Programozás I.

2. előadás
C# bevezető

Sergyán Szabolcs
sergyan.szabolcs@nik.uni-obuda.hu

Óbudai Egyetem
Neumann János Informatikai Kar

2012. szeptember 17.

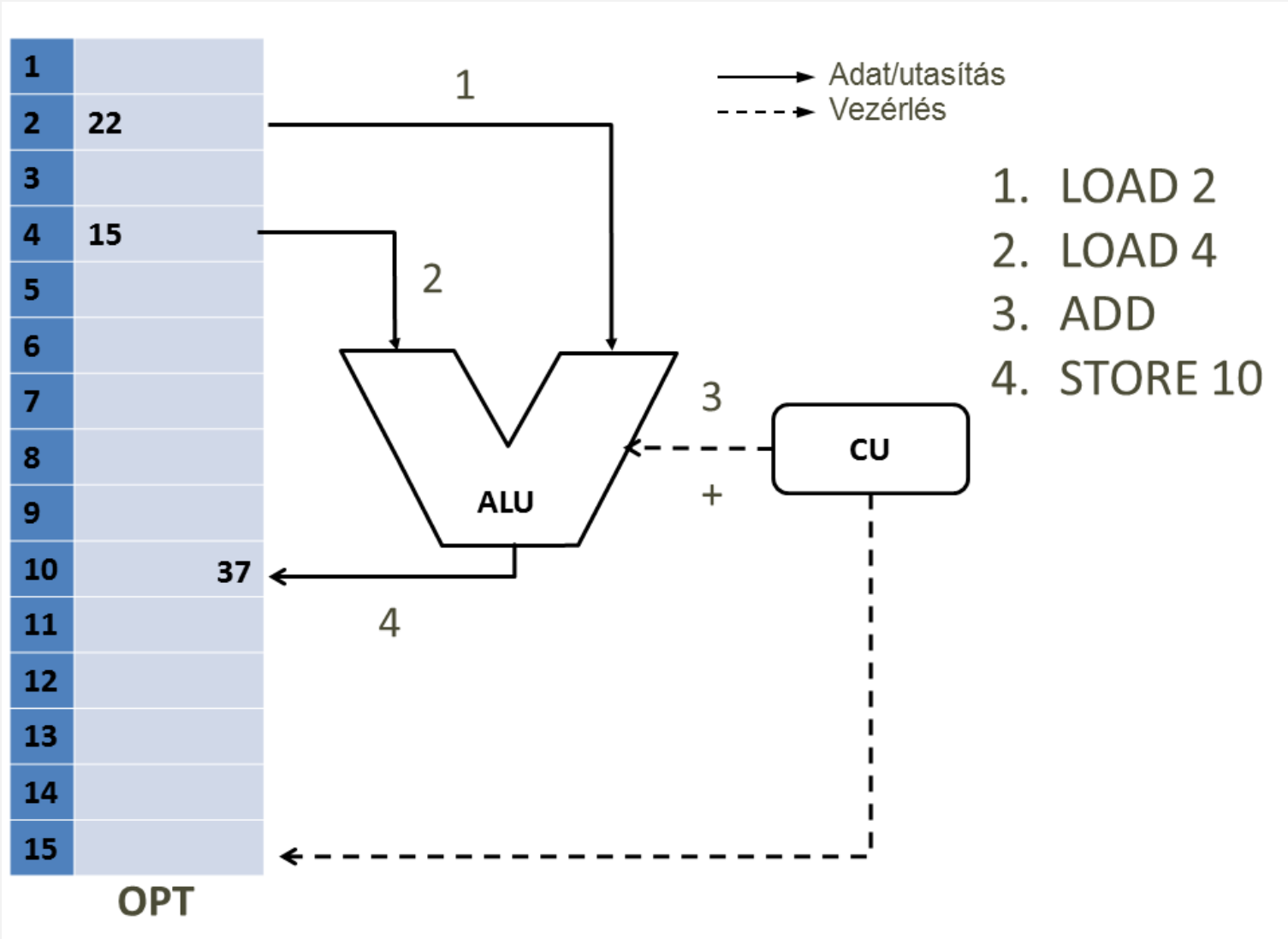
Tartalom

Tartalom

Számítógépes műveletvégzés

- Cél: annak modellezése, hogy az egyszerű adatokat hogyan tárolja, és a műveleteket hogyan végzi el a számítógép
  (részletesebben: Számítógépes architektúrák)
- Használt elemek
  - OPT (Operatív tár: memória, cache, regisztertér is lehetne.
    A legfontosabb: bájtszervezésű tár)
  - ALU (Arithmetical and Logical Unit: Aritmetikai és Logikai Egység;
    a műveletvégző egység. 2 bemeneten tud valamilyen műveletet végezni,
    1 kimenet)
  - CU (Control Unit: Vezérlőegység)
  - Most NEM használjuk a "CPU" elnevezést, mert abban lenne más is
    (buszok, regiszterek, több fajta cache, stb.)

Az ALU sematikus működése



Problémák a közvetlen memóriacímekkel

- Ha így működne, egy nagyobb program megírása szinte lehetetlen komplexitású lenne (a jelenlegi számítógépek címtere óriási)
- Fogalmunk sincs, hogy az operációs rendszer hova helyezi el a programunkat (több program fut egyszerre)
- Neumann-elv: *Az adatok és az utasítások közös operatív tárban vannak* $\rightarrow$ ugyanolyan bináris reprezentációban! Nehéz lenne karban tartani, hogy hol van adat, és hol van utasítás.

Megoldás

A programok csak változókat használnak, a változók konkrét memóriacíme nem érdekes.

Problémák a változókkal

- A fenti példában a memória bájtszervezésű: 1 rekesz = 1 byte, a tárolás bitekben történik
- Ha számot akarunk tárolni, akkor 8 bit csak a 0..255 intervallumban elég. Mi van, ha nagyobb számokat akarunk tárolni?
- Mi van, ha nem számot akarunk tárolni, hanem szöveget?
- Mi van, ha nem szöveget akarunk tárolni, hanem egy képet vagy más bináris adatot?

Megoldás

A változók bevezetése önmagában nem elég. Tudnunk kell, hogy a változó által kijelölt területen mennyi adat van, és azt hogyan kell értelmezni. → TíPUSOK

Tartalom

Parancsok, adatok

- A számítógép minden adatot és utasítást bináris formában tárol a memóriában
- A tárolt bináris adat jelentése értelmezésfüggő
- Az adatokat változókban tároljuk
- A változó deklarációja határozza meg a tárolt adat méretét és értelmezését (típusát)

Változókkal végezhető tevékenységek

- **Deklaráció**: A változó nevének és típusának megadása
  ```
  int szám;
  ```
- **Értékadás**: Érték elhelyezése a változóban
  ```
  szám = 25;
  szám = 6 * 2 - 29;
  ```
- **Érték lekérdezése**: A változó tartalmának kiolvasása. Az érték a kiolvasás után is megmarad a változóban.
  ```
  5 * 10 - szám + 2;
  ```

Változókkal végezhető tevékenységek

- A tevékenységek sorrendje:
  1. Deklaráció
  2. Értékadás
  3. Érték lekérdezése
- A fentiek közül bármelyik tevékenységet is szeretnénk végrehajtani, előbb a sorrendben őt megelőzőt kell elvégezni
- A deklaráció és a kezdeti értékadás összevonható egy utasításba

```
int szám = 25;
```

Egyszerű adattípusok

- Mindent binárisan tárolunk, az összes adattípus mérete a byte többszöröse
- Számok
  - Egész
  - Valós (lebegőpontos)
- Karakterek, karaktersorozatok (string-ek)
- Logikai értékek

Egész (fixpontos) számok

- Két fő kérdés: tárolási méret, előjelesség → e kettőtől függ az ábrázolás értéktartománya
- Relatív kicsi ábrázolás tartomány, de teljes pontosság
- Előjeles ábrázolási mód: kettes komplemens (ld. IRA)

Bitek száma	Előjeltelen	Előjeles
8	`byte`	`sbyte`
16	`ushort`	`short`
32	`uint`	`int`
64	`ulong`	`long`

Egész (fixpontos) számok

Név	Leírás	Értéktartomány
`sbyte`	8 bites előjeles egész	-128 : 127
`byte`	8 bites előjel nélküli egész	0 : 255
`short`	16 bites előjeles egész	-32.768 : 32.767
`ushort`	16 bites előjel nélküli egész	0 : 65535
`int`	32 bites előjeles egész	-2.147.483.648 : 2.147.483.647
`uint`	32 bites előjel nélküli egész	0 : 4.294.967.295
`long`	64 bites előjeles egész	-9.223.372.036.854.775.808 : 9.223.372.036.854.775.807
`ulong`	64 bites előjel nélküli egész	0 : 18.446.744.073.709.551.615

Egész számok speciális értékei

- `MinValue`
  - Az ábrázolható legkisebb szám
  - `byte.MinValue`, `int.MinValue`, stb.
- `MaxValue`
  - Az ábrázolható legnagyobb szám
  - `short.MaxValue`, `long.MaxValue`, stb.
- Túlcsordulás (a változó`++` növeli a változó értékét)
  - `byte a = 255; a++;` ← a változó értéke 0 lesz
  - `sbyte b = -128; b--;` ← a változó értéke 127 lesz

Valós (lebegőpontos) számok

$$\pm m \cdot 2^k$$

- Normalizált szám formájában tároljuk (előjel, mantissza, karakterisztika, részletesebben: IRA)
- Nagy számtartomány, de nem pontos
- A számábrázolás formájából adódóan nem csak abszolút értékben túl nagy, de nullához túlságosan közeli számokat sem tud ábrázolni
- A karakterisztika mérete az ábrázolható számtartomány méretét, a mantissza mérete a pontosságot határozza meg

Valós (lebegőpontos) számok

Név	**Leírás**	**Értékes jegy**	**Értéktartomány**
`float`	32 bites lebegőpontos	7	$\pm 1,5 \cdot 10^{-45} : \pm 3,4 \cdot 10^{38}$
`double`	64 bites lebegőpontos	15	$\pm 5,0 \cdot 10^{-324} : \pm 1,7 \cdot 10^{308}$
`decimal`	128 bites nagypontosságú	28	$\pm 1,0 \cdot 10^{-28} : \pm 7,9 \cdot 10^{28}$

Nullával való osztás

- Egész számtípus használatakor futás idejű hibát dob:

```
int a = 5;
int b = 0;
int c = a / b;
```

- Valós számtípus használatakor hibátlan:

```
float x = 5;
float y = 0;
float z = x / y;
```

- Valós számtípus esetén az eredmény lehet: végtelen (pozitív vagy negatív), illetve "Nem szám"

Valós számok speciális értékei

- 0
  - Külön $+0$ és -0 is ábrázolható, de ezek egyenértékűek
- $\pm\infty$
  - A végtelen elfogadott, bizonyos műveletekhez használható érték
  - Pozitív szám / 0 $\rightarrow$ $+\infty$, Negatív szám / 0 $\rightarrow$ $-\infty$
  - `float.PositiveInfinity`, `double.NegativeInfinity` – `decimal` nincs
- Nem szám
  - $0/0$, illetve ∞/∞ eredménye
  - `float.NaN`, `double.NaN` – `decimal` nincs

Valós számok speciális értékei

- Epsilon
  - A legkisebb ábrázolható pozitív szám
  - `float.Epsilon`, `double.Epsilon` – `decimal` nincs
- Kezdőérték megadása
  - Kódban tizedespont használandó: `double pi = 3.14;`
  - Minden így megadott érték típusa `double`
- Jelzőkarakterek kezdőérték megadásánál
  - `float pi = 3.14f;`
  - `decimal pi = 3.14M;`

Karakterek

- Egy karakter tárolása is binárisan történik $\rightarrow$ kell lennie egy szabálynak, hogy melyik kód melyik karakternek felel meg
- ASCII: kezdetben 7 bites. 0-31: vezérlő karakterek; 32-127: angol ABC kis- és nagybetűi, számok, írásjelek
- 8 bites ASCII: 128-255: rajzoló karakterek, speciális karakterek (ä, ç), csak az országok egy részének megfelelő
  - Hiányzó karakterek: ő, Ő, ű, Ű (csak ô, û)
  - Nincs elég hely: japán, kínai, szír, stb.
  - Alternatíva: kódlapok (cp437, cp850/852, cp1250)
  - Kódlapok szabványosítása (ISO8859-1, -2, -15)
  - Probléma: készítés kódlapja $\leftrightarrow$ feldolgozás kódlapja

Karakterek

- Alternatíva: felejtsük el az 1 byte = 1 karakter szabályt
  - Probléma: hogyan állapítjuk meg egy karakterlánc hosszát? Eddig egyszerű volt, de ezután ... $\rightarrow$ mindent újra kell írni ...
- UNICODE kódolás: UTF-8, UTF-16, UTF-32 kódlapok
  - UTF-8: Az angol ABC betűinek kódolása ugyanaz, a többi karakternek egyedi karakterje van, 2-4 byte / karakter
  - UTF-16: 2 vagy 4 byte / karakter
  - Az UTF-16 a C# nyelv és a .NET keretrendszer belső kódolása (a file-ok kódolása UTF-8
  - Minden karakteres típus, minden szövegkezelő függvény ez alapján működik

Karakterek, karakterláncok

- Karakter: `char` (megadás: aposztróffal)
  - `char c = 'é';`
- Karakterlánc: `string` (megadás: idézőjellel)
  - `string s = "Árvízi tükörfúrógép";`
- Speciális karakterek is megadhatóak (@ jellel kikapcsolható)

Jelölés	Karakter
\0	Null karakter
\a	Sípszó
\b	Visszatörlés
\f	Lapdobás
\n	Soremelés
\r	Kocsi vissza
\t	Vízszintes tabulátor

Jelölés	Karakter
\v	Függőleges tabulátor
\x....	Hexadecimális kód
\u....	Unicode karakter
\U....	Unicode karakter
\'	Aposztróf
\"	Idézőjel
\\	Backslash

Logikai típus

Név	Leírás	Értéktartomány
`bool`	Logikai adattípus	`true` vagy `false` (igaz vagy hamis)

- Teljesítmény okokból általában nem egy biten ábrázoljuk (részletesebb ld. IRA)

Logikai műveletek

A	B	$A \wedge B$	$A \vee B$	$A \oplus B$	$\neg A$
H	H	H	H	H	I
H	I	H	I	I	I
I	H	H	I	I	H
I	I	I	I	H	H

Változók deklarálása és használata

```
int j = -10;
int x = 10, y = 20;
double pi = 3.14159;
const int száz = 100;
char d = 'x';
char UnicodePélda = 'u\0170';
string jegy = "jeles";
string ElérésiÚt = "C:\\Program Files\\";
string ElérésiÚt2 = @"C:\Program Files\";
string vers = @"Hová merült el
   szép szemed világa";
bool igaz = true;
```

Fontos szabály: azonos névvel nem lehet egy változót kétszer deklarálni

A közvetlenül beírt értékek más neve: *literál*

Speciális literálok

- Egész literál
  - Típusuk: `int`, `uint`, `long` vagy `ulong` (ebben a sorrendben) attól függően, hogy melyik típusban fér el a megadott érték
  - Az egész literál típusa is módosítható a literál mögé írt betűkkel
    - U: `uint` vagy `ulong` (pl.: `255U`)
    - L: `long` vagy `ulong` (pl.: `-356L`)
    - UL: `ulong` (pl.: `222UL`)
  - Megadható hexadecimálisan: `0xFF`
- Valós literál, tudományos megadás: `1.23456E-2`

Típuskonverziók

- A számtípusok közötti konverzió miként történik attól függ, hogy történik-e értékvesztés a konverzió során
- Egyszerű értékadás használható, amennyiben biztos, hogy nincs értékvesztés:

```
byte a = 5;      long c = 5;      float f = 3.2f;
int b = a;       float d = c;     double g = f;
```

- Amennyiben értékvesztés történhet, akkor mindenképp jelezni kell a konverziót, ez az ún. típuskényszerítés, "kasztolás" (typecasting):

```
int a = 999;         double d = 3.14;  int i1 = -1;
byte b = (byte)a;    int c = (int)d;   uint i2 = (uint)i1;
```

Típuskonverziók

- A stringgé történő konvertálás a C# nyelven MINDEN változónál ugyanúgy történik:

```
byte b = 250;                    float f = 3.14f;
string s1 = b.ToString();        string s2 = f.ToString();
```

- Stringből számmá tudunk konvertálni:

```
string s = "123";                string s2 = "123,456";
byte b = byte.Parse(s);          float f = float.Parse(s2);
```

- Typecasting esetén (ebben a félévben számok között):
  `célváltozó = (céltípus)forrásváltozó;`
- Stringgé konvertálásnál:
  `célváltozó = forrásváltozó.ToString();`
- Stringből konvertálásnál:
  `célváltozó = céltípus.Parse(stringváltozó);`

Tartalom

Kifejezések

- A kifejezések ("expression") adatokat szolgáltató operandusokból és rajtuk valamilyen műveletet végző operátorokból állnak
  - Operandus: pl. bármely változó vagy konkrét megadott érték
  - Operátor: pl. `+ - / *`
- A kifejezések egymásba is ágyazhatók
  - Egy kifejezés operandusa maga is lehet kifejezés
- Több operátor esetén ezek fontosság sorrendje (precedenciája) határozza meg a kiértékelés sorrendjét
  - Példa: az `x + y * z` kifejezés kiértékelés szempontjából: `x + (y * z)`
  - A sorrend zárójelezéssel explicit módon is meghatározható

Operátorok

Aritmetikai operátorok

Operátor	Kifejezés	Jelentés
+	`+x`	Előjelképzés
	`x + y`	Összeadás vagy kombináció (*szám/string*)
-	`-x`	Előjelképzés
	`x - y`	Kivonás
*	`x*y`	Szorzás
/	`x / y`	Osztás (*egész/tört osztás, nullával osztás!*)
%	`x % y`	Maradékképzés
++	`x++`	Növelés eggyel `x` kiértékelése után
	`++x`	Növelés eggyel `x` kiértékelése előtt
--	`x--`	Csökkentés eggyel `x` kiértékelése után
	`--x`	Csökkentés eggyel `x` kiértékelése előtt

Operátorok

Relációs (összehasonlító) operátorok

Operátor	Kifejezés	Jelentés
==	`x == y`	Egyenlő?
!=	`x != y`	Nem egyenlő?
<	`x < y`	Kisebb?
>	`x > y`	Nagyobb?
<=	`x <= y`	Kisebb vagy egyenlő?
>=	`x >= y`	Nagyobb vagy egyenlő?

Operátorok

Bináris logikai (bitenkénti műveletvégző) operátorok

Operátor	Kifejezés	Jelentés
~	~x	Bitenkénti nem művelet
&	x & y	Bitenként ÉS művelet
^	x ^ y	Bitenkénti KIZÁRÓ VAGY művelet
|	x | y	Bitenkénti VAGY művelet
<<	x << y	Eltolás balra (x eltolása y helyiértékkel)
>>	x >> y	Eltolás jobbra (x eltolása y helyiértékkel)

Operátorok

Logikai (feltételvizsgáló) operátorok

Operátor	Kifejezés	Jelentés
!	`!x`	A kifejezés értéke `x` ellentettje
&&	`x && y`	A kifejezés akkor igaz, ha `x` és `y` is igaz
||	`x || y`	A kifejezés akkor igaz, ha `x` vagy `y` is igaz

Operátorok

Értékadó operátorok

Operátor	Kifejezés	Értékadás típusa
=	x = y	Egyszerű (x értéke legyen egyenlő y-nal)
+=	x += y	Összeadással (x = x + y)
-=	x -= y	Kivonással (x = x - y)
*=	x *= y	Szorzással (x = x * y)
/=	x /= y	Osztással (x = x / y)
%=	x %= y	Maradékképzéssel (x = x % y)
&=	x &= y	Bitenkénti ÉS művelettel (x = x & y)
^ =	x ^ = y	Bitenkénti KIZÁRÓ VAGY művelettel (x = x ^ y)
|=	x |= y	Bitenkénti VAGY művelettel (x = x | y)
<<=	x <<= y	Bitenkénti eltolással balra (x = x << y)
>>=	x >>= y	Bitenkénti eltolással jobbra (x = x >> y)

Tartalom

Implementálás (parancssorból)

1. Algoritmus → C# forráskód
   1. Forráskód megírása szövegszerkesztőben
   2. Forráskód elmentése *fájlnév.cs* néven
2. C# forráskód → Futtatható fájl (.exe)
   1. Parancssori ablak megnyitása (vagy Visual Studio Command Prompt indítása)
   2. Belépés a forráskódot tartalmazó könyvtárba (cd)
   3. Forrásfájl lefordítása (csc *fájlnév.cs*)
3. Futtatható fájl[1] végrehajtása
   1. Parancssori ablak megnyitása
   2. Program futtatása (*fájlnév.exe*)

[1] Ez nem valódi futtatható fájl, hanem ún. köztes kód, melynek futtatásához a .NET Framework megfelelő verziója szükséges

C# program alapja

```
class ProgramNév
{
    static void Main()
    {
    }
}
```

Utasítások

- Egy program alapvetően utasítások sorozatából áll
- Egyszerű utasítások (*statement*)
  - Az egyszerű utasítások lehetnek deklarációk, kifejezések vagy előre definiált utasítások
  - Az egyszerű utasításokat ; karakter zárja le
- Összetett utasítások (*compound statement*)
  - Több utasítás sorozata összefogható egy összetett utasítássá
  - Az összetett utasítások végén nem szerepel ; karakter
  - Az összetett utasítás másik neve: *blokk* vagy *kódblokk*
  - A kódblokkon belül definiált változó csak a kódblokkon belül látszik

Szekvencia

- Minden utasítást ; zár le
- Az utasítások írhatók külön sorokba is:

```
utasítás1;
utasítás2;
...
utasításN;
```

- Az utasítások írhatók egy sorba:

```
utasítás1; utasítás2; ... utasításN;
```

Kiíratás

- `Console.Write("Szöveg")` – Kiírja a Szöveg-et a konzolra
- `Console.WriteLine("Szöveg")` – Kiírja a Szöveg-et a konzolra és a kurzort a következő sor elejére ugratja át
- Használatukhoz szükséges a program legelején a `using System;` utasítás

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello world!");
    }
}
```

Kiíratás

Változók értékének kiíratása

```
using System;

class Program
{
    static void Main()
    {
        int a = 3;
        int b = 4;
        int c = 5;
        Console.WriteLine("A hármoszög oldalai:
            a = {0}, b = {1}, c = {2}", a, b, c);
    }
}
```

Beolvasás

- `Console.ReadLine()` paranccsal
- Ha az `s` `string` típusú változóban szeretnénk eltárolni a beolvasott szöveget, akkor:
  `string s = Console.ReadLine();`
- Ha a beolvasott értéket nem karaktersorozatként (`string`) szeretnénk használni, akkor a szükséges konverziót is végre kell hajtani.
- Használatához szükséges a program elején a `using System;` utasítás

Beolvasás

Példa

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("a = ");
        string s = Console.ReadLine();
        int a = int.Parse(s);
        Console.Write("b = ");
        s = Console.ReadLine();
        int b = int.Parse(s);
        Console.Write("c = ");
        s = Console.ReadLine();
        int c = int.Parse(s);
        Console.WriteLine("A hármoszög oldalai:
            a = {0}, b = {1}, c = {2}", a, b, c);
    }
}
```

Elágazás (szelekció)

```
if (feltétel)
    utasítás1;
[else
    utasítás2;]
```

```
if (feltétel)
{
    utasítás1;
    utasítás2;
}
else
{
    utasítás3;
    utasítás4;
}
```

- `else` ág elhagyható
- Több utasítás esetén kötelező a { }

Elágazás (szelekció)

```
if (feltétel1)
    utasítás1;
else if (feltétel2)
    utasítás2;
else if (feltétel3)
    utasítás3;
...
    ...
else
    utasításN;
```

- `else` ág elhagyható
- Több utasítás esetén kötelező a { }
- `else if` ágakból tetszőleges számú használható

Rövidzár-kiértékelés

- Akkor fordul elő, amikor egy logikai kifejezésben több logikai kifejezést csatolunk össze az ÉS / VAGY (`&&` / `||`) operátorok használatával
- ÉS operátor esetén, ha az első kifejezés hamis, akkor a másodikkal már nem is érdemes foglalkozni, az eredmény mindenképp hamis lesz
- VAGY operátor esetén, ha az első kifejezés igaz, akkor a másodikkal már nem is érdemes foglalkozni, az eredmény mindenképp igaz lesz
- C#-ban az összetett logikai kifejezések (pl. feltételek esetén) ilyen módon történik

Elágazás (szelekció)

```
switch (kifejezés)
{
    case érték1 : utasítás(ok)1; break;
    case érték2 : utasítás(ok)2; break;
    ...
    [default : utasítás(ok)N; break;]
}
```

- Ha a `kifejezés` értéke
  - `érték1`, akkor `utasítás(ok)1`,
  - `érték2`, akkor `utasítás(ok)2`, ... hajtódik végre
- Ha a `kifejezés` értéke egyik `case` ágban megadottal sem egyezik meg, akkor a `default` ágban megadott `utasítás(ok)N` hajtódik végre

Ciklus (iteráció)

```
while (feltétel)
    utasítás;
```

```
while (feltétel)
{
    utasítások;
}
```

- Amíg a `feltétel` igaz, a ciklusmagban található `utasítások` újra és újra végrehajtódnak
- Ha a ciklusmagban több utasítás is van, akkor kötelező a { }

Ciklus (iteráció)

```
do
{
    utasítások;
}
while (feltétel);
```

- Amíg a `feltétel` igaz, az `utasítások` újra és újra végrehajtódnak

Ciklus (iteráció)

```
for (inicializáló_rész; feltétel; módosító_rész)
{
    utasítások;
}
```

- Amíg a `feltétel` igaz, az `utasítások` újra és újra végrehajtódnak
- Több utasítás esetén kötelező a { }
- `inicializáló_rész`: ciklusváltozó(k) inicializálása
- `módosító_rész`: ciklusváltozó(k) értékének módosítása
- A zárójelek közötti három rész közül egyiket sem kötelező megadni (végtelen ciklus)

Felhasznált irodalom

- Korábbi évek OOP diasorai
- Nagy Tibor István diasorai
- Andrew Troelsen: A C# 2008 és a .NET 3.5. *Szak Kiadó*, 2009