A **digitális számítógép** egy elektronikus eszköz, amely adatok fogadására, feldolgozására, továbbítására, tárolására, megjelenítésére alkalmas. Rendelkezik memóriával és utasításfeldolgozó egységgel.

Egy működő számítógépes egységhez szükséges:

**Hardver**: a számítógép fizikai összetevői

**Szoftver**: adatok, programok, programdokumentációk

**Program**: a számítógép (processzor) által feldolgozható gépi kódú utasítások sorozata.

Minden processzornak van egy **utasításkészlet**e és csak azokat a programokat tudja végrehajtani, amik az ő utasításkészletéből állnak.

Programvégrehajtás menete:

* A program utasításai és adatai betöltődnek a memóriába.
* A program utasításait egymás után végrehajtja a processzor.

Utasítások végrehajtása:

* A következő beolvasandó utasítás memóriacímét a programszámláló regiszterben tárolja.
* A végrehajtandó utasítása a processzor vezérlőegységében elhelyezkedő utasításregiszterbe kerül.
* A vezérlőegység értelmezi az utasítást és ennek megfelelően irányítja a gép többi részegységét.

Műveletvégzés:

Ha valamilyen műveletet kell végezni, akkor azt az aritmetikai, logikai egység (ALU) végzi.

Az adatokat a memóriában vannak, műveletvégzés előtt a processzor regisztereibe (akkumulátor regiszter, általános célú regiszterek) kerülnek.

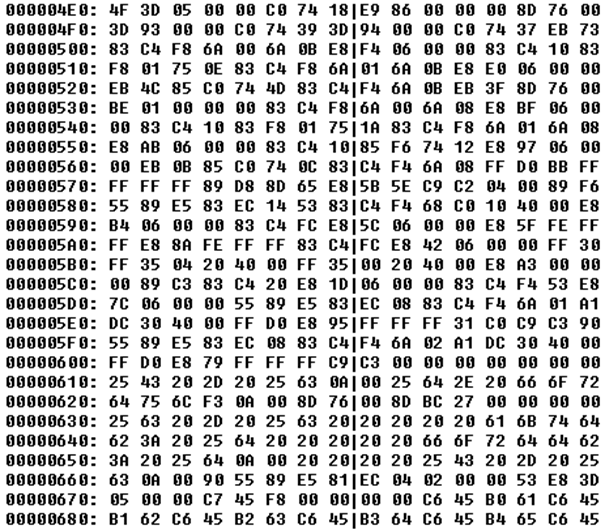
Utasítás általános felépítése:

|  |  |  |  |
| --- | --- | --- | --- |
| Műveleti kód | 1. operandus (címe) | 2. operandus (címe) | eredmény címe |

Az operandusok és az eredmény címe elhagyható, ha nincsenek vagy előre meghatározott helyen, például egy adott regiszterben vannak.

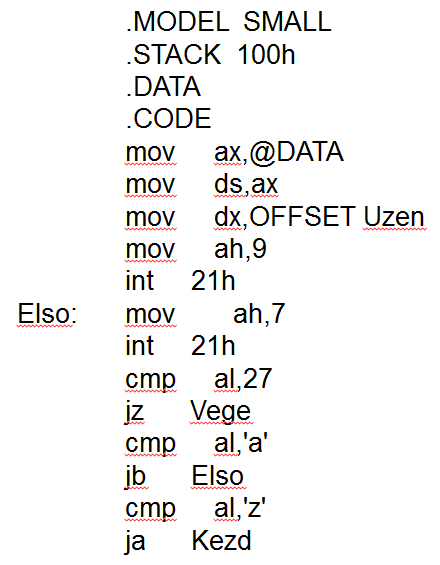
Programozási nyelvek típusai:

**Gépi kód**: közvetlenül a gépi kódú utasításokat írunk.



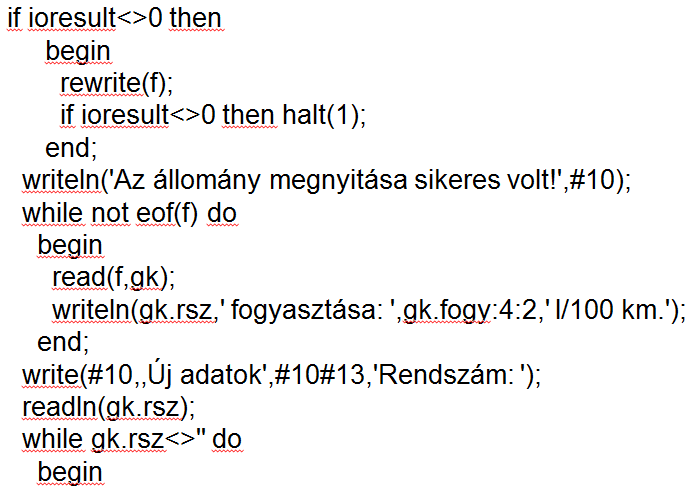
**Assembly nyelv**:

A gépi kódú utasítások helyett néhány karakteres utasításneveket és paramétereket használunk. Közvetlenül nem futtatható, fordításra van szükség.



**Magas szintű programozási nyelvek**:

Könnyebb velük programot írni, használhatók a vezérlési szerkezetek, a fordítás viszont bonyolultabb.



Futtatható program létrehozása:

a programkód elkészítése valamilyen programozási nyelven 🡪 fordítás egy fordítóprogrammal 🡪 futtatható (gépi kódú) program

A fordítás végeredményeként kapott program a végrehajtás szempontjából lehet natív vagy menedzselt kód.

* **Natív kód**: minden további feltétel nélkül futtatható. (pl. C, C++)
* **Menedzselt kód**: a processzor az utasításokat nem tudja közvetlenül végrehajtani, kell valamilyen futtató környezet vagy programkönyvtár. (pl. Java, C#)

**Forráskód**: egy programnyelven létrehozott kód, ami általában egy szöveges adatállomány.

Fordítóprogramok típusai:

* **fordító** (**compiler**): a program forrásszövegéből egy gépi kódú, futtatható programot (pl. exe) állít elő, vagyis a programot lefordítja a gép nyelvére
* **értelmező** (**interpreter**): a program forrásszövegét sorról sorra értelmezi, ellenőrzi, és (ha tudja) végrehajtja

Programozási hibák, hibakeresés

* **szintaktikai hiba** (alaki): a program nem felel meg az adott nyelv szabályainak. Pl. elírás, elfelejtettünk egy zárójelet... Az ilyen hibákat viszonylag könnyű megtalálni, általában a fordító is jelzi.
* **szemantikai hiba** (tartalmi): a program lefordul, elindul, de nem azt csinálja, amit kell, esetleg menet közben le is áll a hiba miatt. (Ezeket nevezik bug-nak, ami az angol poloska szóból származik.) Az ilyen hibák megtalálása és javítása (debuggolás) sokkal nehezebb feladat, erre külön segédprogramok (debugger-ek) állnak rendelkezésre. Szemantikai hiba pl. a 0-val való osztás, rossz adatra hivatkozás, nem megfelelő zárójelezés, stb.

**A C# programozási nyelv**

Az 1990-es években a Microsoft fejlesztette ki a C++ és a Java nyelveket alapul véve.

Magas szintű programnyelv, teljes mértékben objektumorientált.

A C# programok nem natív kódúak, futtatásukhoz szükséges a .NET keretrendszer.

**A C# nyelv alapelemei**

* kulcsszavak
* operátorok
* konstansok v. literálok
* azonosítók
* írásjelek
* térközök

**A C# nyelv jelkészlete**

A C# nyelv a Unicode karakterábrázolásra épül, tehát elvileg tetszőleges karakter használható benne, de célszerű, a betűket tekintve, az angol ábécé karaktereit használni.

**Kulcsszavak** (foglalt szavak)

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 readonly ref return sbyte

sealed short sizeof stackalloc static string struct switch this

throw true try typeof uint ulong unchecked unsafe ushort

using virtual void volatile while

**Operátorok** (műveletek, műveleti jelek):

! != % %= & && &= () \* \*= + ++ += , - -- -=

-> . / /= < <= << <<= = == > >= >> >>= ? [] ^

^= sizeof | |= || ~ ?: :: new checked unchecked defaut delegete

**Térközök**

Olyan karakterek, amelyek üres helyeket biztosítanak a kódban: *szóköz, tabulátor, újsor*.

A fordító a többszörös és sok esetben az egyszeres térköz karaktereket figyelmen kívül hagyja, ezért alkalmasak a forráskód formázására, tagolására, olvashatóbbá és átláthatóbbá tételére.

**Megjegyzések**

Az egysoros megjegyzés:

// Megjegyzés, ami a sor további részére érvényes

Több soros megjegyzés (egy soron belül is használható): /\* és \*/ jelek között található szöveg.

/\* Megjegyzés 1. sora

2. sora

…

megjegyzés vége \*/

Egy soron belül is használható:

programkód /\* Megjegyzés \*/ további programkód;

Dokumentációs megjegyzés: Automatikus külső dokumentálásra alkalmazható megjegyzés, használhatók bent XML kódok is.

/// dokumentációs megjegyzés

Egy programozási nyelv szintaxisát a forráskód felépítésének szabályai alkotják. A C# nyelv un. C stílusú szintaxissal rendelkezik. Ennek legáltalánosabb szabályai:

* Minden utasítást pontosvessző (;) zár le.
* A kicsi és nagy betűk különböző karaktereknek számítanak.
* A különböző programegységeket (ciklusmag, függvénytörzs, osztály elemei, stb.) blokkokat alkotnak, amik {} között állnak.

**Adattípusok**: A C# programokban használt adatok mindegyike valamilyen adattípusba sorolható.

**Egyszerű adattípusok:**

**bool** logikai típus

**char** karakteres típus

**int** egész szám

**float** valós szám (lebegőpontos szám)

**double** valós szám (lebegőpontos szám)

**decimal** valós szám (lebegőpontos szám)

Összetett adattípus:

**string** szöveges adat, Unicode karakterek sorozata

**Egész típusok fajtái:**

Tárolási hossz: az egész számokat tároló típusok különböző méretűek lehetnek: 2, 4, 8 bájt. Elnevezésük: short és long.

Előjel: az egész számok lehetnek előjelesek (pozitív és negatív), illetve előjel nélküliek (0 és annál nagyobb egészek). Elnevezésük: signed, unsigned

Adattípusok jellemzői:

* Minden adattípusnak van egy értékkészlete, ami meghatározza, hogy milyen és hány féle adat tárolható.
* Az adatok a memóriában tárolja a (futó) program. Minden adat a típusának megfelelő méretű helyet (valahány bájtot) foglal a memóriában.
* Lebegőpontos számoknál az eltárolható számjegyek száma, amit pontosságnak hívunk.

**Adattípusok tulajdonságai:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adattípus neve | Értékkészlet | Méret (byte) | Pontosság (számjegy) | Leírás |
| **bool** | false, true | 1 |  | logikai adattípus |
| **char** | 0..65535 | 2 |  | egy Unicode karakter |
| **byte** | 0..255 | 1 |  | előjel nélküli 8 bites egész szám |
| **sbyte** | -128..127 | 1 |  | előjeles 8 bites egész szám |
| **short** | -32768..32767 | 2 |  | előjeles 16 bites egész szám |
| **ushort** | 0..65535 | 2 |  | előjel nélküli 16 bites egész szám |
| **int** | -2147483648..2147483647 | 4 |  | előjeles 32 bites egész szám |
| **uint** | 0..4294967295 | 4 |  | előjel nélküli 32 bites egész szám |
| **long** | -9223372036854775808..9223372036854775807 | 8 |  | előjeles 64 bites egész szám |
| **ulong** | 0..18446744073709551615 | 8 |  | előjel nélküli 64 bites egész szám |
| **float** | ±1.5\*10-45..3.8\*10+38 | 4 | 6 | egyszeres pontosságú lebegőpontos szám |
| **double** | ±5\*10-324..1.7\*10+308 | 8 | 15 | kétszeres pontosságú lebegőpontos szám |
| **decimal** | 1\*10-28..7.9\*10+28 | 16 | 28 | fix pontosságú 28+1 jegyű szám |

**Konstans vagy literál**: a programba beírt konkrét adat, pl. szám, karakter, szöveg.

Egész számok:

* 10-es számrendszerben megadva: 9123 vagy -109
* 16-os számrendszerben megadva: 0x3AF (0x-el kezdődik)

Típusmódosítók: hosszú egész (long): l vagy L; pl.: 18L;

Előjel nélküli egész: u vagy U. Pl.: 456754u

Lebegőpontos szám: 12.547; 5.0; 1.475e2; 12.E3

Karakter konstans: ’a’

Sztring (szöveg) konstans: ”Szöveg”

**Azonosító**: a program összetevőinek a neve (pl. változók, függvények).

Azonosító képzésének a szabályai:

* Az angol ábécé betűi, számjegyek és \_ karakter szerepelhet benne.
* Az első karakter betű vagy \_.
* A kis és nagy betűk különbözőnek számítanak.
* Hossza tetszőleges, de a fordító csak az első 32 karaktert veszi figyelembe.
* Nem lehet foglalt szó.

**Változó**: névvel ellátott memóriaterület, ami a programból elérhető, tartalma megváltoztatható.

Minden változónak van típusa, valamint neve, és ezek megadásával deklarálni kell.

**Változó deklarálása:**

típusnév változónév;

pl.: double szam;

Több, azonos típusú változó deklarálása:

típusnév változónév1, változónév2, változónév3, …;

pl.: int a1, b1, a2, b2;

A változónévre az azonosítók képzési szabályai vonatkoznak.

A deklarációkat az eredeti C szabvány szerint a programkód elején kell megadni, de a C# nyelv bárhol megengedi. Az átláthatóság miatt célszerű a kód elején bevezetni a változókat.

Deklarálás után a változó értéke véletlen (a memóriában lévő véletlen bitsorozat). Ilyen változó használatát a C# fordító nem is engedélyezi.

A deklaráláskor értéket is adhatunk a változónak. Ez a kezdeti értékadás.

Változó deklarálása kezdeti értékadással:

típusnév változónév = érték;

pl.: char ch = ’a’; int min = 0;

Változók értékének megváltoztatása értékadással történik. Az értékadás egy operátor, aminek a jele az = jel.

Az értékadó utasítás felépítése:

változó = kifejezés;

* bal oldalán egy változónak kell lenni, ami értéket kaphat,
* jobb oldalán egy kifejezés áll, amit a program kiértékel és a kapott eredményt eltárolja a változóba. A kifejezés lehet konstans, változó, függvény, illetve ezek, műveleti jelekkel összekapcsolva.

pl.: a\_oldal = b\_oldal + 4.6;

x = (a + c - 2) \* 7;

y = y + 5;

Működése: Először a jobb oldali kifejezés értéke lesz kiszámolva, majd ez a bal oldali változóba kerül. Ezért érvényes az y = y+5; értékadás is: Először növeljük az y értékét 5-tel, majd ez a szám kerül az y változóba.

Többszörös értékadás:

változó\_1 = változó\_2 = változó\_3 = érték;

pl.: b = c = d = 7;

A végrehajtás iránya jobbról balra, vagyis először d változó értéke lesz 7, aztán a c, majd végül a b változó is felveszi a 7-et.

Műveletek prioritása (végrehajtási sorrendje)

Operátorok:

|  |  |  |
| --- | --- | --- |
| **Művelet** | **Operátor** | **Kiértékelési sorrend** |
| Egyváltozós | ! ~ - ++ -- (típus) | Jobbról balra |
| Multiplikatív | \* / % | Balról jobbra |
| Additív | + - | Balról jobbra |
| Viszonyító | < <= > >= | Balról jobbra |
| Egyenlőségi | == != | Balról jobbra |
| Feltételes AND | && | Balról jobbra |
| Feltételes OR | || | Balról jobbra |
| Feltételes művelet | ?: | Jobbról balra |
| Értékadás | = += -= \*= /= %= <<= >>= &= |= ^= | Jobbról balra |

A végrehajtási sorrendet zárójelezéssel módosíthatjuk.

Matematikai műveletek:

összeadás: +

kivonás: -

szorzás: \*

osztás: /

maradékképzés: %

előjel módosítása: - (pl. int a=10; esetén -a értéke -10 lesz)

Egész típusú (int, char) változók értékének 1-gyel növelése és csökkentése a ++ és -- operátorokkal lehetséges.

|  |  |
| --- | --- |
| int v = 5;  ++v; 🡪 v változó értéke 6 lesz | int z = 9;  --z; 🡪 z változó értéke 8 lesz |

A ++ és a -- állhat a változó előtt és után is. Különbség köztük akkor van, ha nem önállóan, hanem összetett kifejezésben használjuk.

Az előre vetett alaknál először történik meg a növelés/csökkentés és a módosított értékkel számolunk.

A hátravetett alaknál a változó eredeti értéket használjuk fel és csak ez után növeljük/csökkentjük.

|  |  |
| --- | --- |
| int a, b = 5;  a = ++b; 🡪 a és b változó értéke is 6 lesz | int c, d = 8;  c = d++; 🡪 z változó 8 lesz, d pedig 9 |

Értékadás egy változónak:

int p, r;

p = 10; 🡪 p értéke 10 lesz

r = p + 1; 🡪 r értéke 10+1 = 11 lesz

p = p + 5; 🡪 az utasítás végrehajtása: a jobb oldali kifejezést kiszámolja a program és a bal oldali változónak ez lesz az új értéke, vagyis p értéke 15 lesz.

A p = p + 5 értékadásokat rövidíteni is lehet C-ben a += operátorral:

|  |  |
| --- | --- |
| **Rövid utasítás** | **Teljes utasítás** |
| p += 5; | p = p + 5; |
| p -= 14; | p = p – 14; |
| p \*= 2; | p = p \* 2; |
| p /= 10; | p = p / 10; |
| p %= 7; | p = p % 7; |

**Konstans változók:**

**const** típus változónév = érték;

A const kulcsszóval megadott változók értéke a program futása során nem módosítható, tartalma csak kezdeti értékadással állítható be.

const double pi = 3.1415;

**Típusátalakítás**

Egész értékekből felépített matematikai műveletek értéke mindig egész szám lesz, akkor is, ha lebegőpontos változónak adjuk eredményül. Ezeknél a műveleteknél az eredmény úgy jön ki, hogy a tizedesrész elveszik. Ezt a C/C++ nyelvben úgy nevezzük, hogy **egész aritmetika**.

pl.: 4/3 értéke a programban 1 lesz.

Ha egy kifejezésben különböző típusú adatok szerepelnek, akkor az eredmény típusa a „bővebb” típus lesz. Vagyis, ha egész és lebegőpontos szám (float vagy double változó, illetve konstans) is van egy műveletben, akkor az eredmény lebegőpontos lesz.

**Típusátalakító operátor**

Egy változónak vagy konstansnak a programban módosíthatjuk a típusát. Az új típust a változó vagy konstans érték elé kell írni zárójelben. Ez a művelet a változó típusát csak az aktuális művelet elvégzésének idejére módosítja.

pl: int x, y; x = 5; y = 4;

esetén x/y értéke 1 lesz, de (double) x/y értéke 1.25 lesz, mert az x változót double típusúnak veszi.

**Egyszerű C# program felépítése:**

using System;

a programban használható névterek

using System.Linq;

using System.Text;

a program névterének megadása

a program osztálya

namespace Programom

{

class Program

{

main függvény, a program kezdőpontja

static void Main(string[] args)

{

double f;

változódeklarációk

int a, b = 5;

char ch;

main függvény törzse,

a program utasításai

f = b / 3.5;

utasítások

a = b \* 3;

Console.WriteLine("a= {0}", a);

Console.ReadKey();

}

}

}

A **using** kulcsszóval névtereket kapcsolunk a forráskódhoz. Az így kijelölt névterek objektumait és függvényeit egyszerűbben (a névtér megadása nélkül) elérjük a programunkban.

A **namespace** definiálja azt a **névteret**, amelyben a programunk adatai érvényesek.

A névtér egy olyan programozási egység, amelyen belül minden azonosítónak egyedinek kell lennie. Különböző névterekben használhatók egyező nevű azonosítók is. A névtér megadásával egyértelmű. hogy melyikre hivatkozunk.

A class kulcsszóval egy objektumosztály definícióját adhatjuk meg. A C# teljes mértékig objektumorientált nyelv, a legegyszerűbb programban is létre kell hozni egy objektumosztályt.

Egy objektumhoz tartozhatnak adatok, amik egy objektum tulajdonságainak is nevezünk, valamit függvények, amik az objektum viselkedését határozzák meg (mit tud csinálni). Ezeket tagfüggvénynek vagy metódusnak nevezzük.

Minden program forráskódja tartalmaz egy darab **main() függvény**t. Ugyan úgy, mint a C/C++ nyelvek esetén, a C#-ban írt programok futása is a main függvény végrehajtásának indításával kezdődik. Ez a függvény a program belépési pontja.

A kódban szereplő {} (kapcsos zárójelpárok) **utasításblok**kok. Logikailag összetartozó programelemeket fogják össze. Pl. egy névtér elemei, egy osztály változói és függvényei vagy egy függvény törzse, vagyis a benne található utasítások. Utasításblokkokat a későbbiekben függvényen belül is fogunk használni.

A beépített függvények meghívásának általános alakja:

*Névtér.Objektumnév.Tagfüggvény(paraméterek)*

A névtér elhagyható, ha a forráskód elején a using kulcsszóval már felvettük.

**A C/C++ program fordítása és futtatása:**

A forráskódot a fordító egy köztes kódra fordítja le, amelynek futtatásához a .NET keretrendszerre van szükség, ez készít futtatáskor végrehajtható natív kódot.

**Adatok kiírása a képernyőre:**

Konzolalkalmazásoknál az alapértelmezett kiviteli eszköz a parancssori ablak, amit a Console objektum Write és WriteLine tagfüggvényével érhetünk el.

Egy szövegkonstans kiírása:

Console.Write(”Szöveg”);

Console.WriteLine(”Szöveg”);

A Write és WriteLine közti különbség, hogy a WriteLine a kiírt szöveg után sortörést tesz, vagyis a kurzor a következő sorba kerül, a Write esetében pedig a szöveg mögött marad.

A paraméterként megadott szövegben escape-szekvenciákat is elhelyezhetünk, amiknek a segítségével különleges karaktereket (sortörés, tabulátor, stb.) írhatunk ki. Alakjuk: \ jel után egy betű.

*Escape-szekvenciák:*

|  |  |  |
| --- | --- | --- |
| **Szekvencia** | **Karakter** | **Funkció** |
| \t | HT | vízszintes tab |
| \n | LF | új sor |
| \r | CR | kocsi vissza |
| \” | ” | macskaköröm |
| \’ | ‘ | aposztróf |
| \\ | \ | fordított per jel |

Változók kiírása:

Ha egy változó értékét szeretnénk kiírni, akkor azt kell megadni paraméterként. A változó típusa ebben az esetben tetszőleges lehet.

double f = 45.265345;

Console.WriteLine(f);

Szöveg és változók kiírása:

Összetettebb adatkiírásnál, ahol több változót vagy szövegkonstanst és egy vagy több változót szeretnénk együtt kiíratni, a Write/WriteLine többparaméteres változatát használhatjuk.

Az első paraméter egy formátumsztring, amely {}-ek között hivatkozásokat tartalmaz a szövegkonstans utáni további paraméterekre, amelyek változók, konstans értékek és műveleteket tartalmazó kifejezések is lehetnek.

A {}-ekben 0-tól kezdődő számokat kell megadni, ahol 0 a formátumszöveg utáni első paramétert jelenti.

int a = 58, b = -4;

char ch = ’$’;

Console.WriteLine(”Első szám: {0}; második szám: {1}; karakter: {2}”, a, b, ch);

Eredmény: Első szám: 58; második szám: -4; karakter: $

**Adatok bekérése a billentyűzetről:**

Adatok beolvasására a Console.ReadLine() függvény használható. A konzolról egy teljes sort olvas be, és azt egy stringben adja vissza.

string s;

s = Console.ReadLine();

Ha nem string adattípust szeretnénk beolvasni, akkor a ReadLine által visszaadott szöveges adatot konvertálni kell, amit a Convert objektum tagfüggvényei végeznek el.

A Convert objektum fontosabb tagfüggvényei:

|  |  |
| --- | --- |
| Céltípus | Átalakító függvény |
| **char** | Convert.ToChar() |
| **byte** | Convert.ToByte() |
| **sbyte** | Convert.ToSByte() |
| **short** | Convert.ToInt16() |
| **ushort** | Convert.UToInt16() |
| **int** | Convert.ToInt32() |
| **uint** | Convert.UToInt32() |
| **long** | Convert.ToInt64() |
| **ulong** | Convert.UToInt64() |
| **double** | Convert.ToDouble() |
| **decimal** | Convert.ToDecimal() |

Adatbekérésnél a függvények paramétere a Console.ReadLine() függvény.

int szam;

szam = Convert.ToInt32(Console.ReadLine());

char ch;

ch = Convert.ToChar(Console.ReadLine());

double f;

f = Convert.ToDouble(Console.ReadLine());

Egy másik megoldás a **típus.Parse()** függvények használata:

int.Parse();

char.Parse();

double.Parse(); stb.

pl.:

szam = int.Parse(Console.ReadLine());

**Véletlen számok előállítása**

Véletlen számok előállításához a System névtér szükséges.

Első lépésként deklarálni kell egy Random objektumot:

Random velszam = new Random();

Majd egy konkrét véletlen számot a Next tagfüggvénnyel nyerhetünk, amit egy változónak értékül adunk:

valtozo = velszam.Next();

Paraméter nélkül egy pozitív int típusú számot ad eredményül. Egy egész paramétert megadva 0-nál nagyobb vagy egyenlő, a paraméternél kisebb egész számot kapunk.

valtozo = velszam.Next(29); // 0-tól 28-ig kapunk egy véletlen számot

Két egész paraméterrel pedig a két szám közti, a kisebbnél nagyobb vagy egyenlő, a nagyobbnál kisebb egész számot kapunk.

valtozo = velszam.Next(51,100); // 51-től 99-ig kapunk egy véletlen számot

**Matematikai függvények**

A System névtér Math objektuma tartalmazza a gyakrabban használt matematikai függvényeket és konstansokat.

Konstansok:

Math.PI –  értéke

Math.E – e értéke

A függvények paramétereikkel:

Math.Sin(double x) az x paraméter szinusza;

Math.Cos(double x) az x paraméter koszinusza;

Math.Tan (double x) az x paraméter tangense;

Math.Exp(double x) az ex exponenciális függvény;

Math.Log(double x) az x paraméter természetes alapú logaritmusa (ln(x)), x>0;

Math.Log10(double x) az x paraméter tízes alapú logaritmusa (lg(x)), x>0;

Math.Pow(alap, kitevő) az xy alakú hatványfüggvény, értelmezési tartomány hiba lép fel, ha x=0 és y<0, vagy ha x<0 és y értéke nem egész szám;

Math.Sqrt(double x) az x paraméter négyzetgyöke, x>0;

Math.Ceiling(double x) az x paraméternél nem kisebb legkisebb egész szám, double típusra konvertálva;

Math.Floor(double x) az x paraméternél nem nagyobb legnagyobb egész szám, double típusra konvertálva;

Math.Truncate(double x) az x paraméternél csonkolása egészre, vagyis a tizedesrészt levágja;

Math.Round(double x) az x paraméter kerekítése egészre;

Math.Round(double x, int y) az x paraméter kerekítése y által meghatározott helyiértékre;

Math.Abs(x) az x szám paraméter abszolút értéke (|x|);

Math.Max(x, y) x és y számok közül a nagyobbat adja vissza;

Math.Min(x, y) x és y számok közül a kisebbet adja vissza;

Math.Sign(x) x szám előjelétől függően -1-gyel vagy +1-gyel tér vissza.

**Feltételes utasítások**

Ha egy utasítás végrehajtása valamilyen feltételtől függ, akkor feltételes utasítást, más néven elágazást kell alkalmaznunk.

**if … else (ha … akkor) utasítás:**

Alakja:

**if** (feltétel)

{

utasítás(ok)\_1; // Ez hajtódik végre, ha a feltétel igaz, …

}

**else**

{

utasítás(ok)\_2; // ha a feltétel hamis, akkor pedig ez.

}

A feltétel egy logikai kifejezés, vagyis olyan kifejezés, amiről a program egyértelműen el tudja dönteni, hogy igaz-e vagy hamis (pl. összehasonlítás, bool adattípus).

Ha a feltétel igaz, akkor az *utasítás(ok)\_1* hajtódik végre, ha hamis, akkor az else utáni *utasítás(ok)\_2*.

Az igaz és a hamis ágban is a tetszőleges számú utasításból álló utasításblokk lehet. Ha csak egy utasítás van az if vagy az else ágban, akkor a {} elhagyható.

**if** (feltétel)

utasítás; // Ez hajtódik végre, ha a feltétel igaz, …

**else**

utasítás; // ha a feltétel hamis, akkor pedig ez.

{} használata nélkül csak az if, illetve az else utáni első utasítás tartozik a feltételes utasításhoz!

Az if utasításból az else ág elhagyható:

**if** (feltétel)

{

utasítás;

}

Akkor használjuk, ha az utasítás egy feltétel bekövetkeztekor kell végrehajtani. Hamis érték esetén nem kell semmit sem csinálni.

**Többszörös elágazás:**

Ha a feltételtől függően kettőnél több lehetőség közül kell választani, akkor használjuk, az if utasításnak ezt az alakját.

**if** (feltétel\_1)

utasítás(blokk)\_1; // Ez hajtódik végre, ha a feltétel\_1 teljesül, …

**else if** (feltétel\_2)

utasítás(blokk)\_2; // ha a feltétel\_1 hamis, de feltétel\_2 igaz, akkor pedig ez,

**else if** (feltétel\_3)

utasítás(blokk)\_3; // ha a feltétel\_1 és feltétel\_2 is hamis, de feltétel\_3 igaz, akkor ez

…

**else if** (feltétel\_n)

utasítás(blokk)\_n; // ha minden eddigi feltétel hamis, de feltétel\_n igaz, akkor pedig ez

**else**

utasítás(blokk); // ha a minden feltétel hamis, akkor ez lesz végrehajtva.

Az „else-if” szerkezetben az első olyan utasítás fog végrehajtódni, amelyikhez tartozó feltétel igaz lesz. A többi feltételt már nem fogja vizsgálni.

Ha minden feltétel hamis, akkor az utolsó else utáni utasítás kerül végrehajtásra. Ez azonban el is hagyható.

Üres utasítás:

Ha az if utasításban a feltétel után rögtön egy pontosvessző áll, akkor a program semmit nem fog végrehajtani igaz esetben. Az önmagában álló pontosvessző ugyanis az üres utasítás.

**if** (feltétel\_1) ; // üres utasítás

**switch utasítás**

Felépítése:

**switch** (változó v. egész kif.)

{

**case** érték\_1: utasítások\_1;

**break**;

**case** érték\_2: utasítások\_2;

**break**;

…

**case** érték\_n: utasítások\_n;

**break**;

**default**: utasítások egyéb esetben;

**break**;

}

A **switch** utasítás egy egész típusú változó (byte, int, long különböző típusai, char) vagy egész eredményt visszaadó kifejezés értékétől függően hajt végre utasításokat. A C# nyelvben használható a string típus is.

A változó v. kifejezés a switch utáni zárójelben van, a vizsgált értékek pedig a {}-ek közötti case ágakban. Minden case ágbna a kettőspont után állnak a végrehajtandó utasítások.

Ha a változó értéke megegyezik az egyik case értékkel, akkor az utána következő utasítások végrehajtódnak. Az egyes case ágakban szereplő utasításokat **break** utasítással kell lezárni, ami kilép a switch utasításból.

Ha egyik case érték sem egyezik meg a switch változójával, akkor a default: utáni utasítás(ok) kerülnek végrehajtásra. Az itt szereplő utasításokat is a break utasítás zárja. A default: ágat nem kötelező használni.

**break utasítás**

A **break** utasítás kilép az adott programblokkból és a blokk utáni első utasításra ugrik. Használható a switch utasítás case ágaiban is, illetve C#-ban kötelező használni.

**Összehasonlító operátorok:**

kisebb: < nagyobb: > kisebb egyenlő: <= nagyobb egyenlő: >=

egyenlőség vizsgálata: == nem egyenlő: !=

**Az egyszeres egyenlőség jel (pl. x=3) az értékadás művelete, nem összehasonlítás!**

Logikai műveletek:

Logikai állítások: olyan állítás, amiről egyértelműen eldönthető, hogy igaz vagy hamis.

A logikai állításokat összekapcsolhatjuk különböző logikai műveletekkel.

Logikai műveletek:

Tagadás (NEM, NOT): Igaz állítás tagadása hamis, hamis tagadása pedig igaz lesz. Jele: !.

pl.: a>=0 tagadása: !(a>=0)

ÉS művelet (AND): Akkor használjuk, ha két állításnak egyszerre kell teljesülnie. Jele: &&

pl. v változó értéke 10 és 50 közé esik-e: v>=10 && v<=50

VAGY művelet (OR): Akkor használjuk, ha két állítás közül elég, ha az egyik teljesül. Jele: ||

pl. ch karakteres változó kicsi vagy nagy t betűt tartalmaz-e: ch==’t’ || ch==’T’

**Ciklus utasítások**

Ciklusok típusai:

növekményes ciklus – előre ismert a végrehajtások száma

feltételes ciklus – az utasítások végrehajtásának száma egy feltételtől függ

előltesztelő ciklus – először vizsgálja a feltételt, utána hajtja végre a ciklusmagot

hátultesztelő ciklus – először hajtja végre a ciklusmagot, utána vizsgálja a feltételt

Ciklusmag: az ismétlődő utasítás vagy utasítások.

C# ciklus utasításai:

while ciklus: előltesztelő ciklus, alakja:

while (feltétel)

{

cuklusmag;

}

Működése: a ciklusmag addig ismétlődik, amíg a feltétel igaz. Ha hamis, akkor a ciklus utáni első utasítással folytatódik a program végrehajtása. Ha a feltétel már induláskor hamis, akkor a ciklusmag egyszer sem fut le.

A feltétel lehet egy összehasonlítás, logikai változó vagy egy összetett logikai kifejezés (and, or, not logikai operátorokkal).

Ha a ciklusmag egy utasításból áll, akkor a {}-ek elhagyhatók.

do-while ciklus: hátultesztelő ciklus, alakja:

do

{

cuklusmag;

}

while (feltétel);

Működése: a ciklus végrehajtása a ciklusmag lefutásával kezdődik, majd ez után jön a feltételvizsgálat. Ha a feltétel igaz, akkor a ciklusmag újból lefut, addig, amíg a feltétel hamis nem lesz. Hamis esetén, a ciklus utáni első utasítással folytatódik a program végrehajtása. A ciklusmag a do-while ciklusnál legalább egyszer mindenképpen lefut.

A feltétel itt is összehasonlítás, logikai változó vagy egy összetett logikai kifejezés lehet.

Ha a ciklusmag egy utasításból áll, akkor a {}-ek elhagyhatók.

for ciklus: növekményes, előltesztelő ciklus, alakja:

➊

➌

➋

for (inícializációs rész; feltétel; léptető rész)

{

cuklusmag;

}

A fejléc részeinek funkciói:

➊ ciklusváltozók, számlálók kezdőértékadása, vesszővel elválasztva több is megadható.

➋ igaz vagy hamis logikai kifejezés (összehasonlítás, logikai változó vagy egy összetett logikai kifejezés)

➌ ciklusváltozók számlálók növelése, csökkentése, több utasítás is lehet vesszővel elválasztva.

Az egyes szakasz ok lehetnek üresek is, a pontosvesszőt viszont ki kell tenni.

pl.: for ( ; a<12; ++a) 🡪 nincs kezdőértékadás; for( ; ; ) 🡪 üres fejléc

A for ciklus működése:

1. inícializációs utasítások végrehajtása
2. feltételvizsgálat
3. ciklusmag lefutása, ha a feltétel igaz
4. léptető utasítások végrehajtása
5. 2-4. pont ismétlése, amíg a feltétel igaz

Ha a feltétel hamis a ciklust követő első utasítással folytatódik a pr. futása.

Ha a ciklusmag egy utasításból áll, akkor a {}-ek elhagyhatók.

A deklarációs részben változó deklaráció is lehet, de ez csak a ciklusmagon belül érhető el.

for (int j = 0; j < 10; ++j)

{ ciklusmag; }

**A ciklust megszakító utasítások:**

**break** utasítás: kilép a ciklusból, megszakítja a ciklusmag végrehajtását és a ciklus utáni első utasításra ugrik. alakja: break;

**continue** utasítás: a ciklusmag pillanatnyi lefutását szakítja meg, de a ciklusból nem lép ki. A while és a do-while ciklusoknál a feltételvizsgálatra ugrik, a for ciklusnál pedig a léptető utasításra.

**Tömbök**

A **tömb** egy összetett adattípus, amelyben több, azonos típusú adat tárolható. A tömbön belül tárolt értékek a sorszámukkal, más néven indexükkel érthetők el.

Mint minden változót, a tömböket is deklarálnunk kell. A deklarációval egy tömb azonosítót hozunk létre, illetve megadjuk, hogy milyen típusú elemeket tárolhat.

Tömb deklarációja:

típus[] tömbnév;

A deklaráció után a tömböt létre kell hozni, amihez meg kell adni az elemszámot is.

A tömb létrehozása:

tömbnév = new típus[méret];

A deklarációt és a tömb létrehozását egy műveletbe is össze lehet vonni:

típus[] tömbnév = new típus[méret];

A típus a tömbben tárolt elemek típusát adja meg. A méret pedig a maximálisan eltárolható elemek számát.

pl.: int[] t = new int[100]; 🡪 A t nevű tömb maximum 100 darab int típusú egész számot tud tárolni.

char[] betuk = new char[50]; 🡪 A betuk nevű tömb legfeljebb 50 darab karaktert tud tárolni.

A tömb elemeinek elérése indexelés segítségével történik. Az első elem indexe 0, az utolsó elemé pedig elemszám-1. Egy tömbelemre a tömb nevével és az indexével hivatkozhatunk.

pl.:

t[0] = 53; 🡪 értékadás az első elemnek.

betuk[3] = ’h’; 🡪 a negyedik elem a h betű lesz.

A helytelen indexelés futási hibát okoz.

Tömb deklarálása kezdeti értékadással:

típus[] tömbnév = {elem\_0, elem\_1, elem\_2, … elem\_n};

A tömb elemszámát a felsorolt elemek száma határozza meg.

pl.:

double[] szamok = {4.2, 6, 1.9, 45.3, 0.5, 10}; 🡪 6 elemű valós tömb.

Egy tömb méretének lekérdezése a Length tulajdonság tárolja. Lekérdezése:

tömbnév.Length

pl.:

meret = szamok.Length;

Tömbök kezelése for ciklussal:

int i; // tömbindex

for ( i = 0; i < elemszám; ++i)

ciklusmag

A ciklusmagban a tömb aktuális elemére tömbnév[i]-vel hivatkozhatunk!

**Többdimenziós tömbök**

Egy kétdimenziós tömb sorokból és oszlopokból álló adatszerkezet (mint egy táblázat).

Deklarációja létrehozással:

típusnév[,] tömbnév = new típusnév[sorszám, oszlopszám];

pl.: char[,] tabla = new char[6, 8]; 🡪 6 sorból és 8 oszlopból álló, karaktereket tároló tömb

int[,] szamok = new int[10, 15]; 🡪 10 sorból és 15 oszlopból álló, egészeket tároló tömb

Kezelésükhöz két indexet használunk, egyet a soroknak, egyet pedig az oszlopoknak. Mindkét index számozása 0-tól kezdődik és sorszám -1 –ig, illetve oszlopszám -1-ig tart.

Elemek elérése kétdimenziós tömbben:

tömbnév[sorszám, oszlopszám]

pl.: tabla[0, 0] = ’A’;

Kétdimenziós tömbök elemein két darab, egymásba ágyazott for ciklussal tudunk végigmenni.

int i, j; 🡪 sor és oszlopindexek

for (i = 0; i < sorok száma; ++i)

for (j = 0; j < oszlopok száma; ++j)

{

ciklusmag; az aktuális tömbelemre tömbnév[i, j] kifejezéssel hivatkozhatunk

}

pl:

for (i = 0; i < 10; ++i)

for (j = 0; j < 15; ++j)

{

szamok[i, j] = rnd.Next(100, 200);

}

**Sztringkezelés**

**Szöveges változó (sztring) deklarálása:**

string str, str\_2 = ”kezdeti érték”;

**Sztringek összefűzése:**

Az összeadás (+) operátorral különböző sztringekből egy szöveggé egyesíthetünk. A műveletet szöveg összefűzésnek is mondjuk. Használható string típusú változókra, szövegkonstansokra, de különböző változók értékeit is hozzáadhatjuk egy szöveghez.

string s, s1 = ”villany”, s2 = ”írógép”;

s = s1+ s2;

s változó tartalma: villanyírógép

s = ”három ” + s1 + s2;

s változó tartalma: három villanyírógép

double v = 5.6;

s = v + ” ” + s2;

s változó tartalma: 5.6 írógép

Sztringek hossza:

Egy sztring hosszát, vagyis karaktereinek számát a Length tulajdonsággal adhatjuk meg. A kapott érték egy egész szám. Használata:

sztingnév.Length;

string st = ”C# programozás”;

int n = st.Length; 🡪 n értéke 14 lesz

Sztringek karaktereinek elérése

A string változók karakteres tömbként is kezelhetők, vagyis az egyes karakterei index segítségével elérhetők stringnév[index] formában. Az első karakter indexe itt is 0, az utolsóé sztringhossz -1. A sztring karakterei viszont ezzel a fajta eléréssel nem módosíthatók.

string st = ”Programozás”;

st[0] értéke a ’P’ betű, st[1] az ’r’ betű, stb., az utolsó karakter st[10]-ben az ’s’.

Sztringek összehasonlítása

Sztingekre két féle összehasonlító reláció alkalmazható. ezek az egyenlő (==) és a nem egyenlő (!=) összehasonlító operátorok.

str1 == str2 kifejezés igaz, ha mindkét változó ugyan azt a karaktersorozatot tartalmazza.

st1 != st2 kifejezés igaz, ha a két változóban különböző szövegek vannak.

Sztringeket összehasonlíthatunk a CompareTo tagfüggvénnyel is. Szöveges adatok összehasonlítsa az ábécé szerinti sorba rendezésen alapul. Amelyik szöveg az ábácé sorrendben előbb szerepel, az kisebb az utána következőknél.

Ha s1 és s2 string típusú változók, akkor s1.CompareTo(s2) alakban használható. A függvény értéke 0, -1 vagy 1 lehet.

Ha s1 < s2, vagyis s1 előbb van az ábécében mint s2, akkor s1.CompareTo(s2) értéke -1

Ha s1 > s2, vagyis s1 hátrébb van az ábécében mint s2, akkor s1.CompareTo(s2) értéke 1

Ha s1 = s2, vagyis s1 és s2 azonosak, akkor s1.CompareTo(s2) értéke 0

Levágás

A Trim()tagfüggvény segítségével eltávolíthatjuk a szöveg elejéről és a végéről a szóközt, illetve a paraméterben megadott karaktert. Használata:

Az s.Trim() utasítás egy olyan sztringet állít elő, amely s sztring elejéről és végéről levágja a szóközöket. Az s.Trim(’karakter’) alak, pedig a megadott karaktert távolítja el. Pl. az s.Trim(’d’) a d betűket vágja le az s változóban lévő szövegből. A függvény a minden esetben egy új sztringet állít elő, amit egy másik szöveges változóba menthetünk, az eredeti változó tartamát nem módosítja.

Hasonlóan működnek a TrimStart() és TrimEnd() függvények is, amelyek csak a szöveg elejéről, illetve csak a végéről távolítják el a szóközöket vagy a paraméterben megadott karaktert.

string st, s;

string st = ” egyenlet ”;

s = st.Trim(); 🡪 s értéke ”egyenlet” lesz.

st = ”aaaaxxxxxxxxxxaaaa”;

s = st.Trim(’a’); 🡪 s értéke ”xxxxxxxxxx”; lesz.

Szövegrész kimásolása

Egy szöveges változóból tetszőleges részt kinyerhetünk a Substring tagfüggvénnyel. Teljes alakja s.Substring(honnéttól, hány karakter), vagyis az első paraméterrel megadható, hogy melyik indexű karaktertől kezdődjön a kimásolandó szöveg, a második paraméter pedig a karakterek számát adja meg. Ha a második paraméter elmarad, akkor a string végéig tart az új szöveg.

Átalakítás kis- vagy nagybetűssé

Egy string típusú változó csupa nagybetűs alakját állítja elő a ToUpper() tagfüggvény, illetve csupa kisbetűs alakját a ToLower() tagfüggvény.

Keresés a szövegben

Az **IndexOf** tagfüggvénnyel egy karaktert vagy egy szöveget kereshetünk egy string változóban. Ha nincs találat, akkor a kapott eredmény -1, egyébként pedig egy 0 vagy nagyobb pozitív szám, ami azt adja meg, hogy hányadik indexű karakternél található meg a keresett szöveg (vagy karakter) a sztringben.

Ha a találat helyére (indexére) nincs szükség, akkor használható a **Contains** tagfüggvény. A visszatérési értéke true vagy false, a találatnak megfelelően.

Használat:

stringváltozó.Contains(”keresendő string”)

Megadja, hogy a keresendő string a stringváltozóban, mint részstring szerepel-e vagy nem.

**Fájlkezelés**

**Szöveges fájlok kezelése**

A fájlok kezeléséhez a System.IO névtérre van szükség, aminek az elérését a forráskód elején a using System.IO; utasítás elhelyezésével biztosíthatjuk.

Ahhoz, hogy egy állományt tudjunk kezelni, létre kell hozni egy fájlváltozót, majd ezt összerendelni egy fizikai állománnyal. a programon belül a fájlváltozóval érhetjük el az állomány tartalmát, illetve végezhetünk különböző fájlműveleteket. A fájlváltozó típusa attól függ, hogy írni vagy olvasni akarjuk-e az állományt.

**Fájlváltozó létrehozása írásra:**

Ha egy állományba írni szeretnénk, akkor azt egy StreamWriter típusú változóhoz kell hozzárendelni.

Első lépésként deklarálni kell egy fájlváltozót:

StreamWriter fájlváltozó; // fájlváltozó létrehozása íráshoz

Majd egy konkrét állomány eléréséhez, össze kell rendelni a fájlváltozóval:

fájlváltozó = new StreamWriter(”fájlnév”);

A deklarálás és a hozzárendelés egy utasítással is elvégezhető:

StreamWriter fájlváltozó = new StreamWriter(”fájlnév”);

**Fájlváltozó létrehozása olvasásra:**

Ha egy állomány tartalmát szeretnénk olvasni, akkor ahhoz egy StreamReader típusú változó kell.

Fájlváltozó deklarálása olvasáshoz:

StreamReader fájlváltozó; // fájlváltozó létrehozása íráshoz

Fájl hozzárendelése a fáljváltozóhoz:

fájlváltozó = new StreamReader(”fájlnév”);

A deklarálás és a hozzárendelés egy utasítással:

StreamReader fájlváltozó = new StreamReader(”fájlnév”);

**Írás fájlba**

A fájlba adatokat írni a Write és WriteLine tagfüggvényekkel lehet:

fájlváltozó.WriteLine(”Szöveg” vagy változó);

**Olvasás fájlból**

Egy fájl tartalmát olvasni a ReadLine tagfüggvénnyel lehet, ami egyszerre egy-egy sort ad vissza a szöveges állományból, amit egy string változóba tudunk elmenteni. Ha más típusra van szükség, akkor konvertálnunk kell az eredményt.

sting st = fájlváltozó.ReadLine();

int szam = int.Parse(fájlváltozó.ReadLine());

Állomány lezárása

Az állományműveletek végén az állományokat le kell zárni. hogy feleslegesen ne foglalja le a program őket. Erre a Close függvény használható:

fájlváltozó.Close();

Írásra megnyitott állományoknál célszerű a Flush függvényt meghívni, ami kiüríti a fájlhoz tartozó puffereket.

fájlváltozó.Flush();

**Függvények**

A függvény a programkód olyan, névvel ellátott része, amelyet a program más részeiről tetszőleges számszor meghívható és egy meghatározott feladatot lát el.

Egy függvény felépítése:

visszatérési érték típusa függvénynév(paraméterlista)

{

függvénytörzs utasításai;

return visszatérési érték; // ha van

}

pl.:

double köb(double x)

{

double k;

k = x \* x \* x;

return k;

}

Függvényt csak egy objektumosztályon belül hozhatunk létre. Egy objektumhoz tartozó függvényeket tagfüggvénynek vagy metódusnak hívunk. A függvények (C/C++/C#) nem ágyazhatók egymásba.

A függvény fenti kódját a függvény definíciójának hívjuk.

Egy függvény törzsén belül új változókat is létrehozhatunk, ezek viszont csak a függvény törzsében használhatók. Élettartamuk csak a függvényre korlátozódik.

Ha egy függvény nem ad vissza értéket, akkor a visszatérési értéke void lesz. Ekkor a return utasítás nem szerepelhet benne.

A return utasítás a függvény végrehajtását mindenképpen megszakítja, és a mögötte található kifejezés értékével tér vissza a hívás helyére. Ennek a kifejezésnek a típusa és a visszatérési értéknél megadott típus meg kell, hogy egyezzen.

Ha a függvény olyan műveleteket végez, amihez valamilyen adat(ok)ra van szükség, akkor azokat a paraméterlistában kell neki átadni. Az itt megadott változók a függvénytörzsön belül használhatók (azon kívül nem érhetők el). A paraméterek megadásánál a típussal és a nevet is meg kell adni.

A függvény hívása:

függvénynév(paraméterek);

A függvény hívása a nevének és a paramétereinek megadásával történhet. Ekkor a vezérlés a függvény törzsére ugrik, az átadott paramétereket felhasználva hajtja végre a függvénytörzs utasításait. Ha van visszatérési érték, akkor a hívás helyére ezzel tér vissza.

A paraméterek számának, típusának és sorendjének meg kell egyezni a függvény kódjában megadottakkal.

Ha egy függvényt nem a saját objektumán belül hívjuk, akkor a meghívásához az objektum vagy osztály nevét is meg kell adni.

Paraméterek típusai

Egy függvénynek adatokat a paraméterein keresztül adhatunk át.

A függvény definíciójában szereplő paramétereket formális paraméterlistának mondjuk, a meghíváskor átadott konkrét adatokat pedig aktuális paraméterlistának nevezzük.

Paramétereket két módon adhatók át. Az egyik az érték szerinti, a másik a cím szerinti paraméterátadás.

Érték szerinti paraméterátadás esetén megadhatunk konstansokat, változókat vagy tetszőleges kifejezéseket paraméterként. Ekkor ezek értékeinek egy másolata kerül a megfelelő paraméterváltozóba.

Cím szerinti paraméterátadásnál az átadott adat csak egy változó címe lehet, amelyen keresztül a függvény eléri a változóban tárolt adatot.

Érték szerinti paraméterátadásnál a függvénynek átadott változók értéke nem módosul, mivel a függvény törzsébe csak azok másolata kerül.

Cím szerinti paraméterátadásnál a függvény az eredeti változókat tudja módosítani.

**Érték szerinti paraméterátadás:**

visszatérési érték típusa függvénynév(típus1 változónév1, …)

{

függvénytörzs;

}

A függvény meghívása: függvénynév(paraméter1, …);

A paraméter lehet konstans, változó vagy számított kifejezés is.

**Cím szerinti paraméterátadás:**

visszatérési érték típusa függvénynév(ref típus1 változónév1, …)

{

függvénytörzs;

}

A függvény meghívása: függvénynév(ref változónév, …);

A paraméter csak változónév lehet.

Egy paraméterlistán belül előfordulhat cím és érték szerinti paraméter is.

Ha egy függvénynek a paraméterén keresztül tömböt, stringet vagy objektumot adunk át, azok alapból cím szerinti paraméterátadással kezeli a program, ezt nem kell a ref kulcsszóval jelölni.

**Osztályok, objektumok**

A C# teljesen objektum-orientált programnyelv. A program minden eleme egy objektum része. Maga a program is egy objektum.

Az **osztály** adatokat és az azokon végzett vagy azokkal kapcsolatos műveleteket (függvények) egy egységbe foglaló programösszetevő.

Egy osztályhoz tartozó adatokat **adattagoknak** vagy tulajdonságoknak nevezzük, a műveleteket pedig **tagfüggvénynek** vagy **metódusnak** hívjuk.

Az osztályokat a programban definiálni kell.

Az osztály definiálása:

class osztálynév

{

osztály törzse

}

Az osztálytörzsön belül kell megadni az adattagokat és a metódusokat.

Ahhoz, hogy egy osztályt használni tudjunk, egy objektumváltozót kell belőle létrehozni deklarálás segítségével.

Objektum deklarálása:

osztálynév objektumváltozó\_név = new osztálynév();

Az objektumokat felhasználhatjuk olyan összetett adatszerkezetek létrehozására is, amelyek több, különböző típusú adatot tartalmaznak.

class osztálynév

{

public típus adattag1;

public típus adattag2;

…

}

Példa:

class diák

{

public string név;

public int évfolyam;

public string lakhely;

}

Objektum deklarálása a diák osztályhoz:

diák András = new diák();

A deklarált objektum használata:

András.név = ”Kovács András”;

András.évfolyam = 10;

András.lakhely = ”Ajka”;

Console.WriteLine(András.név);

Az adattagokat az osztály függvényeiből a nevükkel érhetjük, más objektumokból viszont az objektumváltozót is használni kell.

Adattag elérése az osztályon kívül: objektumváltozó.adattag

Metódusok elérése:

objektumváltozó.tagfüggvény(paraméterek)

Az osztályok tagfüggvényei:

A osztályokhoz tagfüggvényeket is rendelhetünk, amelyek segítségével különféle feladatokat is végre tudnak hajtani. A tagfüggvényeket az osztálydefiníciójában, az adattagok után kell megadni.

Tagfüggvény megadása:

class struktúranév

{

adattagok;

public visszatérési\_érték függvénynév(paraméterek)

{

függvénytörzs;

}

}

pl.:

class téglalap

{

public double a;

public double b;

public double terület()

{

return a\*b;

}

}

Tagfüggvények meghívása:

Az osztály tagfüggvényeit az objektumváltozó és a függvénynév segítségével érhetjük el.

vátozónév.tagfüggvénynév(paraméterek);

pl.:

téglalap tl = new téglalap();

tl.a = 45.6;

tl.b = 12;

double ter = tl.terület();

**Konstruktor**

A konstruktor egy olyan tagfüggvény, ami egy objektumpéldány létrehozásakor automatikusan lefut. Fő feladata a kezdeti értékadás, illetve különböző beállítások elvégzése.

Minden osztálynak van alapértelmezett konstruktora, akkor is, ha nem hozunk létre.

A konstruktor metódus neve mindig megegyezik az osztály nevével.

A konstruktort gyakran használjuk arra, hogy deklaráláskor paraméterként értéket adjunk át a létrejövő objektum adattagjainak.

Egy objektum deklarációjában a new kulcsszó után tulajdonképpen a konstruktort hívjuk meg.

Példa:

class téglalap

{

public double a;

public double b;

public téglalap(double param1, double param2)

{

a = param1;

b = param2;

}

}

Új objektum létrehozása:

téglalap tl\_1 = new téglalap(2.5, 9.1);

**Osztályok tagjainak láthatósága**

Egy osztály adattagjai és metódusai esetén is szabályozható, hogy az adott elemet honnét érhetjük el. A két legfontosabb típus a private és a public.

private: nem lehet elérni másik osztályból.

public: bárhonnét elérhető elem.

**Statikus adattagok és függvények**

A statikus adattagok nem az egyes objektumpéldányokhoz tartoznak, hanem az osztályhoz. Ami azt jelenti, hogy minden objektumban ugyan az az értéke. Kívülről elérni az osztály nevével lehet.

A statikus függvényeket az osztály nevével lehet elérni. Használatukhoz nem kell objektumpéldányt létrehozni. Ilyenek pl. a Math objektum függvényei is.

**Öröklés**

Az objektumorientált programozás lehetővé teszi, hogy a már létező osztályokból újakat hozzunk létre. Az új osztály megkapja a szülő osztály adattagjait és metódusait, ha szükséges ezeket felülírhatja, módosíthatja, és új tagokat is létrehozhat.

Egy szülőosztályból több származtatott osztályt is létrehozhatunk, de C#-ban minden származtatott osztálynak csak egy szülője lehet.

A szülőosztályt alaposztálynak, a származtatott osztályt gyermekosztálynak is nevezzük.

class alaposztály

{

alaposztály törzse

}

class származtatott\_osztály : alaposztály

{

új adattagok

új metódusok

}

*Az alaposztály metódusának kiegészítése:*

public visszatérési érték típusa metódusnév (paraméterek) : base(paraméterek)

{

alaposztály utasításainak a kiegészítése

}

*Az alaposztály metódusának felülírása:*

public new visszatérési érték típusa metódusnév (paraméterek)

{

új utasítások

}

**Struktúrák**

A struktúra egy olyan összetett adatszerkezet, amely több, akár különböző típusú elemeket tartalmazhat. Ezek, a struktúrában lévő elemek lehetnek egyszerű típusú adatok (double, int, char, stb.), de összetett adatok is, pl. string, tömbök vagy akár más struktúrák is.

Struktúrákat több programozási nyelv is használ (pl. C, C++), illetve a legtöbb nyelvben megtalálható a struktúrához hasonló adatszerkezet.

A C# nyelv a struktúrákat az objektumosztályokhoz hasonlóan kezeli: lehetnek egy struktúrának adattagjai és metódusai, valamint rendelkezhet konstruktorral is.

Struktúra adattagjai: a struktúra által tárolt adatok, meg kell adni az adat típusát és egy változónevet.

Struktúra metódusai (tagfüggvényei): a struktúrához rendelhetők különféle függvények, amik a struktúrához köthető műveleteket (értékadás, kiírás, számítások stb.) végzik el.

A struktúra deklarálása a program objektumán kívül történik.

Struktúra definiálása:

struct struktúranév

{

struktúra törzse

}

Adattagok megadása:

struct struktúranév

{

public típus1 változónév1;

public típus2 változónév2;

…

}

pl.:

struct diák

{

public string név;

public int évfolyam;

public string lakhely;

}

A public minősítővel azt adhatjuk meg, hogy az egyes adattagok kívülről elérhetők legyenek.

A struktúra deklarálása egy struktúratípust ad meg. Ha használni szeretnénk, akkor deklarálni kell egy ilyen struktúratípusú változót. Deklarálni a használat helyén kell, egyszerűbb esetekben tehát a Main függvényben.

Struktúra deklarálása:

struktúranév változónév = new struktúranév();

pl.:

diák András = new diák();

A struktúraváltozó használata:

A struktúraváltozó adattagjai tárolják a konkrét adatokat. Ezek elérése a következő:

változónév.adattag\_neve

pl.: András.név, András.évfolyam, András.lakhely.

András.név = ”Kovács András”;

András.évfolyam = 10;

András.lakhely = ”Ajka”;

Struktúrák tagfüggvényei:

A struktúrákhoz tagfüggvényeket is rendelhetünk, amelyek segítségével különféle feladatokat is végre tudnak hajtani. A tagfüggvényeket a struktúra definíciójában, az adattagok után kell megadni.

Tagfüggvény megadása:

struct struktúranév

{

adattagok;

public visszatérési\_érték függvénynév(paraméterek)

{

függvénytörzs;

}

}

pl.:

struct téglalap

{

public double a;

public double b;

public double terület()

{

return a\*b;

}

}

Tagfüggvények meghívása:

A struktúra tagfüggvényeit a változónév és a függvénynév segítségével érhetjük el.

vátozónév.tagfüggvénynév(paraméterek);

pl.:

téglalap tl = new téglalap();

tl.a = 45.6;

tl.b = 12;

double ter = tl.terület();

A struktúrák tartalmazhatnak konstruktort is, ami egy speciális tagfüggvény. A konstruktor olyan metódus, ami egy struktúraváltozó létrehozásakor automatikusan lefut. Arra használható, hogy a deklaráció után kezdőértéket adjunk a struktúra adattagjainak, illetve egyéb kezdeti műveleteket végrehajtsunk, ha szükséges.

A konstruktor neve meg kell, hogy egyezzen a struktúra nevével, visszatérési értéket viszont nem kap.

struct struktúranév

{

adattagok;

public struktúranév(paraméterek)

{

függvénytörzs;

}

}

pl:

struct téglalap

{

public double a;

public double b;

public téglalap(double p1, double p2)

{

a = p1;

b = p2;

}

}

A konstruktor meghívása:

téglalap tl = new téglalap(3, 7);

**Kivételkezelés**

A program végrehajtása során keletkező hibák (futásidejű hibák) kivételeket okoznak. Ezek a hibák, ha nem kezeljük őket, az a program leállásához vezetnek.

Ilyen súlyos futási hibák például a nullával való osztás, túlcsordulás, egy változó nem a típusának megfelelő értéket kap (pl. szám helyett szöveg), egy tömb túlindexelése, nem elérhető állományt akarunk használni stb.

A kivételkezelés a futásidejű hibák kezelését jelenti. A programkódban olyan utasításokat adhatunk meg, amelyek a keletkező hibákat elfogják és lehetővé teszik a felhasználó általi kezelésüket. Így nem okozzák a program váratlan leállását.

Kivételkezelés megvalósítása a programban:

try

{

programblokk, amelyben el akarjuk kapni a hibákat

}

catch

{

hibakezelő blokk

}

A hibalehetősséget tartalmazó programrészt egy try blokkba kell helyezni. Ha ebben a blokkban hiba keletkezik, akkor nem az alapértelmezett hibaüzenetet kapjuk programleállás kíséretében, hanem az utasítás-végrehajtás átadódik a try utáni catch blokkra, amelybe saját hibakezelő kódot írhatunk. A try blokk hiba utáni része nem hajtódik végre.

Hibakód átvétele

Arra is van lehetőség, hogy a hibakódot a catch blokk átvegye és a blokkon belül felhasználjuk. Ekkor a kivételkezelés alakja:

try

{

programblokk, amelyben el akarjuk kapni a hibákat

}

catch (Exception error)

{

hibakezelő blokk

}

A hibakódot az Exception típusú, error nevű változó tartalmazza.

finally blokk:

try

{

programblokk, amelyben el akarjuk kapni a hibákat

}

catch

{

hibakezelő blokk

}

finally

{

minden esetben végrehajtásra kerülő kód

}

A finally blokkban elhelyezett kód minden esetben lefut. Akkor is, ha a try blokk hibátlanul lefut, de ha hiba lesz és a catch blokkra kerül a vezérlés, akkir is végrehajtódik az itt elhelyezett kód.

**Konzol utasítások**

Console.BackgroundColor a háttérszínt tároló változó, értékét a ConsoleColor objektum konstansaiból veszi.

Console.ForegroundColor az előtér, vagyis a szövegszínt tároló változó, értékét a ConsoleColor objektum konstansaiból veszi.

Console.ResetColor() az alapszíneket visszaállító függvény

Console.Clear() A konzol ablak törlése.

Console.CursorVisible a kurzor megjelenítése, értéke true vagy false lehet.

Console.SetCursorPosition(vízsz. poz, függ. poz.) A kurzor pozícióját beállító függvény

Console.Beep(frekvencia, ezredmásodperc) hangkeltés, a paraméterben megadott frekvenciájú hangot valahány ezredmásodpercig szólaltatja meg.

Console.Title a konzolablak címét állítja be, értéke szöveg.

Console.WindowHeight ablak magassága karakterben

Console.WindowWidth ablak szélessége karakterben

Console.SetWindowSize(magasság, szélesség) A konzolablak magasságát és szélességét egy lépésben beállító függvény.

Console.ReadKey() várakozás billentyűleütésre.

**Listák kezelése**

A lista a tömbhöz hasonló összetett adatszerkezet, mérete azonban dinamikusan változtatható, vagyis nem kell előre megadni a méretét, az mindig az aktuális elemszámhoz igazodik. A listák is azonos típusú elemekből állnak.

A lista deklarációja:

List<típus> listanév = new List<típus>();

pl.: List<int> szamlista = new List<int>();

Új elemet az Add metódussal tudunk a listához adni. Ez a lista végére kerül.

listanév.Add(új elem);

pl.: szamlista.Add(7);

A lista méretét a Count tulajdonsággal kérdezhetjük le:

listanév.Count;

pl.: int db = szamlista.Count;

A lista elemeinek elérése a tömbökével azonos, vagyis listanév és index szükséges hozzá:

listanév[index]

A lista első tagjának indexe a 0, az utolsóé pedig az elemszám-1, vagyis listanév.Count-1.

Az Insert medódussal egy adott indexű helyre szúrhatunk be új elemet:

listanév.Insert(index, új lestaelem);

Egy adott elemet a Remove metódussal törölhetünk. Paramétere a törlendő elem, ha több ilyen van, akkor az első előfordulást törli.

listanév.Remove(listaelem);

Index szerint is törölhetünk egy elemet a RemoveAt tagfüggvénynel:

listanév.RemoveAt(index);

A teljes lista törlése a Clear metódussal történik, ekkor minden elem törlődik és a listánk üres lesz:

listanév.Clear();

Egy adott elem keresése a listában:

listanév.Contains(keresett elem);

A metódus visszatérési értéke bool, vagyis a visszatérési érték true vagy false logikai konstans.

Egy adott feltételnek megfelelő elem keresése a listában

Ezt a műveletet a Find metódus végzi el:

listanév.Find(feltétel);

A visszaadott érték a lista feltételnek megfelelő eleme, ha van találat.

A paraméterben megadott feltétel alakja:

változónév => összehasonlítás

A FindIndex metódus működése hasonló, viszont itt a visszaadott érték a talált elem indexe.

Ha a keresés során több találat is van és ezeket szeretnénk egy új listába menteni, akkor ezt a műveletet a FindAll függvény végzi el. Visszatérési értéke ugyan olyan típusú lista, mint amire alkalmazzuk.

Rendezés

Egy lista elemeit a Sort metódussal tudjuk rendezni:

listanév.Sort();

A rendezés a tartalmazott elemek típusától függő növekvő rendezés.

Ha ettől eltérő rendezést szeretnénk készíteni, akkor a paraméterében egy rendező kifejezést kell megadni, amely értéke a CompareTo metódus eredményével egyezik meg:

listanév.Sort(rendező kifejezés);

A rendező kifejezés:

(változó1, változó2) => -1, +1 vagy 0-át visszaadó összehasonlító kifejezés