Fogalmak

**Alapok**

**compiler:** fordítóprogram, ami lefordítja az adott nyelvet gépi byte nyelvre.

**program.cs**: C#-ban ezzel a nevezetű fájlal kezdődik az egész, amit később lefordítunk.

**szintaktikai hiba:** Abban az esetben, ha a programozási nyelv "nyelvtanában" hibáztunk valamit pl. sor végén ; kell mindig a C#-ban, és a compiler nem fordítja le programkódot, hanem valamilyen hibát dob.

**szemantikai hiba:** ha a programunk hiba nélkül lefordul, de működése közben valami nem stimmel vele (pl.: prímszámokat kiíró program esetén kiírja az 1-et is), erre NEM figyelmeztet a fejlesztőeszközünk, ezért a program futása közben ellenőrizzük a kimenet valóságát.

**namespace:** névtéraz a logikai egység, melyben az azonosítónak egyedinek kell lennie a többi névtértől.

**Adatszerkezetek**

**változó:** A változó a memóriában lefoglalt bitek azonosítója és tartalma.

**deklarálás:** egy változónévvel ellátott típust hozunk létre érték nélkül pl: int szam1;

**definiálás**: más szóval inicializálás a változónévvel ellátott típusnak értéket is adunk. pl: int szam1=1

**típusok:** egy-egy algoritmusban a megadott típus értékének tárolására használt memória tartományának elnevezése pl int,double,string,char stb.

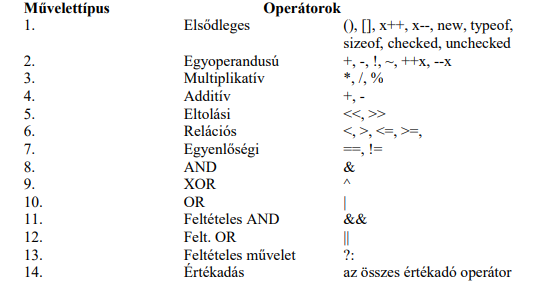
**prefix:** használati alakja pl: ++a itt először növeli az ’a’ változó értékét majd a többi részt.

**postfix:** használati alakja pl: a++ itt csak utólag adódik hozzá az ’a’ változóhoz az 1.

**Túlcsordulás:** Akkor beszélünk túlcsordulásról, ha az adott érték már nem fér el a változónak fenntartott tárterületen például: byte (0-255; 8bit) típusnál még a 255 belefér a tárterületbe, de ha eggyel (vagy többel) növeljük, akkor túlcsordulás következik be: a 255 kettes számrendszerben: 11111111 (8bit), ha 1-el növeljük, akkor már nem fér bele a 8bitbe így (1)00000000 lesz tehát 0 tizesszámrendszerben. C#-ban az adattípus.MaxValue/MinValue megtudhatjuk az adott típus legnagyobb/legkisebb MÉG eltárolható értékét, példa: Console.Writeline(byte.MaxValue); Ami a konzolra annyit ír ki, hogy 255.

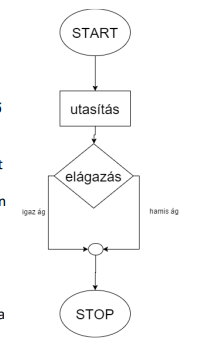
**Precedenciai rangsor**

A precedenciai rangsor egy olyan rangsor, amelyet a nyelv követ és olyan sorrendben értelmezi a műveleteket, amilyen sorrendben le van írva ebben a táblázatban. Például a matematikában is a () jeles részek után a szorzást, majd az egyéb műveleteket vesszük figyelembe.

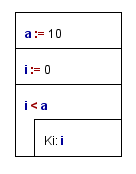


**Algoritmust leíró eszközök**

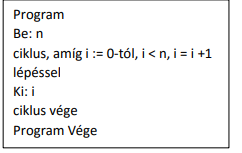
**A folyamatábra:** olyan algoritmust leíró eszköz, mely a programot gráfként írja le. Ez egy olyan irányított gráf, amely csomópontokból áll, egyetlen induló és befejező része van. A következő kép megmutatja ezeket a csomópontokat, utasításokat…, és talán így már könnyebben megérthető lesz később a ciklusok használata. Minden egyes program egy Start ponttal indul és egy Stop ponttal fejeződik be. A nyilak a program útját mutatják, az utasítást egy téglalap írja le, a csomópontokat a kör jelöli. Később megismerkedünk a feltétellel (rombusz), ami szinte kihagyhatatlan lesz a programunkból, mert még a ciklusok is tartalmazzák!

****

**Struktogram:** jóval könnyebben rajzolható, viszont nehezebben áttekinthető, mint a folyamatábra. Itt egy téglalap írja le az egész programot, amelyet további téglalapokra, háromszögekre osztanak.



**Mondatszerű leírás:** programot mondatokkal írjuk le a Program szóval és a Program vége szavakkal ér véget például:

****

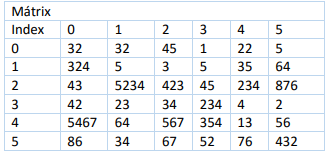
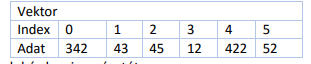
**Összetett adatszerkezetek: Tömbök**

**tömb:** adatok tárolására alkalmas, viszont a sima adatszerkezetekkel ellentétben ez már egy összetett adatszerkezet, ami képes egy bizonyos változónév alatt több elemet is tárolni egy index segítségével. Képzeljünk el egy szekrényt, aminek a neve „ruhák”. A szekrény fiókjai pedig meg vannak számozva, és minden számhoz tartozik egy ruhanemű, például az 1-es fiókban vannak a zoknik, a 2-esben a nadrágok, a 3-asban a pólók… C#- ban így lehetne valahogy létrehozni ezt a „ruhás” szekrényt:

****

Ha a tömbnek nem adunk értéket alapértelmezetten 0-val fog feltöltődni szám típusok esetében, egyéb típusoknál a null értéket kapja. A tömbök indexelése 0-tól kezdődik, a 0. elem az első, az 1. elem a második…, ezért az utolsó elemet mindig az eltárolható elemek n-1 indexe mutatja meg. A tömböket az indexük segítségével tudjuk használni, általános formája: tömbnév[index] Ugyanúgy használhatók, mint a változók. Ha a tömb egy dimenziós (egy indexe van) **vektornak**, ha 2, akkor már **mátrixnak** nevezzük.

Valahogyan így lehetne elképzelni egy vektort és egy két dimenziós mátrixot:

C#-ban a tömbök tudják a méretüket, így azt le lehet kérni, vektor esetében a tömbnév.Length-el. Mátrixok esetén a mátrixnév.GetLength(dimenzió száma)-val lehet lekérdezni a méretét. A már deklarált tömb méretét már nem tudjuk befolyásolni.

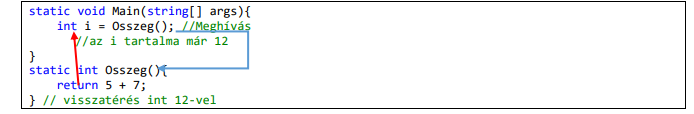
**Bővebben a mátrixról:** : a mátrix egy olyan kétdimenziós tömb melynek van szélessége és magassága is, míg a vektor csak 1 magas volt és bármekkora széles addig ez már legegyszerűbben egy táblázatként fogható fel, használata majdnem ugyan az, mint a vektornál csak itt már két indexet kell megadnunk, hogy elérjünk egy cellát. Deklarálása hasonló, mint a vektorok esetében, csak egy vesszőt kell beszúrnunk a négyzetes zárójelek közé. Pl.: int[,] mátrix = new int[4, 5]; Ilyenkor egy olyan int típusú mátrix jön létre, melynek szélessége 4 cella, magassága pedig 5. Mint már mondtam a több dimenziós tömbök esetében a tömbnév.GetLength(dimenzió szám) tudjuk lekérni az aktuális dimenzió méretét, a dimenziók számozása is 0- tól kezdődik és a megadott dimenziókig tart. Szintén tudunk kapcsos zárójelekkel létrehozni egy már 0-tól eltérő adatokat tartalmazó tömböt, ilyenkor szintén nem kell megadnunk a tömb méretét. A mátrix elképzelhető egy olyan vektorként melynek celláiban újabb vektorok vannak, viszont ezek a vektoroknak meg kell egyezniük a többi cellában lévő vektorok méretével, ezek a tömböket szabályos tömböknek, négyzetes mátrixoknak, multidimenziós tömböknek nevezzük. Léteznek olyan többdimenziós tömbök, melyeknek a mutató vektoron (fő vektor) belül lévő vektorok mérete nem azonos, ezek a mutatóvektorok.

**konstans:** létrehozhatunk c# nyelvben olyan változókat melyeknek az értéke állandó, tehát biztos, hogy nem fog változni, így a program csak a változó értékét foglalja le a memóriában. Létrehozhatunk konstans tömböt is (pl.: a magyar abc-t tartalmazó betűkészlet tömböt), általános létrehozási alakja, amit általában a deklaráláskor kell megadnunk: const változótípus változónév = érték;

**Alprogramok**

**alprogram:** A programok felépítése alapvetően szekvenciális a főprogram a Main a programunk belépési pontja, innen kezdődik a futás, az itt meghívott függvények hajtódnak végre. Az alprogramok a programkód lerövidítésére és átláthatóságára szolgál, minden egyes saját eljárásnak vagy függvénynek meg van a maga szerepe és a szerepére utaló neve, így, ha valamilyen javítandó probléma lenne a programunkban, akkor könnyen nyomon követhető a program felépítése és jobban korrigálható. Az alprogramok általános alakja: static típus Név(paraméter1, p2,..){utasítás1; utasítás2… (ha függvény return valami;)} 2 fő alprogram típust különböztetünk meg, minden alprogram a static kulcsszóval kezdődik, ennek lényegéről majd szó esik az objektum orientált programozás résznél, a paraméterezésről majd később bővebben.

**eljárás:** : nincs visszatérési értéke, ezért void (üres) típusú:



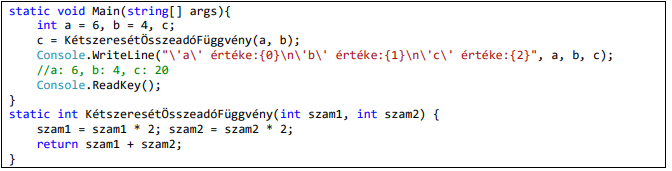
**függvény:** van visszatérési értéke, a visszaadott értéktől függ, a return utasítással adjuk meg a visszaadott értéket, ami minden esetben kötelező a megadása.

****

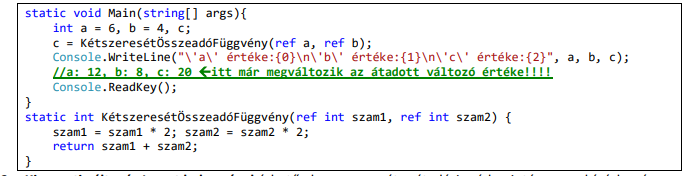
**rekurzió:** Az olyan műveletet, melynek végrehajtásakor a saját műveletsorait hajtja végre, tehát önmagát ismétli, rekurziónak nevezzük. Programozás esetén a függvények saját magukat hívják meg. Amikor egy függvény nem önmagát, hanem egy másik függvényt hív meg, majd ez a függvény pedig meghívja azt a függvényt, amiből meghívták (…), akkor azt kölcsönös rekurziónak nevezzük (pl. A() meghívja B()-t, majd B() meghívja A()-t…).

**Az alprogramok paraméterezése:** A metódus deklarálásánál megadott paramétereket formális, a metódus meghívásakor megadott paramétereket pedig aktuális paraméternek hívjuk. Az eljárásokat és függvényeket 3-féle módon paraméterezhetjük: érték, cím és egy u. kimeneti paraméterként, a paramétereket vesszővel választjuk el egymástól, ha értéket is adunk az alprogramunk fejlécében a paramétereknek, akkor a meghíváskor nem lesz kötelező megadni az adott paramétert.

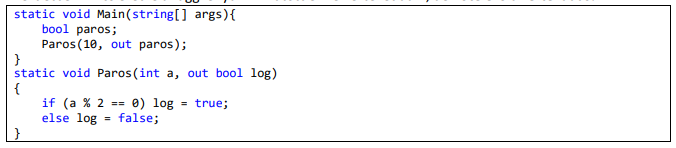
**Érték szerinti paraméterátadás:** Ha nem adunk meg semmilyen kulcsszót, akkor ilyen módon adódnak át a változók (kivéve tömbök, objektumok... ). Ilyenkor az alprogram meghívásakor megadott aktuális paraméterek értéke átmásolódik az alprogram egy-egy megfelelő formális paraméterbe, ezután a formális paramétert ugyanúgy használhatjuk, mintha egy változó lenne.

****

**Cím (Referencia) szerinti paraméterátadás:** A ref kulcsszóval érhető el a cím sz. p., a ref kulcsszót az aktuális és a formális paraméter előtt is kell használni! Ebben az esetben az átadott aktuális paraméterre, változóra egy hivatkozás jön létre a memória tartományára, csak a memóriaterület címe/azonosítója adódik át, így az alprogramon belül végzett változtatások érvénybe lépnek az átadott változóban is! Fontos: az összetett adatszerkezetek, tömbök, objektumok (…) minden esetben referencia, tehát cím szerint adódnak át!

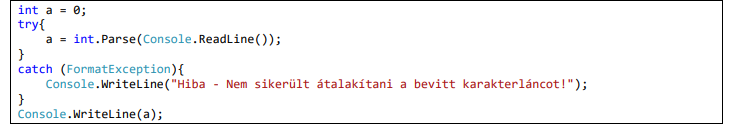
****

**Kimeneti változó:** Az out kulcsszóval érhető el ez a paraméter átadási mód, szintén a meghíváskor és az alprogram deklarálásakor is meg kell adni! Hasonlít a cím sz. p. -ra viszont az átadott paraméternek kezdetben nincs érétke a függvényünk hívatott ennek értéket adni, de kötelező az értékadás!

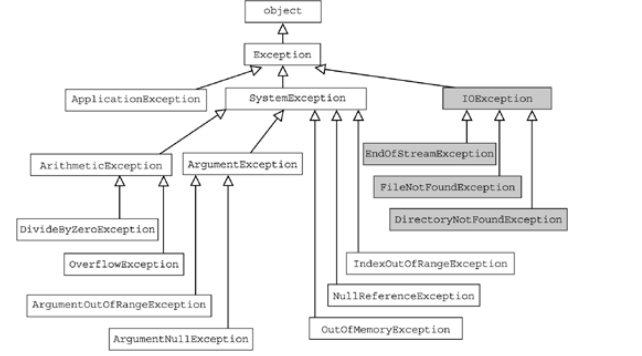
****

**Hibakezelés**

Abban az esetben eddig, ha egy nem kezelt kivétel (Exception) keletkezett be, akkor kifagyott a programunk és a hibát vagy a visual studio, vagy az operációs rendszer kezelte le, és egy hibaüzenetet dobott. A programunk, ekkor kifagyott. Például a string int-re való parsolásánál, ha nem egész számot adtunk meg, vagy karaktereket is megadtunk a string literálba, akkor FormatException hiba adódott, bár a TryParse erre megoldást ad, de nem minden kivételnek van egy-egy megfelelő, a hibát kezelni képes függvénye. C#-ban a kivétel egy objektum, amit akkor indít útjára egy függvény, ha valamilyen hiba történt, ekkor a programunk hibakezelés nélkül elszáll. A kivételek elkapását és lekezelését a **try-catch** blokk végzi, a try blokkot védett blokknak nevezik, itt keletkezhet a kivétel, melyet majd a **catch blokk** fog majd lekezelni, a catch-nak meg kell adni egy kivétel objektumot, ami utal a hiba típusára, egy try-blokk után több catch-blokk is következhet, hiszen érdemes a különböző kivételekre, különböző módon reagálni, nem csak azt kiírni, hogy Hiba! Amint egy kivétel keletkezik a try blokkban az utána következő utasítások nem hajtódnak végre! Nézzünk egy példát: Kérjünk be egy int számot, és kezeljük le az esetleges FormatException-t, mivel a try is egy blokk, el ne felejtsük a bekért változót már deklarálni a try előtt, különben nem tudjuk később használni!



Minden kivétel őse (eredete) az Exception osztály, ha ezt állítjuk be a catch blokkba, akkor bármilyen kivétel is dobódik, minden hiba lekezelésre kerül. A kivételek „családfája”:



**Finally blokk:** a finally blokkot minden esetben az utolsó blokk-ként kell használni, az itt megadott utasítások minden esetben lefutnak, ha keletkezett kivétel, ha nem! Példa:

