# az új .NET keretrendszer bázisnyelve. Tipikusan ehhez a keretrendszerhez tervezték, nem véletlen, hogy a szabványosítási azonosítójuk is csak egy számmal tér el egymástól. A nyelv teljesen komponens orientált. A fejlesztők számára a C++ hatékonyságát, és a Visual Basic fejlesztés gyorsaságát, egyszerűségét ötvözték ebben az eszközben.

A C# nyelv legfontosabb elemei a következők:

- Professzionális, Neumann-elvű. Nagy programok, akár rendszerprogramok írására alkalmas.
- A program több fordítási egységből – modulból – vagy fájlból áll. Minden egyes modulnak vagy fájlnek azonos a szerkezete.
- Egy sorba több utasítás is írható. Az utasítások lezáró jele a pontosvessző (;). Minden változót deklarálni kell. Változók, függvények elnevezésében az ékezetes karakterek használhatóak, a kis- és nagybetűk különböznek.

I. Alapismeretek

- A keretrendszer fordítási parancsa parancssorból is egyszerűen használható. (pl. `csc /out:alma.exe alma.cs`).
- Minden utasítás helyére összetett utasítás (blokk) írható. Az összetett utasítást a kapcsos zárójelek közé `{}` írt utasítások definiálják.
- Érték (alaptípusok, *enum*, *struct*, *value*) illetve referencia (*class*) típusú változók.
- Nincs mutatóhasználat; biztonságos a vektorhasználat.
- Boxing, unboxing. Minden típus öse az *object*, így például egy egész típust (*int*) csomagolhatunk objektumba (boxing) illetve vissza (unboxing).
- Függvények definíciói nem ágyazhatók egymásba, önmagát meghívhatja (rekurzió). Tehát függvénydefiníció esetén nincs blokkstruktúra. Blokkon belül statikus vagy dinamikus élettartamú változók deklarálhatók. Függvénytípolimorfizmus megengedett.
- Érték, referencia (*ref*) és output (*out*) függvényparaméterek.
- Kezdő paraméter-értékkadás, változó paraméterszámú függvény deklarálása.
- Delegáltak, események használata.
- Hierarchikus névterekben használt osztályok. Mivel minden osztály, ezért a „program”, a *Main* függvény *public static* hozzáférésű. Több osztály is tartalmazhat *Main* függvényt, de ilyen esetben a fordításkor meg kell mondani, hogy melyik osztálybeli *Main* függvény legyen az aktuális induló (*/main:osztálynév*).
- Új operátorok: *is* operátor egy objektum típusát ellenőrzi (*x is Lambda*), *as* operátor a bal oldali operandust jobb oldali típussá konvertálja (*Lambda l = x as Lambda;*). A hagyományos konverziós operátortól abban különbözik, hogy nem generál kivételt!
- Privát konstruktor (nem akarunk egy példányt se), statikus konstruktor (statikus mezők inicializálása, mindig példány konstruktor előtt hívódik meg, futási időben az osztálybetöltő hívja meg) használata.
- Nincs destruktork, helyette a keretrendszer szemétygyűjtési algoritmusa van. Szükség esetén az osztály *Dispose* metódusa újradefiniálható.
- Egyszeres öröklés, interface-ek használata.
- Operátorok definíálásának lehetősége, *property*, *indexer* definíálás.
- Kivételkezelés megvalósítása.
- Párhuzamos végrehajtású szálak definíálhatósága.

I.3. A .NET k

A Visual Studio Microsoft legfriss hálózati munka kapta. Jelenleg a szempontjából niki a különbségek!

Az új eszköz nyosnak tekinthet alkalmazásokat csak olyan operát keretrendszert (.NET rendszerek egyike felfelé utólag feltes kódra fordítja Language) néven

A Visual Studio

A .NET keret

- Webszabvány
- Univerzális támogatás tetszőpatibilitás
- Minden nyelvi System)
- Minden .NET

A keretrendszer végez, míg az így (Common Language alkalmazások szállítását (Garbage Co

A keretrendszer portba osztható:

- ADO.NET,
- ASP.NET,
- XML Web
- Windows alkalmazó

egyszerűen használ-

írható. Az összetett
efiniálják.

encia (*class*) típusú

t.

például egy egész tí-
ing) illetve vissza

önmagát meghívhatja
blokkstruktúra. Blok-
ltozók deklarálhatók.

éterek.

ú függvény deklarál-

minden osztály, ezért
érésű. Több osztály is
fordításkor meg kell
legyen az aktuális in-

nörzi (*x is Lambda*), *as*
á konvertálja (*Lambda*
rátortól abban külön-

, statikus konstruktor
struktor előtt hívódik
ználata.

jtési algoritmusa van.
finiálható.

xer definiálás.

I.3. A .NET környezet áttekintése

A Visual Studio 6.0 fejlesztőrendszer átdolgozásaként 2002-ben jelent meg a Microsoft legfrissebb fejlesztőeszköze. Utalva a lényeges változtatásokra, a hálózati munka integrálására, a fejlesztőeszköz a Visual Studio.NET nevet kapta. Jelenleg az eszköz 2003-as frissítése a legutolsó verzió. A könyv szempontjából nincs lényeges különbség a két verzió között, így nem is térünk ki a különbségek bemutatására.

Az új eszköz a korábbi fejlesztőrendszerekkel ellentétben nem a hagyományosnak tekinthető ún. Win32 alapú, hanem a .NET környezet alatt futtatható alkalmazásokat készít. Ez azt jelenti, hogy az új eszközzel készült alkalmazások csak olyan operációs rendszer alatt futtathatók, melyek támogatják a .NET keretrendszert (.NET Framework). Alapértelmezésként a jelenlegi operációs rendszerek egyike sem támogatja ezt, viszont Windows 98 operációs rendszertől felfelé utólag feltelepíthető. A fordító a forráskódot nem natív, hanem egy köztes kódra fordítja le. Ezt a köztes kódot MSIL (Microsoft Intermediate Language) néven szokták említeni.

A Visual Studio.NET telepítése tehát a keretrendszer telepítésével kezdődik.

A .NET keretrendszer legfontosabb erényei:

- Webszabványokon alapuló (XML, HTML, SOAP).
- Univerzális alkalmazási modellt használ, azaz egy .NET osztály, szolgáltatás tetszőleges .NET kompatibilis nyelvből használható. A .NET kompatibilitást a Common Language Specification (CLS) definiálja.
- Minden nyelvből ugyanazok a típusok használhatók (Common Type System)
- Minden .NET osztály a fejlesztők rendelkezésére áll.

A keretrendszer a fejlesztőeszköz számára fordítási idejű szolgáltatásokat végez, míg az így lefordított alkalmazásoknak futási idejű környezetet biztosít (Common Language Runtime). A keretrendszer biztosítja a fentiek mellett az alkalmazások számára a már nem használt objektumok memóriabeli felszabadítását (Garbage Collection).

A keretrendszer osztálykönyvtára az alkalmazások típusa alapján több csoportba osztható:

- ADO.NET, adatbázis alkalmazások új generációs könyvtára.
- ASP.NET, webes alkalmazások (Web Forms) készítésének könyvtára.
- XML Web Service, interneten keresztül elérhető könyvtár.
- Windows alapú felhasználói (Windows Forms, Console Application), alkalmazói könyvtár.

I. Alapismeretek

Az osztálykönyvtárak használatához egy fejlesztő nyelvre van szükség. A Visual Studio.NET alapértelmezésben három nyelvet biztosít a fejlesztők számára. A Visual Basic és a Visual C++ nyelveket, mint az előd keretrendszerből megmaradt bázisnyelveket, és egy új nyelvet, amit a .NET keretrendszerhez fejlesztettek ki, a Visual C#-ot. Ma már a Java utódnyelv, a J# is a Visual Studio.NET 2003 fejlesztőrendszer része. Az új C# nyelvet szabványosították, ami a széleskörű elterjedésnek fontos feltétele. A C# nyelv szabványosított dokumentációjához ECMA-334 kód alatt férhetünk hozzá.

A .NET keretrendszer lényege, hogy a közös nyelvi definíciók már szabványosítottak. Ebben a szabványosításban a Microsoft mellett az informatikában érdekelt legnagyobb szervezetek is (HP, Intel, IBM, Sun, Fujitsu, Netscape) részt vettek. Ez ECMA-335-ös azonosítóval, mint a közös nyelvi infrastruktúra vagy eredeti nevén Common Language Infrastructure (CLI) nemzetközi szabványává vált.

I.4. A program szerkezete

Egy C# program tetszőleges számú fordítási egységből (modulból) állhat. Szokás ezen modulok vagy fájlok összességét projektnek nevezni.

A program ezen modulokban definiált osztályok összessége.

Egy fájl tartalmazza az egyes modulok közti hivatkozásokat (*using*), osztálytípus-, változó- és függvénydeklarációkat.

Egy modul egy tetszőleges névtér része lehet. A névtér (*namespace*) az a logikai egység, amiben az azonosítónknak egyedinek kell lennie. Nem kötelező a névtér definiálása, ebben az esetben az ún. név nélküli névtér része lesz az adott kód.

Névtér szerkezete:

```
namespace új_névtérnév
{
    class új_osztálynév
    {
        Típusdefiníció;
        Függvénydefiníció;
    }
    ...
}
```

Függvények

```
{
}
}
```

Az osztályok futtatáskor az op hogy több típusb ben fordításkor l főprogram.

A programb. majd a második nyelvben a kis- é azonos:

a
a

Az azonosító igényeknek megf

A .NET kere magyar ékezetes betűket is megen

k

Azonosítók l szavak vagy véd

abst
byte
clas
dele
ever
fixe
if
inter
new
over

relvre van szükség. A
tosít a fejlesztők szá-
előd keretrendszerből
NET keretrendszerhez
elv, a J# is a Visual
vet szabványosították,
nyelv szabványosított

Függvények definíciójának formája:

```

visszaadott_típus név(argumentumlista, ha van)
{
    változó definíciók,
    deklarációk
    és utasítások
}

```

definíciók már szabvá-
lett az informatikában
in, Fujitsu, Netscape)
s nyelvi infrastruktúra
e (CLI) nemzetközi

Az osztályok egyikében kell egy *Main* nevű függvénynek szerepelnie, amely futtatáskor az operációs rendszertől a vezérlést megkapja. A nyelv megengedi, hogy több típusban (osztályban) is definiáljunk ilyen függvényt. Ebben az esetben fordításkor kell megmondani, hogy melyik típus *Main* függvénye legyen a főprogram.

A programban használt neveknek betűvel vagy _ jellel kell kezdődniük, majd a második karaktertől tetszőleges betű és szám kombinációja állhat. A nyelvben a kis- és nagybetűk különbözőek, így például a következő két név sem azonos:

```

alma
aLma

```

ből (modulból) állhat.
revezni.
ssége.
ozásokat (*using*), osz-

Az azonosítónevek hossza általában 32 karakter lehet, de sok rendszerben az igényeknek megfelelően lehet ezen változtatni.

rtér (*namespace*) az a
lennie. Nem kötelező
li névtér része lesz az

A .NET keretrendszer 16 bites unikód karakterábrázolást használ, amiben a magyar ékezetes karakterek is benne vannak. A nyelvi fordító ezeket az ékezetes betűket is megengedi, így az alábbi név is megengedett:

```

körte

```

Azonosítók használatára nem megengedettek a C#-ban a következő kulcsszavak vagy védett azonosítók:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public

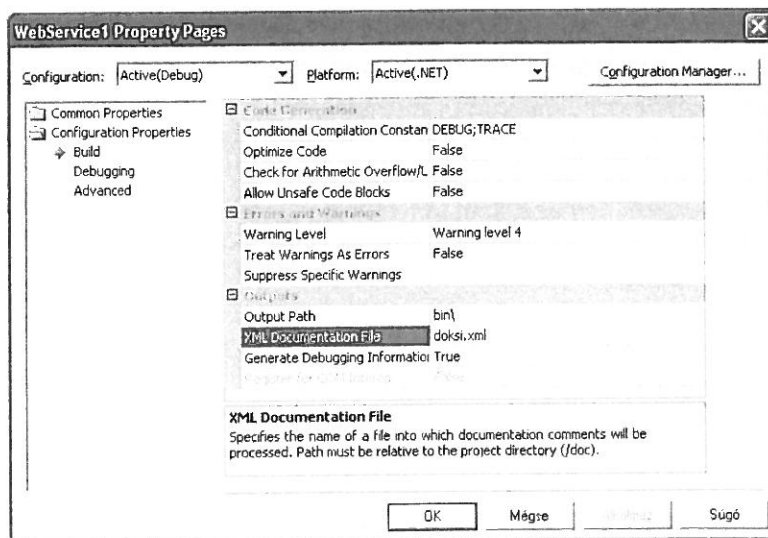
I. Alapismeretek

readonly	ref	return	sbyte	sealed
short	stackalloc	static	string	struct
switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe
ushort	using	virtual	void	while

Egy forrásállomány szerkesztése során magyarázatokat is elhelyezhetünk a `/*` és `*/` jelek között. Az ilyen megjegyzés több soron át is tarthat. Egy soron belül a `//` jel után írhatunk magyarázatot.

A mai fejlesztőkörnyezetekben egyáltalán nem ritka, hogy valamilyen speciális megjegyzést arra használjanak fel, hogy ennek a programszövegből történő kigyűjtésével a forrásállománynak vagy magának a programnak egy dokumentációját kapják. Ezt a kigyűjtést a fordító végzi el.

A C# fordítónak ha a `/doc:fájlnév` formában adunk egy fordítási paramétert, akkor `///` karakterek utáni információkat XML fájlba (a megadott névvel) kigyűjti. Ha nem parancssori fordítót használunk, ami a Visual Studio.NET környezet használatakor gyakran (majdnem mindig) előfordul, akkor ezt a beállítást a projekt Tulajdonságok dialógusablakában az *XML Documentation File* paraméter értékeként állíthatjuk be.



1. ábra

Ha a keretrendszerben egy változó vagy függvény definíciója elé beszurjuk a `///` karaktereket, akkor azt a szövegszerkesztő automatikusan kiegészíti a következő formában:

Példa:

```
/// <summary>
/// ez egy
/// </summary>
int i;
/// <summary>
/// ez me
/// </summary>
/// <param name="i">
public void
{
    ...
}
```

A `/** ... */` f
előző módszer n
gyakran.

Ezek alapján
forráskódját.

Példa:

```
using System;
class Program
{
    static
    {
        ...
    }
}
```

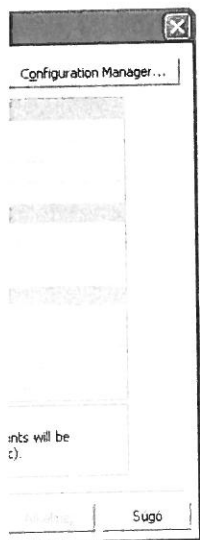
Programunk
dunk a legegysze
névteret, hanem
Ebben az osztály
függvényt. Mive
hozásig (a definí
dány nélkül is lé
A C stílusú nyelv
kiíró utasítások,
System névter rés
kus képernyőre ír
feladatát.

sealed
struct
try
checked unsafe
while

hat is elhelyezhetünk a
it is tarthat. Egy soron

átka, hogy valamilyen
k a programszövegből
ak a programnak egy
l.

gy fordítási paramétert,
a megadott névvel) ki-
a Visual Studio.NET
előfordul, akkor ezt a
z *XML Documentation*



definíciója elé beszurjuk a
tikusan kiegészíti a kö-

Példa:

```
/// <summary>
/// ez egy változó
/// </summary>
int i;
/// <summary>
/// ez meg egy függvény
/// </summary>
/// <param name="i"></param>
public void szamol(int i)
{
    ...
}
```

A `/** ...*/` forma is megengedett, szintén dokumentációs célokra. Mivel az előző módszer nem gyűjti ki az ilyen formájú megjegyzést, nem is használják gyakran.

Ezek alapján nézzük meg a szokásosnak nevezhető bevezető programunk forráskódját.

Példa:

```
using System;
class foci
{
    static void Main()
    {
        Console.WriteLine("Hajrá Fradi!");
    }
}
```

Programunk rövid magyarázataként annyit mondhatunk, hogy a forráskódunk a legegyszerűbb esetben egyetlen állományból áll, amiben nem definiálunk névtérrel, hanem csak egy *foci* osztályt. (Valamilyen típust kell definiálnunk!) Ebben az osztályban egyetlen függvényt definiálunk, a programot jelentő *Main* függvényt. Mivel egyetlen osztálytípus egyetlen példánya sem létezik a létrehozásig (a definíció még nem létrehozás!), ezért ahhoz, hogy a függvényünk példány nélkül is létezzen, a *static* jelzővel kell a hozzáférési szintet meghatározni. A C stílusú nyelvek hagyományaként a C# nyelvnek sem részei a beolvasó és kiíró utasítások, így könyvtári szolgáltatásokra kell hagyatkoznunk, ami a *System* névtér része, és ennek a névtérnek a *Console* osztálya biztosítja a klasszikus képernyőre írás, *Console.WriteLine()* és billentyűzetolvasás, *Console.ReadLine()* feladatát.

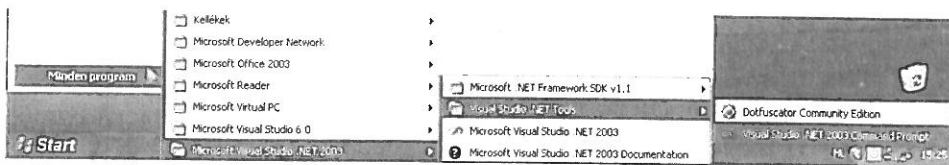
I. Alapismeretek

I.5. Parancssori környezet használata

A tetszőleges szövegszerkesztővel megírt forráskódot egy *alma.cs* (a nyelv a cs kiterjesztést szereti...) fájlba menthetjük, majd az alábbi utasítással fordíthatjuk le:

```
csc alma.cs
```

A fordítás eredménye egy *alma.exe* állomány lesz, amit futtatva a képernyő következő sorában láthatjuk kedvenc buzdító mondatunkat. Természetesen akkor kapunk hibamentes fordítási eredményt, ha a fordítónk és a könyvtárak használatához szükséges útvonali, környezeti változó beállítások helyesek. Ezt a *vcvars32.bat* állomány elvégzi. Ilyen parancssori környezetet legkönnyebben a Visual Studio.NET programcsoporthoz tartozó segédeszközök közül tudunk elindítani. (Ekkor lefut a *vcvars32.bat*, nem kell nekünk kézzel indítani!)



2. ábra

A parancssori fordítónak a */?* paraméterrel történő futtatása, mint egy segítség funkció, megadja a fordító fontosabb kapcsolóit.

Ezek közül a legfontosabbak a következők:

- /t:exe* alapértelmezés, exe állományt fordít.
- /t:library* a fordítandó állomány könyvtár (dll) lesz.
- /out:név* a fordítás eredményének nevét határozhatjuk meg, alapértelmezésben ez a név megegyezik a fordított fájl nevével.
- /r:valami.dll* egy könyvtár felhasználása fordításkor, ha több könyvtári állományt kell hozzáfordítani, akkor az állománynevek közé vesszőt kell tenni.
- /main:osztálynév* a nyelv megengedi, hogy több osztály is tartalmazzon *Main* függvényt, ekkor ezzel a kapcsolóval mondhatjuk meg, hogy a sok *Main* függvény közül melyik legyen a „főprogram”.

A keletkezett elegendő egy ha, úgy fogalmazzák, hanem szükséges! Ezt természetesen nem tudjuk menedzselni!

Általában a nyelvi eszközei keretrendszerre. Ennek kivételek. Illeg meg kell mondanunk fordítsa-e a forráskódot a menedzser processzor szintjén kódolva a menedzser nevezik.

I.6. A környezeti

Mielőtt a képernyőn kezdenénk, nézzük meg a környezeti változókat, hogy miképpen lehet őket kipróbálni.

Természetesen a parancssori fordító (pl: emacs) használatához a környezeti változókra sem, hiszen a fordító a környezeti változókra támaszkodik.

A fejlesztői környezetben indíthatunk a környezeti változók jellemző ablakokat.

A jelenlegi fordító a környezeti változók fedőnév alatt képes a környezeti változók ablakban végezni a legutóbbi parancsok nyithatunk meg. Ezek a feladatok.

A keletkezett exe állományról el kell mondani, hogy annak futtatásához nem elegendő egy hagyományos 32 bites Microsoft operációs rendszer, gyakran ezt úgy fogalmazzák meg, hogy a keletkezett program nem natív kódot tartalmaz, hanem szükséges az operációs rendszerhez hozzátelepíteni a .NET keretrendszert! Ezt természetesen a Visual Studio.NET installálása elvégzi, így a saját gépünkön nem tűnik fel annak hiánya sem. Az irodalom ezt az exe kódot gyakran menedzselt (managed) kódnak nevezi.

Általában elmondható, hogy az új fejlesztőkörnyezet minden egyes nyelvi eszköze olyan kódot fordít, aminek szüksége van erre a keretrendszerre. Természetesen mint az élet minden területén, úgy itt is vannak kivételek. Ilyen kivétel például a C++ nyelv, ahol alapértelmezés szerint meg kell mondani, hogy a programunkat menedzselt vagy natív kódra fordítsa-e a fordítónk. Natív kód alatt értjük azt a fordított eredményt, ami processzor szintű utasításokat tartalmaz. Ebben a fejlesztőkörnyezetben ezt a kódot a menedzselt ellentétéként, nem menedzselt (unmanaged) kódnak is nevezik.

I.6. A környezet használata

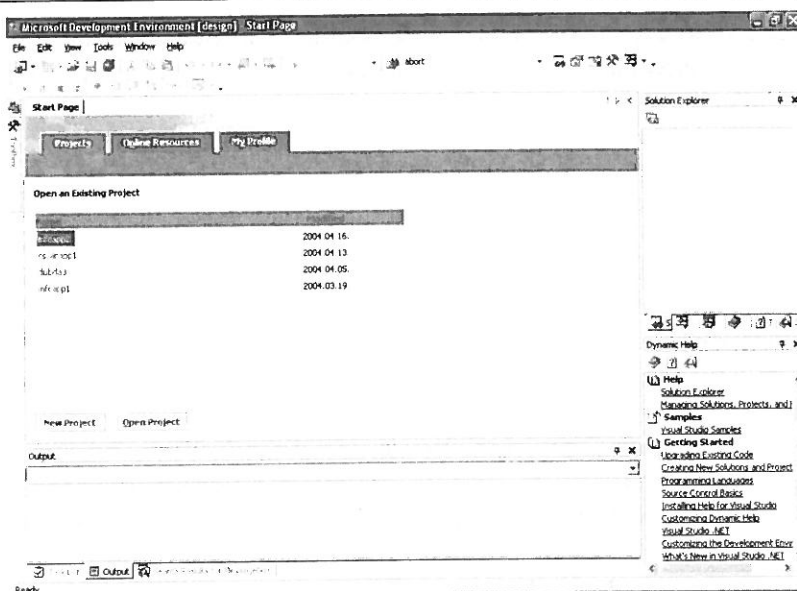
Mielőtt a következő részben a nyelv részletes jellemzőinek ismertetését kezdenénk, nézzük, hogy a felinstallált Visual Studio.NET környezet segítségével miképpen tudjuk az eddigi egy, illetve a későbbi példaprogramokat kipróbálni.

Természetesen nincs szükségünk egy tetszőleges szövegszerkesztő (pl: emacs) használatára, hiszen a fejlesztőkörnyezet ad egy jól használható belső szövegszerkesztőt, és nincs szükség a parancssori fordítás parancs kiadására sem, hiszen ez a parancs egy menüpontra van rádefiniálva.

A fejlesztőkörnyezet installációja után a Start \ *Programok* menüpontra keresztül indíthatjuk el a Visual Studio.NET környezetet, ami a 3. ábrán látható jellemző ablakkal jelenik meg.

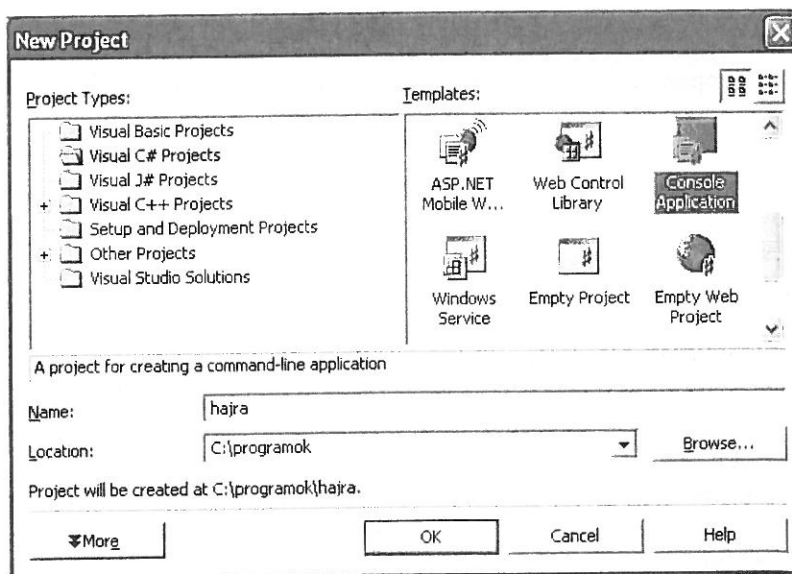
A jelenlegi fejlesztőkörnyezetekben minden program valójában egy projekt-fedőnév alatt készül el. Egy vagy több projekt felügyelete a *Solution Explorer* ablakban végezhető. A fő munkaterületen egy induló *Start Page* látható, amiben a legutóbbi projektek szerepelnek, illetve új vagy korábbi létező projektet nyithatunk meg. Természetesen a *File* menüpontra keresztül is elvégezhetők ezek a feladatok.

I. Alapismeretek



3. ábra

A *New Project* menüpont kiválasztása után, ahogy az az alábbi ablakban is látszik, három fő kérdésre kell válaszolnunk:



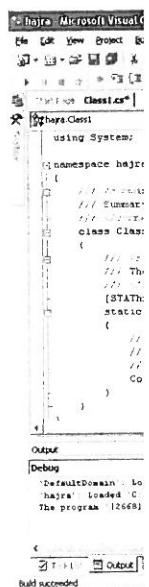
4. ábra

1. Ki kell válni
2. Az adott r... szeretnének
3. Meg kell a

Ebben a könyv választjuk, amit a

A *Templates* a illetve a követke választjuk, ami eg Ezek az alkalmazá

A munkakönyv hozza a munkakör ben a *c:\programc* mányt, aminek tar



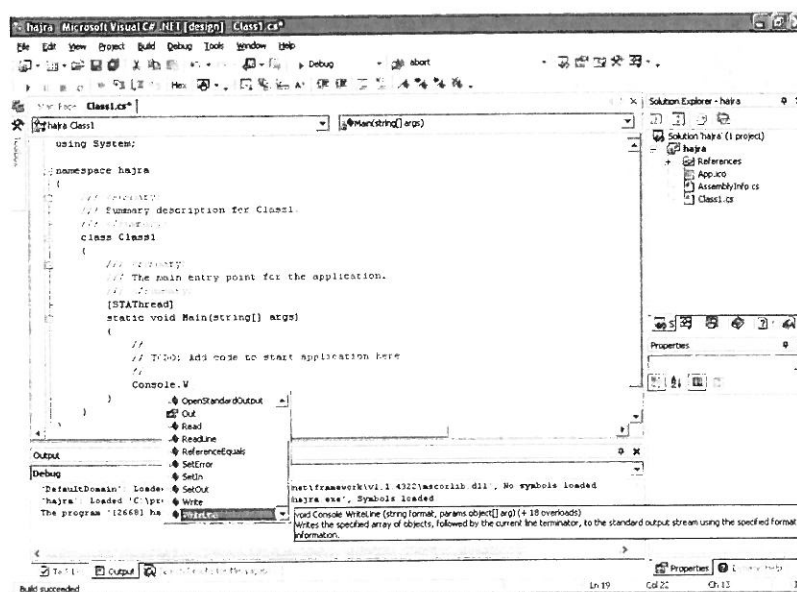
Ahogy látható, hogy csak a val lényegében mege nem kötelező név vény paraméterét

1. Ki kell választani a programozási nyelvet.
2. Az adott nyelvhez meg kell mondani, hogy milyen jellegű alkalmazást szeretnénk készíteni.
3. Meg kell adni a munkakönyvtárat és a projekt nevét.

Ebben a könyvben a nyelvet illetően mindig a *Visual C# Project* lehetőséget választjuk, amit a nyitott mappajel is mutat.

A *Templates* ablakban választhatjuk ki az alkalmazás jellegét. Jelen esetben, illetve a következő rész példáiban az ún. *Console Application* lehetőséget választjuk, ami egy hagyományos karakteres felületű programkörnyezetet jelent. Ezek az alkalmazások parancssori ablakban futnak.

A munkakönyvtár és a projekt nevének megadása azt jelenti, hogy létrehozza a munkakönyvtáron belül a projekt névvel megadott könyvtárat, esetünkben a *c:\programok\hajra* könyvtárat, és ezen belül egy önkényes *class1.cs* állományt, aminek tartalma a következő:



5. ábra

Ahogy látható, a keretrendszer a *class1.cs* állományba mindent előkészít, hogy csak a valódi feladatra kelljen koncentrálnunk. Az állomány tartalma lényegében megegyezik a korábbi első programkóddal. Ahogy már említettem, nem kötelező névteret (*namespace*) definiálni, illetve nem kötelező a *Main* függvény paraméterét jelölni, és a végrehajtási szál attribútum jelölése is opcionális.

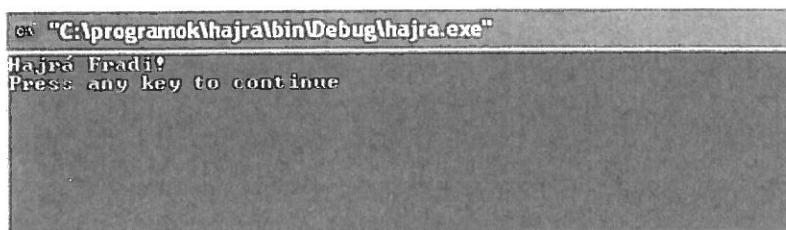
I. Alapismeretek

Ezeket, illetve a dokumentációs megjegyzéseket figyelembe véve látható, hogy a két kódrészlet azonos.

Amint a 3. ábráról látható, a fejlesztőkörnyezet rendelkezik a már szokásosnak nevezhető beíráskori értelmezéssel (*IntelliSense*), és ha általa ismert típust fedez fel, akkor egy helyi ablakban megmutatja az aktuális objektum elérhető jellemzőit.

A projekt a 3. ábrán látható forráskód mellett még egy *assemblyinfo.cs* állományt is tartalmaz, amiben három jellemző attribútum beállítását végezhetjük el. Beállíthatjuk programunk jellemző adatait, verziószámát, illetve a digitális aláírás kulcsállományra vonatkozó információt is. Ezek a tulajdonságok a .NET keretrendszer szolgáltatásain alapulnak, aminek részletezése meghaladja ennek a könyvnek a kereteit.

A programot a *Debug* menüpont *Start* (F5) vagy *Start without debugging* (CTRL-F5) menüpontjával fordíthatjuk le, illetve futtathatjuk. Ez utóbbi menüpontot használva a parancssori ablak nem tűnik el a program futása után, lehetőséget biztosítva a program eredményének megnézésére.



6. ábra

A fordítás hatására létrejön a *c:\programok\hajra* könyvtárban egy *bin* és egy *obj* könyvtár. Ez utóbbiban a lefordított állományok, míg a *bin* könyvtárban egy *Debug* vagy *Release* könyvtárban létrejön a kívánt exe program is. A két változat közti különbség a nevéből adódik, nevezetesen a *Debug* változatban a nyomkövetés elősegítését biztosítva készül el a futtatható állomány. Ez a lehetőség a programfejlesztések során nagyon hasznos, hiszen logikai vagy egyéb hibák keresésében a lépésenkénti, nyomkövető módszerrel történő végrehajtás nélkülözhetetlen. A *Release*, kiadási végleges változatot a program befejezéseként az 5. ábra fejlécében látható *Debug-Release* választómező állításával készíthetjük el.

Többfelhasználós környezetben, például Windows XP operációs rendszer alatt, a *Debugger Users* csoportba minden fejlesztőt bele kell rakni.

A könyv második részében a C# nyelv elemeinek megismerésekor jellemző projekt típusként konzolalkalmazást (6. ábra) használunk.

I.7. Windows

Az alkalmazás a grafikus alkalmazás is szokták nevezni

Ahogy a karal szítve a *Windows* megadjuk a nevet,

New Project

Project Types:

- ☐ Visual Basic
- ☐ Visual C# P
- ☐ Visual J# P
- ☒ Visual C++
- ☐ Setup and I
- ☐ Other Proje
- ☐ Visual Stud

A project for creat

Name:

Location:

☐ Add to Solution

Project will be creat

More

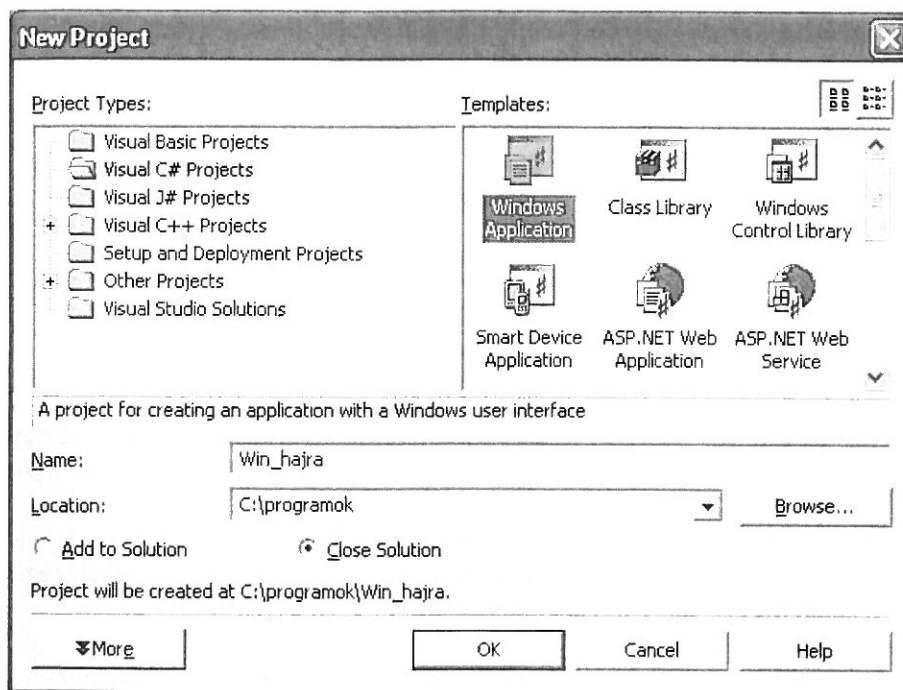
Miután a fen
juk, hogy a kar
hiszen ebben a r
induló program.

Ez valójában
amibe az eszköz
letre visszük, és
alakításnak megf

I.7. Windows alkalmazások készítése

Az alkalmazások másik, és talán ma már szélesebb körben használt csoportja a grafikus alkalmazás készítése. Ezeket a programokat Windows alkalmazásnak is szokták nevezni.

Ahogy a karakteres alkalmazásoknál már láttuk, új feladatot (projektet) készítve a *Windows Application* (Windows alkalmazás) lehetőséget választjuk és megadjuk a nevet, ahogy az a következő képen látható.



7. ábra

Miután a fenti dialógusablakban befejezzük az adatok megadását, azt láthatjuk, hogy a karakteres alkalmazásokhoz képest a munkafelület megváltozik, hiszen ebben a rendszerben az az alap elképzelés, hogy egy üres ablak (*form*) az induló program.

Ez valójában azt jelenti, hogy egy grafikus tervezőablakot kapunk (*Form1*), amibe az eszköztárból (*Toolbox*) a szükséges vezérlőelemeket a grafikus felületre visszük, és közben a forráskódot (*form1.cs*) a keretrendszer a felületalakításnak megfelelően automatikusan módosítja.

I. Alapismeretek

Az első grafikus programunkhoz három lépést végezzünk el!

1. Írjuk át az ablak fejlécszövegét. Ehhez kattintsunk egyet a *form*-ra, majd a jobb alsó *Properties* (Tulajdonságok) ablakban írjuk át a *Text* tulajdonságot. (*Első program!*)
2. A *Toolbox* ablakból helyezzünk fel egy *Button*, nyomógomb objektumot. A *Tulajdonság* ablakban állítsuk át a *Text* mezőt a *Vége* szövegre.
3. Kattintsunk kettőt a nyomógomb objektumra, ezzel a nyomógomb alapértelmezett kattintás esemény függvényét definiáltuk. Ebbe írjuk be a *Close()*, ablakbezárás utasítást. Ekkor a *form1.cs* forráskód részlete a következőképpen néz ki:

```
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void button1_Click(object sender, System.EventArgs e)
{
    Close();
}
```

Ezen három lépés után fordítsuk le, majd futtassuk a programot. Ezt, ahogy a korábbi karakteres program esetében is a *Debug* menü *Start* menüpontjával tehetjük meg.

Ahhoz, hogy további grafikus alkalmazásokat tudjunk készíteni, elengedhetetlenül szükséges, hogy ismerjük egyrészt a választott nyelv lehetőségeit, másrészt a rendszer könyvtári szolgáltatásait.

Az előbbi, a nyelvi lehetőségek megismerése általában a legkönnyebb és leggyorsabb feladat. Ez reményeim szerint a következő rész áttanulmányozása után a kedves Olvasónak is sikerül. Viszont az utóbbi, a rendszer könyvtári szolgáltatásainak megismerése hosszabb, több időt, sok egyéni munkát jelentő feladat. Azt remélem, hogy a befejező, a grafikus alkalmazások alapjaival foglalkozó rész után mindenki elég bátorságot érez magában a további önálló munka folytatásához.

I.8. Feladatok

1. Milyen l
2. Mit csin
3. Lehet-e
4. Mik a M
5. Egy pr
lefordíta

gezzünk el!

tsunk egyet a *form*-ra, majd
ekben írjuk át a *Text* tulaj-

tton, nyomógomb objektu-
t mezőt a *Vége* szövegre.
, ezzel a nyomógomb alap-
efiniáltuk. Ebbe írjuk be a
ml.cs forráskód részlete a

System.EventArgs e)

k a programot. Ezt, ahogy a
menü *Start* menüpontjával

udjunk készíteni, elenged-
lasztott nyelv lehetőségeit,

ltalában a legkönnyebb és
ező rész áttanulmányozása
őbbi, a rendszer könyvtári
sok egyéni munkát jelentő
kalmazások alapjaival fog-
nagában a további önálló

I.8. Feladatok

1. Milyen kódú programot fordít a Visual Studio.NET fejlesztőkörnyezet?
2. Mit csinál a Garbage Collection (szemétgyűjtő) algoritmus?
3. Lehet-e ékezetes karaktereket használni a C# programban?
4. Mik a *Main* függvény jellemzői?
5. Egy program hány *Main* függvényt tartalmazhat, hogy tudjuk lefordítani?

II. A C# nyelv alaptípusai

Egy programozási nyelvben az egyik legfontosabb tulajdonság az, hogy programkészítés során hogyan és milyen típusú adatokat használhatunk. Meg kell jegyezni, hogy van néhány olyan programozási nyelv is (pl. PHP), ahol ez a terület nem igazán fontos, típusok gyakorlatilag nincsenek, adatot szükség szerinti típusban a programozó rendelkezésére bocsát.

A C# szigorúan típusos nyelv, ebben a nyelvben csak ennek figyelembevételével használhatunk saját változókat.

II.1. Változók definiálása

A nyelvben a változók definiálásának alakja:

`típus változónév ;`

Azonos típusú változókat egymástól vesszővel elválasztva definiálhatunk. A típusok ismerete nélkül nézzünk néhány példát változók definiálására.

Példa:

```
char ch;           // ch változó karakter típusú
int egy, tizenegy; // az egy és tizenegy egész típusú
```

Változók lehetnek külsők, azaz függvényen kívüliek, vagy belsők, azaz függvényen belüliek. A függvényen kívüli változók is természetesen csak osztályon belüliek lehetnek. Gyakran hívják ezeket a változókat adatmezőknek is.

Belső változók, az adott blokkon (függvényen) belüli lokális változók lehetnek dinamikus vagy statikus élettartamúak. Módosító jelző nélküli definiálás esetén dinamikusnak vagy automatikusnak nevezzük, s ezek élettartama a blokkban való tartózkodás idejére korlátozódik. Ha a blokk végrehajtása befejeződött, a dinamikus változók megszűnnek.

Statikus élettartamú belső változót a *static* szó használatával (ahogy külső változó esetén igen) nem definiálhatunk. Láttuk, a függvények lehetnek statikusak (lásd *Main* függvény), melyekre ugyanaz igaz, mint az osztályváltozókra, ezen függvények élettartama is a programéval egyezik meg, más szóval ezen függvények a program indulásakor jönnek létre.

A változóka
cia típusúak, n
irodalomban ez
típusú, az úgyi
tozókat szokták

Az értéktíp
null értékre állít

A változók

II.2. Álland

Állandók de
gével tehetjük n

Példa:

```
const fl  
g=2.3;  
...  
const in
```

II.3. Változ

Változók ke
a következő for

Példa:

```
char s='  
char[] b
```

A változók
példát.

vi

abb tulajdonság az, hogy
tokat használhatunk. Meg
yelv is (pl. PHP), ahol ez a
incsenek, adatot szükség

1 csak ennek figyelembe-

A változókat két kategóriába sorolhatjuk, az osztálytípusú változók referen-
cia típusúak, melyek mindig a dinamikus memóriában helyezkednek el (az
irodalomban ezt gyakran *heap*-nek nevezik), míg minden más nem osztály-
típusú, az úgynevezett értéktípusú (*value type*) változó. Az értéktípusú vál-
tozókat szokták *stack változóknak* is nevezni.

Az értéktípusú változók kezdőértékét, az inicializálás hiányában, *0*, *false*,
null értékre állítja be a fordító.

A változók definiálásakor rögtön elvégezhető azok direkt inicializálása is.

II.2. Állandók definiálása

Állandók definiálását a típusnév elé írt *const* típusmódosító szócska segítsé-
gével tehetjük meg. A definíció alakja:

const típus név = érték;

Példa:

```
const float g=9.81;    // valós konstans
g=2.3;                // !!! HIBA
...
const int min=0;      // egész konstans
```

álasztva definiálhatunk. A
k definiálására.

er típusú
y egész típusú

üliek, vagy belső, azaz
természetesen csak osztá-
ókat adatmezőknek is.
lülő lokális változók lehet-
ó jelző nélküli definiálás
s ezek élettartama a blokk-
végrehajtása befejeződött,

asználatával (ahogy külső
üggvények lehetnek statik-
mint az osztályváltozókra,
zik meg, más szóval ezen

II.3. Változók inicializálása

Változók kezdő értékadása, inicializálása azok definiálásakor is elvégezhető
a következő formában:

típus változó = érték;

Példa:

```
char s='a'; int []a={1,2,3};
char[] b="Egy";
```

A változók, konstansok és függvények használatára nézzük a következő
példát.

II. A C# nyelv alaptípusai

Példa:

```
// A változó.cs fájl tartalma:
using System;
// A kiíráshoz szükséges deklarációt tartalmazza.
class változó
{
    static int alma=5;           //alma változó definíciója
                                //és inicializálása
    static float r=5.6F          //statikus valós típusú változó
                                //valós konstans zárhat az
                                //F (float) betű
    const float pi=3.1415;       //konstans definíció

    static void Main()
    {
        Console.WriteLine( "Teszt"); //eredmény Test
        Console.WriteLine( alma );    //eredmény 5
        int alma=6;                   //helyi változó
        Console.WriteLine( alma );     //eredmény 6
        Console.WriteLine( változó.alma ); //a külső takart (5)
                                        //változóra hivatkozás
        Console.WriteLine( terület(r)); //a terület függvény
                                        // meghívása
    }

    static float terület(float sugár) // függvénydefiníció
    {
        return(pi*sugár*sugár);
    }
}
```

Bár még nem definiáltunk függvényeket, így talán korainak tűnhet ez a példa, de inkább felhívnom még egyszer a figyelmet az ékezetes karakterek használatára. A későbbi mintafeladatokban, ha előfordul ékezet nélküli változó, függvénydefiníció, akkor az csak a korábbi környezetek „rossz utóhatásának” köszönhető.

II.4. Elemi típusok

char – karakter típus (2 byte hosszú)

A karakter típus 16 bites karakterábrázolást (unikód) használ. Általában igaz, hogy minden alaptípus mérete rögzített.

A karakter típusú változó értékét aposztróf (') jelek között tudjuk megadni.

Példa:

```
char c;
c='a';
```

A backslash ter(eke)t, mint escape szekvenc

```
\a
\b
\f
\r
\n
```

Az új sor k hatásával.

```
\t
\v
\\
\'
\"
\?
\uxxxx
```

string – karakt

A **System.St** szövegösszefűzé indexű karakter szöbőli a szövb backslash (\) kar

Példa:

```
char c='\n';
string s=
string n=
s+="fa";
char c1=s
char c2="
string f=
string fi
```

Példa:

```
char c;
c='a';
```

talmazza.

ma változó definíciója
inicializálása
atikus valós típusú változó
lós konstanst zárhat az
(float) betű
nstants definíció

```
//eredmény Test
//eredmény 5
//helyi változó
//eredmény 6
); //a külső takart (5)
//változóra hivatkozás
//a terület függvény
// meghívása

// függvénydefiníció
```

A backslash (\) karakter speciális jelentéssel bír. Az utána következő karakter(ek)e)t, mint egy escape szekvenciát dolgozza föl a fordító. Az így használható escape szekvenciák a következők:

\a	-	a 7-es kódú csipogás
\b	-	backspace, előző karakter törlése
\f	-	formfeed, soremelés karakter
\r	-	kocsi vissza karakter
\n	-	új sor karakter (soremelés+kocsi vissza)

Az új sor karakter hatása azonos a formfeed és a kocsi vissza karakterek hatásával.

\t	-	tabulátor karakter
\v	-	függőleges tabulátor
\\	-	backslash karakter
\'	-	aposztróf
\"	-	idézőjel
\?	-	kérdőjel
\uxxy		xx és yy unikódú karakter

string – karaktersorozat

talán korainak tűnhet ez a
met az ékezetes karakterek
rdul ékezet nélküli változó,
zetek „rossz utóhatásának”

A *System.String* osztálynak megfelelő típus. Szövegek között a + és a += szövegösszefűzést végző operátorok ismertek. A [] operátor a szöveg adott indexű karakterét adja meg. Az indexelés 0-val kezdődik. A @ karakter kiküszöböli a szövegen belüli „érzékeny” karakterek hatását. Ilyen például a backslash (\) karakter.

Példa:

```
char c='\u001b';           // c= a 27-es kódú (Esc) karakterrel
string s="alma";
string n="alma"+"barack";
s+="fa";                   //s= almafa
char c1=s[2];              // c1= 'm'
char c2="szia"[1];         // c2='z'
string f="c:\\programok\\alma.txt";
string file=@"c:\programok\alma.txt";
```

unikód) használ. Általában

ek között tudjuk megadni.

II. A C# nyelv alaptípusai

A *String* osztály a fenti lehetőségeken kívül egy sor függvénnyel teszi használhatóbbá ezt a típust. Ezen függvények közül a legfontosabbak:

Length

Csak olvasható tulajdonság, megadja a szöveg karaktereinek a számát:

```
string s="alma";
int i=s.Length;      //i=4
```

CompareTo(string)

Két szöveg összehasonlítását végzi. Ha az eredmény 0, akkor azonos a két szöveg:

```
string str1="egyik", str2="másik";
int cmpVal = str1.CompareTo(str2);
if (cmpVal == 0)      // az értékek azonosak
{...}
else if (cmpVal > 0)  // str1 nagyobb mint str2
{...}
else                 // str2 nagyobb mint str1
```

Equals(string)

Megadja, hogy a paraméterül kapott szöveg azonos-e az eredeti szöveggel:

```
string str1="egyik", str2="másik";
if (str1.Equals(str2)){...} // azonos
else{...}                  // nem azonos
```

IndexOf(string)

Több alakja van, megadja, hogy az eredetiben melyik indexnél található a paraméterül kapott szöveg. A visszaadott érték -1 lesz, ha nincs benn a keresett szöveg:

```
int i="almafa".IndexOf("fa"); //4
```

Insert(string,int)

A második paraméterül kapott szöveget, az első paraméterrel megadott indextől beszúrja az eredeti szövegbe:

```
string s="almafa alatt";
string ujs=s.Insert(4," a "); // alma a fa alatt
```

A szöveget (*System.String*) még azt, hogy a szövegobjektum marad.

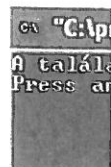
Szöveges fe szövegminták se tibilis reguláris reguláris kifeje *System.Text.Reg*

Példa:

```
using Sys
using Sys
class rex
{
    publ
    {
```

A fenti pélc betű különbözik szerint:

Regex rex akkor a futási er



egy sor függvénnyel teszi a legfontosabbak:

araktöreinek a számát:

lmény 0, akkor azonos a két

osak

nt str2

nt str1

os-e az eredeti szöveggel:

melyik indexnél található a k -I lesz, ha nincs benn a

első paraméterrel megadott

a fa alatt

A szövegtípus további szolgáltatásait, függvényeit a szövegosztály (*System.String*) online dokumentációjában érdemes megnézni. Meg kell említeni még azt, hogy a szövegosztály függvényei nem módosítják az eredeti szöveget, szövegobjektumot, példaként az előző *s* szöveges változó értéke változatlan marad.

Szöveges feldolgozási feladatok során a reguláris kifejezésekkel megadható szövegminták segítségét is igénybe vehetjük. A .NET Framework a Perl5 kompatibilis reguláris kifejezéshasználattal támogatja. A *Regex* osztály szolgáltatja a reguláris kifejezést, míg a *Match* a találati eredményt. Az osztályokat a *System.Text.RegularExpressions* névtérben találjuk.

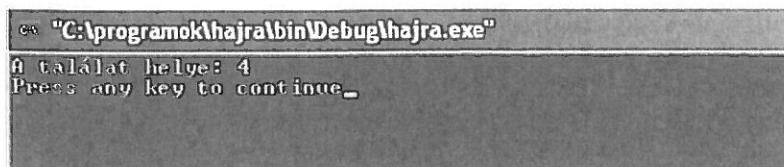
Példa:

```
using System;
using System.Text.RegularExpressions;
class reguláris
{
    public static void Main()
    {
        Regex reg_kif = new Regex("fradi");
        // a fradi keresése
        Match találat = reg_kif.Match("A Fradi jó csapat?");
        if (találat.Success)
        {
            // (hol találtuk meg
            Console.WriteLine("A találat helye: " + találat.Index);
        }
    }
}
```

A fenti példa nem ad eredményt, hiszen alapértelmezésben a kis- és nagybetű különbözik, míg ha a reguláris kifejezést egy kicsit módosítjuk, az alábbiak szerint:

```
Regex reg_kif = new Regex("adi");
```

akkor a futási eredmény az alábbi lesz:



8. ábra

II. A C# nyelv alaptípusai

Ha azt szeretnénk elérni, hogy az eredeti szövegünk is változzon, akkor ehhez a *System.Text* névtér *StringBuilder* osztályát tudjuk igénybe venni.

Példa:

```
using System.Text;
...
StringBuilder s1 = new StringBuilder("almafa alatt");
s1.Insert(4, " a ");
Console.WriteLine(s1);    // "alma a fa alatt"
```

int – egész típus(4 byte)

Az egész típusú értékek négy bájtot foglalnak – a ma leggyakrabban használt nyelvi környezetben –, így értelmezési tartományuk -2^{31} , $2^{31} - 1$ között van.

long	hosszú egész (8 byte)
short	rövid egész (2 byte)
sbyte	előjeles (signed) byte
float	valós (4 byte)
double	dupla pontosságú valós (8 byte)
decimal	„pénzügybeli” típus (16 byte), 28 helyiérték

Mindkét egész típus előjeles, ha erre nincs szükségünk, használhatjuk az előjel nélküli változatukat, melyek: *uint*, *ushort*, *byte*, *ulong*.

Valós számok definiálásakor a 10-es kitevőt jelölő *e* konstans használható.

Példa:

```
float a=1.6e-3;    // 1.6 * 10-3
```

void – típus nélküli típus

Az olyan függvénynek (eljárásnak) a típusa, amelyik nem ad vissza értéket.

Példa:

```
void növel(ref int mit)
{
    mit++;
}
```

Az iménti függvény paraméterének jelölése referencia szerinti paraméterátadást jelent, amiről a *Függvények* c. fejezetben részletesen szólnunk.

bool – logikai típus

A logikai típus Ahogy a nyelv fő nyelvi kulcsszó a

Általában elh az egyes implemen gesen pótolják, é mint a függvénye

II.5. Felsorol

A felsorolás mák definiálása.

Ennek a kul meg kell adni. segítségével defi

Példa:

```
enum szín
enum betu
betuk saj
enum vala
```

Ekkor a 0,1, kiírásakor ezen kell ezeket feltét

Példa:

```
class Szi
{
    enum
    publ
    {
    }
}
```

vegünk is változzon, akkor adjuk igénybe venni.

afa alatt");

alatt"

a ma leggyakrabban használt -2^{31} , $2^{31} - 1$ között van.

yte)
), 28 helyiérték

égünk, használhatjuk az elő-
long.
lő e konstans használható.

lyik nem ad vissza értéket.

ferencia szerinti paraméter-
zletesen szólunk.

bool – logikai típus (1 byte)

A logikai típus értéke a *true* (igaz) vagy *false* (hamis) értéket veheti fel. Ahogy a nyelv foglalt alapszavainál láhattuk, ezeket az értékeket a *true* és *false* nyelvi kulcsszó adja meg.

Általában elmondható, hogy míg a nyelv nem definiál sok alaptípust, addig az egyes implementációk, főleg azok grafikus felület alatti könyvtárai ezt bősé-
gesen pótolják, és gyakran több időt kell az 'új típusok' megfejtésének szentelni,
mint a függvények tanulmányozásának.

II.5. Felsorolás típus

A felsorolás típus gyakorlatilag nem más, mint egész konstans(ok), szinoni-
mák definiálása. Felsorolás típust névtéren vagy osztályon belül definiálhatunk.

Ennek a kulcsszava az *enum*. A kulcsszó után a felsorolás típus azonosítóját
meg kell adni. A *System.Enum* osztály szinonimáját adja az ezen kulcsszó
segítségével definiált típus.

Példa:

```
enum szinek {piros,kék,zöld,sárga};
enum betuk {a='a', b='b'};
betuk sajátbetű=betuk.a;           // változó kezdőértéke a
enum valami { x="alma"};           // hiba
```

Ekkor a 0,1, ... értékek rendelődnek hozzá a felsorolt nevekhez, és a nevek
kiírásakor ezen számokat is látjuk. A kezdőértékdás lehetőségével élve nem
kell ezeket feltétlenül elfogadnunk, hanem mi is megadhatjuk az értéküket.

Példa:

```
class Szinek
{
    enum újszinek {piros=4, kék=7, zöld=8, sárga=12};

    public static void Main()
    {
        Console.WriteLine(újszinek.zöld); // kiírja a színt
        //ciklus a színeken történő végighaladásra
        for(újszinek i=újszinek.kék; i<újszinek.sárga; i++)
        {
            //kiírja a színeket
            Console.WriteLine(i);
        }
    }
}
```

II. A C# nyelv alaptípusai

A fenti ciklus eredményeként azon egészek esetén, ahol az egész értékéhez egy „nevet” rendeltünk hozzá, a név kerül kiírásra, egyébként a szám. Az eredmény a következő lesz:

```
Kék
Zöld
9
10
11
```

Mivel ez a típus a *System.Enum* megfelelője, ezért rendelkezik annak jellemzőivel is. Ezek közül a két legjellemzőbb a *GetHashCode()* és a *ToString()* függvény. Az előbbi a belső tárolási formát, a megfelelő egész számot adja meg, míg a *ToString()*, mint alapértelmezett reprezentáció, a szöveges alakot (kék,zöld, stb) adja meg.

Példa:

```
újszinek s=újszinek.piros;
Console.WriteLine(s.GetHashCode()); // eredmény: 4
```

A felsorolás típus alapértelmezésben egész típusú értékeket vesz fel. Ha ez nem megfelelő, akár *byte* (vagy tetszőleges egész szinonima) típusú értékeket is felvehet úgy, hogy a típusnév után kettősponttal elválasztva megadom a típusnevet. (Nem adhatom meg a valós vagy karakter típust!)

Példa:

```
enum színek:byte {piros,kék,zöld,sárga};
...
enum hibás:float {rossz1, rossz2}; // ez fordítási hiba
```

Előfordulhat, hogy olyan felsorolásadatokra van szükségem, melyek nem egy egész konstanshoz, hanem egy bithez kötődnek, hiszen ekkor a logikai és/vagy műveletekkel a felsorolásadatok kombinációját határozhatjuk meg. Ebben az esetben a *[Flags]* attribútumot kell az *enum* szócska elé szúrni.

Példa:

```
[Flags] enum alma
{piros=1,jonatán=2,zöld=4,golden=8};
...
alma a=alma.piros | alma.jonatán;
Console.WriteLine(a); // piros, jonatán
a=(alma) System.Enum.Parse(typeof(alma), "zöld,golden");
```

```
// a felsoro
// a könyvtá
//
Console.Writ
...
// eredmény
```

Bithez kötött ér
adni a neki megfelel

Az alaptípusoka
környezetnek egy n
zésre. Az természet
hivatkozni, de a ny
kód) fordítja le a kei

Az alaptípusok ;
zett, ahogy a C++ v
ben megengedett, a
context). Erről rövid

II.6. Feladatok

1. Milyen alap
2. Mi a változ
3. Mi a külön
zött?
4. Definiáljon
5. Ábrázolja fe

etén, ahol az egész értékéhez, egyébként a szám. Az ered-

```
// a felsorolás típus egyszerű értékadása helyett
// a könyvtári hívás lehetőségét is használhatjuk
//
Console.WriteLine(a.GetHashCode());
...
// eredmény 12 lesz
```

ért rendelkezik annak jellem-
HashCode() és a ToString()
felelő egész számot adja meg,
entáció, a szöveges alakot

// eredmény: 4

usú értékeket vesz fel. Ha ez
szinonima) típusú értékeket is
választva megadom a típus-
ust!)

// ez fordítási hiba

an szükségem, melyek nem
nek, hiszen ekkor a logikai
nációját határozhatjuk meg.
um szócska elé szűrni.

atán
"zöld,golden");

Bithez kötött értékek esetén kötelező minden egyes konstans névhez megadni a neki megfelelő bites ábrázolást!

Az alaptípusokat a .NET keretrendszer biztosítja, így ennek a fejlesztési környezetnek egy másik nyelvben pontosan ezek a típusok állnak rendelkezésre. Az természetesen előfordulhat, hogy nem ezekkel a nevekkal kell rájuk hivatkozni, de a nyelvi fordító biztosan ezen értelmezés szerinti kódot (köztes kód) fordítja le a keretrendszer, mint futtató környezet számára.

Az alaptípusok zárásaként meg kell jegyezni, hogy a mutató típus is értelmezett, ahogy a C++ világában, de ennek a használata csak olyan C# programrészben megengedett, amely nem biztonságos környezetben helyezkedik el (*unsafe context*). Erről röviden a könyv XIV. fejezetében olvashat.

II.6. Feladatok

1. Milyen alaptípusokat ismer a C# nyelv?
2. Mi a változó is az állandó között a különbség?
3. Mi a különbség egy statikus és egy dinamikus élettartamú változó között?
4. Definiáljon két szöveg típusú változót, adja meg a hosszukat!
5. Ábrázolja felsorolás típussal az NB 1 bajnokság csapatait!

III. Kifejezések, műveletek

A nyelv jellemzője a korábban (például C++ nyelvben) megismert, megszokott kifejezésfogalom, amit tömören úgy is fogalmazhatunk, hogy a vezérlési szerkezeteken kívül a nyelvben minden kifejezés.

Egy kifejezés vagy elsődleges kifejezés, vagy operátorokkal kapcsolt kifejezés.

Elsődleges kifejezések:

- azonosító
- (kifejezés)
- elsődleges kifejezés[]
- függvényhívás
- elsődleges kifejezés.azonosító

A kifejezéseket egy-, két- vagy háromoperandusú operátorokkal kapcsolhatjuk össze.

Egy kifejezésen belül először az elsődleges kifejezések, majd utána az operátorokkal kapcsolt kifejezések kerülnek kiértékelésre.

A nyelvben használható operátorok prioritási sorrendben a következők.

III.1. Egyoperandusú operátorok

new

A dinamikus helyfoglalás operátora. A helyfoglalás operátorának egyetlen operandusa az, hogy milyen típusú objektumnak foglalunk helyet. Sikeres helyfoglalás esetén a művelet eredménye a lefoglalt memória címe. Sikertelen helyfoglalás esetén a *null* mutató a visszatérési eredmény.

Példa:

```
int[] a;  
a = new int;           //egy egész tárolásához  
                        //elegendő hely foglalása  
a = new int[5]         //öt egész számára foglal helyet
```

A *new* utasítás gyakran szerepel a vektorokkal összefüggésben, amiről később részletesen szólunk.

A .NET ké
dinamikusan fog
használható mem
szüksége. Sok ny

-
!
~
++,--
(típus) ki

Példa:

```
double d=  
int i=(in
```

A ++ és --
esetben a váltó
kiértékelésekor, i

Példa:

```
a= ++b;  
a= b++;
```

Az egyopera
jobbról balra
precedenciánál b

III.2. Kétop

*
/
%

E három op
adja.
(% operátor
azonos.)

etek

C++ nyelvben) megismert,
, is fogalmazhatunk, hogy a
ifejezés.
vagy operátorokkal kapcsolt

A .NET keretrendszer egyik legfontosabb tulajdonsága az, hogy a dinamikusan foglalt memóriaterületeket automatikusan visszacsatolja a felhasználható memóriatartományba, ha arra a memóriára a programnak már nincs szüksége. Sok nyelvi környezetben erre a *delete* operátor szolgál.

-	aritmetikai negálás
!	logikai negálás (not)
~	bitenkénti negálás,
++,--	inc, dec.
(típus) kifejezés	típuskényszerítés

Példa:

```
double d=2.3;
int i=(int) d;    // i=2
```

A ++ és -- operátor szerepelhet mind pre-, mind postfix formában. Prefix esetben a változó az operátorral változtatott értékét adja egy kifejezés kiértékelésekor, míg postfix esetben az eredetit.

Példa:

```
a= ++b;    // hatása: b=b+1; a=b;
a= b++;    // hatása: a=b; b=b+1;
```

Az egyoperandusú operátorok és az értékadás operátora esetén a végrehajtás jobbról balra haladva történik, minden más operátor esetén azonos precedenciánál balról jobbra.

III.2. Kétooperandusú operátorok

*	szorzás
/	osztás
%	maradékképzés

E három operátorral mindig igaz: az $(a/b)*b+a\%b$ kifejezés az a értékét adja.

(% operátor esetén az operandusok nem lehetnek valósak, és a prioritásuk azonos.)

III. Kifejezések, műveletek

+	összeadás, string esetén azok összefűzése
-	kivonás
<<	bitenkénti balra léptetés
>>	bitenkénti jobbra léptetés

A jobbra léptetés operátorához példának tekintsük az $a \gg b$; utasítást. Ekkor ha az a unsigned, akkor balról nullával, különben pedig az előjelbittel tölt a bitenkénti jobbra léptetés operátora.

Példa:

```
int a = -1, b;  
b = a >> 2;           // b értéke -1 marad
```

<, >	kisebb, nagyobb relációs operátorok
<=, >=	kisebb vagy egyenlő, ill. a nagyobb vagy egyenlő operátorok
==, !=	egyenlő, nem egyenlő relációs operátorok
&	bitenkénti és
^	bitenkénti kizáró vagy
	bitenkénti vagy
&&	logikai és
	logikai vagy

is

Logikai operátor, egy objektumról megmondja, hogy a bal oldali operandus a jobb oldali típusnak egy változója-e.

Példa:

```
...  
int i = 3;  
if (i is int) Console.WriteLine("Bizony az i egész!");  
    else Console.WriteLine("Bizony az i nem egész!");
```

as

Kétooperandusú típuskényszerítés. A bal oldali változót a jobb oldali referencia típusra alakítja, ha tudja. Ha sikertelen az átalakítás, akkor eredményül a *null* értéket adja.

Példa:

```
...  
double d = 2.5;  
Object o = d as Object;  
Console.WriteLine(o);           // eredmény: 2.5  
// Object-re mindent lehet alakítani, aminek van toString kiíró  
// függvénye. Erről bővebben később esik szó.
```

typeof

Egy *System* mezőfüggvény információ

Példa:

```
using System;  
class Test  
{  
    stat
```

stat

A program f

```
System.Ir  
System.St  
System.Dc  
System.Vc
```

A típuskonv
létrehozhatunk
boxing-nak, csom
zárójeles típus
szakirodalom.

Példa:

```
int i = 5;  
Object o =  
int j = (ir
```

Az *Object*
őseként tekinthe
használt típusho

ok összefűzése

intsük az $a >> b$; utasítást.
 nben pedig az előjelbittel tölt

-1 marad

perátorok
 nagyobb vagy egyenlő

ációs operátorok

u, hogy a bal oldali operandus

az i egész!");
 nem egész!");

zót a jobb oldali referencia típusra
 eredményül a *null* értéket adja.

2.5
 minek van toString kiíró
 szó.

typeof

Egy *System.Type* típusú objektumot készít. Ennek az objektumnak a mezőfüggvényeivel, tulajdonságaival (*FullName*, *GetType()*) tudunk típusinformációhoz jutni.

Példa:

```
using System;
class Teszt
{
    static void Main(){
        Type[] t = {
            typeof(int),
            typeof(string),
            typeof(double),
            typeof(void)
        };
        for (int i = 0; i < t.Length; i++)
        {
            Console.WriteLine(t[i].FullName);
        }
    }
}
```

A program futása a következő eredményt produkálja:

```
System.Int32
System.String
System.Double
System.Void
```

A típuskonverzióval kapcsolatban elmondható, hogy minden értéktípusból létrehozhatunk egy neki megfelelő *Object* típusú objektumot. Ezt gyakran *boxing*-nak, csomagolásnak, míg az ellenkező kicsomagoló műveletet, amihez a zárójeles típuskonverzió operátort kell használni, *unboxing*-nak nevezi a szakirodalom.

Példa:

```
int i=5;
Object o=i;           // boxing, becsomagolás
int j=(int) o;         // unboxing, kicsomagolás
```

Az *Object* típus, ami a *System.Object* típusnak felel meg, minden típus őseként tekinthető. Ennek a típusnak a függvényei ily módon minden általunk használt típushoz rendelkezésre állnak.

III. Kifejezések, műveletek

Az *Object* típus tagfüggvényei a következők:

ToString()

Megadja az objektum szöveges reprezentációját. Ha erre van szükség, például kiírásnál, akkor ezt automatikusan meghívja. Ha ezt egy új típushoz újradefiniáljuk, akkor ez hívódik meg kiírás esetén.

```
Console.WriteLine(o);          // indirekt ToString hívás
Console.WriteLine(o.ToString());
```

GetType()

Megadja az objektum típusát, hasonló eredményt ad, mint a korábban látott *typeof* operátor.

Equals(object)

Megadja, hogy két objektum egyenlő-e, logikai értéket ad eredményül.

```
int i=5;
object o=i;
Console.WriteLine(o.Equals(5));
```

Az *Equals* függvénynek létezik egy statikus változata is, aminek a formája: *Object.Equals(object, object)*

GetHashCode()

Megadja az objektum *hash* kódját, ami egy egész szám. Tetszőleges saját utódtípusban újradefiniálhatjuk, amivel a saját típusunk *hash* kódját tudjuk számolni.

III.3. Háromoperandusú operátor

Az operátorral képezhető kifejezés alakja a következő: *e1 ? e2 : e3*, ahol *a ?* és *a :* az operátor jelei, míg *e1, e2, e3* kifejezések.

A kifejezés értéke *e2* ha *e1* igaz (nem 0), különben *e3*.

Példa:

```
char c;
int a;
...
a=((c>='0' && c<='9') ? c-'0' : -1);
```

Az *a* változó
A háromoperandusú
hatékony kifejezés

III.4. Kétop

=

A nyelvben
oldalán olyan k
vehet fel. Gya
egyenlőségjel j
érték, *rvalue*).
korlátozás nincs
Az értékadá
egyben ez lesz a

Példa:

```
...
char []d;
while (i<n)
{
    d[i] = c;
}
char []c;
d=c;
```

Az értékadá
hatására *a* és *b* i
Az egyenlőségjel
'kombinált' értéke
Ezek az ope

+=, -=, *=, /=

Jelentésük:
e1 = e1 op e2 ki

Az *a* változó vagy a 0-9 vagy -1 értéket kapja.

A háromoperandusú operátor valójában egy logikai elágazás utasítás. A hatékony kifejezőkészítés, tömörebb írásmód kiváló eszköze.

III.4. Kétopoperandusú értékadó operátorok

= értékadás operátora

A nyelvben az értékadás sajátos, önmaga is kifejezés. Az egyenlőségjel bal oldalán olyan kifejezés, változó állhat, amely által képviselt adat új értéket vehet fel. Gyakran hívja a szakirodalom ezt balértéknek is (*lvalue*). Az egyenlőségjel jobb oldalán egy értéket adó kifejezés kell, hogy álljon (jobbérték, *rvalue*). Ez gyakorlatilag azt jelenti, hogy a jobbértékre semmilyen korlátozás nincs.

Az értékadás során a bal oldal megkapja a jobb oldali kifejezés értékét, és egyben ez lesz a kifejezés értéke is.

Példa:

```
...
char []d=new char[80]; // másoljuk az s forrásszöveget d-be
while (i<s.Length)
{
    d[i]=s[i]; i++;
}
char []c={'L','a','l','i'};
d=c; // hiba!! mivel d már egy 80 karakteres
//területet jelöl, ezért új területet már nem
// jelölhet, d nem lehet balérték
```

Az értékadás operátora jobbról balra csoportosít, ezért pl. az $a=b=2$; utasítás hatására *a* és *b* is 2 lesz.

Az egyenlőségjel mellett még néhány, az aritmetikai operátorokkal 'kombinált' értékadó operátor is használható.

Ezek az operátorok a következők:

$+=, -=, *=, /=, \%, >>=, <<=, \&=, \^{\wedge}=, |=$

Jelentésük: az *e1* operátor *e2* kifejezés, ahol *e1* és *e2* is kifejezések, az $e1 = e1 \text{ op } e2$ kifejezésnek felel meg, ahol *op* az egyenlőségjel előtti operátor.

III. Kifejezések, műveletek

Példa:

```
int a=2;
a *= 3;      // a= a*3, azaz a értéke 6 lesz
string b="alma";
b+="fa";     // b=almafa
```

A típusok egymás közti konverziójával kapcsolatban azt lehet mondani, hogy a C vagy C++-ban ismert automatikus konverziók nem léteznek. Egy egész típusú változót egy valósba gond nélkül beírhatunk, míg fordítva már hibát jelez a fordító. A konverziós lehetőségek széles skáláját nyújtja a fejlesztőeszközünk.

Konverziókkal kapcsolatban két esetet szoktak megkülönböztetni. Ezek közül az első szám szöveggé alakítása. Ez gyakorlatilag nem igényel semmilyen segítséget, a számtípushoz tartozó osztály *ToString* függvényét hívja meg a fordító. Ehhez nem kell semmit sem tenni.

Példa:

```
int a=2;
string s="Az eredmény:" + a;
```

A másik eset, mikor szövegből szeretnénk számot kapni, már nem ilyen automatikus, de semmiképpen nem lehet bonyolultnak nevezni. Többféle módszer lehet az alakításra, de a legáltalánosabb talán a *Convert* osztály használata. Az osztály konvertáló függvényei azonos névkonvencióval az alábbi formájúak:

Convert.ToCTStípusnév

ahol a CTS (Common Type System) típusnév az alábbi lehet: *Boolean*, *Int16* (short int), *Int32* (rendes int), *Int64* (hosszú int), *Float*, *Double*, *String*, *Decimal*, *Byte*, *Char*.

Példa:

```
string s="25";
int i=Convert.ToInt32(s); // i=25 lesz
```

Érdekességgént megemlítem, hogy létezik a *Convert.ToDateTime(object)* függvény is, ami egy könyvtári dátumtípust készít a paraméterül kapott objektumból.

Ha bármelyik konvertáló függvény hibás paramétert kap, nem tud eredményt produkálni, akkor *InvalidCastException* kivételt vagy más, a hiba okára utaló kivételt generál. A kivételkezelésről később részletesen is fogunk beszélni.

Általában ig
gesek, így mind
ezért a konvertá
Hasonló kor
vényét használj

Gyakran elő
igényeinket. A
műveleteket biz

Math.Si

Math.Co

Math.Ta

Math.Ex

Math.Lo

Math.Sc

Math.Ar

Math.Ra

Math.Co

Math.Fl

Math.Po

Math.Pl

Math.E

Példa:

```
...
double d
Console.'
dd=Math.'
Console.'
```

Matematika
igény. A Syste
generálásának l
időhöz (paramé
A véletlenobje
NextDouble a k
míg a *NextDou*
bemutat.

Általában igaz, hogy grafikus alkalmazások során a beírt adataink szövegesek, így minden esetben az azokkal történő számolás előtt konvertálnunk kell, ezért a konvertáló függvények használata elég gyakori.

Hasonló konvertáló szolgáltatásokat kapunk, ha az alaptípusok *Parse* függvényét használjuk (*int.Parse(szöveg)*, *double.Parse(szöveg)*).

Gyakran előfordul, hogy az alapl műveletek nem elégítik ki a számolási igényeinket. A *System* névtér *Math* osztálya a leggyakrabban használt számolási műveleteket biztosítja. A leggyakrabban használt függvények a következők:

<code>Math.Sin(x)</code>	<code>sin(x)</code> , ahol az <code>x</code> szög értékét radiánban kell megadni
<code>Math.Cos(x)</code>	<code>cos(x)</code>
<code>Math.Tan(x)</code>	<code>tg(x)</code>
<code>Math.Exp(x)</code>	<code>ex</code>
<code>Math.Log(x)</code>	<code>ln(x)</code>
<code>Math.Sqrt(x)</code>	<code>x</code> négyzetgyöke
<code>Math.Abs(x)</code>	<code>x</code> abszolút értéke
<code>Math.Round(x)</code>	kerekítés a matematikai szabályok szerint
<code>Math.Ceiling(x)</code>	felfelé kerekítés
<code>Math.Floor(x)</code>	lefelé kerekítés
<code>Math.Pow(x,y)</code>	hatványozás, <code>xy</code>
<code>Math.PI</code>	a <code>PI</code> konstans (3.14159265358979323846)
<code>Math.E</code>	az <code>e</code> konstans (2.7182818284590452354)

Példa:

```
...
double dd=Math.Sin(Math.PI/2);
Console.WriteLine(dd);    // értéke 1.
dd=Math.Pow(2,3);
Console.WriteLine(dd);    // 8
```

Matematikai, statisztikai feladatoknál a véletlenszámok használata gyakori igény. A *System* névtér *Random* osztálya nyújtja a pszeudo-véletlenszámok generálásának lehetőségét. Egy véletlenszám-objektum létrehozását a rendszer-időhöz (paraméter nélküli konstruktor) vagy egy adott egész számhoz köthetjük. A véletlenobjektum *Next* függvénye a következő véletlen egész számot, a *NextDouble* a következő valós véletlenszámot adja. A *Next* függvénynek három, míg a *NextDouble* függvénynek egy változata van, amit a következő példa is bemutat.

III. Kifejezések, műveletek

Példa:

```
Random r=new Random();  
//r véletlenszám objektum rendszeridő alapú létrehozása  
Random r1=new Random(10);  
// r1 véletlenobjektum generátor a 10 értékből indul ki  
//  
int v= r.Next();  
// véletlen egész szám 0 és MaxInt (legnagyobb egész) között  
//0 lehet az érték, MaxInt nem  
//  
int v1=r.Next(10);  
// véletlen egész szám 0 és 10 között, 0<=v1<10  
//  
int v2=r.Next(10,100);  
// véletlen egész szám 10 és 100 között, 10<=v2<100  
//  
double d=r.NextDouble();  
// véletlen valós szám 0 és 1 között, 0<=d<1
```

III.5. Feladatok

1. Mit nevezünk kétoperandusú operátornak?
2. Melyik operátor három operandusú?
3. Milyen szám-szöveg konverziós lehetőségeket ismer?
4. Fogalmazza meg két elem közül a nagyobb kiválasztását *operator* segítségével!
5. Ismerjük egy háromszög két oldalát és közbezárt szögét. Számoljuk ki a szöggel szemközti oldal hosszát!

IV. Össz

IV.1. Tömböl

A feladataink szeretnénk használni, hogy két vagy három szükségünk, akkor nem is kivitelezhető összekapcsolásáról az adatszerkezetek memóriában sorolva annak szinonímái.

A C# nyelv a tömbből származtatja. A tömbök defini...

Az egyes tömbök kezdődik. Az index...

Egy tömb általános nyelv környezetében jelenti. A vektor referencia típusú (példányosítani). Minden vektornak az elemszáma a L...

Példa:

```
int[] numk  
numbers =  
...  
numbers =
```

Mivel a nyelv szabványosítja a refer...

IV. Összetett adattípusok

IV.1. Tömbök

A feladataink során gyakori igény az, hogy valamilyen típusú elemből többet szeretnénk használni. Amíg ez a „több” kettő vagy három, addig megoldás lehet, hogy két vagy három változót használunk. Amikor viszont tíz, húsz adatra van szükségünk, akkor ez a fajta megoldás nem igazán kényelmes, és sok esetben nem is kivitelezhető. Lehetőség van azonos típusú adatok egy közös névvel való összekapcsolására, és az egyes elemekre index segítségével hivatkozhatunk. Ezt az adatszerkezetet tömbnek nevezzük. A tömbök elemeinek elhelyezkedése a memóriában sorfolytonos. A tömb helyett gyakran használjuk a vektor szót, annak szinonímjaként.

A C# nyelv minden tömb- vagy vektortípus definíciót a *System.Array* osztályból származtat, így annak tulajdonságai a meghatározóak.

A tömbök definíciójának formája a következő:

típus[] név;

Az egyes tömbelemekre való hivatkozás, a tömbök indexelése mindig 0-val kezdődik. Az index egész típusú kifejezés lehet, a típus pedig tetszőleges.

Egy tömb általános definíciója nem tartalmazza az elemszám értékét. Bár a nyelv környezetében nem lehet mutatókat definiálni, de ez gyakorlatilag azt jelenti. A vektor definíciója egy referencia típus létrehozása, és a nyelv minden referencia típusát a *new* operátorral kell konkrét értékekkel inicializálni (példányosítani). Ekkor kell megmondani, hogy az adott vektor hány elemű lesz. Minden vektornak, ellentétben a C++ nyelvvel, a létrehozása után lekérdezhető az elemszáma a *Length* tulajdonsággal.

Példa:

```
int[] numbers;           // egész vektordefiníció
numbers = new int[10];    // 10 elemű lesz az egész vektorunk
...
numbers = new int[20];    // most meg 20 elemű lett
```

Mivel a nyelv minden dinamikus referencia típusának memóriabeli felszabadítását a rendszer automatikusan végzi – ezt az irodalom gyakran szemét-

IV. Összetett adattípusok

gyűjtési algoritmusnak (*garbage collection*) nevezi –, ezért a példabeli 10 elemű vektor memóiahelyét automatikusan visszakapja az operációs rendszer.

A vektorelemek a fenti dinamikus létrehozás után 0 kezdőértéket kapnak. Ha egyéb elemekkel szeretnénk az inicializációt elvégezni, kapcsos zárójelek közé írva adhatjuk meg azokat.

típus[] név={érték, érték,...};

Példa:

```
int[] n={3,4,5,6};  
int hossz= n.Length; // vektorhossz meghatározás
```

Ha logikai vektort definiálunk, akkor a vektorelemek kezdőértékadás hiányában logikai hamis (*false*), míg ha referencia vektort definiálunk, akkor a referenciaelemek a *null* kezdőértéket kapják meg.

Példa:

```
int[] numbers = {1, 2, 3, 4, 5};  
int[] numbers = new int[5] {1, 2, 3, 4, 5};  
string[] nevek = new string[3] {"Ali", "Pali", "Robi"};  
int[] numbers = new int[] {1, 2, 3, 4, 5};  
string[] nevek = new string[] {"Ali", "Pali", "Robi"};
```

A C# nyelvben nem csak a fenti egydimenziós vektor definiálható. Ezenkívül még definiálható két- vagy többindexű, multidimenziós vektor, illetve vektorok vektora.

Multidimenziós vektor:

Példa:

```
string[,] nevek;  
...  
nevek= new string[2,4];
```

A vektor dimenzióját a *Rank* tulajdonsággal kérdezhetjük le.

```
Console.WriteLine(nevek.Rank); // 2
```

Természetesen a normál vektornak (*int[] v*) is lekérdezhetjük a dimenzió értékét.

Példa:

```
int[,] num;  
string[,] k  
int[,] num  
string[,] }  
int[,] szá  
string[,] }
```

Használat:

```
numbers[1,
```

Vektorok vektor

A más nyelv sorban azonos a definíciónak fele definíció, ahol ezután egyenként jelennek.

Példa:

```
byte[][] i  
for (int :  
{  
    meres  
}  
  
int[][] s  
{  
    new i  
    // ez  
    new i  
    // ez  
};  
  
int[][] n  
new int[]  
int[][] n
```

Használat:

```
numbers[1
```

Természetesen csak kétdimenzió

zi –, ezért a példabeli 10 elemű az operációs rendszer. itán 0 kezdőértéket kapnak. Ha égezni, kapcsos zárójelek közé

ék,...);

ssz meghatározás

toelemek kezdőértékadás hiá-
a vektort definiálunk, akkor a

```
, 5};
"Pali", "Robi"};
5};
"Pali", "Robi"};
```

iók vektor definiálható. Ezen-
tidimenziós vektor, illetve vek-

érdezhetjük le.

2

is lekérdezhetjük a dimenzió

Példa:

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] kapocs = new string[2, 2] { {"Miki", "Ani"}, {"Mari", "Albert"} };
int[,] numbers = new int[, ] { {1, 2}, {3, 4}, {5, 6} };
string[,] kapocs = new string[, ] { {"Miki", "Ani"}, {"Mari", "Albert"} };
int[,] számok = { {1, 2}, {3, 4}, {5, 6} };
string[,] kapocs = { {"Miki", "Ani"}, {"Mari", "Albert"} };
```

Használat:

```
numbers[1,1] = 667;
```

Vektorok vektora:

A más nyelvekbeli analógiát tekintve, a multidimenziós vektorban minden sorban azonos darabszámú elem van, ez tehát a szabályos, négyzetes mátrix stb. definíciónak felel meg. A vektorok vektora pedig valójában egy mutató vektor–definíció, ahol először megmondjuk, hogy hány elemű a mutató vektor, majd ezután egyenként meghatározzuk, hogy az egyes elemek milyen vektorokat jelentenek.

Példa:

```
byte[][] meres = new byte[5][];
for (int x = 0; x < meres.Length; x++)
{
    meres[x] = new byte[4];
}

int[][] számok = new int[2][]
{
    new int[] {3,2,4},
    // ez a három elemű vektor lesz a számok első vektora
    new int[] {5,6}
    // ez a két elemű vektor lesz a számok második vektora
};

int[][] numbers =
new int[][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
int[][] numbers = { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

Használat:

```
numbers[1][1] = 5;
```

Természetesen mindkét esetben – multidimenziós, vektorok vektora – nem-csak kétdimenziós esetről beszélhetünk, hanem tetszőleges méretű vektorról.

IV. Összetett adattípusok

Példa:

```
int[, ,] három = new int[4,5,3];
```

Lehetőségünk van a kétféle vektordefiníció kevert használatára is. A következő példa egy olyan egyszerű vektort definiál, aminek minden eleme 3x2 dimenziós szabályos mátrix.

Példa:

```
int[[],[],[]] mix;
```

Ezek után írjunk egy példaprogramot, amely bemutatja a korábban megbeszélt vektorjellemzők használatát:

Példa:

```
//vektor.cs
using System;

class vektorpelda
{
    public static void Main()
    {
        // Sima vektordefiníció, a definíció pillanatában
        // 5 elemű vektor lesz
        // A vektorelemeknek külön nem adtunk kezdőértéket,
        // ezért azok 0 értéket kapnak.
        int[] numbers = new int[5];

        // Kétdimenziós vektor, sima 5x4-es szöveges,
        //ezen elemek inicializálás hiányában null értéket kapnak.
        string[,] names = new string[5,4];

        // Vektorok vektora, ezt az irodalom gyakran
        // "jagged array", (csipkés, szaggatott vektor) néven említi
        byte[][] scores = new byte[5][];

        // Az egyes vektorok definiálása
        for (int i = 0; i < scores.Length; i++)
        {
            scores[i] = new byte[i+3]; // elemről elemre nő az elemszám.
        }
        // Egyes sorok hosszának kiírása
        for (int i = 0; i < scores.Length; i++)
        {
            Console.WriteLine("Az {0} sor hossza: {1}", i, scores[i].Length);
        }
    }
}
```

A program futása u

```
Az 0 sor hossza
Az 1 sor hossza
Az 2 sor hossza
Az 3 sor hossza
Az 4 sor hossza
```

Példa:

```
string honapnev
{
    string[] né
    "
    "
    "
    "
    "
    return((n
}
```

Az iménti példa r
nevét.

IV.2. Struktúra

Gyakran szükségü
leírni kívánt adatunka
ha a talán legkézenfek
megadni? Egy lehets
szögű koordinátarend
dinátákat.

A struktúradefiníci
hogy az összetartozó a
A struktúradefiníci

```
struct
```

Hivatkozás egy m

A program futása után a következő eredményt kapjuk:

Az 0 sor hossza: 3
 Az 1 sor hossza: 4
 Az 2 sor hossza: 5
 Az 3 sor hossza: 6
 Az 4 sor hossza: 7

Példa:

```
string honapnev(int n)
{
    string[] név={"Nem létezik", "Január",
                 "Február", "Március",
                 "Április", "Május", "Június",
                 "Július", "Augusztus",
                 "Szeptember", "Október",
                 "November", "December"};
    return((n<1)|| (n>12)) ? név[0] : név[n];
}
```

Az iménti példa meghatározza egy tetszőleges sorszámú hónaphoz annak nevét.

IV.2. Struktúra

Gyakran szükségünk van az eddigiektől eltérő adatszerkezetekre, amikor a leírni kívánt adatunkat nem tudjuk egy egyszerű változóval jellemezni. Elég, ha a talán legkézenfekvőbb feladatot tekintjük: hogyan tudjuk a sík egy pontját megadni? Egy lehetséges megadási mód, ha a pontot mint a sík egy derékszögű koordináta-rendszerének pontját tekintem, és megadom az X és Y koordinátákat.

A struktúra-definiálás az ilyen feladatok megoldása esetén lehetővé teszi, hogy az összetartozó adatokat egy egységként tudjuk kezelni.

A struktúra-definíció formája a következő:

```
struct név {
    hozzáférés típus mezőnevek;
    ...
    hozzáférés függvénydefiníció ;
};
```

Hivatkozás egy mezőre: *változónév.mezőnév*

IV. Összetett adattípusok

A struktúrákon azonos típus esetén az értékadás elvégezhető, így ha például a és b azonos típusú struktúra, akkor az $a=b$ értékadás szabályos, és az a minden mezője felveszi b megfelelő mezőértékeit.

Példa:

```
struct pont
{
    int x;
    int y;
};
...
pont a;
...
struct egy
{
    string name;
    int kor;
};
egy[] c=new egy[10]; // 10 elemű struktúra vektor
```

A fenti definíció teljesen jó, csak a hozzáférési szint megadásának hiányában minden mező privát, azaz a struktúra mindkét adata csak belülről látható. A struktúra alapértelmezett mezőhozzáférése privát (*private*). Minden taghoz külön meg kell adni a hozzáférési szintet.

Struktúra esetén a megengedett hozzáférési szintek:

private	csak struktúrán belülről érhető el
public	bárki elérheti ezt a mezőt
internal	programon belülről (assembly) elérhető

Példa:

```
struct személy
{
    public string név;
    public int kor;
};
```

Az egyes mezőkre hivatkozás formája:

Példa:

```
személy st=new személy();
System.WriteLine( st.kor); // fizetésiírás
```

Struktúra-definiálást is meg lehet csinálni a struktúrának, köztük definiálható, azt lehet használni és nem a struktúra magját tekintve megmarad. A struktúrák donságot:

- Egy struktúra

Példa:

```
struct al
{
    string strir;
    ...
}
```

- Struktúra a beállítás módokkal

Példa:

```
struct po
{
    public:
}
pont p=new
p.x=2; p.y=3;
```

- Bár a struktúra konstruktor elhelyezése Befejezésül zünk egy teljes

Példa:

```
using System;
struct st
{
    public:
    public:
}
```

lás elvégezhető, így ha például
adás szabályos, és az *a* minden

Struktúra definíció esetén a nyelv nem csak adatmező, hanem függvénymező definiálást is megenged. Azt a függvénymezőt, aminek ugyanaz a neve, mint a struktúrának, konstruktornak nevezzük. Paraméter nélküli konstruktor nem definiálható, azt mindig a környezet biztosítja. A struktúra értéktípusú adat, így a vermen és nem a dinamikus memóriában jön létre. Általában elmondható, hogy a struktúra majdnem úgy viselkedik, mint egy osztály, de funkcionalitását tekintve megmarad a különböző típusú adatok tárolásánál.

A struktúrákkal kapcsolatosan érdemes megjegyezni három alapvető tulajdonságot:

- Egy struktúra adatmezőt a definíálás pillanatában nem inicializálhatunk.

Példa:

```
struct alma
{
    string nev="jonatán"; // fordítási hiba
    ...
}
```

- Struktúra adatmezőket a *default* konstruktor nem inicializál. A kezdőérték beállításáról, ha szükséges, saját konstruktorral vagy egyéb beállítási móddal kell gondoskodni.

Példa:

```
struct pont
{
    public int x,y;
}
pont p=new pont();
p.x=2; p.y=4;
```

- Bár a struktúra érték típusú, azért a *new* operátorral kell biztosítani a saját konstruktorral történő inicializációt. Ez ebben az esetben a vermen fog elhelyezkedni.

Befejezésül a struktúra definiálására, használatára, struktúra vektorra nézzünk egy teljesebb példát:

Példa:

```
using System;
struct struktúra_példa
{
    public int kor;
    public string név;
}
```


V. Utasítások

A programvezérlés menetét az utasítások szekvenciája szabja meg. Ahogy korábban is láttuk, az operációs rendszer a *Main* függvénynek adja át a vezérlést, majd a függvénytörzs egymás után következő utasításait (szekvencia) hajtja végre, ezután tér vissza a vezérlés az operációs rendszerhez.

Az utasítások fajtái:

- összetett utasítás
- kifejezés utasítás
- elágazás utasítás
- ciklus utasítás
- ugró utasítás

V.1. Összetett utasítás

Ahol utasítás elhelyezhető, ott szerepelhet összetett utasítás is.

Az összetett utasítás vagy blokk a `{ }` zárójelek között felsorolt utasítások listája. Blokkon belül változók is deklarálhatók, amelyek a blokkban lokálisak lesznek. Blokkok tetszőlegesen egymásba ágyazhatók.

V.2. Kifejezés utasítás

Az utasítás formája:

kifejezés ;

Az utasítás speciális formája az, amikor a kifejezés elmarad. Az ilyen utasítást `(;)` üres vagy nulla utasításnak nevezzük. Ennek természetesen általában nagyon sok értelme nincs. Az alábbi példa egy változó értékét növeli egyesével egy ciklusban.

Példa:

```
int k=5;
for(i=0; i<10; i++)
    k++;
```

V. Utasítások

V.3. Elágazás utasítás

Az elágazások két típusa használható, a kétfelé és a sokfelé elágazó utasítás. A kétfelé elágazó utasítás formája:

```
if (kifejezés) utasítás;  
if (kifejezés) utasítás; else utasítás;
```

Először a kifejezés kiértékelődik, majd annak igaz értéke esetén a kifejezés utáni utasítás, hamis értéke esetén az *else* – ha van – utáni utasítás hajtódik végre.

Egymásba ágyazás esetén az *else* ágat a legutóbbi *else* nélküli *if*hez tartozónak tekintjük.

Példa:

```
if (c=='a')  
    Console.WriteLine("Ez az a betű");  
else  
    Console.WriteLine("Nem az a betű");
```

Ha az igaz ágon összetett utasítást használunk, akkor utána pontosvessző nem kell, illetve ha mégis van, az az *if* lezárását jelentené, és hibás lenne az utasításunk az *if* nélküli *else* használata miatt.

Példa:

```
if (c=='a')  
    { Console.WriteLine("Ez az a betű");}  
else  
    Console.WriteLine("Nem az a betű");
```

A többirányú elágazás utasítása, a *switch* utasítás formája:

```
switch (kifejezés) {  
    case érték1: utasítás1 ;  
    case érték2: utasítás2 ;  
    ...  
    default: utasítás ;  
};
```

A *switch* utáni *break* is. Ha a kifejezés értéke a következő *case* után a kifejezés értékével nem egyezik meg, akkor fordítási hiba.

A C++ nyelvet is lehet vezetni.

Minden elágazást átadással befejezni!

Példa:

```
switch (para)  
{  
    case "r":  
        Run  
        break;  
    case "s":  
        Sav  
        break;  
    case "q":  
        Qui  
        break;  
    default:  
        Ros  
        break;  
}
```

A *case* belépési érték megadása nem kötelező, a címké szerepét tölti ki, akkor két címkét lehet használni.

Példa:

```
...  
switch (a)  
{  
    case 1:  
    case 2:  
        Console.WriteLine("Ez az a betű");  
        break;  
    default: Console.WriteLine("Nem az a betű");  
};  
...
```


é és a sokfelé elágazó utasítás.

utasítás;

igaz értéke esetén a kifejezés
van – utáni utasítás hajtódik

többi *else* nélküli *if*hez tartozó-

nk, akkor utána pontosvessző
át jelentené, és hibás lenne az

");}

);

ítás formája:

tás1 ;
tás2 ;

A *switch* utáni kifejezés nem csak egész értékű lehet, hanem szöveg (*string*) is. Ha a kifejezés értéke a *case* után megadott értékkel nem egyezik meg, akkor a következő *case* utáni érték vizsgálata következik. Ha egy *case* utáni érték azonos a kifejezés értékével, akkor az ez utáni utasítás végrehajtódik. Ha egy *case* ágban nincs egyértelmű utasítás arra vonatkozóan, hogy hol folytatódjon a vezérlés, akkor fordítási hibát kapunk.

A C++ nyelvet ismerők tudhatják, hogy abban a nyelvben egy *case* ág végét nem kellett vezérlésátadással befejezni, és ebből gyakran adódtak hibák.

Minden elágazáságot, még a *default* ágot is, kötelező valamilyen vezérlésátadással befejezni! (*break*, *goto*, *return*...)

Példa:

```
switch (parancs)
{
    case "run":
        Run();
        break;
    case "save":
        Save();
        break;
    case "quit":
        Quit();
        break;
    default:
        Rossz(parancs);
        break;
}
```

A *case* belépési pont után csak egy érték adható meg, intervallum vagy több érték megadása nem megengedett, hiszen ezek az 'értékek' gyakorlatilag egy címke szerepét töltik be. Ha két értékhez rendeljük ugyanazt a tevékenységet, akkor két címkét kell definiálni.

Példa:

```
...
switch (a)
{
    case 1:
    case 2:
        Console.WriteLine("Egy és kettő esetén...");
        break;
    default: Console.WriteLine("Egyébként...");
        break;
};
...
```

V. Utasítások

V.4. Ciklus utasítás

A ciklus utasításnak négy formája alkalmazható a C# nyelvben, ezek a *while*, a *do*, a *for* és a *foreach* ciklus utasítások.

V.4.1. „while” utasítás

A *while* ciklus utasítás formája:

while (kifejezés) utasítás;

Először a zárójelben lévő kifejezés kiértékelődik, ha értéke igaz, akkor a zárójeles kifejezést követő utasítás – amit szokás ciklusmagnak nevezni –, végrehajtódik. Ezután újból a kifejezés kiértékelése következik, igaz érték esetén pedig a ciklusmag végrehajtása. Ez az ismétlés addig folytatódik, amíg a kifejezés hamis értéket nem ad.

Mivel az utasítás először kiértékeli a kifejezést és csak utána hajtja végre a ciklusmagot (ha a kifejezés igaz értéket adott), ezért a *while* ciklus utasítást előltesztelő ciklusnak is szokás nevezni.

Példa:

```
while(true)
{
    Console.WriteLine("Végtelen ciklus!");
    Console.WriteLine("Ilyet ne nagyon használj!");
}
...
static void Main()           // betűnkénti kiírás
{
    string szöveg[]="Szöveg!";
    int i=0;
    while(i<szöveg.Length)    // a string végéig
    {
        Console.WriteLine(szöveg[i]);
        i++;
    }
}
```

A második példában a ciklusmagban használhattam volna egy utasítást is, kihasználva a ++ operátor postfix alakját az alábbi módon:

```
while(i<szöveg.Length)
    Console.WriteLine(szöveg[i++]);
```

V.4.2. „do” utas

Az utasítás for

A ciklusmag kifejezést, és ami hajtását. A *do* cikl

Példa:

```
string név
do
{
    Console.
    név=Cx
}
while (név
Console.Wr

do
    Console.
while(fals
```

V.4.3. „for” uta

Az utasítás fo

Ez az utasítás

kifejezés1
while (kif

Mindegyik k
(true)-ként tekinti
A kifejezés1
ható forráskódbai

V.4.2. „do” utasítás

Az utasítás formája:

do utasítás; while (kifejezés) ;

A ciklusmag – a *do* és a *while* közötti rész – végrehajtása után kiértékeli a kifejezést, és amíg a kifejezés igaz értéket ad, megismétli a ciklusmag végrehajtását. A *do* ciklust szokás hátultesztelő ciklusnak is nevezni.

Példa:

```
string név;
do
{
    Console.WriteLine("Ki vagy?");
    név=Console.ReadLine();
}
while (név!="Zoli");
Console.WriteLine("Szia {0}!",név);

do
    Console.WriteLine("Biztos egyszer lefut!");
while(false);
```

V.4.3. „for” utasítás

Az utasítás formája:

for (kifejezés1; kifejezés2; kifejezés3) utasítás;

Ez az utasítás egyenértékű a következő alakkal:

```
kifejezés1;
while (kifejezés2) {
    utasítás;
    kifejezés3;
};
```

Mindegyik kifejezés elmaradhat. Ha *kifejezés2* is elmarad, akkor *while (true)*-ként tekinti a ciklust. A kifejezések közötti pontosvessző nem maradhat el.

A *kifejezés1* típusdefiníciós kifejezés utasítás is lehet. Nagyon gyakran látható forráskódban az alábbi forma:

V. Utasítások

Példa:

```
for (int i=0; i<10; i++)
    Console.WriteLine("Hajrá Fradi!");
```

Ez a fajta ciklus használat felel meg például a *Basic For...Next* ciklusának.

V.4.4. „foreach” utasítás

Az utasítás formája:

foreach (azonosító in vektor) utasítás:

Ez az utasítás egyenértékű a vektor-elemeken történő végiglépdeléssel. Az azonosító, mint ciklusváltozó felveszi egyenként a vektor elemeit.

Példa:

```
public static void Main()
{
    int paros = 0, paratlan = 0;
    int[] v = new int [] {0,1,2,5,7,8,11};
    foreach (int i in v)
    {
        if (i%2 == 0)
            paros++;
        else
            paratlan++;
    }
    Console.WriteLine("Találtam {0} páros, és {1} páratlan  
számot.",paros, paratlan);
}
```

A *foreach* ciklus az általunk definiált vektorokon kívül az ehhez hasonló könyvtári adatszerkezeteken is (*ArrayList*, *Queue* stb.) jól használható. Ezzel kapcsolatos példát a *Helyi könyvtárak használata* fejezetben láthatunk.

A *foreach* utasítás nemcsak közvetlen vektortípuson, mint például a fenti *v* vektoron alkalmazható, hanem minden olyan „végigjárható” típuson, amely az alábbi feltételeknek megfelel:

- A típus rendelkezzen egy *GetEnumerator()* függvénnyel, aminek eredményül kell adni azt a típust, amilyen típusú elemeken lépdelünk végig.
- Definiáljunk egy *MoveNext()* függvényt, amelyik az indexet aktuálisra állítja, és logikai visszatéréssel megadja, hogy van-e még maradék elem.
- Definiáljunk egy *Current* tulajdonságot, ami az aktuális indexű elemét adja eredményül.

Példa:

```
using System;
public class Program
{
    int[] elements;

    public Program(int[] elements)
    {
        this.elements = elements;
    }

    public void Print()
    {
        for (int i = 0; i < elements.Length; i++)
        {
            Console.WriteLine(elements[i]);
        }
    }
}

public class ProgramTest
{
    public static void Main()
    {
        int[] elements = { 1, 2, 3, 4, 5 };
        Program program = new Program(elements);
        program.Print();
    }
}
```

Példa:

```

using System;
public class adatsor
{
    int[] elemek;

    public adatsor()
    {
        elemek = new int[5] {12, 44, 33, 2, 50};
    }

    public vektor GetEnumerator()
    {
        return new vektor(this);
    }

    // Az "enumerator", class definiálás:
    public class vektor
    {
        int Index;
        adatsor a;
        public vektor(adatsor ad)
        {
            a = ad;
            Index = -1;
        }

        public bool MoveNext()
        {
            Index++;
            return (Index < a.elemek.GetLength(0));
        }

        public int Current
        {
            get
            {
                return a.elemek[Index];
            }
        }
    }
}

public class MainClass
{
    public static void Main()
    {
        adatsor adatok = new adatsor();
    }
}

```


V. Utasítások

```
        Console.WriteLine("Az adatsor elemei:");

        // elemek kiírása
        foreach (int i in adatok)
        {
            Console.WriteLine(i);
        }
    }
}

/*Eredmény:
12
44
33
2
50*/
```

V.5. Ugró utasítások

V.5.1. „break” utasítás

Az utasítás formája:

```
break;
```

Hatására befejeződik a legbelső *while*, *do*, *for* vagy *switch* utasítás végrehajtása. A vezérlés a következő utasításra adódik.

Példa:

```
int i=1;
while(true)                // látszólag végtelen ciklus
{
    i++;
    if (i==11) break;       // Ciklus vége
    Console.WriteLine( i);
}
```

A *break* a többirányú elágazás (*switch*) utasításban is gyakran használt, így kerülhetjük el, hogy a nem kívánt *case* ágak végrehajtódjanak.

V.5.2. „continue” utasítás

Az utasítás formája:

```
continue;
```

Hatására a legkisebb értékű elem kerül kiírásra.

Példa:

```
int i=1;
while(i<10)
{
    i++;
    if (i%2==0)
        continue;
    Console.WriteLine(i);
}
```

A fenti példában a páros számok nem kerülnek kiírásra.

V.5.3. „return” utasítás

Az utasítás formája:

A vezérlés visszatér a hívóhoz.

V.5.4. „goto” utasítás

Az utasítás formája:

A vezérlés arra a címről, ahol a *goto* utasítás szerepel, folytatódik.

Példa:

```
goto tovább;
Console.WriteLine(i);
...
int i=1;
switch (i)
{
    case 0:
        goto tovább;
    case 1:
        Console.WriteLine(i);
        break;
}
```

```
elemek:");
```

Hatására a legbelső *while*, *for*, *do* ciklus utasításokat vezérlő kifejezések kerülnek kiértékelésre. (A ciklus a következő ciklusmag végrehajtásához készül.)

Példa:

```
int i=1;
while(true)          // 10 elemű ciklus
{
    i++;
    if (i<10) continue; // következő ciklusmag

    if (i==11) break; // Ciklus vége
    Console.WriteLine( i);
}
```

A fenti példában a *continue* utasítás miatt a *Console.WriteLine(i)* utasítás egyszer sem hajtódik végre.

V.5.3. „return” utasítás

Az utasítás formája:

```
return ;
return kifejezés;
return (kifejezés);
```

A vezérlés visszatér a függvényből, a kifejezés értéke a visszaadott érték.

V.5.4. „goto” utasítás

Az utasítás formája:

```
goto címke;
```

A vezérlés arra a pontra adódik, ahol a *címke*: található.

Példa:

```
goto tovább;
Console.WriteLine("Ezt a szöveget sohase írja ki!");
tovább;

...
int i=1;
switch (i)
{
    case 0:
        nulla();
```

V. Utasítások

```
        goto case 1;
case 1:
    egy();
    goto default;
default:
    valami();
    break;
}
```

A *goto* utasításról záráskeppen meg kell jegyezni, hogy a strukturált programkészítésnek nem feltétlenül része ez az utasítás, így használata sem javasolt. A könyv későbbi példaprogramjaiban sem fordul elő egyszer sem ez az utasítás.

V.5.5. „using” utasítás

A *using* kulcsszó kétféle nyelvi környezetben szerepelhet. Egyrészt ún. direktíva, könyvtári elemek, adott névtér, osztály használataként. Ennek formája:

using névtér_vagy_osztály;

Példa:

```
using System; // a keretrendszer fő névtérének használata
// a C++ #include-hoz hasonlóan megmondja a
// fordítónak, hogy ennek a névtérnek a típusait is
// használja. Ezen típusokat azért nélkül is teljes
// névvel (System.Console...) használhatjuk
```

A másik eset a *using* utasítás. Egy ideiglenesen használt objektum esetén a *using* utasítás után a keretrendszer automatikusan meghívja az objektum *Dispose* metódusát. A *using* utasítás alakja a következő:

using (objektum) { utasítások; }

Példa:

```
using (Objektumom o = new Objektumom())
{
    o.ezt_csinald();
}
```

Ez a használata
ket később ismert

```
...
Objektumom
try
{
    {o.ez
finally
{
    if (o
}
```

A *Dispose* uti
részletezzük.

V.5.6. „lock” ut

Ha egy kriti
van szükség a biz
jük.

A *lock* kulcs
következik a kriti

Példa:

```
lock(this)
{
    a=5;
}
```

A *lock* utasítá
ha a védett változ

Ha statikus
(lockolni), hogy
lock referenciánál

Példa:

```
class adat
{
    stati
    publi
```

Ez a használat az alábbi kódrészletnek felel meg, a nem ismert nyelvi elemeket később ismertetjük:

```
...
Objektumom o = new Objektumom();
try
    {o.ezt_csinald();}
finally
{
    if (o != null) ((IDisposable)o).Dispose();
}
```

A *Dispose* utasítást és környezetét bővebben a destruktorokkal kapcsolatban részletezzük.

V.5.6. „lock” utasítás

Ha egy kritikus utasítás blokk végrehajtásakor egy referencia blokkolására van szükség a biztonságos végrehajtáshoz, akkor ezt a *lock* utasítással megtehetjük.

A *lock* kulcsszó egy referenciát vár, ez jellemzően a *this*, majd utána következik a kritikus blokk. Ennek formája:

lock(ref) utasítás;

Példa:

```
lock(this)
{
    a=5; // biztonságos
}
```

A *lock* utasításnak, ahogy láttuk, a leggyakrabban használt paramétere a *this*, ha a védett változó vagy függvényutasítás nem statikus. (Osztálypéldányok.)

Ha statikus változókat vagy függvényutasításokat szeretnénk biztosítani (lockolni), hogy egyszerre csak egy végrehajtási szál tudjon hozzáférni, akkor a *lock* referenciának az osztály típusát kell megadni.

Példa:

```
class adatok
{
    static int szamlalo=0;
    public void mentes(string s)
```

V. Utasítások

```
{
    lock(typeof(adatok))
    {
        szamlalo++;
        Console.WriteLine("Adatmentés elindul!");
        for(int i=0;i<50;i++)
        {
            Thread.Sleep(1);
            Console.Write(s);
        }
        Console.WriteLine("");
        Console.WriteLine("Adatmentés {0}.alkalommal
                           befejeződött!",szamlalo);
    }
}
```

A példa bővebb magyarázata, teljes környezetbeli alkalmazása a IX. rész *Párhuzamos programvégrehajtás* fejezetében található.

V.6. Feladatok

1. Milyen típusú adat szerint készíthetünk többirányú (*switch*) elágazást?
2. Milyen előtesztelő és hátulatesztelő ciklusokat használhatunk?
3. Mire használható a *break* utasítás?
4. Írjon rövid programot, amely beolvassa egy focicsapat, adott fordulóban szerzett pontszámát (0,1,3) egy egész típusú változóba, majd felhasználva a *switch* utasítást a beolvasott érték alapján kiírja a következő szövegeket: *győzelem, döntetlen, vereség, hibás adat*.
5. Írjon programot, amelyik beolvas egy kettővel osztható, 10 és 100 közé eső egész számot! (Ha rossz értéket adnak meg, akkor addig folytassa a beolvasást, amíg a feltételeknek megfelelő számot nem sikerül megadni!)

VI. Függvények

VI.1. A függvény

Ha egy függvény különböző esettről kerülhet átadásra, másolata kerül a másolatra.

VI.1.1 Érték szerinti átadás

Általánosan ez az átadási módszer, a függvény paraméterei a hívóérték másolatai.

Tekintsük meg, hogy a kapott két

Példa:

```
class maxf
{
    static
    {
        r
    }
    static
    {
        C
    }
}
```

Az így megnevezzük, a *maxi* paramétert kapja érték is megadhat. A függvény hív meg. Szokászik meg a klasszikus

VI. Függvények

VI.1. A függvények paraméterátadása

Ha egy függvénynek adatot adunk át paraméterként, akkor alapvetően két különböző esetről beszélhetünk. Egy adat érték szerint és hivatkozás szerint kerülhet átadásra. Az első esetben valójában az eredeti adatunk értékének egy másolata kerül a függvényhez, míg a második esetben az adat címe kerül átadásra.

VI.1.1 Érték szerinti paraméterátadás

Általánosan elmondhatjuk, hogyha nem jelölünk semmilyen paraméterátadási módszert, akkor érték szerinti paraméterátadással dolgozunk. Ekkor a függvény paraméterében, mint formális paraméterben, a függvény meghívásakor a hívóérték másolata helyezkedik el.

Tekintsük meg a következő maximum nevű függvényt, aminek az a feladata, hogy a kapott két paramétere közül a nagyobbát adja vissza eredményként.

Példa:

```
class maxfv
{
    static int maximum(int x,int y)
    {
        return (x>y?x:y);
    }

    static void Main()
    {
        Console.WriteLine(maximum(4,3));
    }
}
```

Az így megvalósított paraméterátadást érték szerinti paraméterátadásnak nevezzük, a *maximum* függvény két (*x* és *y*) paramétere a híváskor megadott két paramétert kapja értékül. Érték szerinti paraméterátadásnál, híváskor konstans érték is megadható.

A függvény hívásának egy sajátos esete az, mikor egy függvényt saját maga hív meg. Szokás ezt rekurzív függvényhívásnak is nevezni. Erre példaként nézzük meg a klasszikus faktoriálisszámoló függvényt.

VI. Függvények

Példa:

```
class faktor
{
    static int faktorialis(int x)
    {
        return (x<=1?1: (x* faktorialis(x-1)));
    }
    static void Main()
    {
        Console.WriteLine(faktorialis(4));
    }
}
```

VI.1.2. Referencia (cím) szerinti paraméterátadás

Amikor egy függvény paramétere nem a változó értékét, hanem a változó tárolási helyének címét, hivatkozási helyét kapja meg a függvény meghívásakor, akkor a paraméterátadás módját cím szerinti paraméterátadásnak nevezzük.

A cím szerinti paraméterátadást gyakran hivatkozás (*reference*) szerinti paraméterátadásnak is szokás nevezni.

Ha a paraméternév elé a *ref* (hivatkozás) kulcsszót beszurjuk, akkor a hivatkozás szerinti paraméterátadást definiáljuk. Ezt a kulcsszót be kell szúrunk a függvénydefinícióba és a konkrét hívás helyére is!

Példaként írjunk olyan függvényt, ami a kapott két paraméterének értékét felcseréli.

Példa:

```
...
// a ref kulcsszó jelzi, hogy referencia
// paramétereket vár a függvény
void csere(ref int x, ref int y)
{
    int segéd=x;
    x=y; y=segéd;
}
static void Main()
{
    int a=5, b=6;
    Console.WriteLine("Csere előtt: a={0}, b={1}.", a , b);
    // függvényhívás, a ref kulcsszó itt is kötelező
    csere(ref a,ref b);
    Console.WriteLine("Csere után: a={0}, b={1}.", a , b);
}
```

A *csere(a,b)* hívás megcseréli a két paraméter értékét, így az *a* változó értéke 6, míg *b* értéke 5 lesz.

VI.1.3. Függvény

A referencia :
értéke kerül ki a l
a függvény nem k

Az eredmény
paraméter elé ki l
egy függvénynek

Példa:

```
using Syst
public cla
{
    publi
    {
        j
        r
    }
    publi
    {
        c
        c
        c
    }
}
```

Képernyő e
-1
B

VI.1.4. Tömbök

A nyelv a tö
tömb paraméterá
Egy függvény te
az eredmény egy
egyszerű változó
szerinti paraméte

Az elmondot

VI.1.3. Függvényeredmény paramétere

A referencia szerinti paraméterátadáshoz hasonlóan, a függvénybeli változó értéke kerül ki a hívó változóba. A különbség csak az, hogy ebben a változóban a függvény nem kap értéket.

Az eredmény paramétert az *out* kulcsszóval definiálhatjuk. A híváskor is a paraméter elé ki kell írni az *out* jelzőt. A használata akkor lehet hasznos, amikor egy függvénynek egynél több eredményt kell adnia.

Példa:

```
using System;
public class MyClass
{
    public static int TestOut(out char i)
    {
        i = 'b';
        return -1;
    }

    public static void Main()
    {
        char i; // nem kell inicializálni a változót
        Console.WriteLine(TestOut(out i));
        Console.WriteLine(i);
    }
}
```

Képernyő eredmény:

-1

B

VI.1.4. Tömbök paraméterátadása

A nyelv a tömböt annak nevével, mint referenciával azonosítja, ezért egy tömb paraméterátadása nem jelent mást, mint a tömb referenciájának átadását. Egy függvény természetesen tömböt is adhat eredményül, ami azt jelenti, hogy az eredmény egy tömbreferencia lesz. Egy tömb esetében sincs másról szó, mint egyszerű változók esetében, a tömbreferencia – egy referenciaváltozó –, érték szerinti paraméterátadásáról.

Az elmondottak illusztrálására lássuk a következő szematikus példát.

VI. Függvények

Példa:

```
void módosít(int[] vektor)
{
    // a vektort, mint egy referenciát kapjuk paraméterül
    // ez érték szerint kerül átadásra, azaz ez a referencia
    //nem változik, változik viszont a 0. tömbelem, és ez a
    // változás a hívó oldalon is látszik
    vektor[0]=5;
}
```

Ahogy a fenti példán is látható, a vektor paraméterként a referenciájával kerül átadásra. Ez érték szerinti paraméterátadást jelent. Ha egy függvényben a mutatott értéket (vektorelemet) megváltoztatom, akkor a változás a külső, paraméterként átadott vektorban is látható lesz!

A nyelvben a vektor tudja magáról, hogy hány eleme van (*Length*), ezért – ellentétben a C++ nyelvvel –, az elemszámot nem kell átadni.

Tömb esetében is alkalmazhatjuk a referencia vagy *out* paraméter átadásának lehetőségét. Ennek illusztrálására nézzük a következő példákat:

1. példa:

```
using System;
class teszt
{
    static public void feltölt(out int[] vektor)
    {
        // a vektor létrehozása, és inicializálása
        vektor = new int[5] {1, 2, 3, 4, 5};
    }

    static public void Main()
    {
        int[] vektor; // nem inicializáltuk
        // meghívjuk a feltölt függvényt:
        feltölt(out vektor);
        // A vektorelemek kiírása:
        Console.WriteLine("Az elemek:");
        for (int i=0; i < vektor.Length; i++)
            Console.WriteLine(vektor[i]);
    }
}
```

2. példa:

```
using System;
class Reftesz
{
    public static void Main()
    {
        // ...
        // ...
        if (arr != null)
        {
            // ...
            arr = new int[5];
            // ...
        }

        static public void Feltölt(out int[] vektor)
        {
            // ...
            int[] vektor = new int[5];
            // ...
            for (int i=0; i < vektor.Length; i++)
            {
                // ...
            }
        }
    }
}
```

VI.2. A Main fi

A parancsok, pi
paraméter(ek)rel in
paramétereit a képe
méterül kapott fájl t

Ezeket a paran
Amikor elindítunk
átveszi a vezérlést),
a paraméter egy szö
A paramétertömböt

2. példa:

```

using System;
class Refteszt
{
    public static void feltölt(ref int[] arr)
    {
        // Ha hívó fél még még nem készítette el,
        //akkor megteesszük itt.

        if (arr == null)
            arr = new int[10];
        // néhány elem módosítás
        arr[0] = 123;
        arr[4] = 1024;
        // a többi elem értéke marad az eredeti
    }

    static public void Main ()
    {
        // Vektor inicializálása
        int[] vektor = {1,2,3,4,5};

        // Vektor átadása ref, paraméterként:
        feltölt(ref vektor);

        // Kiírás:
        Console.WriteLine("Az elemek:");
        for (int i = 0; i < vektor.Length; i++)
            Console.WriteLine(vektor[i]);
    }
}

```

VI.2. A Main függvény paraméterei

A parancsok, programok indításakor gyakori, hogy a parancsot valamilyen paraméter(ek)rel indítjuk. Ilyen parancs például a DOS *echo* parancsa, amely paramétereit a képernyőre „visszhangozza”, vagy a *type* parancs, mely a paraméterül kapott fájl tartalmát írja a képernyőre.

Ezeket a paramétereket parancssor-argumentumoknak is szokás nevezni. Amikor elindítunk egy programot (a *Main* függvény az operációs rendszertől átveszi a vezérlést), akkor híváskor a *Main* függvénynek egy paramétere van. Ez a paraméter egy szöveges (*string*) tömb, amelynek elemei az egyes paraméterek. A paramétertömböt gyakran *args* névre keresztelik.

VI. Függvények

A parancssor-argumentumok használatára nézzük az imént már említett *echo* parancs egy lehetséges megvalósítását.

Példa:

```
static void Main(string[] args)
{
    int i=0;
    while (i<args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

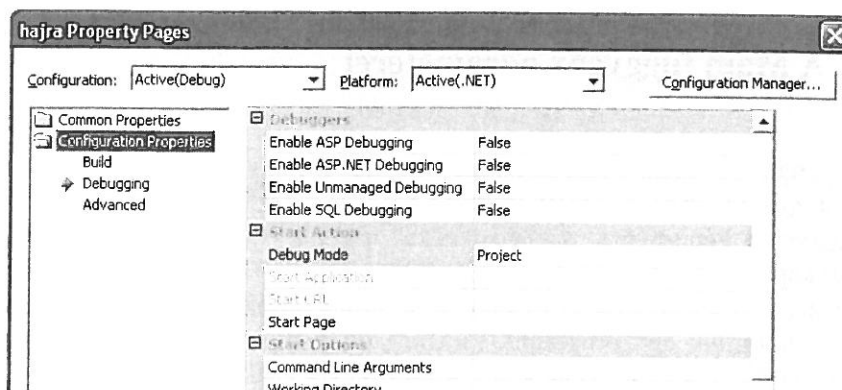
Ha a fenti programot lefordítjuk, és a neve *parancs.exe* lesz, akkor a következőképpen próbálhatjuk ki:

```
c:\parancs.exe fradi vasas újpest
```

A program futtatása után az alábbi eredményt kapjuk:

```
fradi
vasas
újpest
```

A parancssori paraméterek Visual Studio.NET környezetet használva a projekt *Tulajdonság* ablakában is megadhatók: *Configuration Properties \ Debugging \ Command Line Arguments*, ahogy az alábbi képen is látható.



9. ábra

A *Main* függvényt, akarjuk feldolgozni, a p

```
static void Mai
{...}
```

VI.3. Függvények

A feladatmegoldás eldönteni, hány elemet hány paraméterrel kés segítségével egy vektor paramétereit tartalma: Nézzünk néhány pé

Példa:

```
using System;
public class v
{
    public sta
    {
        for (
        (
        Conso
    }

    public sta
    {
        for (
        (
        Consc
    }

    public st
    {
        intPe
        objPe
        int[
        intPe
    }
}
```

nézzük az imént már említett

A *Main* függvényt, abban az esetben, ha a parancssori paramétereket nem akarjuk feldolgozni, a paraméterei nélkül is deklarálhatjuk:

```
static void Main()
{...}
```

VI.3. Függvények változó számú paraméterrel

A feladatmegoldások során gyakran előfordulhat, hogy előre nem tudjuk eldönteni, hány elemet kell a függvénynek feldolgoznia. Tehát nem tudjuk, hogy hány paraméterrel készítsük el a kívánt függvényünket. A *params* kulcsszó segítségével egy vektorparamétert definiálhatunk a függvénynek, ami ezeket a paramétereket tartalmazza majd.

Nézzünk néhány példát ilyen függvény definiálására, majd a használatára.

Példa:

```
using System;
public class váltparaméter
{
    public static void intParaméterek(params int[] list)
    {
        for ( int i = 0 ; i < list.Length ; i++ )
            Console.WriteLine(list[i]);
        Console.WriteLine();
    }

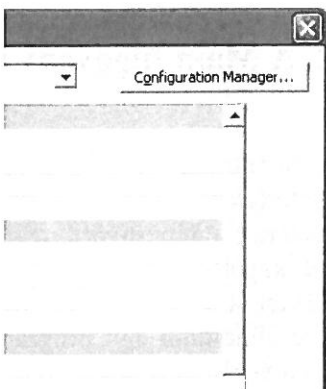
    public static void objParaméterek(params object[] list)
    {
        for ( int i = 0 ; i < list.Length ; i++ )
            Console.WriteLine(list[i]);
        Console.WriteLine();
    }

    public static void Main()
    {
        intParaméterek(1, 2, 3);
        objParaméterek(1, 'fradi', "teszt", 3.5);
        int[] myarray = new int[3] {10,11,12};
        intParaméterek(myarray);
    }
}
```

eve *parancs.exe* lesz, akkor a

t kapjuk:

NET környezetet használva a
k: *Configuration Properties* \
z alábbi képen is látható.



VI. Függvények

Az eredmény a következő lesz:

1
2
3

1
fradi
teszt
3.5

10
11
12

Egy változó paraméterszámú függvénynek lehetnek fixen megadott paraméterei is. A fixen megadott paramétereknek kötelezően meg kell előzniük a változó paramétereket. A következő példa egy ilyen függvénydefiníciót mutat.

Példa:

```
using System
class parameters
{
    public static void valtozo(int x, params object[] list)
    {
        Console.WriteLine(x);
        foreach (object o in list)
            Console.WriteLine(o);
    }

    public static void Main()
    {
        valtozo(25, "alma", "barack", "szilva");
    }
}
```

Eredményül kapjuk a függvény paramétereinek kiírását egymás alá.

VI.4. Függvénynevek átdefiniálása

Egy-egy feladat esetén gyakorta előfordul, hogy a megoldást adó függvényt különböző típusú vagy különböző számú paraméterrel jellemezhetjük. Természetesen az ezt megoldó függvényt azonos névvel szeretnénk elkészíteni minden esetben, hisz ez lenne a legkifejezőbb. A lehetőséget gyakran függvény „overloading” névvel is megtaláljuk az irodalomban, vagy a polimorfizmus kapcsán kerül megemlítésre.

Határozzuk meg a konvertálást megvalósító függvény egész és különleges paraméterét.

Kényelmetlenebb lenne megírni a függvénynevet használó függvények egész és különleges paraméterét fordító a paraméterek az adott esetben.

Példa:

```
// valós n
double max
{
    if (x
    {
        else
    }

// egész n
int maxim
{
    if (x
    {
        else
    }

...
int a=2,b
double c=
Console.W
Console.W
Console.W
Console.W
...
```

Meg kell jegyezni, hogy a (template) tulajdonság C# ezt nem támogatja.

Határozzuk meg két szám maximumát! Ha el szeretnénk kerülni a típuskonvertálást mondjuk egész típusról valósba és fordítva, akkor külön az egész és külön a valós számok esetére is meg kell írunk a *maximum* függvényt.

Kényelmetlen dolog lenne, ha mondjuk *egészmax* és *valósmax* néven kellene megírni a két függvényt. Mindkét esetben szeretnénk a *maximum* függvénynevet használni, hogy ne nekünk kelljen eldönteni azt, hogy az egészek vagy valósak maximumát akarjuk-e meghatározni, hanem döntse el a fordító a paraméterlista alapján automatikusan, hogy melyik függvényt is kell az adott esetben meghívni.

Példa:

```
// valós maximum függvény
double maximum(double x,double y)
{
    if (x>y)
        return x;
    else
        return y;
}

// egész maximum függvény
int maximum(int x,int y)
{
    if (x>y)
        return x;
    else
        return y;
}

...
int a=2,b=4;
double c=3.123, d=2;
Console.WriteLine(maximum(4.1,2)); // valós hívás
Console.WriteLine(maximum(4,2)); // egész hívás
Console.WriteLine(maximum(a,b)); // egész hívás
Console.WriteLine(maximum(c,d));
...
```

Még kell jegyezni, hogy ezt a tulajdonságot sok nyelvben függvénysablon (*template*) tulajdonság segítségével elegánsabban oldhatnánk meg, de a jelenlegi C# ezt nem támogatja.

VI.5. Delegáltak, események

Ahogy korábban is szó volt róla, a biztonságosabb programkód készítésének érdekében a mutató aritmetika a C# nyelvben nem engedélyezett. Ebbe beletartozik az is, hogy a függvénymutatók sem lehetnek kivételek.

Ez utóbbi esetben viszont olyan nyelvi tulajdonságok nem implementálhatók, mint például egy menüponthoz hozzárendelt függvény, amit a menüpont választása esetén kell végrehajtani. Ezen utóbbi lehetőséget hivatott a delegált típus biztosítani. Ez már csak azért is fontos, mert többek között a Windows programozás „callback” jellemző paramétere is ezt használja.

A C++ környezetben ezt a lehetőséget a függvénymutató biztosítja.

A delegált valójában egy függvénytípus-definíció, aminek alakja a következő:

```
delegate típus delegáltnév(típus paraméternév,...);
```

A delegált referencia típus. Ha paramétert is megadunk, akkor a paraméter nevét is meg kell adni.

Példa:

```
delegate int pelda(int x, string s);
```

Az előbbi példában tehát a *pelda* delegált típust definiáltuk. Ez olyan függvénytípus, amelyiknek egy egész és egy szöveg paramétere van, és eredményül egy egész számot ad.

Egy delegált objektumnak vagy egy osztály statikus függvényét, vagy egy osztály példányfüggvényét adjuk értékül. Delegált meghívásánál a hagyományos függvényhívási formát használjuk. Ezek után nézzünk egy konkrét példát:

Példa:

```
using System;
class proba
{
    public static int negyzet(int i)
    {
        return i*i;
    }
    public int dupla(int i)
    {
        return 2*i;
    }
}
```

```
class foprogram
{
    delegate
    public :
    {
        eme
        Cor
        pro
        eme
        Cor
    }
}
```

A delegáltakat a *void*, akkor egy dele; delegált), ha nem voi egy végrehajtandó fü Az egyszerű és

Példa:

```
using System;
class proba
{
    public
    {
        re
    }
    public
    {
        r
    }
    public
    {
        r
    }
    public
    {
        C
    }
    publi
    {
        C
    }
}
```


osabb programkód készítésének
en nem engedélyezett. Ebbe
ehetnek kivételek.

ajdonságok nem implementál-
elt függvény, amit a menüpont
lehetőséget hivatott a delegált
nert többek között a Windows
ezt használja.

vénymutató biztosítja.

finíció, aminek alakja a követ-

paraméternév,...);

s megadunk, akkor a paraméter

pust definiáltuk. Ez olyan függ-
paramétere van, és eredményül

' statikus függvényét, vagy egy
legált meghívásánál a hagyo-
zek után nézzünk egy konkrét

```
class foprogram
{
    delegate int emel(int k); // az emel delegált definiálása

    public static void Main()
    {
        emel f=new emel(proba.negyzet);
                                // statikus függvény lesz a delegált
        Console.WriteLine(f(5)); // eredmény: 25
        proba p=new proba();
        emel g=new emel(p.dupla);
                                // normál függvény lesz a delegált
        Console.WriteLine(g(5)); // eredmény: 10
    }
}
```

A delegáltakat a környezet két csoportba sorolja. Ha a delegált visszatérési típusa *void*, akkor egy delegált több végrehajtandó függvényt tartalmazhat (*multicast*, összetett delegált), ha nem *void* a visszatérési típus, mint a fenti példában, akkor egy delegált csak egy végrehajtandó függvényt tud meghívni (*single cast*, egyszerű delegált).

Az egyszerű és összetett delegáltakra nézzük a következő példát:

Példa:

```
using System;
class proba
{
    public static int negyzet(int i)
    {
        return i*i;
    }
    public int dupla(int i)
    {
        return 2*i;
    }
    public int tripla(int i)
    {
        return 3*i;
    }
    public void egy(int i)
    {
        Console.WriteLine(i);
    }
    public void ketto(int i)
    {
        Console.WriteLine(2*i);
    }
}
```

VI. Függvények

```
class foprogram
{
    delegate int emel(int k);
    delegate void meghiv(int j);

    public static void Main()
    {
        emel f=new emel(proba.negyzet);
        Console.WriteLine(f(5));
        proba p=new proba();
        emel g=new emel(p.dupla);
        Console.WriteLine(g(5));
        emel h=new emel(p.tripla);
        g+=h; //g single cast, g a tripla lesz
        Console.WriteLine(g(5)); // eredmény: 15
        meghiv m=new meghiv(p.egy); // multicast delegált
        m+=new meghiv(p.ketto);
        m(5); // delegált hívás, eredmény 5,10
    }
}
```

Ha függvénytípus (delegált) deklarálunk, akkor értékadás esetén a mutató típusa határozza meg, hogy a fordítónak melyik aktuális függvényt is kell az azonos nevűek közül választania.

```
delegate double vmut(double,double);
// vmut olyan delegált amelyik egy valós értéket visszaadó,
// és két valós paramétert váró függvényt jelent

delegate int emut(int,int);
// emut olyan delegált, amelyik egy egész értéket visszaadó
// , és két egész paramétert váró függvényt jelent
vmut mut=new vmut(maximum); // valós hívás
Console.WriteLine(mut(3,5));
```

A *multicast* delegált (típus) definiálási lehetőséget, igazítva az igényeket az eseményvezérelt programozási környezethez (mind a Windows, mind az X11 ilyen), az egyes objektumokhoz, típushoz úgy kapcsolhatjuk, hogy esemény típusú mezőt adunk hozzájuk. Ezt az alábbi formában tehetjük meg:

public event meghiv esemeny;

ahol a *meghiv* összetett delegált. Az esemény objektum kezdő értéke *null* lesz, így az *automatikus meghívás (esemeny(...))* nem ajánlott!

Az eseményhez a felhasználói tevékenység Windows alkalmazásunkban. Példaként nézzük a generált kódot. A fent nevére hallgat. Ennek a kódnak (a keretrendszer) Mikor ehhez az *ampl_SelectedIndex* tervező:

Példa:

this.ampl.Se

VI.6. Feladatol

1. Milyen para
2. Mikor defin
3. Hogyan kel
4. Írjon egy fü
a leghosszal
5. Definiáljon
hetnek. Ku
mérékzésre

Az eseményhez zárasképpen meg kell jegyezni, hogy az egész keretrendszer a felhasználói tevékenységet ehhez a lehetőséghez rendeli akkor, amikor klasszikus Windows alkalmazást akarunk készíteni.

Példaként nézzük meg egy Windows alkalmazás esetén a form tervező által generált kódot. A form (*this*) objektuma egy választómező (*ComboBox*), *ampl* névre hallgat. Ennek könyvtári eseménye, a *SelectedIndexChanged* végrehajtódik (a keretrendszer hajtja végre), ha módosítunk a választáson.

Mikor ehhez az eseményhez egy eseménykezelő függvényt definiálunk *ampl_SelectedIndexChanged* néven, akkor az alábbi sort adja a programhoz a tervező:

Példa:

```
this.ampl.SelectedIndexChanged += new
    System.EventHandler(this.ampl_SelectedIndexChanged);
```

VI.6. Feladatok

1. Milyen paraméterátadási módokat ismer?
2. Mikor definiálhatunk azonos névvel függvényeket egy osztályban?
3. Hogyan kell változó paraméterszámú függvényt használni?
4. Írjon egy függvényt, amely a paraméterként megadott néhány név közül a leghosszabb hosszát adja meg!
5. Definiáljon kupaforduló eseményt, melyekre a szurkolók jelentkezhetnek. Kupaforduló esetén hívjuk meg a feliratkozott szurkolókat a mérkőzésre!

VII. Osztályok

C#-ban az osztályok a már ismert összetett adatszerkezetek (struktúrák) egy természetes kiterjesztése. Az osztályok nemcsak adattagokat, hanem operátorokat, függvényeket is tartalmazhatnak. Az osztályok használata esetén általában az adatmezők az osztály által éppen leírni kívánt esemény „állapotjelzői”, míg a függvénymezők az állapotváltozásokat írják le, felhasználói felületet biztosítanak. A függvénymezőket gyakran metódusoknak is nevezi a szakirodalom. Osztálymezőket lehetőségünk van zártta nyilvánítani (*encapsulation*), ezzel biztosítva az osztály belső „harmóniáját”. Ezek az osztálymezők általában az „állapotjelző” adatmezők.

Új igények, módosítások esetén lehetőségünk van az eredeti osztály megtartása mellett, abból a tulajdonságok megőrzésével egy új osztály származtatására, amely öröklí (*inheritance*) az ősoosztály tulajdonságait (az osztályok elemeit). A C++ nyelvben lehetőségünk van arra, hogy több osztályból származtassunk utódosztályt (*multiple inheritance*), esetünkben ezt a jellemzőt a CLR nem támogatja, így ennek a fejlesztőkörnyezetnek egyik nyelve sem támogatja a többszörös öröklés lehetőségét. Lehetőségünk van *interface* definícióra, amiket egy osztály implementálhat. Egy osztály több *interface*-t is implementálhat.

Az öröklés lehetőségének a gyakorlatban legelőször jelentkező haszna talán az, hogy a programunk forráskódja jelentősen csökkenhet.

Egy osztálytípusú változót, az osztály megjelenési alakját gyakran objektumnak is szokás nevezni.

Az osztályok használatához néhány jótanácsot, a nyelv „jobbkez-szabályát” a következő pontokban foglalhatjuk össze. Ezek a tanácsok természetesen nem a nyelv tanulási időszakára, hanem a későbbi, a nyelvben történő programozási feladatokra vonatkoznak elsősorban.

- Ha egy programelem önálló értelmezéssel, feladattal, tulajdonságokkal rendelkezik, akkor definiáljuk ezt az elemet önálló osztályként.
- Ha egy programrész adata önálló objektumként értelmezhető, akkor definiáljuk őt a kívánt osztálytípus objektumaként.
- Ha két osztály közös tulajdonságokkal rendelkezik, akkor használjuk az öröklés lehetőségét.
- Általában is elmondható, hogy ha a programokban az osztályok közös vonásokkal rendelkeznek, akkor törekedni kell univerzális bázisosztály létrehozására. Gyakran ezt *absztrakt bázisosztálynak* is nevezzük.
- Az osztályok definiálásakor kerüljük a „nyitott” (publikus) adatmezők használatát.

VII.1. Osztály

Az osztályok c

Az osztálykulc
azonosító neve. F
osztálynév, majd k
Az osztályok t

- private
- public
- protected
- internal
- protected in

A *private* tago
Az osztályok p
A *protected* m
utódosztályból ige
Az *internal* m
A *protected in*
adott hozzáférési e
tály utódosztályát
származtatni, hogy

Ha az „osztály
ban adatok csop
olvashatjuk, hogy
adatszerkezetet sz
definiáljuk.

Az egyes mez
a struktúrák esetéb

Az osztálydefi

VII.1. Osztályok definiálása

Az osztályok deklarációjának alakja a következőképpen néz ki:

```

osztálykulcsszó osztálynév [: szülő, ...]
{
    osztálytaglista
};

```

Az osztálykulcsszó a *class*, *struct* szó lehet. Az osztálynév az adott osztály azonosító neve. Ha az osztály valamely bázisosztályból származik, akkor az osztálynév, majd kettőspont után az ősosztály felsorolása történik.

Az osztályok tagjainak öt elérési szintje lehet:

- private
- public
- protected
- internal
- protected internal.

A *private* tagokat csak az adott osztályon belülről érhetjük el.

Az osztályok *publikus* mezőit bárhonnan elérhetjük, módosíthatjuk.

A *protected* mezők az osztályon kívüliek számára nem elérhetőek, míg az utódosztályból igen.

Az *internal* mezőket a készülő program osztályaiból érhetjük el.

A *protected internal* elérési módjában egy egyszerű vagy kapcsolattal megadott hozzáférési engedély. A mező elérhető a programon belülről, vagy az osztály utódosztályából! (Egy osztályból természetesen tudunk úgy utódosztályt származtatni, hogy ez nem tartozik az eredeti programhoz.)

Ha az „osztály” definiálásakor a *struct* szót használjuk, akkor abban elsősorban adatok csoportját szeretnénk összefogni, ezért gyakran tanácsként is olvashatjuk, hogy ha hagyományos értelemben vett struktúrát, mint összetett adatszerkezetet szeretnénk használni, akkor azt „*struct* típusú osztályként” definiáljuk.

Az egyes mezőnevekre való hivatkozás ugyanúgy történik, ahogy korábban a struktúrák esetében.

Az osztálydefiníció alakjának ismeretében nézzünk egy konkrét példát!

VII. Osztályok

Példa:

```
...
class első {
    private int x;                // az x mező privát
    public void beállít(int mennyi)
    { x=mennyi;};                // függvénymező
    public int kiolvas()
    { return x;};
};
```

Mivel az *x* az osztály privát mezője, ezért definiáltunk az osztályba két függvénymezőt, amelyek segítségével be tudjuk állítani, illetve ki tudjuk olvasni az *x* értékét. Ahhoz, hogy ezt a két függvényt az osztályon kívülről el tudjuk érni, publikusnak kellett őket deklarálni.

```
static void Main()
{
    első a;                      // legyen egy a nevű első típusú osztályunk
    a.x=4;                       // hiba!!! x privát mező
    a.beállít(7);                // az x mező értékét 7-re állítjuk
    Console.WriteLine(a.kiolvas());
}
```

Osztályt egy osztályon vagy egy függvényen belül is definiálhatunk, ekkor azt belső vagy lokális osztálynak nevezzük. Természetesen ennek a lokális osztálynak a láthatósága hasonló, mint a lokális változóké.

Mielőtt a statikus mezőkről szólnánk, szót kell ejteni a *this* mutató szerepéről. Ez a mutató minden osztályhoz, struktúrához automatikusan létrejön. Tételezzük fel, hogy definiáltunk egy osztálytípust. A programunk során van több ilyen típusú objektumunk. Felvetődik a kérdés, hogy amikor az egyes osztályfüggvényeket meghívjuk, akkor a függvény végrehajtása során honnan tudja meg a függvényünk, hogy ő most éppen melyik aktuális osztályobjektumra is kell, hogy hasson? Minden osztályhoz automatikusan létrejön egy mutató, aminek a neve *this*, és az éppen aktuális osztályra mutat. Így, ha egy osztályfüggvényt meghívunk, amely valamilyen módon például a privát változókra hat, akkor az a függvény a *this* mutatón keresztül tudja, hogy mely aktuális privát mezők is tartoznak az objektumhoz. A *this* mutató konkrétabb használatára a későbbiek során láthatunk példát.

Tételezzük fel, hogy egy mérőeszközt, egy adatgyűjtőrendszert egy osztállyal akarunk jellemezni. Ennek az osztálynak az egyes objektumai a mérőeszközünk meghatározott tulajdonságait képviselik. Viszont az eszköz jelerősítési együtthatói azonosak. Ezért szeretnénk, hogy az osztály erősítésmezője közös legyen, ne objektumhoz, hanem az osztályhoz kötődjön. Ezt a lehetőséget statikus mezők segítségével valósíthatjuk meg.

Statikus mező minden objektum inicializálás hiány

Példa:

```
class tesz
{
    public
    public
    ...
}
```

```
...
teszt.a=5;
```

Statikus mező hiszen a statikus vonatkozóan) köz nem tudnak elérni normál függvényt

VII.2. Konstrukt

Adatok, adat kezdeti értékadás ban tárgyalt típus értékadás elvége; olyan egyszerű, függvény kapja n osztály „születés tornak vagy konstruktorként osztály nevével a nélküli automatik

A konstruktor is több lehet, ha r

Az osztály 1 kötelező a new meghívását végz név után zárójele

A bináris f konstruktor haszn

Statikus mezőket a *static* jelző kiírásával definiálhatunk. Ekkor az adott mező minden objektum esetében közös lesz. A statikus mező kezdőértéke inicializálás hiányában *0*, *null*, *false* lesz.

Példa:

```
class teszt {
    public static int a;          // értéke még 0
    public static string s="Katalin";
    ...
};

...
teszt.a=5;                      // statikus mező értékadása, 5 az érték
```

Statikus mezők a *this* mutatóra vonatkozó utalást nem tartalmazhatnak, hiszen a statikus mezők minden egyes objektum esetén (azonos osztálybelire vonatkozóan) közösnek. Tehát például statikus függvények nem statikus mezőket nem tudnak elérni! Fordítva természetesen problémamentes az elérés, hiszen egy normál függvényből bármely statikus mező, függvény elérhető.

VII.2. Konstruktor- és destruktorkonstruktor függvények

Adatok, adatszerkezetek használata esetén gyakori igény, hogy bizonyos kezdeti értékadási műveleteket, kezdőérték-állításokat el kell végezni. A korábban tárgyalt típusoknál láttuk, hogy a definícióval „egyekötve” a kezdőértékadás elvégezhető. Osztályok esetén ez a kezdőértékadás nem biztos, hogy olyan egyszerű, mint volt elemi típusok esetén, ezért ebben az esetben egy függvény kapja meg az osztály inicializálásával járó feladatot. Ez a függvény az osztály „születésének” pillanatában automatikusan végrehajtódik, és konstruktornak vagy konstruktor függvénynek nevezzük. A konstruktor neve mindig az osztály nevével azonos. Ha ilyet nem definiálunk, a keretrendszer egy paraméter nélküli automatikus konstruktort definiál az osztály számára.

A konstruktor egy szabályos függvény, így mint minden függvényből, ebből is több lehet, ha mások a paraméterei.

Az osztály referencia típusú változó, egy osztálypéldány létrehozásához kötelező a *new* operátort használni, ami egyúttal a konstruktor függvény meghívását végzi el. Ha a konstruktornak vannak paraméterei, akkor azt a típusnév után zárójelek között kell megadni.

A bináris fájl feladat egyfajta megoldását tekintsük meg példaként a konstruktor használatára.

VII. Osztályok

Példa:

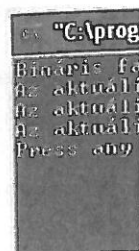
```

using System;
class binfa
{
    int x; // elem
    int db; // elemszám
    public binfa(int i) // konstruktor
    {
        x=i; db=1;
        bal=jobb=null;
    }
    public binfa bal, jobb;
    public void illeszt(int i)
    {
        if (x==i) db++;
        else if (i<x) if (bal!=null) bal.illeszt(i);
                       else bal= new binfa(i);
        // bal oldalra illesztettük az elemet
        else if (jobb!=null) jobb.illeszt(i);
                       else jobb= new binfa(i);
        // jobb oldalra illesztünk
    }
    public void bejar()
    {
        // először bejárjuk a bal oldali fát
        if (bal!=null) bal.bejar();
        // ezután jön az aktuális elem
        Console.WriteLine("Az aktuális elem: {0} darabszám: {1}",x,db);
        // végül a jobb oldali fa következik
        if (jobb!=null) jobb.bejar();
    }
}

class program
{
    public static void Main()
    {
        binfa bf=new binfa(7);
        Console.WriteLine("Bináris fa példa.");
        bf.illeszt(5);
        bf.illeszt(9);
        bf.illeszt(5);
        bf.bejar();
    }
}

```

Futási eredmér



VII.2.1. Statikus

Egy osztály, a függvénymezőket példányoktól függően végezheti a statikus keretrendszer nem

Ha definiálunk indulásakor megtérő nánk.

Statikus kons
írní. Ennek a kons

Példa:

```
using System;
class Stat
{
    public
    static
    {
        ;
    }
    public
    {
        ;
    }
}
```

Futási eredményül az alábbiakat kapjuk:

```

C:\programok\hajra\bin\Debug\hajra.exe
Bináris fa példa.
Az aktuális elem: 5 darabszám: 2
Az aktuális elem: 7 darabszám: 1
Az aktuális elem: 9 darabszám: 1
Press any key to continue

```

10. ábra

VII.2.1. Statikus konstruktor

Egy osztály, ahogy azt korábban is említettük, tartalmazhat statikus adat- és függvényezőket is. Ezen osztályozók a dinamikusan létrejövő objektum-példányoktól függetlenül jönnek létre. Ezeknek a mezőknek az inicializálását végezheti a statikus konstruktor. Definiálása opcionális, ha nem definiáljuk, a keretrendszer nem hozza létre.

Ha definiálunk egy statikus konstruktort, akkor annak meghívása a program indulásakor megtörténik, még azelőtt, hogy egyetlen osztálypéldányt is definiálnánk.

Statikus konstruktor elé nem kell hozzáférési módosítót, visszatérési típust írni. Ennek a konstruktornak nem lehet paramétere sem.

Példa:

```

using System;
class statikus_osztaly
{
    public static int x;
    static statikus_osztaly()
    {
        x=2;
    }
    public int getx()
    {
        return x
    }
}

```