Eötvös Loránd Tudományegyetem
Természettudományi Kar

Alkalmazott Modul III

2. előadás

Procedurális programozás: adattípusok, elemi programok

© 2011.09.26. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

A procedurális programozás

Koncepciója

- A *procedurális programozás* a felülről lefelé tervezés elvét követi, vagyis a feladatot részfeladatokra bontja
  - a részfeladatokhoz az imperatív megközelítés elvén megoldó *algoritmusok*at rendel
  - az algoritmusok a *programkonstrukció*k, vagy vezérlési szerkezetek (szekvencia, elágazás, ciklus) segítségével épülnek fel
  - a megoldó algoritmusok az algoritmus állapotterét alkotó *változók* segítségével kommunikálnak egymással
  - a változók változtatásával a program állapotot vált, és a program futása az *állapotváltozás*ok sorozata, amelyek elvezetnek a végállapothoz

A procedurális programozás

Alprogramok

- Az algoritmusok *alprogram*okként (*subroutine*) jelennek meg, és meghatározott részfeladatot végeznek el, lehetnek:
  - *eljárás*ok (*procedure*): valamilyen utasítássorozatot futtat, végeredmény nélkül
  - *függvény*ek (*function*): valamilyen matematikai számítást végez el, és megadja annak eredményét
- Alprogramok meghívhatnak más alprogramokat, és kommunikálhatnak velük adatok átadásával
- Van egy kiemelt szerepű alprogram, amelyet a program indítások futtat, ez a *főprogram*
  - feladata a teljes program vezérlése és az adatok összefogása, rá hárul a teljes felelősség

A procedurális programozás

A C# nyelvben

- A C# *tisztán objektumorientált programozási nyelv*, amely lehetőséget procedurális módon történő programozásra
  - beleágyazva az objektumorientált környezetbe, ezért bizonyos objektumorientált környezeti elemek kényszerűen megjelennek a procedurális kódban is
  - az egyszerű típusait is felruházza intelligens lehetőségekkel, ezért sok utasítás, vagy érték típuson, vagy változók keresztül érhető el, pl.:

    `<típusnév>.<utasítás>(<paraméretek>)`,

    `<típusnév>.<érték>`,
    `<változónév>.<utasítás>(<paraméterek>)`
  - a programok logikai szerveződésére a névtereket használja

A procedurális programozás

A C# nyelvben

```
namespace MyProgram {
// névtér, a programszerveződés szintje

   class MyClass {
   // típus, a procedurális program határa

       // a procedurális alprogramok helye

       static void Main() { // főprogram

           // a procedurális utasítások helye
       }
   }
}
```

A procedurális programozás

Névterek

- A névterek biztosítják a rendszer strukturáltságát, a típusok és utasítások logikai szerkezetbe foglalását
  - mindennek névtérben kell elhelyezkednie, a keretrendszer által biztosított típusok és utasítások is ezekben helyezkednek el
  - a névterek hierarchikusan egymásba ágyazhatóak, és ezt a névtérben pont elválasztóval jelöljük, pl.:

```
namespace Outer { ... }
namespace Outer.FirstInner { ... }
// a fenti névtéren belüli névtér
namespace Outer.FirstInner.DeepInner { ... }
// a belső névtéren belüli névtér
namespace Outer.SecondInner { ... }
```

A procedurális programozás

Névterek

- A mi projektjeink általában egy névtérben helyezkednek el, de lehetőség van ennek tagolására is
- Névtereket felhasználni a **`using <névtér>`** utasítással lehet, ekkor a névtér összes típusa és utasítása elérhető lesz
  - pl.: **`using System;`**
    `using System.Collections.Generic;`
  - az utasítás a teljes fájlra vonatkozik, így általában a névtér-használattal kezdjük a kódfájlt
  - a típusnév előtt is megadhatjuk a használandó névteret (így nem kell **`using`**), pl.: **`System.Collections.Stack s;`**
  - típusnév ütközés esetén mindenképpen ki kell írnunk a teljes elérési útvonalat

A procedurális programozás

Adatok

- Az adatok kétféle fajtáját tarjuk nyilván:
  - *konstans* (*constant*): értéke és típusa rögzített, nem változhat a program futása során
  - *változó* (*variable*): értéke (esetleg típusa is) változtatható a program futása során, mindig a memóriában tároljuk, és megfelelő *azonosító*val (változónév) látjuk el
- Minden adat a programban meghatározott típussal rendelkezik, amely megadja, milyen értékeket vehet fel és milyen műveletek végezhetőek rajta
  - vagyis a típus megadható egy értékhalmaz és egy művelethalmaz rendezett párjaként
  - Pl.: `bool = ({and, or, not} , {true, false})`

Adattípusok

Primitív típusok

- A típusoknak két csoportját tartjuk nyilván:
  - *Primitív típus*nak nevezzük a nyelv alaptípusait, amelyek a központi könyvtárban vannak megvalósítva, ezek a következők:
    - logikai: **`bool`**
    - előjeles egész számok: **`sbyte`**, **`short`** , **`int`** , **`long`** ,
    - előjel nélküli egész számok: **`byte`**, **`ushort`**, **`uint`**, **`ulong`**
    - lebegőpontos számok: **`float`**, **`double`**
    - fixpontos szám: **`decimal`** ($1.0*10^{-28}$ - $7.9*10^{28}$)
    - karakter: **`char`**
    - objektum: **`object`**

Adattípusok

Összetett típusok

- *Összetett típus*nak nevezzük azokat, amelyek már létező típusok, valamint típuskonstrukciók (iterált, direktszorzat, unió) segítségével valósulnak meg
  - ezekből néhány található a központi könyvtárban, pl.: **`string`**
  - rengeteg található további beépített könyvtárakban, pl.: **`Stack`**, **`Console`**, **`StreamReader`**
- Minden rövidített C# típusnév ekvivalens egy .NET típusnévvel, amely a **`System`** névtérben található, pl.:

```
bool == System.Boolean,
int == System.Int32,
float == System.Single,
object == System.Object
```

Adattípusok

Példányosítás

- Változókat bárhol létrehozhatunk a programkódban a típus, a név, illetve a kezdőérték megadásával
  - pl.: **`Int32 myInt = 10;`**
  - a kezdőérték megadása nem kötelező, de a változó addig nem használható fel, amíg nem kap értéket (ez történhet beolvasással is)
  - összetett típusok (pl. tömbök) esetén használni kell a **`new`** utasítást a létrehozáshoz
- Konstansokat pusztán az érték leírásával hozhatunk létre, vagy elnevezett esetben használnunk kell a **`const`** kulcsszót, ekkor kötelező a kezdőérték megadása
  - pl.: **`const Int32 myConstInt = 10;`**

Adattípusok

Operátorok

- A típusok műveleteik egy részét *operátor*okon keresztül végzik el, amelyek egyszerűsítik a művelet meghívását
- Az operátorok olyan utasítások, amelyek vezérlőkaraktereken, vagy kulcsszavakon keresztül érhetőek el
  - a meghívás rögzített formában történik (*prefix*, *postfix* vagy *infix* jelölés mellett)
  - az operátorok *precedenciá*val rendelkeznek, amely halmozás esetén megszabja a hívási sorrendet
  - az operandusok száma minden esetben rögzített, vannak egy-, két-, illetve háromoperandusú műveletek, és ez rögzített minden operátorhoz (a **+** és **–** operátortoknak van egy-, illetve kétoperandusú változata is)

Adattípusok

A leggyakrabban használt operátorok

- A leggyakrabban használt operátorok:
  - *aritmetikai*: összeadás (`a + b`), negáció (`-a`), kivonás (`a - b`), szorzás (`a * b`), osztás (`a / b`), maradékképzés (`a % b`), értéknövelés (`a++`, `++a`), értékcsökkentés (`a--`, `--a`)
  - *értékadás*: egyszerű (`a = b`), összetett (`a += b`, `a -= b`, `a *= b`, `a /= b`, `a %= b`, `a <<= b`, `a >> b`, `a &= b`, `a |= b`, `a ^= b`)
  - *logikai*: érték összehasonlítás (`a < b`, `a > b`, `a <= b`, `a >= b`, `a == b`, `a != b`), tagadás (`!a`), és (`a && b`), vagy (`a || b`)
  - *indexelés* (`a[b]`)
  - *feltételes kiértékelés* (`a ? b : c`)

Adattípusok

Operátorok precedenciája

- Az operátorok *precedenciája* meghatározza, a műveletek halmozása esetén milyen sorrendben történik a végrehajtás
  - a precendencia típusfüggetlen, és rögzített
  - a magasabb precendenciájú hajtódik előbb végre, ezen túl a végrehajtás balról jobbra történik (kivéve az értékadást, amely jobbról balra történik)
  - zárójelek használatával befolyásolhatjuk a végrehajtási sorrendet
- Pl.:

  `a * b - c == 7` jelentése: `((a * b) - c) == 7`

  `c = a == b % 7 % 2` jelentése: `c = (a == ((b % 7) % 2))`

  `++a++` jelentése: `++(a++)`

  `a = b = c = 3` jelentése: `a = (b = (c = 3))`

Adattípusok

Intelligens típusok

- Már a primitív típusok is intelligensek C#-ban, azaz támogatnak számos műveletet és speciális értéklekérdezést a típusokon, illetve változókon keresztül, pl.:
  - szöveggé alakítás bármely típusra: **`intValue.ToString()`**
  - speciális értékek lekérdezése: **`Int32.MaxValue`**, **`Double.NaN`**, **`Double.PositiveInfinity`**, **`String.Empty`**
  - konverziós műveletek: **`Double.Parse(myString)`**
  - karakter átalakító és lekérdező műveletek: **`Char.ToLower(myChar)`**, **`Char.IsDigit(myChar)`**
  - szöveg átalakító és lekérdező műveletek: **`myString.Length`**, **`myString.Find(char)`**, **`myString.Replace(oneChar, ahotnerChar)`**

Adattípusok

Konstansok példányosítása

- A nem elnevezett konstansok a megfelelő automatikus típust kapják meg, ez módosítható karakterliterálok (**`L`**, **`U`**, **`F`**, …) segítségével, pl.:

```
10 // típusa Int32 lesz
10L // típusa Int64 lesz az L karakter miatt
10.0 // típusa Double lesz
10.5 // típusa Double lesz
10.5F // típusa Single lesz az F karakter miatt
```

- Ezen konstansok is megkapják a típus összes utasítását, így ők is intelligensek lesznek, pl.:

```
10.ToString() // eredménye: "10"
"Hello World".SubString(0, 5)
   // eredménye: "Hello"
```

Adattípusok

Típuskonverziók

- A C# nyelv *szigorúan típusos*
  - minden értéknek fordítási időben ismert a típusa
  - az implicit (automatikus) típuskonverziók korlátozva vannak a nagyobb típushalmazba
    - pl.: **`byte`** ⟶ **`short`**, **`ushort`**, **`int`**, …, **`double`**

      `int` ⟶ **`long`**, **`float`**, **`decimal`**, **`double`**

      `float` ⟶ **`double`**
  - nem lehet automatikus konverzióra támaszkodni olyan típusok között, ahol nem garantált, hogy nem történik értékvesztés, és ez fordítási időben kiderül
    - pl.: **`float`** ⟶ **`int`**
    - ekkor explicit konverziót kell alkalmaznunk

Adattípusok

Típuskonverziók

- az explicit típuskonverzió fordítási időben felügyelt, és kompatibilitást ellenőriz, pl.:

```
int x; double y = 2, string z;
x = (int)y; // engedélyezett
z = (string)y;
// HIBA, int és string nem kompatibilisek
```

- tetszőleges primitív típuskonverzióra a **`Convert`** típus műveletei használhatóak, illetve szövegre történő konverzió több módon is elvégezhető:

```
int x; double y = 2, string z;
x = Convert.ToInt32(y);
z = Convert.ToString(y); // z = y.ToString();
x = Int32.Parse(z); // x = Convert.ToInt32(z);
```

Adattípusok

Példa

Feladat: Kérjünk be két valós számot a konzol képernyőn, és írjuk vissza az összegüket.

- a valós számok közül használjuk az egyszeres pontosságút (**`Single`**), a változóink legyenek **`a`** és **`b`**
- a konzol képernyőről beolvasni a **`Console.ReadLine()`** utasítással tudunk, amely szöveget ad vissza, így azt konvertálnunk kell (**`Convert.ToSingle`**)
- a konzolra kiírni a **`Console.Write()`** és **`Console.WriteLine()`** utasításokkal tudunk bármilyen típusú változót, vagy konstansot
- a kiíráshoz szöveget is társítunk, amelyhez a **`+`** operátor segítségével tudunk további értékeket fűzni

Adattípusok

Példa

Megoldás:

```
using System; // felhasznált névtér

namespace SimpleSummation { // saját névtér
   class Program {

      static void Main(string[] args) {
         Single a, b, c;
         // a két beolvasandó szám, valamint az
         // eredmény

         Console.Write("Kérem az első számot: ");
         a = Convert.ToSingle(Console.ReadLine());
         // beolvasás és konvertálás
```

Adattípusok

Példa

Megoldás:

```
            Console.Write("Kérem a második számot:
                          ");
            b = Convert.ToSingle(Console.ReadLine());

            c = a + b; // összeadás elvégzése
            Console.WriteLine("A két szám összege: "
                              + c); // kiírás
            Console.ReadKey(); // várakozás
        }
    }
}
```

Adattípusok

Példa

Feladat: Olvassunk be egy szöveget, írjuk ki a hosszát, valamint magát a szöveget nagy kezdőbetűvel és ponttal a végén.

- a szöveg hosszát lekérdezni a **`Length`** utasítással tudjuk
- nagy kezdőbetűt a **`Char.ToUpper()`** utasítással tudunk kialakítani, amely megkapja az átalakítandó karaktert
- szöveg egy karakterét a **`[ ]`** operátorral tudjuk lekérdezni, ennek meg kell adni a karakter sorszámát 0-tól kezdődően (azaz pontosabban azt adjuk meg, hány hellyel van arrébb a karakter a kezdőpozícióhoz képest)
- a kiolvasott 0. karakter után következik a többi rész, amelyet a **`SubString()`** utasítással kérünk le, majd a **`+`** operátorral hozzáfűzzük a pontot

Adattípusok

Példa

Megoldás:

```
static void Main(string[] args){
    String myString;

    Console.Write("Kérem a szöveget: ");
    myString = Console.ReadLine();
    // szöveg beolvasása

    Console.WriteLine("A szöveg hossza: "
                      + myString.Length);
    // hossz lekérdezése és kiírása
```

Adattípusok

Példa

Megoldás:

```
        myString = Char.ToUpper(myString[0]) +
                   myString.Substring(1) + ".";
        // első karakter nagybetűssé alakítása,
        // valamint a további szöveg és a pont
        // hozzáfűzése

        Console.WriteLine("Az eredmény: " + myString);
        Console.ReadKey();
    }
```

Adattípusok

Tömbök

- Amennyiben egy típusú elemből sokat szeretnénk eltárolni, az iterált típuskonstrukciót használjuk, a programozási nyelvekben ennek megvalósítását a *tömb*ök jelentik
- A tömb tehát azonos típusú elemek rögzített hosszúságú sorozata, amely egy összetett típust fog adni
  - bármely elemét elérhetjük, lekérdezhetjük, módosíthatjuk, de elemeket nem vehetünk hozzá, vagy törölhetünk belőle
  - az elemek indexszel rendelkeznek, ami 0-tól indul (hasonlóan a **`String`** típus karaktereihez)
  - létrehozása elemszámmal:

```
<típusnév>[] <változónév> =
    new <típusnév>[<elemek száma>];
```

Adattípusok

Tömbök

- létrehozása elemekkel:
  <*típusnév*>[] <*változónév*> =
  new <*típusnév*>[] { <*elemek felsorolása*> };
- elemelérése: **<*változónév*>[<*index*>]**
- méret lekérdezése: **<*változónév*>.Length**

- A nyelv lehetőséget ad több dimenziós tömbök létrehozására is (mátrixok, térbeli mátrixok), ekkor pusztán fokoznunk kell az indexek és a méretek számát, vesszővel elválasztva
  - mátrix létrehozása elemszámmal:
    <*típusnév*>[,] <*változónév*> =
    new <*típusnév*>[<*oszlopszám*>, <*sorszám*>];
  - mátrix elemelérése: **<*változónév*>[<*oszlop*>, <*sor*>]**

Elemi programok

Vezérlési szerkezetek

- A legtöbb program nem írható le utasítások sorozataként, ekkor jönnek képbe a *vezérlési szerkezetek*, amelyek az utasításabsztrakció alapvető eszközei
  - *szekvencia*: utasítások egymásutánja, ahol a ; tagolja az utasításokat
  - *programblokk*: **`{ <utasítások> }`** utasítások csoportosítására szolgál, valamint meghatározza a változók élettartamát (a blokkban létrehozott változók csak a blokkon belül érhetőek el)
  - *elágazás*: lehetővé teszi valamilyen feltétel függvényében különböző tevékenységek végrehajtását, esetei:

Elemi programok

Vezérlési szerkezetek

- *kétágú elágazás*:

```
if (<feltétel>) // logikai típusú feltétel
   <utasítás>; // igaz ág
else
   <utasítás>; // hamis ág
```

  - ha a feltétel teljesül, az igaz ág kerül végrehajtásra, különben a hamis ág
  - a hamis ág elhanyagolható
  - a csellengő **`else`** mindig az utolsó elágazáshoz tartozik
  - felváltható triáris operátor használatával:

```
<feltétel> ? <utasítás>; : <utasítás>;
```

Elemi programok

Vezérlési szerkezetek

- *többágú elágazás*:

```
switch(<változó>){
    case <konstans> : <utasítások>; break;
    …
    default: <utasítások>; break;
}
```

  - egy adott változó értéke függvényében kerülnek különböző ágak végrehajtásra
  - alkalmazható egész, karakter és szöveg típusú változókra
  - alapértelmezett (**`default`**) ág nem kötelező
  - a lezárás (**`break`**), vagy továbbadás (**`goto`**) kötelező

- ciklusok: utasítások (ciklusmag) többszöri végrehajtására alkalmasak valamilyen feltétel (ciklusfeltétel) függvényében
  - *számláló ciklus*:
    `for (<inicializálás>; <feltétel>; <léptetés>)`
    `<utasítás>;`
    - előre meghatározott, hányszor futtatja le a ciklusmagot
    - a léptetés során egy általában egész típusú változót (ciklusszámláló) növel, vagy csökkent, amíg az el nem ér egy megadott küszöböt

- *előtesztelő ciklus*:
  `while(<feltétel>) <utasítás>;`
  - először ellenőrzi a feltételt, és ha az igaz, lefuttatja a ciklusmagot, majd ezt futtatja körkörösen, amíg a felvétel hamissá nem válik
- *utántesztelő ciklus*:
  `do <utasítás>; while(<feltétel>);`
  - először mindenképpen lefuttatja a ciklusmagot, majd ellenőrzi a feltételt
  - az első futtatást követően ugyanúgy hajtódik végre, mint az előtesztelő ciklus

Elemi programok

Vezérlési szerekezetek

- *bejáró ciklus*:
  foreach(<*deklaráció*> in <*kifejezés*>)
  <*utasítás*>;
  - egy gyűjtemény értékein tud végighaladni
  - kifejezés értékének kell **`GetEnumerator()`** metódus (az **`IEnumerable`** interfészből)
- ciklusból kilépés bármikor lehetséges a felvételtől függetlenül (**`break`**), valamint feltétel kiértékeléshez történő ugrás (**`continue`**) is lehetséges

Elemi programok

Példa

Feladat: Olvassunk be 10 egész számot a konzolról, és írjuk vissza a páros számokat a képernyőre.

- az adatok tárolására egészeket tároló tömböt használunk, mérete 10 lesz
- két számláló ciklusra van szükségünk, az elsővel beolvassuk az elemeket, a másodikkal visszaírjuk őket a képernyőre
- a második ciklusban elágazást használunk, amely csak páros szám esetén végez kiírást (ezért a hamis ágra nem lesz szükségünk), az elágazás feltétele a kettővel való oszthatóság lesz

Elemi programok

Példa

Megoldás:

```
   static void Main(string[] args){
      Int32[] values = new Int32[10];

      // beolvasás
      Console.Write("Kérem a számokat: ");
      for (Int32 i = 0; i < values.Length; i++)
         // számláló ciklus, amely a tömb végéig
         // megy

         values[i] =
            Convert.ToInt32(Console.ReadLine());
            // belül használhatjuk a ciklusszámlálót
```

Elemi programok

Példa

Megoldás:

```
        // kiírás
        Console.WriteLine("Páros számok:");
        for (Int32 i = 0; i < values.Length; i++)
            // használhatjuk ugyanazt a ciklusváltozót

            if (values[i] % 2 == 0)
               // elágazás, a feltétel a párosság

               Console.WriteLine(values[i]);

        Console.ReadKey();
    }
```

Elemi programok

Algoritmusok

- *Algoritmus*nak nevezzük azt a műveletsorozatot, amely a feladat megoldásához vezet
  - a program lényegi része, amely nem tartalmazza az adatok beolvasását és kiírását
  - egy programban több algoritmus is szerepelhet, amelyek valamilyen kombinációja oldja meg a feladatot
- A megoldandó feladatokban gyakorta fedezünk fel hasonlóságokat (pl. többen valamit kell keresni)
  - ennek köszönhetően a megoldó algoritmusuk is teljesen hasonló, csupán néhány eltérést fedezhetünk fel közöttük
  - általában a megfelelő adat, illetve feltétel változtatásokkal megkapjuk az új feladat megoldását a korábbi alapján

Elemi programok

Algoritmusok

- Az algoritmusokat ezért célszerű általánosan (absztraktan) megfogalmazni, hogy a változtatások (transzformációk) könnyen véghezvihetők legyenek
  - amennyiben a feladatra találunk megoldó algoritmust, és azt átalakítjuk az aktuális feladatra, akkor azt mondjuk, hogy a feladatot *visszavezettük az algoritmus*ra
  - az algoritmus lehet nagyon egyszerű (pl. szám szorzása), és nagyon összetett
- Az algoritmust két részre szeparáljuk:
  - *inicializálás*: változók kezdőértékeinek megadása
  - *feldolgozás* (mag): műveletvégzés a bemenő adatokkal és az inicializált változókkal

Elemi programok

Programozási tételek

- Algoritmusokat azért célszerű használni, mert jó, bizonyított megoldását adják a feladatnak, nem kell újabb algoritmust kitalálni
  - már több ezer algoritmus létezik, amelyek mind nevesítettek
  - az algoritmusok bemenete általában egy adatsorozat, vagyis adatok egymásutánja (pl. tömbben)
- Az egyszerű, sorozatokra alkalmazott algoritmusokat nevezzük *programozási tételek*nek, ezek a következők:
  - összegzés, számlálás
  - lineáris keresés, bináris keresés
  - maximum keresés, feltételes maximumkeresés
  - elemenkénti feldolgozás

Elemi programok

Összegzés

- Az *összegzés programozási tétele* lehetővé teszi tetszőleges sorozat $(a_1, \ldots, a_n)$ adott függvény (f) szerint vett értékének összesítését (sum)

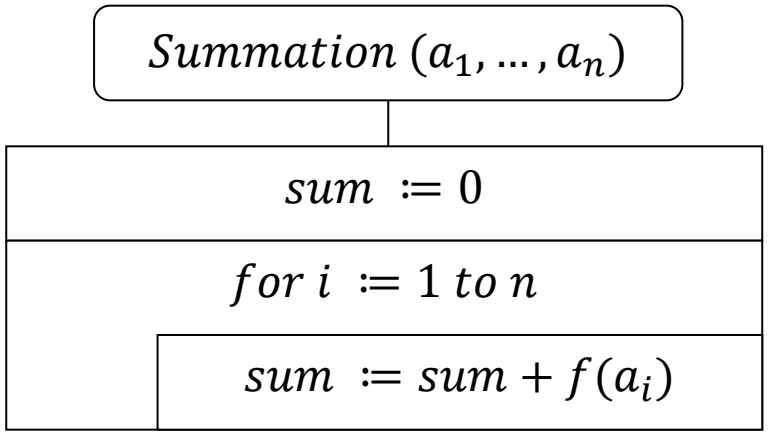
$$sum = \sum_{i=1}^{n} f(a_i)$$

  - az összegzés egy ciklusban történik, az összeget egy külön változóhoz adjuk hozzá minden lépésben, amelyet egy kezdeti értéken inicializálunk
  - általában az összegző művelet az összeadás, ekkor az összeg változó 0-ról indul
  - a függvény általában az identitás, de lehet nagyon összetett is

Elemi programok

Összegzés

- Az összegzés absztrakt megfogalmazása:



- A ciklus nem csak számláló lehet, hanem bármilyen feltétellel vezérelt előtesztelő
- A konkrét feladattól függően az egyes konstansok és műveletek változhatnak, pl. faktoriális számítás esetén az összeg 1-től indul, a művelet a szorzás

Elemi programok

Példa

Feladat: Adjuk meg egy pozitív egész szám faktoriálisát.

- alkalmazzunk összegzést, amelyben a szorzás műveletét használjuk
- készüljünk fel arra, hogy a felhasználó nem garantált, hogy pozitív számot ad meg, ezért egy elágazással előbb válasszuk le a hibás eseteket, és írjunk ki figyelmeztető üzenetet
- az eredményváltozót egy nagyobb értékhalmazban vesszük fel (`Int64`), hogy garantáltan elférjen az eredmény

Elemi programok

Példa

Megoldás:

```
   static void Main(string[] args){
      Int32 number;
      Int64 sum; // az eredményhez nagyobb
                 // értéktartományt veszünk

      Console.WriteLine("Kérek egy pozitív egész
                         számot: ");
      number = Convert.ToInt32(Console.ReadLine());

      if (number <= 0)
         Console.WriteLine("Mondom, pozitív egész
                            számot!");
```

Elemi programok

Példa

Megoldás:

```
      else {
         // itt már programblokk szükséges, mivel
         // több utasítás is helyet kap

         sum = 1; // összegzés
         for (Int32 i = 1; i <= number; i++)
            sum *= i; // vagy sum = sum * i;

         Console.WriteLine("A szám faktoriálisa: " +
                                   sum);
      }
      Console.ReadKey();
   }
```