Forrás: http://www.doksi.hu

1. Bevezetés a C programozási nyelvhez

A C nyelvet eredetileg a UNIX operációs rendszerhez fejlesztették ki, a hetvenes évek legelején. Az azóta eltelt idő bebizonyította, hogy a C nemcsak UNIX környezetben életképes, hanem más operációs rendszerek alatt is rendkívül jól használható programnyelv.

A C magasszintű programnyelv, amelyet nagyfokú tömörség, korszerű vezérlési szerkezetek, igen jól használható adatstruktúrák, bőséges operátorkészlet, előre a gyártó által elkészített könyvtári függvények jellemeznek. Ez a programnyelv nem kötődik egyetlen felhasználási területhez sem, viszont a felhasználó nem ütközik lépten-nyomon korlátokba, mint más magasszintű nyelvek esetén. Ezért a C sok más nyelvnél kényelmesebb és hatékonyabb. Bizonyíték erre, hogy sok felhasználási területen kiszorította az assembly-t, illetőleg más magasszintű nyelvek is C-ben íródnak. A C másik rendkívüli előnye, hogy egy programon, egyidőben több fejlesztő is dolgozhat.

A PC kompatibilis számítógépeken több C megvalósítás látott napvilágot, ezek közül a legjelentősebbek a Microsoft cég és a Borland cég termékei. A Microsoft C rendszerei MSC xx márkanévvel - az utóbbi idõben a Windows alá írt verziók VISUAL C++ x.xx néven - míg a Borland implementációk régebben TURBO, manapság már Borland márkanévvel szerepelnek.

2. Alapfogalmak

Elsőnek vizsgáljuk meg hogyan keletkezik a forrásnyelvű szövegből futtatható program. A folyamat a következő:

-A felhasználó által megírt forrásnyelvű állományt egy u.n. **előfeldolgozó** mintegy átszűri, kiveszi belőle a fordítási művelethez szükségtelen információkat, a fölösleges szóközöket, a megjegyzéseket és a feltételes fordítás kondícióinak megfelelően egy "csupasz" forráslistát ad át a tényleges fordító számára.

-Az előfeldolgozó által szolgáltatott állományt a **compiler** úgynevezett object állománnyá fordítja. Ez az object még nem futtatható, még sok járulékos információt tartalmaz arra az esetre, ha több ilyen állományt akarunk összeszerkeszteni.

-A **linker** feladata, hogy compiler által létrehozott objecteket összeszerkessze és futtatható kódot állítson elő.

A fenti összefoglalás természetesen nagyon vázlatos, de ennek a segédletnek nem célja a fordító és szerkesztő programok működésének leírása. Azonban ahhoz, hogy később bizonyos fogalmakat tisztán lássunk, okvetlenül szükséges ezeket a fordítási fázisokat ismerni.

Egy C program egy, vagy több függvényből, változókól, operátorból és utasításból épül fel. Ezekből a legszükségesebb az úgynevezett **main()** függvény, amelyet minden futó programnak tartalmaznia kell. Ezt a függvényt vezetőfüggvénynek hívjuk. A függvény megnevezés lefedi a más programozási nyelvekből már ismert szubrutin fogalmát, illetve a PASCAL-hoz hasonlítva az eljárásokat és annak függvényeit. A függvényekkel részletesebben egy következő fejezetben foglalkozunk. Az előzőek alapján nézzünk meg egy egyszerű C programot.

```
#include    <stdio.h>                    /* inklúzió */

main()                                   /* a main kulcsszó */
{                                        /* nyitó jel */
```

Forrás: http://www.doksi.hu

```
    printf("Ez egy C program !");        /* a program törzse */
}                                        /* a lezáró jel */
```

A 2. sorban láthatjuk a **main()**-t. A main()-t egy { karakter követi, ez nyitja meg a main() függvényt, ezután következik a program törzse, amely ebben az esetben egy printf() hívást tartalmaz, ezek után a programot egy } karakter zárja le. Ebből már kiderül, hogy minden C program tulajdonképpen egy main() függvény

A két kapcsos zárójel jelen esetben egy **függvény törzset** zár közre, de **általánosságban** amit a két "{..}" jel közé írunk azt a C fordító **logikailag egy egységnek** fogja tekinteni.

Láthatjuk, hogy a printf() függvény után egy **";"** található. **Általános érvényű szabály, hogy az utasításokat és a függvényeket mindig ";"-vel zárjuk le**. A C megengedi, hogy egy sorba **több kifejezést** írjunk. Ezalól csak az előfeldolgozó számára szóló utasításcsoport kivétel.

A fenti mintaprogramunk nem tartalmaz sem **operátort**, sem **változót**. Az operátorok és a változók, illetőleg az utasítások és a függvények **közé tetszőleges számú szóközt** lehet tenni. Az is általános szabály, hogy **ahová szóközt** tehetünk, oda tehetünk **tabulátort**, vagy **újsor karaktert** is.

A legelső sorban található a #include utasítás az előfeldolgozó számára szól. Minden **előfeldolgozó utasítás** a # karakterrel kezdődik.

Mintaprogramunkban minden sor mellé fűztünk **megjegyzést**. A megjegyzések a következő jelpárok közé kerülnek /*.....*/.

A C programozási nyelv, ellentétben sok más magas- és alacsonyszintű programnyelvvel, **különbséget tesz a nagybetűvel és a kisbetűvel írt karakterek és szavak között.** Tehát például nem tekinti azonosnak az "alma" és az "ALMA" karakterláncokat. Erre érdemes figyelni, hogy később sok kellemetlenségtől kímélhessük meg magunkat.

A következő alapfogalom az **objektum** fogalma. Objektum-nak nevezünk a nyelv minden olyan elemét, amely a fordítás után helyet foglal el a tárban. Az objektum lehet **kódgeneráló** és lehet **területfoglaló**. A kódgeneráló objektumok a függvények, a területfoglaló objektumok a változók.

3. Változók és típusaik, típuskonverzió, típusdefiniálás

A C és ezen belül a Borland C++ is az alábbi, tovább már nem bontható alaptípusokkal rendelkezik:

-karakter, jelölése: **char**
-egészek, jelölése: **int**
-lebegőpontos, jelölése: **float**, illetve **double**

Ezek a típusok ugyan tovább már nem bonthatók, de többféle mérettel és ábrázolási móddal rendelkezhetnek.

Egy változó **deklarálása** a következőképpen történik:

```
char      a;   /* Az a változó karakter típusú */
int       i;   /* Az i változó egész típusú */
float     f;   /* Az f változó float típusú */
double    d;   /* A d változó double típusú */
```

Forrás: http://www.doksi.hu

A típusok **adatábrázolási módja gép** és **implementáció függő**. IBM-PC-s környezetben az alaptípusok adatméretei az alábbiak:

```
char       8 bit      -128  ...    127
int       16 bit    -32768  ...  32767
float     32 bit    ±3.4e±38
double    64 bit    ±1.7e±308
```

Más gépek esetén ezek a típusok természetesen különbözhetnek. A fentiekből kiderül, hogy a változók **alapesetben előjeles** értelmezésűek. A változók **méretét** és **értelmezését** megváltoztathatjuk, ezt úgy érhetjük el, hogy a típust jelző kulcsszó elé egy **módosító jelzőt** teszünk.

Az **előjeltelen ábrázolási mód** módosító jelzője az **unsigned**. Használható **char** és **int** típusú változók esetén. A változó által lefoglalt terület **mérete nem** változik meg, csak az **adatábrázolás módja**.

```
unsigned char   8 bit    0          -      255
unsigned int   16 bit    0          -    65535
```

Az egész típusú változók számára több módosító jelző adható ki. Az első az u.n. **rövid ábrázolási mód** jelzője a **short**. Ennek a Borland C++ esetén jelentősége igazából nincs, mert sem az adat méretét, sem az ábrázolását nem változtatja meg.

```
short int       16 bit   -32768    -    32767
```

A következő módosító jelző, a **hosszú ábrázolási mód** jelzője a **long**. A long hatására az adatábrázolás mérete kétszeresére növekszik.

```
long int       32 bit    -2147483648  -  +2147483647
```

Az **egész típusú változók esetén** mindig a **hosszt módosító jelző elé** ki lehet tenni az **unsigned** jelzőt. Pl.:

```
unsigned long int l;
```

Egész típusú adatábrázolás esetén nem szükséges az egész deklarációs sort kiírni, elegendő csak a módosító jelzővel jelezni, hogy mit szeretnénk deklarálni. **Alapértelmezésben** a C minden változót **int** típusúnak tekint, tehát ha nem írjuk ki a teljes sort, azt automatikusan int-nek értelmezi. Pl.:

```
unsigned int u;     egyenértékű    unsigned u;
unsigned long int lu;    -"-       unsigned long lu;
```

Egész típusú változók esetén a C nyelv megengedi, hogy ne csak decimális, hanem **oktális** és **hexadecimális** formátumban is **adjunk értéket a változóknak**. **Oktális** ábrázolás esetén a változó számjegyei **0-7** tartományon belül kell legyenek és az első számjegy elé mindig egy 0-át kell írni. Például a 255d szám 0377 lesz a C-ben. **Hexadecimális** ábrázolás esetén a számjegyek **0-f**, vagy **0-F** tartományba kell essenek és a számot egy **0x**, vagy egy **0X** előzi meg. A 255d tehát 0xff, vagy 0XFF lesz.

A lebegőpontos ábrázolási módban két típust különböztetünk meg. Ezek a **float** és a **double**. A lebegőpontos ábrázolás hosszú típusa a **double**.

Ha valakinek ez a típus nem elég nagy, használhatja a **long double** típust. A long itt módosító jelző.

```
long double    80 bit ±3.4e±4932
```

Forrás: http://www.doksi.hu

Az alaptípusokon kívül létezik egy speciális felsorolási típus, az **enum**. Ez szintén egész értéket használ. Akkor célszerű alkalmazni, ha nem a változó értéke a fontos, hanem csak az például, hogy két érték egyenlő e, vagy nem. Használata a következő:

```
enum napok
{hetfo,kedd,szerda,csutortok,pentek,szombat,vasarnap};
```

Ebben a példában a hetfo értéke 0, a kedd értéke 1, a szerda értéke 2, stb lesz. Az enum 16 biten előjelesen van ábrázolva, ezért az értéktartománya megegyezik az int értéktartományával.

Az változók deklarálásánál lehetőség van a **Borland C++** esetén a változó kezdeti értékének megadására a következő módon:

```
int  i=51;
```

Ez bármilyen alap és módosított változótípusra igaz, így az enum típusra is. Az **enum** esetén nem kötelező a számlálásnak 0-ról indulnia, adhatunk **más kezdőértéket** is, nézzünk egy példát a rendszerből:

```
enum modes {LASTMODE=-1,BW40,C40,BW80,C80,MONO=7};
```

Az előző példában a LASTMODE értéke -1 lesz, a BW40 0, a C40 1, a BW80 2, a C80 3, a MONO 7.

Megadhatunk tetszőleges típusú konstanst is a **const** módosító jelző segítségével. A const azt eredményezi, hogy az adott nevű **objektumra** (nem írunk változót, mert konstans) történő minden **értékadást** a compiler **fordítási időben megakadályoz**, vagyis hibajelzést ad. Használata:

```
const int konstans=5;
const float pi=3.1415926;
```

A változókkal kapcsolatos speciális jelző a Borland C++ esetében a **volatile**, amely azt jelzi a fordító számára, hogy a változó, amelyet ezzel a jelzővel ellátunk, bármely pillanatban módosulhat. A fordító ennek hatására a változót olyan tárterületen helyezi el, amelynek helye a program futása során nem változik meg. Ezt a jelzőt megszakítási függvények változóinál szoktuk kiadni.

Az alaptípusok egymás között **konvertálhatók**, tehát a típusok adott műveletek esetén átalakulnak, illetőleg átalakíthatók. A típuskonverzió alapesetei a következők:

-A műveletekben a **char** és az **int** kifejezések szabadon keveredhetnek, a kifejezésekben a **char értéke automatikusan int lesz.** Nézzük az alábbi példát.

```
char c;
int  i;
int  e;
     .
     .
e=i+c;
```

Ebben az esetben, a művelet végzése alatt a c értéke int lesz úgy, hogy az előjelét megtartja. Az IBM PC-ken futó C változatok esetén a char és az int típusok kettes komplemens számábrázolással szerepelnek, ebben az esetben ez azt jelenti, hogy a kibővített c nevű változó magasabb helyiértékű byte-jának összes bitje felveszi az eredeti c változó

Forrás: http://www.doksi.hu

MSB-jének értékét. **Ez természetesen csak a művelet idejére igaz, a c változó végérvényesen nem módosul.**

-Ha a változók típusa a művelet során **float és double a float automatikusan double lesz.**

-Típuskonverzió lehetséges értékadás esetén is. Nézzük az alábbi példát.

```
float x;
int   i;
      .
      .
x=i;        /* x felveszi i értékét float formában */
      .
      .
i=x;        /* i felveszi x egészrészének értékét */
```

-Lehetséges **típuskonverziót** úgymond **kikényszeríteni** az u.n. **cast** szerkezettel. Ez szintaktikailag a következő:

```
(kívánt_típus)változó_neve    Pl.
(float)i;      /* i értéke float-tá konvertálódik */
```

A cast szerkezetet akkor használjuk, ha a típuskonverzió **automatikusan nem hajtódik** végre, például egy függvény paraméterezésénél.

Nézzünk egy összetettebb példát a típuskonverzióra:

```
char        c;
int         i;
float       f;
double      d,result;
      .
      .
result=(c/i)+(f*d)-(f-i);
```

-A **(c/i)** művelet alatt a **c** int típusúvá alakul át.
-Az **(f*d)** alatt az **f** double lesz.
-Az **(f-i)** alatt az **i** float lesz.

-Az első és a második zárójeles kifejezés összeadása közben az aritmetikai művelet double típusban hajtódik végre éppúgy, mint a harmadik kifejezés esetén. Az eredmény kiszámításánál a műveletek nagy része double típuson történik, igy az eredmény, amely egyébként is double lenne, viszonylag pontos lesz.

A felhasználó számára a C nyelv lehetőséget biztosít **típusok definiálására**. Ehhez a **typedef** kulcsszó segít hozzá. A típusdefiniálás módja a következő:

```
typedef   int     egesz;
   |       |        |
   |       |       Az új típus.
   |      Az "ős" típus.
  A kulcsszó.
```

Itt kell megjegyeznünk, hogy a typedef kulcsszóval nem csak egyszerű típusok definiálhatók, de ezt a struktúrákkal és unionokkal kapcsolatos fejezetben tárgyaljuk.

4. Mutatók

Forrás: http://www.doksi.hu

A C nyelvben nem csak a **változókat**, hanem a **tárcímüket** is elérhetjük ezeket a tárcímeket speciális változókban tároljuk. Ezeket a speciális változókat **mutatóknak** nevezzük. A mutatók **általában valamilyen típushoz tartoznak**. Deklarációjuk az alábbi alapján történik:

```
típus     *mutató_neve;
  |        |      |
  |        |     A mutatót azonosító név.
  |       A csillag jelzi, hogy mutatóról van szó.
  A mutató típusa (milyen típusú változó címét
  tartalmazza).

Pl.:
int       *ip;       /* Egész típusra mutat */
float     *fp;       /* float-ra mutat */
```

A C **nemcsak** az **alaptípusokhoz**, hanem a **definiált típusokhoz** is engedélyezi **mutatók rendelését**. Nézzük a következő példát:

```
typedef   int  EGESZ;
EGESZ     *E_MUT;
```

Látható, hogy a példa első sorában egy új típust definiáltunk EGESZ néven, majd a második sorban deklaráltunk egy mutatót, amely egy egész típusú változóra fog mutatni.

Felmerül a kérdés, hogy a mutató hogyan kap értéket. Erre a célra a C rendelkezik egy **operátorral**, amely egy változóhoz rendelve megadja a **változó tárbeli helyét.** Figyeljük meg ennek menetét az alábbi példán.

```
int         i;    /* Egy egész */
int         *ip;  /* Egy egészre mutató pointer */
.
.
ip=&i;            /* Az értékadás sora */
```

Az **&** operátor az u.n. címképző operátor.

A mutatónak már tudtunk értéket adni, kérdés az, hogyan tudunk **a mutató által címzett tárrekesz tartalmához hozzájutni.** Erre a C nyelvben szintén egy operátor szolgál, melynek használatát ismét egy példán mutatjuk be.

```
int         i=5;  /* Egy egész típusú változó, értéke 5 */
int         j;    /* Szintén egy egész érték nélkül */
int         *ip;  /* Egy egészre mutató pointer */
     .
     .
ip=&i;            /* Az ip értéket kap */
j=*ip;
```

Figyelem nagyon fontos, hogy ne keverjük össze az előbbi indirekciós operátort a deklarációnál használatos csillaggal, bár formailag azonosnak tűnnek, de jelentésük teljesen más. Az egyik jelzi a fordító számára, hogy mutatóról van szó, a másik operátor.

A C nyelvben mutatókat nem csak egyszerű és származtatott változótípusokra lehet rendelni, hanem gyakorlatilag minden **objektum** rendelkezhet mutatóval. Ezeket a "kritikus" helyeken ismertetjük.

Forrás: http://www.doksi.hu

5. Operátorok

A C nyelv következő lényeges alkotóelemei az u.n. **operátorok**. Az operátorok műveleteket jelölnek ki, illetőleg logikai és szintaktikai elválasztásokat jeleznek. Az operátorok csoportjai az alábbiak:

-elválasztást végző operátorok,

-aritmetikai operátorok,

-relációs operátorok,

-logikai operátorok,

-bitműveletek operátorai,

-speciális operátorok.

5.1. Elválasztást végző operátorok

Az elválasztást végző operátorok a következők:

"{}" A **logikai blokkhatárt** jelző operátorpár, amely kijelöli a fordító számára az **egységként kezelendő részeket**, például egy program törzsét. Lásd a következő példán.

```
main()          /* A program vezető függvénye */


         {         /* A prg. törzsét nyitó op. */
         .
         .         /* A program törzse. */
         .
         .         /* A prg. törzsét záró op. */
         }
```

"()" A **hagyományos zárójelezést** végző operátor, amelynek jelentése gyakorlatilag megfelel a **matematikai zárójelezésnek**.

"[]" **Tömbök indexeléséhez** használt operátorpár. A tömbökről a későbbiekben lesz szó, addig is használatát az alábbi példán megtekinthetjük.

```
tomb[i]
2dtomb[i][j]
3dtomb[i][j][k]
```

"," A vessző operátor esetén be kell vezetnünk egy új fogalmat a **"kötési szabályt"**.

A kötési szabály azt mondja meg, hogy a kérdéses operátor és a körülötte lévő operandusok **milyen sorrendben** kerülnek kiértékelésre.

A kiértékelés iránya lehet **balról-jobbra**, **jobbról-balra**, vagy **nem értelmezhető**. A vessző esetén a kiértékelés iránya balról-jobbra történik. **(rövidítve BJ).** A vessző használata legegyszerűbben egy példa segítségével sajátítható el.

```
int   a;
int   b;
int   c;
    .
    .
a=b,b=c;
```

Forrás: http://www.doksi.hu

Ez az utasítássor egyenértékű az alábbival:

```
a=b;
b=c;
```

Ebben az esetben egyértelműen látszik a BJ kiértékelési irány, de vizsgáljuk meg ezt a példát.

```
a=(b=c=2,c=c+3);
```

Az kifejezés kiértékelése után az "a" értéke 5, a "b" értéke 2 és a "c" értéke szintén 5 lesz, miért?

Azt mondtuk, hogy a kiértékelés iránya BJ, ezért ilyen irányban járjuk végig a fenti kifejezést. A zárójelen belül baloldalt van egy kettős értékadás, melynek eredményeként a "b" és a "c" felveszi a kettes értéket, ha jobbra lépünk a "c" értéke egy rekurzív összeadás eredményeként hárommal megnő. Azt mondtuk a zárójel operátor esetében, hogy jelentése megegyezik a matematikában használatos zárójel jelentésével, tehát ebben az esetben a zárójeles kifejezés értékelődik ki először. Ezek alapján az "a" a "c" legutolsó értékét veszi fel. A kifejezés teljesen egyenértékű a következő kifejezéssel:

```
c=2;  b=c;  c=c+3; a=c;
```
[1]

A vessző operátor másik felhasználási módja a **deklarációknál** fordul elő, ahol **az egy típusba tartozó változókat egy sorba is deklarálhatjuk**. Az alábbi két kifejezés teljesen egyenértékű.

```
int   a;
int   b;  ≡  int a,b,c;
int   c;
```

5.2. Aritmetikai operátorok

Az aritmetikai operátorok, mint nevük mutatja **aritmetikai műveleteket** jelölnek ki. Ezek az alábbiak:

+	összeadás operátor,
-	kivonás operátor,
*	szorzás operátor,
/	osztás operátor,
%	modulo (maradék) képzés operátor,
=	értékadás operátor,
-	Egyoperandusú mínusz.

A **"+"**, a **"-"**, a **"*"** operátor a matematikában megszokott értelmezéssel rendelkezik, kiértékelési sorrendjük **BJ**.

Az osztás operátora a **"/"** az előzőektől eltérően nem azonos módon viselkedik különböző változó típusok esetén. Ha **egészként ábrázolt számokat osztunk egymással az eredmény a törtrészeket nem fogja tartalmazni**, ezek elvesznek.

A **modulóképzés** operátora a **"%"**, ez az operátor egész típusú változók osztása során keletkező **maradékot** adja meg.

---

[1] A véleményem az, hogy ha valaki programot ír, kerülje ezeket a vesszős trükköket, mert egy-két nap múlva ő maga sem fogja saját programját érteni.

Forrás: http://www.doksi.hu

Az **"="** operátor az **értékadás operátora**, amely valamely **balértékhez**, valamely **jobboldali kifejezés értékét rendeli** hozzá. Használatát a következő példákon mutatjuk be:

```
a=5;       /* Az "a" felveszi az 5 értéket */
a=b+c;     /* "a", "b" és "c" összegével lesz egyenlő */
```

```
b+c=a;     Ez teljesen rossz !!!!!!!!!!!!!!!
```

Az **egyoperandusú mínusz "-"** egyetlen változó, vagy kifejezés értéknek **előjelét változtatja meg**.

Az aritmetikai operátorok a **rekurzív** jellegű műveletekre egy **rövidítést engednek meg**, amelyet a következő példánkon mutatunk be:

```
a=a+1;   =   a+=1;
b=b-1;   =   b-=1;
c=c*3;   =   c*=3;    stb.
```

5.3. Relációs operátorok

A C nyelv is, más programozási nyelvekhez hasonlóan, ismeri a relációs operátorokat. Egy relációs vizsgálat eredménye **akkor igaz, ha a keletkező érték nem nulla**, ellenkező esetben hamisnak minősül a vizsgálat. Ezt az információt a későbbiekben felhasználhatjuk. A relációs operátorok a következőek:

<	kisebb,
>	nagyobb,
<=	kisebb vagy egyenlő,
>=	nagyobb vagy egyenlő
==	egyenlő,
!=	nem egyenlő,

Gyakran előforduló hiba, hogy programírás közben az egyenlőség vizsgálatot végző "==" operátor helyett az értékadás operátorát használjuk fel. A C fordító erre **figyelmeztető üzenetet küld,** de megengedi ezt a fajta műveletet, hiszen a relációs vizsgálathoz a művelet eredményét használja fel és nem magát az operációt. A compiler joggal feltételezheti, hogy a programot író személy azt a trükköt akarja alkalmazni, hogy egy értékadást és egy relációs műveletet egyetlen lépésben végez el. Ez az operáció ugyan megengedett, de mi a magunk részéről mindenkit óvunk tőle.

Ezeknek az operátoroknak is a **kiértékelési sorrendje BJ**.

5.4. Logikai operátorok

A logikai operátorok alapvetően a **relációs kifejezések összefűzésére szolgálnak**. Ezek segítségével képezhetünk **többszörös feltételrendszereket**. A logikai operátorok az alábbiak:

&&	"és" logikai művelet,
||	"vagy" logikai művelet,
!	"nem" logikai művelet.

Az **"&&"** és **"||"** operátor két kifejezés logikai értéke között végzi el a kijelölt logikai műveletet. Ha több operáció van kijelölve a **műveletsort csak addig végzi, míg az eredmény biztos "nem" lesz**. Nézzünk erre egy egyszerű példát.

```
elso - igaz, masodik - hamis, harmadik -hamis
```

Forrás: http://www.doksi.hu

```
elso || masodik || harmadik
      ⇑A műveletsor idáig hajtódik végre

elso - hamis, masodik - igaz, harmadik -hamis
elso || masodik || harmadik
                 ⇑A műveletsor idáig hajtódik végre

elso - hamis, masodik - hamis, harmadik -igaz
elso || masodik || harmadik
                            ⇑A műveletsor idáig
                             hajtódik végre
```

A fenti példából látható, hogy "vagy" műveletek esetén az első igaz kifejezésig, "és" művelet esetén az első hamis kifejezésig értékelődik ki a kifejezés.

A **"!"** operátor az igaz értékből hamisat, a hamisból igaz értéket csinál.

5.5. Bit operátorok

A bit operátorok egész és karakter típusú változókon végeznek műveleteket. Az operátorok a következőek:

&	bitenkénti és,
|	bitenkénti vagy,
^	kizáró vagy,
<<	balra siftelés,
>>	jobbra siftelés,
~	egyes komplemens képzés (bitenkénti negáció).

Az **"&,"** a **"|,"** a **"^"** operátorok **kétoperandusú** műveleteket jelölnek ki, ami azt jelenti, hogy a két változó **minden egyes bitjére a kijelölt műveletet végbemegy** és ennek az eredménye kerül a balértékbe.

Itt kell **felhívnunk a figyelmet** arra, hogy hasonlóan az egyenlőséget vizsgáló és értékadó operátorokhoz, itt is egy **hibalehetőség adódik**. Nem mindegy, hogy egy feltételcsoport vizsgálatakor a logikai **"&&"**, vagy **"||"**, vagy a bitenkénti **"&"** vagy **"|"** operátorokat írjuk be.

A **"<<","">>"** operátorpár a változó bitjeinek léptetésére szolgál, használatát a következő példán mutatjuk be.

```
a=b << 6;
```

A balérték "a" ebben az esetben felveszi a "b" változó **hatszor balrasiftelt értékét**. A **"b" értéke nem változik meg**.

Ha "a" értéke 2, akkor "b" értéke 128 lesz a művelet után.

Balrasiftelésnél az alsó bitek helyére **nullák kerülnek**, viszont ha **jobbra siftelünk**, **ez nem egyértelmű**, mert a belépő bitek örökölhetik az MSB értékét, illetőleg lehetnének nullák is. Sajnálatos módon ez **implementáció függő**.

Az egyeskomplemens képző operátor **"~"** a változó minden bitjét invertálja és ez a negált érték kerül a balértékbe. Ez az operátor **egyoperandusú** és a kötési szabály szerint **BJ**.

5.6. Speciális operátorok

Forrás: http://www.doksi.hu

Ez a csoport azokkal az operátorokkal foglalkozik, amelyeket nem lehetett nyugodt szívvel egyik csoportba sem sorolni, mert valamely tulajdonságuk réven kilógnak az eddig felsorolt típusok közül. A következő operátorokról lesz szó:

++	inkrementáló operátor,
--	dekrementáló operátor,
&	címképző operátor,
*	indirekció operátor,
.	elem kijelölõ operátor struktúráknál és unionoknál,
->	indirekció operátor struktúráknál,
?:	három operandusú operátor,
(cast)	típus konverzió "operátor",
sizeof	méret megadó operátor.

A "++" és a "**--**" operátorok jellegüket tekintve **aritmetikai operátornak** minősülhetnének, azonban használatukban van egy lényeges különbség, ez pedig az, hogy alkalmazásuk esetén **nincs szükség balértekre**. Nézzünk egy példát használatukra :

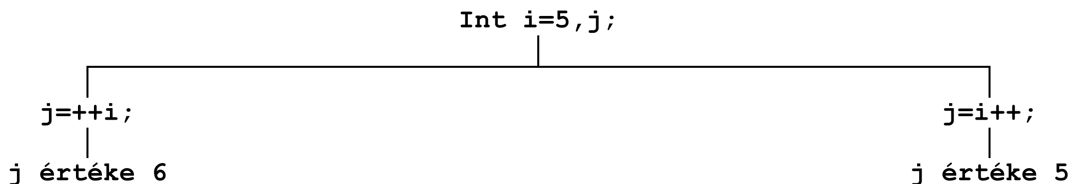
```
int   i=5;              /* Az i értéke 5 */
        .
        .
i++;                    /* Az i értéke 6 lett */
```

Ez alapján kideríthető, hogy az **i++;** kifejezés aritmetikai jelentésében teljesen megegyezik az **i=i+1;** kifejezéssel. A különbség az, hogy az **eredmény azonnal képződik** az i változóban.

Ez a két operátor kiadható **más formában** is, nézzük hogyan:

```
++i;       --i;
```

Mi a különbség az **i++;** és a **++i;** között? Első lépésben ehhez meg kell vizsgálni az operátorokra érvényes **kötési szabályt**, amely **JB**. Ez egyszerű felhasználás esetén, amikor a teljes kifejezést az inkrementálás, illetve a dekrementálás alkotja nem okoz problémát, gyakorlatilag a két írási mód egyenértékű. Azonban, ha a **kifejésben más elemek is vannak**, például egy értékadás, rögtön jelentkezik a **kiértékelés irányából történő különbség.** Próbáljuk ezt a kijelentést egy példával megvilágítani.



Az "**&**" és a "*" operátorokról a mutatóknál már ejtettünk szót. Egyetlen dolgot kell róluk még megjegyezni, hogy ellentétben az operátorok nagy többségével, az operátor és a változó közé **nem kerülhet szóköz**.

> Itt is adódik hibalehetőség, vigyázni kell, hogy **ne keverjük össze a szorzás és az indirekciós, valamint a bites és a tárcím operátorokat.**

A "**.**" operátort a struktúráknál és a unionoknál fogjuk tárgyalni

Forrás: http://www.doksi.hu

A **"->"** operátor az u.n. **mutatóképző operátor**, amelynek használatát a **struktúráknál** fogjuk tárgyalni.

A **"?:"** az u.n. **háromelemű operátor**, amely tulajdon-képpen egy rendkívül egyszerű vezérlési szerkezetet valósít meg. Felépítése a következő:

```
(1.kifejezés) ? (2.kifejezés) : (3.kifejezés)
```

Az **1. kifejezés egy relációs kifejezés**, amely ha **igaznak bizonyul a 2. kifejezés** teljesül, amennyiben a feltétel **hamis a 3. kifejezés** teljesül. Nézzünk egy példát:

```
c=(a>b)?(a):(b);
```

Ha "a" nagyobb, mint "b", "c" értéke "a" értékét veszi fel, ha "b" nagyobb, vagy egyenlő "a" értékével "c"-be "b" értéke kerül.

A példa alapján nyilvánvaló, hogy ennek az operátornak a **kiértékelési iránya BJ**.

A "**cast**"szerkezetről a típusoknál esett szó.

A "**sizeof"** operátor egy objektum **méretét adja meg**. Alkalmazása a következő:

sizeof(objektum_neve);

Az objektum mérete a **PC-s környezetben byte**-ban jelenik meg.

5.7. Operátorok precedenciái és kötési szabályai

Operátorok	Kötési szabály
{}	Értel-metlen
() [] . ->	BJ
- ~ ! * & ++ -- sizeof (cast) | | | (címképző operátor) (indirekciós operátor)	JB
* / %	BJ
+ -	BJ
<< >>	BJ
< > <= >=	BJ
== !=	BJ
^	BJ
&	BJ
|	BJ
&&	BJ
||	BJ
?:	BJ
= += -= *= /=	BJ
,	BJ

6. Utasítások

Forrás: http://www.doksi.hu

A C nyelvben más, programozási nyelvekhez viszonyítva, rendkívül **kevés utasítást** találunk. Ennek oka az, hogy a C-ben az utasítások helyét a függvények vették át. A C utasításai alapvetően **két csoportra** oszthatók. Az első csoportba az u.n. **program utasítások**, a másikba az **előfeldolgozó utasításai** tartoznak. A program utasítások a program **tényleges futása során** fejtik ki hatásukat, míg az előfeldolgozó utasításai a **fordítás első fázisában** játszanak szerepet.

6.1. Program utasítások, szerkezetek

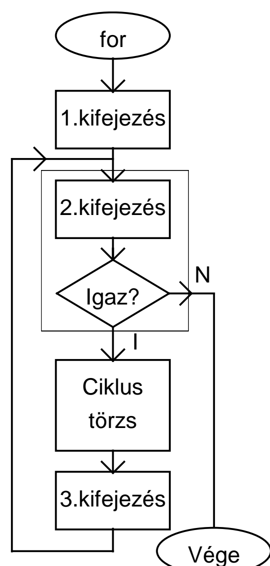
A C programozási nyelvben a program utasítások (**ezentúl utasítások**) alapvetően **vezérlési szerkezetek** létrehozásához szükségesek, ezért ezeket együtt tárgyaljuk.

Az első utasításcsoport, amelyről beszélünk, a **ciklus-szervező** utasítások.

6.1.1. A for utasítás

A **for** utasítás egy **általános ciklusszervező utasítás**, amely rendkívül könnyűvé teszi ennek a vezérlési szerkezetnek a megvalósítását. Használata a következőképpen néz ki:

```
for(1.kifejezés;2.kifejezés;3.kifejezés)
     { ciklustörzs }
```



A folyamatábrából már világosan látszik a for utasítás működése, de a pontosabb megértés céljából kövessük végig szóban is.

-Az **első** kifejezés **csak egyszer** hajtódik végre, ezért ezt leginkább a **ciklusváltozó kezdeti értékadására** használjuk fel.

-A **második** kifejezés tulajdonképpen **feltételvizsgálatot** hajt végre. Ha a kijelölt feltétel **igaz**, a vezérlés a **ciklustörzsre** adódik, ha **hamis**, a ciklust **befejezi**.

-A **harmadik** kifejezésre a **ciklustörzs végrehajtása után** kerül a vezérlés, ezért célszerű a **ciklusváltozó értékét** ennek segítségével változtatni.

Nézzünk erre egy konkrét példát:

Ez a kis programrészlet 0-tól 9-ig összeadja a számokat.

```
int   i,j;
        .
        .
j=0;
for(i=0;i<10;i++)
        {
        j=j+i;
        }
```

A for utasítás **nem követeli** meg, hogy mindhárom kifejezés helye **ki legyen töltve**, ezeken a helyeken lehet **üres utasítás** is. Pl.:

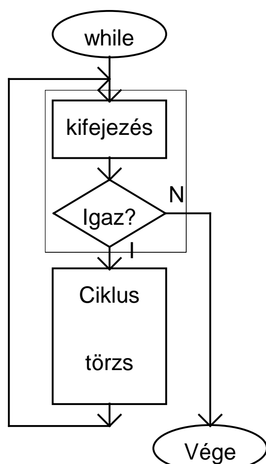
```
for(;;)/* így csinálunk végtelen ciklust */
```

6.1.2. A while utasítás

A **while** utasítás **elöltesztelő** ciklus létrehozásában játszik szerepet. Az utasítás szerkezete a jobboldalon látható.

```
while(kifejezés)
      {
      ciklustörzs
      }
```

Forrás: http://www.doksi.hu



A működése a következő: az utasítás argumentumában lévő kifejezés kiértékelése során, ha azt **igaznak találja**, a **ciklustörzsre** kerül a vezérlés, ha a feltétel **hamis**, **kilép** a ciklusból. Lásd a folyamatábrát:

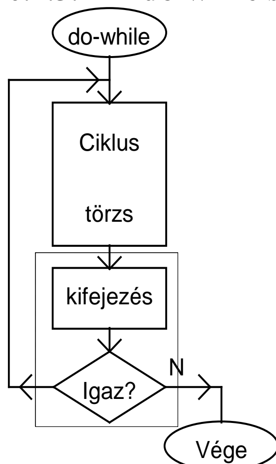
Nézzük a következő példát:

```
int   i=0;j=6;

while(i<j)
      {
      i+=3;
      j+=2;
      }
```

A while segítségével is létrehozhatunk **végtelen ciklust** a **while(1)** kifejezés használatával. Ne feledjük, ami nem nulla az igaz.

6.1.3. A do-while szerkezet



Az előző utasítások mind elöltesztelő ciklus létrehozására voltak alkalmasak. Ez az utasításpár a **hátultesztelő** ciklust támogatja. Szerkezete a következő:

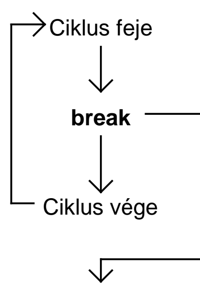
```
do
      {
      cilustörzs
      }
while(kifejezés);
```

A cilustörzs **egyszer mindenképpen** lefut, csak ezután történik meg a kifejezés vizsgálata, melynek eredményétől függ, hogy a ciklus tovább fut, vagy befejeződik.

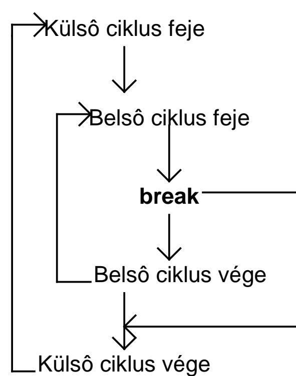
6.1.4. A break utasítás

A break utasítás az éppen **aktuális ciklus elhagyására** szolgál. Amikor erre az utasításra a vezérlés rákerül, a program a ciklusból kilép, de csak az **aktuális ciklusból**. Tehát ha több ciklust ágyazunk egymásba, a break hatására mindig **csak eggyel feljebb** kerülünk. Nézzük meg a break működési gráfját.

Egy ciklus esetén

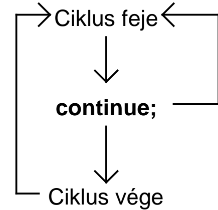


Egymásba ágyazott ciklusok esetén

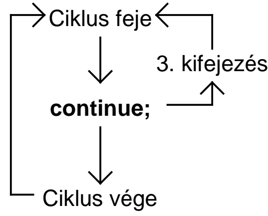


Forrás: http://www.doksi.hu

6.1.5. A continue utasítás



A continue utasítás szintén a ciklusok **"átszervezésére"** használható. Amikor a vezérlés a continue-ra kerül a ciklus törzse **nem folytatódik tovább, hanem elölről kezdődik.** Nézzük meg a continue működési gráfját.

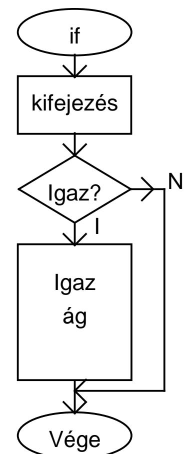


Felmerül a kérdés, hogy a **for szerkezet** esetén mi történik, ha kiadunk egy **continue** utasítást. Ebben az esetben a 3. kifejezés végrehajtódik, tehát a fent látható futási gráf így módosul.

6.1.6. Az if utasítás

A break és continue során felmerülhetett az a kérdés, hogyan kerülhet a vezérlés a két kérdéses utasításra, hiszen ha egyenesen írjuk be a végrehajtás sorába, nem igazán van értelmük. Azt a kérdést elegánsan elhallgattuk, **hogyan tud** a program valamely feltételtől függően **eltérni az u.n. egyenes** végrehajtási iránytól.

A program **feltételes elágaztatására** az **if** utasítás szolgál. Az utasítás szerkezete a következő:




```
if(kifejezés)     kifejezés;
```

vagy

```
if(kifejezés)
      {
      igaz ág
      }
```

Ha a kifejezés **igaz**, a vezérlés az **if utasítást követő kifejezésre**, vagy a **{} operátorpár közé zárt blokk-ra adódik**. Amennyiben a feltétel **hamisnak bizonyul**, a program futása az u.n. **igaz ág után folytatódik**. A működés a folyamatábrán követhető.

Ezek után elhelyezhetjük a break és continue utasításokat egy if igaz ágában, mert egy adott feltételtől függően fog csak a vezérlés rájuk kerülni.

Nézzünk egy példát:

```
int i=0;
      .
      .
while(1)
      {
      i++;
      if(i==12) break;
      }
```

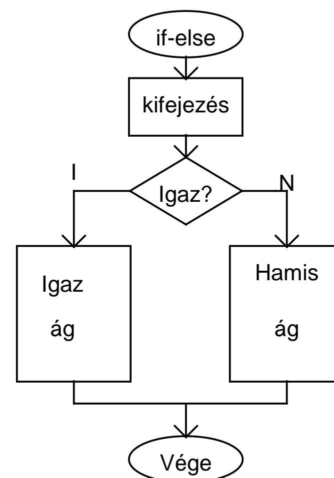
6.1.7. Az else utasítás és az if-else szerkezet

Ha egy **feltételvizsgálat során** olyan igény merül fel, hogy a feltétel **teljesülésekor** a program **egy utasítás csoportot**, **nemteljesülése** során **egy másik utasításcsoportot** hajtson végre, akkor kell az if-else szerkezetet használni. Az **else** utasítás jelöli ki a **hamis ág** kezdetét. **Jellemző tulajdonsága**, hogy **if nélkül nem létezhet**.

A if-else szerkezet a folyamatábrán nyomon követhető:

Forrás: http://www.doksi.hu

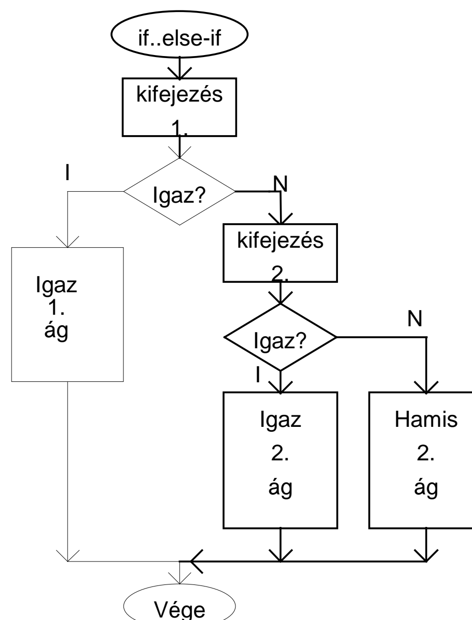
Használatát egy példán nézzük meg.




```
int    i=0,j=12;
       .
       .
while(1)
       {                  /* A ciklus kezdete */
         .
       if(i<j)
              {           /* Az igaz ág kezdete */
              i++;
              }           /* Az igaz ág vége */
       else
              {           /* A hamis kezdete */
              i-=5;
              }           /* A hamis ág vége */
         .
       }                  /* A ciklus vége */
```

6.1.8. Az else-if szerkezet

Az **else-if** szerkezet **nem jelent új utasítást**, de annyira elterjedt a használata, hogy célszerű néhány szót ejteni róla. Az else-if szerkezet az egymásba ágyazott feltételek esetén fordul elő, használatára nézzünk egy egyszerű programrészletet, amely ASCII karakterekből hexadecimális számjegyet állít elő.



A jobboldalon látható folyamatábra analóg a program-részlettel

```
char   c;    /* A karakter */
int    i;    /* A számjegy */
       .     .
if(c>='0'&&c<='9')      /* kifejezés 1 */
       {
       i=c-'0';
       }
else if(c>='A'&&c<='F') /* kifejezés 2 */
             {
             i=c-'A'+10;
             }
       else
             {
             i=c-'a'+10;
             }
       .
```

6.1.9. A switch-case szerkezet

Abban az estben, ha **egy változó értékétől** függően **sok elágazást** kell csinálni, az if-es kifejezések nagyon bonyolult, sok írással járó programokat eredményeznének, amelyet természetesen senki sem szeret. Erre a célra készült a fejezetcímben szereplő **switch-case** szerkezet. Nézzük ennek a használatát.

Forrás: http://www.doksi.hu

```
switch(változó_neve)
      {
      case 1.érték : amit_csinálni_kell_1.értékre;
      case 2.érték : amit_csinálni_kell_2.értékre;break;
      case 3.érték :
      case 4.érték : amit_csinalni_kell_3.,4.értékre;
             .
             .
      default:amit_egyébként_csinálni_kell;
      }
```

A **switch** utasítás a **változó értékét** az **összes case értékével összehasonlítja**. A vezérlés arra a **case** sorra kerül, **amelynél** a **változó** és a **case értéke megegyezik**. A case úgy viselkedik, **mint** egy **címke**, tehát ha a vezérlés rákerült egy kérdéses case-ra a program onnan fog tovább futni. Amennyiben a case-hez tartozó utasítás sorban nem találunk break utasítást a program a következő case-en folytatódik.

A **default**-ra akkor kerül a vezérlés, ha **egyetlen case sem** teljesült. A **default elhagyható**.

Nézzük a megszokott példánkat, a hexadecimális karakter egész konverziót:

```
int   i;
char  c;
      .
      .
switch(c)
      {
      case '0':               /* számok */
      case '1':               /* Lehet így is!*/
      case '2':               /* Vagyis egymá alá írjuk a */
      case '3':               /* case-ket. */
/* De lehet egymás mellé is írni. őket */
      case '4':case '5':case '6':case '7':case '8':
      case '9':i=c-'0';break;
      case 'A':case 'B':case 'C':case 'D':case 'E':
      case 'F':i=c-'A'+10;break;
      case 'a':case 'b':case 'c':case 'd':case 'e':
      case 'f':i=c-'a'+10;break;
      default:c=0xff;break;   /* nem hexa */
      }
```

6.1.10. A goto utasítás

A C nyelv készítői a **goto** utasítás létrehozását azzal indokolták, hogy a break utasítás egymásba ágyazott ciklusok esetén mindig csak egy szintet lép vissza, ezért ha egy belső ciklusból akarjuk az egész szerkezetet elhagyni szükség lehet egy címre történő ugrásra. **Mindazonáltal a goto utasítás használatát, ha csak egy módunk van rá kerüljük**. Használatához egy **címkére** is szükség van. A címke egy **string**, amelyet **kettősponttal** zárunk le. A goto használata az alábbi egyszerű példán látszik:

Forrás: http://www.doksi.hu

```
            .
            .
      GOTO CIMKE;
            .
            .
CIMKE:           /* FUJJ */
            .
            .
```

6.2. Az előfeldolgozó utasításai

Az előfeldolgozó utasításai, mint már említettük olyan utasítások, amelyek **nem a futási**, hanem a **fordítás időben** fejtik ki hatásukat. Ezek az utasítások a felhasználó "életét" teszik könnyebbé, javítják a program olvashatóságát, rövidítik a forrásprogram hosszát. Minden előfeldolgozó utasítás **# karakterrel kezdődik** és mindig **csak egy sort foglalhat el**.

6.2.1. A #define utasítás

A **#define** utasítás **két alapvető** feladatot lát el. Az első, hogy **fordítás időben konstanst** tudjunk **definiálni**. Nézzük hogyan:

```
#define konstans_neve érték
```

A konstans, amelyet így definiálunk, fordítás időben felveszi a helyettesítési értéket. Ez a konstans definíció **filozófiáját** tekintve **teljesen** más, mint a **változó konstans deklarációja**, mert ebben az esetben **csak értékbehelyettesítés** történik, **nincs lefoglalva** a konstans számára **memóriaterület**, tehát ez **nem** hoz létre **objektumot**.

A következő példán keresztül vizsgáljuk meg a #define működését:

```
#define ZERO 0          /* 0 helyett ZERO */
.
char  c;
.
if(c==ZERO)             /* Forrásszövegben konstans */
      {                 /* ZERO helyére 0 kerül */
      .
      }
.
```

A **#define** másik felhasználási módja az u.n. **makro definiálás**, ebben az esetben is tulajdonképpen egy érték behelyettesítés történik, csak **számérték** helyett egy **kifejezést helyettesítünk**. A makro definiálásra két példát mutatunk be, az egyik egy egyszerű példa, a másik egy tipikus felhasználás.

a.

```
#define     PRINT(c)    bdos(5,(c),0)     /* definíció */
      .
      .
PRINT('a');                         /* felhsználás */
      .
```

A compiler a következő szöveget fogja fordítani:

```
bdos(5,('a'),0);        /* Nem kell ezt a szép hosszú és */
                        /* keveset mondó függvényhívást */
                        /* leírni */
```

Forrás: http://www.doksi.hu

b.

```
#define     max(a,b)    ((a>(b)?(a):(b))
        .
        .
int     i,j;   .
        .
        .
max(i,j);
        .
```

A fordítandó szöveg:

```
((i)>(j)?(i):(j));       /* A max név többet mond */
```

6.2.2. A definició megszüntetése

Egy érvényes definíció megszüntetésére az **#undef** utasítás szolgál. **Attól** a **sortól kezdve**, ahol ez **le volt írva**, a **definícó érvénytelen lesz**, vagyis a #define hatása megszűnik.

Használata a következő:

```
#undef      azonosító
```

6.2.3. Az #include utasítás

A következő előfeldolgozó utasítás az **#include**, amelynek szerepe az, hogy a fordítás folyamán egy **megjelölt file-t beépít a fordítási folyamatba**. Az #include-ot szinte minden C programban **megtaláljuk**, mert a gyártók által definiált **könyvtári függvények deklarációi** ilyen beépítendő file-okban foglalnak helyet, más **közhasznú** információ mellet. Természetesen mi is készíthetünk ilyen file-okat, ha vannak sűrűn előforduló kifejezéseink.

Használata nagyon egyszerű, gyakorlatilag csak le kell írni, a helyes szintaxis a következő:

```
#include    <Beépítendő_file> ,vagy
#include    "Beépítendő_file"
```

A két leírási mód között gyakorlatilag nincs különbség. A rendszerhez tartozó úgynevezett **header** file-okat szoktuk a <> karakterek közé tenni, az **""** közé pedig **saját termékeink** nevét írjuk.

Ügyeljünk arra, hogy a beépített file-ban lévő forrásszövegek **szintaktikailag hibátlanok legyenek.**

6.2.4. A feltételes fordítás utasításai

A feltételes fordítás utasításai a következőek:

```
#if, #endif, #else, #ifdef, #ifndef, #elif
```

Minden egyes **feltételes fordításra kijelölt** blokkot az **#endif** utasítás **zár le**.

Az **#if** utasítás **feltételes fordítást nyit meg**, ha az utána írt kifejezés **igaz**, az előfeldolgozó az **#endif**, **#else**, **#elif** utasításokig a megfelelõ szövegrészt **átadja** a fordítónak. Amennyiben a feltétel **hamis** a szövegrész **rejtve marad** a compiler előtt, még szintaktikai hibát is tartalmazhat.

Az **#else** utasítás hasonló működésű, mint azt a program else utasításánál láthattuk. Ha az **#if** utasítás argumentumában a **kifejezés értéke hamis**, az előfeldolgozó az **#else**-től az **#endif**-ig adja át a szöveget a fordítónak.

Az **#elif** utasítás hatása a program **else if szerkezetéhez hasonlatos**, tulajdonképpen megfelel egy **#else** és egy őt követő **#if** utasításnak.

Nézzünk ezekre az utasításokra egy-egy példát:

Forrás: http://www.doksi.hu

```
#if kifejezés          #if kifejezés          #if kifejezés
     |                      |                      |
#endif                 #else                     #elif
                            |                        |
                       #endif                    #endif
                                               #endif
```

Az **#ifdef** és az **#ifndef** utasítások hatása gyakorlatilag azonos az **#if**-ével, csak **nem kifejezést vizsgálnak**, **hanem szimbólumot**, mégpedig olyan szempontból, hogy volt e a kérdéses szimbólum definálva, vagy nem. Az **#ifdef** akkor lesz **igaz**, ha szimbólum **volt definálva**, az **#ifndef** akkor, ha **nem**. Utánuk **minden** más, az **#if esetében kiadott** feltételes fordító szerkezet **használható**.

7. Függvények

A C programozási nyelvnek egyik leglényegesebb alkotó-részei a **függvények**. A függvények segítségével végezhetünk minden olyan műveletet, amelyet az operátorokkal és az utasításokkal nem tudunk elvégezni.

7.1. A függvények általános jellemzői

A függvények alapvetően **két** csoportra oszthatók, úgynevezett **könyvtári** és a saját magunk által írt **felhasználói függvények**.

A **könyvtári függvényeket** az adott C implementáció előállítója **adja** a **rendszerrel**. (Illetőleg nekünk is van lehetőségünk a könyvtárakat bővíteni.)

A felhasználói függvényeket **mi írjuk** forrásnyelven. Ezek szerepelnek a programunkban.

A függvények rendelkeznek **típussal** és **paraméterekkel**. A függvény **típusa** attól függ, hogy **milyen típusú változóval tér vissza** a függvény. A **paraméterek** a függvény **bemeneti értékei**.

Az előzőek szerint, ha egy függvény int típusú változóval tér vissza, akkor int típusú függvényről beszélünk. Azonban előfordulhat, hogy a **függvényünk nem ad vissza értéket**, ekkor típusa u.n. **void** lesz.

7.2. Függvények definiálása

A függvények **definiálásán** azt "műveletet" értjük, amikor a függvényt tulajdonképpen **megírjuk**. Nézzük meg hogyan épül fel egy függvény:

```
típus azonosító(paraméterek_azonosítói)
      paraméterk_deklarációi;
      {
      belső_változók_deklarációi;
      függvény_törzs
      return(visszatérési_érték);
      }
```

Típus: megjelölést az előzőekben már megbeszéltük.

Azonosító: a függvény **neve**.

Paraméterek azonosítói: azoknak az változóknak a **nevei**, amelyeket **átadunk a függvénynek**. A függvény törzsén belül ezekkel a nevekkel hivatkozunk a kérdéses értékekre.

Forrás: http://www.doksi.hu

Paraméterek deklarációi: azok a sorok, amelyek az **átadott paramétereket deklarálják**, **típus** szerint. **Lényeges**, hogy ez a deklaráció még a **kapcsos zárójeleken kívül** van.

A belső változók deklarácói: azok a sorok, amelyek a **függvény által használt** és **csak** a **függvény által látott** változók deklarálását végzik. (Részletesen lásd a láthatóság fejezetnél.)

Függvény törzse: amely tartalmazza azokat az **utasításokat**, esetlegesen azokat a **függvény hívásokat**, amelyek tulajdonképpen a függvény **funkcionális működését** biztosítják.

return: utasítás, vagy függvény (nem egyértelmű) **nem kötelező**, ennek a segítségével tudunk **adatot átadni** a hívónak, ha **függvényünk void, nincs szükség a return**-ra.

Példaként írjuk egy egyszerű függvényt, amely három egész típusú változóból kiválasztja a legnagyobbat, és azt adja vissza.

```
int   max3(a,b,c)        /* Figyelj!! Nincs ";" */
      int   a;           /* Átadottak deklarációja */
      int   b;
      int   c;
      {
      int   i,j;         /* Belső változók */
      i=(a>b?a:b);       /* a, vagy b nagyobb */
      j=(i>c?i:c);       /* c, vagy i nagyobb */
      return(j);
      }
```

Ha a **függvénynek nincsenek paraméterei**, akkor a definícónál a paraméter sorba, vagy **nem írunk semmit** (a zárójelek kötelezőek), vagy a jól ismert **void** kulcsszót. A Borland C++ az utóbbit szereti.

A Borland C++ az **ANSI** szabványhoz igazodva megengedi az átadott paraméterek **deklarálását** a **paramétersorban**. Az előző példa ebben a formában a következő lesz:

```
int max3(int a,int b, int c)
      {
      int   i,j;
      i=(a>b?a:b);
      j=(i>c?i:c);
      return(j);
      }
```

Bár ez a leírás rövidebb, de eltér a C megszokott szintaxisától. Ismerete viszont fontos, hogy tudjuk olvasni mások programjait.

7.3. Függvények deklarációja

Amikor egy függvényt deklarálunk, **tulajdonképpen** a fordító számára **szolgáltatunk információt** az alábbiakról:

-A függvény **alkalmazásra kerül** a kérdéses C programban. (Ha a függvényt mi írtuk természetesen nyilvánvaló, de gondoljunk arra az esetre is, ha könyvtári, vagy más programozó által írt szegmensben szereplő függvényt akarunk használni).

-Megmondja milyen **típusú** a **függvény**. (Alapértelmezésben a C minden függvényt int típusúnak tekint, ezért ha mi írtunk egy függvényt és nem deklaráltuk a compiler figyelmeztető üzenetet küld, de nem jelez hibát.)

-Megmondja, hogy **hány darab** és **milyen típusú** paraméterekkel történik a hívása.

Egy függvény deklarációja a következőképpen néz ki:

Forrás: http://www.doksi.hu

```
típus azonosító(paraméterek_típusai);
  |       |         |                 |
  |       |         |                 A pontos vessző
  |       |         Típusfelsorolás
  |       Függvény neve
  Függvény típusa
```

A **típus** és az az **azonosító jelentése megegyezik** a **definícónál** elmondottakkal. A **paraméterlistában elegendő** felsorolni a változók **típusát**, nincs szükség a nevükre, bár ez is megadható. **A deklarációs sort mindig pontosvessző zárja le, ellentétben a definíciós sor fejlécével.**

Nézzük meg a maximumkereső függvényünk deklarációját:

```
int   max3(int ,int ,int);
```

A **deklarációnak mindig** az **adott program**, vagy **modul elején** kell lennie.

7.4. Függvények mutatói

A **függvényekhez** is, mint ,más **objektumokhoz**, lehet **rendelni mutatót**. Ennek értelmét első pillanatban az ember nem is érti, azonban ha arra gondol, hogy a függvényét megszakítás kiszolgálásra akarja használni, rögtön értelmet nyer ez a mutató fajta is. Természetesen ez csak egy a lehetséges példákból, dörzsölt programozó igen sok apró és nagy trükköt tud a **függvénymutatók** segítségével csinálni.

Hogyan **deklarálunk** egy függvénymutatót? Nézzünk erre egy példát:

```
int  fgv(void); /* Ez egy int típusú függvény */

int  (*fpnt)()  /* Ez egy mutató, amely egy int */
                /* típusú függvényre mutat */
```

Adjunk a mutatónak **értéket**.

```
fpnt=fgv;       /* Az étékadás */
```

Hogyan lehet egy függvényt indirekt módon meghívni? Erre ad választ a következő sor.

```
(*fpnt)();      /* Ennyi az egész */
```

Rögtön felmerül a jogos kérdés, hogy lehet e **átvenni**, illetőleg **átadni** paramétert ilyen **indirekt függvényhívás esetén**. Természetesen **lehet** és nem is túl bonyolult. Nézzük példának a sokat koptatott maximum kereső függvényünket.

Forrás: http://www.doksi.hu

```
int   max3(int,int,int);        /* deklaráció */
int   i1,i2,i3,e;               /* változók */
int   (*fpnt)();                /* függvénymutató */
      .
fpnt=max3;                      /* étékadás */
      .
      .
e=(*fpnt)(i1,i2,i3);            /* indirekt hívás */
      .                         /*  paraméterekkel */
int   max3(a,b,c)
      int   a;
      int   b;
      int   c;
      {
      int   i,j;
      i=(a>b?a:b);
      j=(i>c?i:c);
      return(j);
      }
```

Ez az utóbbi fejezet (7.4.) **nem igazán a kezdő programozók számára szól**. Ahhoz, hogy az ilyen trükköket alkalmazzunk alaposan kell ismernünk a nyelv struktúráját. A további ismeretek a 12. fejezetben találhatók.

8. Összetett adatformátumok, címarimetika

Aki járatos más magasszintű programozási nyelvben, biztosan felfigyelt egy-két úgymond hiányosságra. Ezek közül a legszembetűnőbb a string típus hiánya. Lehetséges, hogy ez a C-ből hiányzik? Igen, de a C nyelv tervezői ezt a kérdést áthidalták, lássuk hogyan.

8.1. Tömbök

Az operátoroknál már említettük a tömböket, a szögletes zárójelekkel kapcsolatban. A **tömbök** egyszerű **alap**, vagy **definiált**, illetőleg **összetett** változókból létrehozott **rendezett halmazok**, melynek elemei **azonos tulajdonságokkal** rendelkeznek. A **tömbök deklarációja** a következőképpen néz ki:

```
char tomb_neve[x];
  |       |     |
  |       |     Az tömb elemeinek az adott
  |       |     dimenzióban a maximális száma
  |      A tömb azonosítója
 A tömb típusa
```

Nézzünk egy példát egy egydimenziós, egy kétdimenziós és egy háromdimenziós tömb deklarációjára.

```
int         itomb[5];         /* 5 elemű int tömb */
float ftomb[3][5];            /* 3x5-ös float tömb */
char        ctomb[4][6][2];   /* 3 dim.char tömb */
```

Egy **N elemű** tömb esetén az **index változó** értéke **0-tól N-1-ig terjedhet**, tehát egy 5 elemű tömb első indexe a 0-ás, az utolsó a 4-es értéket veheti fel. Ezek a tömbelemek a memóriában **egymásután folytonosan helyezkednek el** olymódon, hogy a **legalacsonyabb indexű** elem van a **legalsó** és a **legmagasabb indexű** a **legfelső címen**.

Forrás: http://www.doksi.hu

Többdimenziós tömbök esetén, a memóriában lefoglalt helyet a tömb, az index szerint úgy tölti ki, hogy a **legutolsó indexváltozó** szerint **egymásután** elhelyezkedő elemek vannak **közvetlenül** egymás **mellett**.

Nézzünk erre egy példát:

```
char  t[3][2];    /* 3x2-es karaktertömb */
```

t[0][0]	A legalacsonyabb cím.
t[0][1]	
t[1][0]	
t[1][1]	
t[2][0]	
t[2][1]	

A tömb elemeinek adhatunk **értéket egyenként**, illetőleg van lehetőség a tömb összes elemének **egylépésben történő ér-tékadására** is. Az **egyenként** történő értékadás semmiféle problémát nem okoz, **ugyanúgy** történik, mintha **közönséges változót töltenénk fel**. Az **egylépésben történő feltöltést** a **kezdeti érték** megadásánál alkalmazzuk. Használatát egydimenziós tömb esetén az alábbi példa segítségével mutatjuk be:

```
int   tomb[5]={ 9,8,7,6,20 }; /* Ez deklaráció */
```

A példánkban tehát a tomb[0] elem értéke 9, a tomb[1] elem értéke 8, stb lett. A fenti értékadási **példában** előre megadtuk a tömb **maximális méretét**, ha **kevesebb** elemet írunk a kapcsos zárójelek közé, a **fennmaradó elemek értéke 0** lesz, ha **többet** a **compiler panaszkodni** fog. Ha **el akarjuk kerülni** a kezdeti értékadás esetén azt, hogy a fordító **ne küldjön hibaüzenetet** és **nem kötött** az **elemek száma**, akkor **ne írjunk** a **szögletes zárójelek** közé a **deklarációnál semmit**. Nézzünk újra egy példát:

```
int   tomb[]={ 9,8,7,6,20,30 };
```

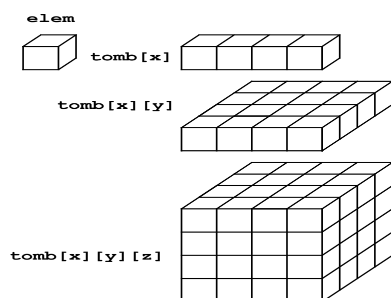
A tömb **automatikusan** 6 elemű lett, hibaüzenetet nem kaptunk.

A többdimenziós tömbök inicializálásánál, a következő **gondolatmenetet** kell figyelemmel kísérnünk:

-Az egydimenziós (**1D**) tömb **egyedi elemekből épül fel**.
-A **2D** tömb **1D tömbökből** épül fel.
-A **3D** tömb **2D tömbökből** jön létre.

.
.

És így tovább !!!



Ha a fenti okfejtést végiggondoltuk, könnyen megértjük, hogy miként kell egy többdimenziós tömbnek kezdeti értéket adni. Az ábrán egy 3D tömböt láthatunk, melynek mind a három irányban 4 "egység" a kiterjedése. Próbáljuk ezt a tömböt inicializálni. (Szépen olvashatóan !!!)

Forrás: http://www.doksi.hu

```
int   tomb[4][4][4]=
      {                         /* 3D keret */
       {                        /* 1. 2D keret */
        { 1,2,3,4 },            /* 1. 1D keret */
        { 5,6,7,8 },
        { 9,4,7,9 },
        { 7,0,2,6 }
       },                       /* 1. 2D keret */
       {                        /* 2. 2D keret */
        { 1,0,2,6 },
        { 2,3,8,6 },
        { 1,5,8,0 },
        { 3,6,9,1 }
       },                       /* 2. 2D keret */
       {
        { 3,5,1,5 },
        { 9,1,6,4 },
        { 3,5,8,6 },
        { 7,4,6,1 }
       },
       {
        { 1,0,6 },              /* Figyelj !! Kimaradt elem */
        { 2,3,8,6 },
        { 1,5,8,0 },
        { 3,6,9,1 }
       }
      };                        /* 3D keret */
```

Az **azonos szinten** lévő tömbelemeket egymástól **vesszővel választjuk** el, egy szint **felsorolását** mindig a **"{"** operátorral **kezdjük**, és a **"}"** operátorral **zárjuk**. (Nagyon munkaigényes megoldás, de megéri.)

Ezt a "tömbszörnyeteget" feltölthetjük másként is. A **legegyszerűbb módon**, két kapcsos zárójel között **felsoroljuk** az **elemeket**. Azonban ebben az esetben a **kimaradt elem helyére nullát kell írnunk**, mert az elemeket a fordító "ész nélkül" **egymás után be fogja rakni a memóriába**. (Honnan is tudhatná, hogy van kimaradt elem.)

```
int   tomb[4][4][4]={ 1,2,3,4,5,6,7,8,9,4,7,9,
      7,0,2,6,1,0,2,6,2,3,8,6,1,5,8,0,3,6,9,1,
      3,5,1,5,9,1,6,4,3,5,8,6,7,4,6,1,1,0,6,0,
      2,3,8,6,1,5,8,0,3,6,9,1,      };      ↑
                                       A hiányzó elem
```

Nagyon rosszul olvasható.

8.2. Karaktertömbök, stringek

A **karaktertömböket**, vagy más néven **stringeket** a C nyelvben **alaptípusként nem** értelmezték. Ez az oka, hogy a változó tömbök közül ezek bizonyos mértékig kitüntetett szerepet játszanak.

A **tömbökről** általában **tudjuk milyen méretűek**, ugyanezt **nem mondhatjuk** el minden esetben a **stringekről**. Azért, hogy tudjuk, hogy a **string vége hol található** a C megalkotói egy karaktert kiemeltek a karaktersorból és elnevezték **EOS**-nak (**E**nd **O**f **S**tring). Az EOS ASCII környezetben a nulla kódú karakter. Ha tehát egy karaktertömb olvasása közben egy nullát találunk, akkor vége a karaktersornak.

A másik érdekes eltérés az, hogy a stringek **kezdeti értékadása** is **eltér** a megszokottól. Elég nehézkes lenne karakter kódonként egy adott üzenetet bevinni egy

Forrás: http://www.doksi.hu

stringbe, még akkor is, ha felhasználjuk a C-nek a karakterkonverziós szolgáltatását. Ezért a tömböt nem a karakterértékekkel inicializáljuk, hanem a memóriában valahol, nekünk nem kell tudni, hogy hol, elhelyezzük a stringet, majd a tömböt tesszük egyenlővé a stringgel, vagyis a tömb mutatóját tegyük egyenlővé a karaktersorozat kezdőcímével. Ez igen bonyolultan hangzik, de megvalósítása jóval egyszerűbb. Hasonlítsuk össze a karakterenkénti és a mutató átadásával történő inicializálást.

```
char  string[]={'E','z',' ','','a','','k','a','r','a',
                'k','t','e','r','e','n','k','é','n','t','i',0 };
```

Ez nagyon nehézkes.

```
char string[]="Ez a mutatós megoldás";
```

Ez azért kényelmesebb. **Általában nem szükséges** az **EOS** karaktert a string végére tenni, mert automatikusan generálódik.

8.3. A karakter kódja, mutatója és karakterállandók

Ha egy **karakter kódjára van szükségünk** a karaktert két aposztróf **('')** közé tesszük és a fordító rögtön szolgáltatja a kódot.

```
char  c;     /* A karakter deklarációja. */
    .
    .
c='u';       /* Az u betü kódja a c változóba. */
```

Ha **nem** a **kódra**, hanem a **karakter helyére** vagyunk kíváncsiak az **idézőjeleket** kell használnunk, ekkor a mutató kerül a balértékbe.

```
char *cp;    /* Karakter mutató deklarálása. */
    .
    .
cp="u";      /* Van egy u betünk, amelynek a címét adjuk át. */
```

Ha egy **függvény** a **paramétersorában** egy **string kezdőcímét igényli**, **nem kell külön létrehozni** a **stringet**, ha az nem változik, hanem **azonnal beírhatjuk** magába az **argumentumba**. Erre a legjobb példa a printf függvény.

```
printf("Na látod ?");
```

A C bizonyos nem látható karakterek könnyű kezelésére úgynevezett karakter konstansokat vezettek be, ezeknek sajátossága, hogy mind a **"\"** nyitó karakterrel kezdődnek és egy betű, vagy egy számjegy követi a nyitó karaktert. A karakter konstansok a következők:

\a	Csengő karakter
\b	Visszalépés karakter
\f	Lapdobás karakter
\n	Újsor karakter
\r	Kocsi vissza karakter
\t	Tabulátor karakter
\v	Függőleges tabulátor karakter
****	Back-slash karakter
\'	Aposztróf karakter
\"	Idézőjel karakter
\?	Kérdőjel karakter
\ooo	Karakter, melynek kódja oktálisan ooo
\xhh	Karakter, melynek kódja hexadecimálisan hh

Forrás: http://www.doksi.hu

8.4. Címaritmetika

A tömbök és a **címaritmetika szorosan összefüggő** részei a C-nek. Általában **címszámítási** műveleteket **tömbökön** szoktunk végrehajtani, hogy **indirekt módon** tudjuk a **tömbelemekhez hozzáférni**. A pointerekre az alábbi műveletek alkalmazhatók:

-egész hozzáadása és kivonása a pointerhez,

-a pointer inkrementálása és dekrementálása,

-két pointer kivonása egymásból.

A különböző **változók különböző tárterületet foglalnak** el, például egy char egy byte-ot, egy int két byte-ot, egy float négy byte-ot. Ha a hozzájuk tartozó pointerhez egészt adunk, vagy vonunk ki, illetőleg inkrementáljuk, vagy dekrementáljuk, a pointer értékét ez **nemcsak egyszerűen módosítja**, hanem olymódon változtatja meg, hogy az így **keletkezett mutató annyiadik tömbelemre mutat**, **amennyi** a **módosítás értéke volt**. Nézzünk erre egy példát:

	char		int		float
F000h		F000h		F000h	
F001h					
F002h		F002h			
F003h					
F004h		F004h		F004h	
F005h					
F006h		F006h			
F007h					

Ha tehát a változónk char, az első inkrementálásra a F000H címből F001h cím lesz, ha int, akkor F002H, ha float, akkor F004H.

A címaritmetika nemcsak az alap változókra, hanem a származtatott és összetett változókra is igaz.

8.5. Struktúrák

A **struktúra** egy **összetett adatszerkezet**, ami arra szolgál, hogy **különböző típusú**, de valamilyen oknál fogva **összetartozó adatokat egy egységként** tudjunk kezelni. **A** struktúrán **belül tetszõleges típusokat** alkalmazhatunk, **még struktúrákat** is, illetőleg bizonyos megszorításokkal egy **struktúra lehet rekurzív** is.

8.5.1. Struktúrák definíciója, deklarációja, elem hozzáféres

A struktúrákat, miután mi készítjük, természetesen definiálni kell. A definíció formája az alábbi:

```
        A nyelvi kulcsszó
        |           A hivatkozási név
        |           |
struct struktúra_név
     {                          ┐
     típus változó_neve_1;      │
     típus változó_neve_2;      │ A    struktúra    elemeinek
     típus változó_neve_3;      │ deklarációi.
               .                │
     típus változó_neve_N;      │
     };                         ┘
```

Forrás: http://www.doksi.hu

Nézzünk egy konkrét példát a struktúra definícióra:

A példában egy képernyőre írandó színes karakter adatait helyezzük el egy struktúrában. A szükséges információk a karakter kódja, színe, x és y koordinátája.

```
struct color_char
      {
      char  code;       /* A kód */
      int   color;      /* Szin */
      int   xpos;       /* X koord. */
      int   ypos;       /* Y koord. */
      };
```

A **deklaráció**, amely a definíciót követi a következő-képpen történik:

```
struct struktúra_név hivatkozási_nevek;
   |             |            |
   |             |           A változóként használt
   |             |           struktúrák megnevezése.
   |
   |            A struktúra  típusának
   |            megnevezése
  A struct kulcsszó
```

Az előző példa alapján egy felhasználási példa:

Öt karakter értékeit akarjuk tárolni, először különálló egységenként, majd tömbként.

```
struct color_char chr_a,chr_b,chr_c,chr_d,chr_e;
struct color_char chr[5];
```

A **deklaráció** és a **definíció egy lépésben** is **végbemehet** nézzük az előző példák alapján hogyan:

```
struct color_char
      {
      char  code;       /* A kód */
      int   color;      /* Szin */
      int   xpos;       /* X koord. */
      int   ypos;       /* Y koord. */
      }chr_a,chr_b,chr_c,chr_d,chr_e,chr[5];
```

A megfelelő formátumokat már definiálni tudjuk, le tudjuk foglalni a területeket a srtuktúrák számára, a következő kérdés, hogyan férhetünk a struktúra egyes elemeihez. Az elemhozzáférés az alábbi módon történik:

```
hivatkozási_név.elem_neve
            |   |    |
            |   |    Melyik elem a struktúrából
            |   Pont operátor
           Melyik struktúrából
```

Az előző példánknál maradva, ha a chr_b struktúrából a color elemre van szükségünk a következő módon érhetjük el:

```
chr_b.color
```

8.5.2. Struktúrapointerek

A **struktúrákhoz** szintén lehet rendelni **mutatókat**, **nemcsak** egy **teljes struktúrához**, hanem az **elemeihez** is. Ekkor használhatjuk az oprátoroknál megemlített **"->"**

Forrás: http://www.doksi.hu

operátort. Ennek bemutatásához használjuk fel az előzőekben már definiált color_char struktúrát.

```
struct color_char chr;      /* deklaráció */
struct color_char *cpnt;    /* mutatót deklarálunk */
          .
          .
cpnt=&chr;                  /* értékadás */
          .
cpnt->code='A';             /* indirekt hivatkozás */
          .
```

A fenti példán egy **struktúra elemre** egy **mutató** és a **"->" operátor** segítségével **indirekt módon hivatkoztunk**. Miért is jó ez, ezt egyből elég nehéz belátni. Ezért egy kicsit eltérve a témától, nézzünk bele a függvények lelkivilágába.

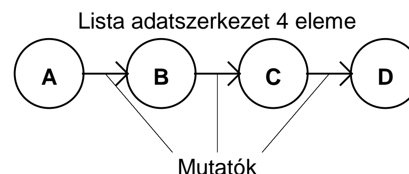
A függvény, a paramétermezőt a számítógép stackjén keresztül (lásd 12.6.2.1 fejezet) kapja meg, ha egy teljes struktúrát akarunk átadni a függvénynek, a stacket jelentősen terheljük, nem beszélve arról az esetről, ha esetleg struktúra tömböt kellene paraméterként átadni. Ezért célszerű a struktúrára a kezdőcímével hivatkozni, mert ez a legrosszabb esetben is csak 4 byte helyet foglal el. Tömb esetén kényelmesen használhatjuk a pointer aritmetika szolgáltatásait. A pointer aritmetika viszont a tömbhatárokkal nem foglalkozik, ekkor legfeljebb azt tehetjük meg, hogy átadjuk a tömb maximális méretét is (no mondd már). Ezek a gondolatok, amelyeket leírtunk nem csak a struktúrákra vonatkoznak, hanem az összes összetett adatszerkezetre is.

8.5.3. Rekurzív struktúrák

A struktúrában előfordulhat gyakorlatilag minden alap, származtatott és összetett típus, csak a saját deklarációja nem, ha ez megengedhető lenne a fordító egy ilyen esetben egy végtelen adatterület foglalásba "csavarodhatna" bele. Azonban van lehetőség ennek aproblémának a kikerülésére úgy, hogy nem az alap struktúrát definiáljuk saját magába, hanem egy olyan mutatót, amely egy ilyen jellegű struktúrára mutat.

Nézzünk egy példát erre az esetre:

Az úgynevezett lista adatszerkezet egy elemét hozzuk létre, mint struktúrát. Az elem felépítése olyan, hogy tartalmazza az adatot és azt a mutatót, amely a következő elemre mutat. Ezt az adatszerkezetet nevezik lista adatszerkezetnek (lásd az ábrát).




```
struct list_item
      {
      int item;                     /* a lista eleme */
      struct list_item *chain;      /* láncolási cím */
      };
```

8.5.4. Speciális field struktúrák

A C programozási nyelv lehetőséget ad olyan struktúrák létrehozására, amelyekben nem a típusokból áll a struktúra, hanem egy adott szélességű, általában 16 bites, terület bitjeiből. Szintaktikailag a field definíciója majdnem azonos a közönséges struktúra definíciójával.

Forrás: http://www.doksi.hu

```
    A struct kulcsszó
     |       A field struktura típusa
     |       |
  struct field_neve A mező azonosítók
       {         |
       unsigned  mezőnév1:szélesség;
       unsigned  mezőnév2:szélesség;
                 .
                 .
       unsigned  mezőnévN:szélesség;
       };                 |    |
                         |    A mező szélesség bitben.
                        Kettőspont
```

Azt, hogy egy mező a kérdéses 16 biten hol helyezkedik el **célszerű kipróbálni**, mert ez erősen implementáció függő.

Hogy mire jó ez a field szerkezet, erre mutatunk be példát, amely a képernyő egy karakterének kód és attribútum byte-ját írja le.

```
struct scr_chr
     {
     unsigned    c_code:8;   /* ASCII */
                 fg_clr:4;   /* előtér színe */
                 bg_clr:3;   /* háttér színe */
                 c_blnk:1;   /* villogtatás */
     };
```

A példánkban a bitmintát egy **unsigned int** típusba raktuk bele. Beletehettük volna **előjeles egészbe** is.

A következő kérdés az, hogy mi történik akkor, ha nagyobb területre van szükségünk, mint 16 bit. Ez sem okoz semmi problémát, mert a C megengedi, hogy ha kifutottunk a méretből az eddigi 16 bitünkhöz még tetszőleges számú 16 bites szót csatoljunk. Ezt is egy példa segítségével mutatjuk be.

```
struct bit_pattern           /* a neve */
     {
     unsigned    patt1:1;
                 patt2:7;
                 patt3:4;
                 patt4:0;    /* Az első szó kilőve. */
     unsigned    patt5:9;    /* A 2. szó */
                 patt6:7;
     };
```

A "0" értékű minta lezárja az aktuális szót, az utána következő szóba kerül a fennmaradó rész.

8.5.5. Struktúrák, mint típusok

A típusoknál ismertetett **typedef** kulcsszóval a struktúrákból is definiálhatunk típusokat. Ilyen jellegű nagyon fontos típus az I/O kezelésnél használt FILE. Nézzük hogyan történik egy ilyen típusdefiniálás egy ismert példa segítségével:

Forrás: http://www.doksi.hu

```
typedef struct
      {
      char  code;
      int   color;
      int   xpos;
      int   ypos;
      } color_char;
```

A **deklaráció** pedig így történik:

```
color_char a_chr,b_chr;
```

8.6. Unionok

A unionok igen sajátságos összetett adatszerkezetek. A **szintaktikai leírásuk gyakorlatilag megegyezik** a **struktúrák leírásával**, azonban a **union elemei ugyanazt** a **memóriaterületet foglalják le**, tehát **egyszerre csak egy elem lehet** a **unionban**, ha egy elem van már benne és új elemet akarunk bele írni, a régi felül fog íródni.

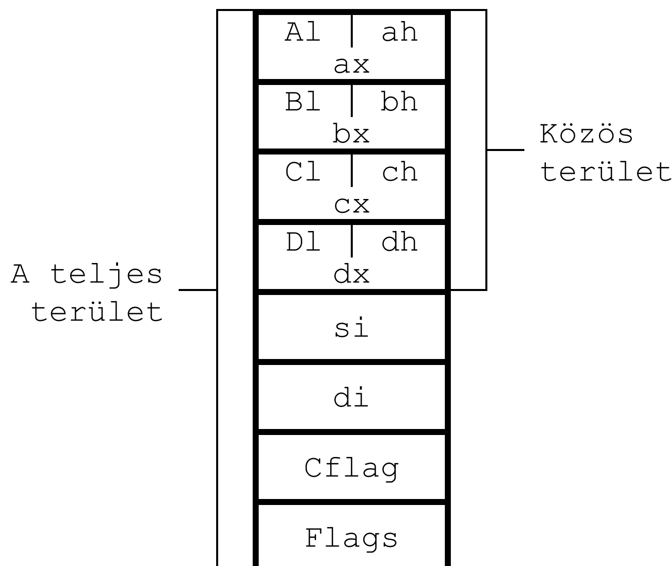
Vegyük példának a **dos.h** header file-ban található **REGS uniont**, és a hozzá tartozó struktúrákat:

```
struct WORDREGS { /* A 16 bites regiszterek */
      unsigned int ax,bx,cx,dx,si,di,cflag,flags;
                  };

struct BYTERGS {  /* 8 bites regiszterek */
      unsigned char al,ah,bl,bh,cl,ch,dl,dh;
                  };

union REGS {
      struct      WORDREGS x; /* 16 bitesek */
      struct      BYTEREGS h; /* 8 bitesek */
            };
```

A fenti példán láthatunk egy uniont, amelyet két struktúrából állítottunk össze. Nézzük meg, hogy a struktúra elemek hogyan foglalnak helyet ebben a unionban.



A WORDREGS struktúra a nagyobb, ezért ő határozza meg a union méretét, ez általánosságban is így van, a **legnagyobb elem az**, **amely a union méretet meghatározza**.

A fenti példa a uniont definiálta, a **deklaráció** ugyan-olyan egyszerű, mint a struktúrák esetén. Használjuk az előző definíciót:

Forrás: http://www.doksi.hu

```
union REGS inregs;
  |     |      |
  |     |      |
  |     |     A union neve
  |    A union típusa
 A union kulcsszó
```

A union eleméhez a struktúráknál megszokott módon jutathatunk hozzá. Nézzük ezt előző példáink segítségével. Először adjunk értéket az ax 16 bites regiszternek, majd olvassuk ki az ah értéket:

```
int   r16;          /* 16 bites regiszter értéke lesz */
char  r8;           /* 8 bites regiszter értéke lesz */
      .
r16=32767;
inregs.x.ax=r16;
      .
      .
r8=inregs.h.ah;   /* ah-ba 127 került */
   |    ||| |
   |    ||| A struktúra elem azonosítója
   |    ||A struktúra pontja
   |    |Az union elem azonosítója
   |   A union pontja
  A union azonosítója
```

8.6.1. Unionok, mint típusok

A uniont is, hasonlóan a struktúrához, lehet típusként definiálni. Vegyük a következő példát. Van egy float típusú számunk, amelynek az egyes byte-jaira vagyunk kíváncsiak.

```
   A typedef kulcsszó
   |        A union kulcsszó
   |        |
typedef     union
      {
      float f;          /* Ez a float */
      char  b[4];       /* Ezek a byte-ok */
      } fbyte;
        |
        A definiált típus neve
      .
      .
  A típus neve
  |    A változó neve
  |    |
fbyte unit;             /* A változó deklarálása */
     .
     .
unit.f=39.27;           /* Értékadás */
     .
```

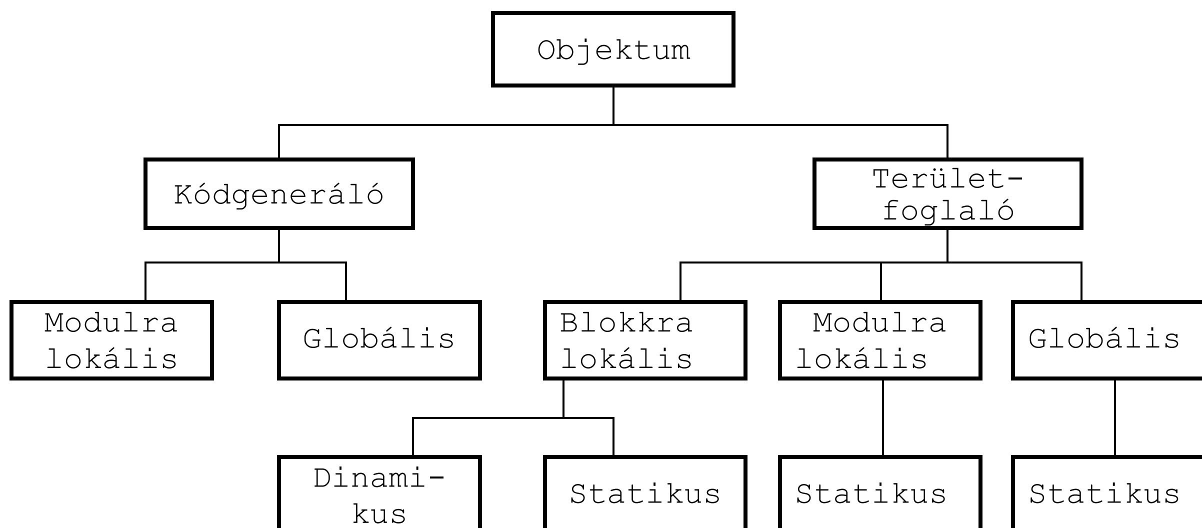
9. Objektumok

Az alapfogalmaknál már ismertettük az objektum fogalmát, ebben a fejezetben az objektumokkal és a velük szorosan összefüggő fogalmakkal ismerkedünk meg, ezzel le is zárjuk a C nyelv felépítésének ismertetését.

9.1. Az objektumok osztályozása

Forrás: http://www.doksi.hu

Az objektumokat az alábbi módon osztályozhatjuk:



Ezek után sorban tisztázzuk az ismeretlen fogalmakat:

-**Kódgeneráló** az az objektum, amely **hívása** után, vala-milyen **értékkel tér vissza**. Ez a C esetében a **függvényeket** jelenti, **ide soroljuk** a **void** típusú **függvényeket** is.

-**Területfoglaló** az az objektum, amely **változók** és **konstansok tárolására szolgál**.

-**Modulnak** nevezzük a programnak azon részét, amely **egy forráslistában található**. (Említettük, hogy a program több külön fordítható modulból állhat, amelyekből majd a linker készíti el a végleges futtatható kódot.)

-**Blokknak** nevezzük a **modulon belül található kisebb egységeket**, amely a C esetében a **függvényeket jelenti**.

-**Globálisnak** nevezzük azokat az objektumokat, amelyek **bármely modulból és blokkból eléretők**.

-**Modulra lokális** az az objektum, amely az **adott modul-ból**, de csak az adott modulból, **bárhonnan elérhető**.

-**Blokkra lokális** az az objektum, amelyet csak az **adott blokkból**, praktikusan csak az **adott függvényből**, **lehet elér**ni.

-**Statikus** egy **objektum**, ha számára **állandó tárolóhely van kijelölve**, már a fordítás folyamán, és ez a hely **nem szűnik** meg a program **futása során**.

-**Dinamikus** egy objektum, ha **élettartama** csak **addig tart**, míg az **adott blokkban tartózkodik** a **program**, a **blokk elhagyása után** az objektum **megszűnik** és **akkor jön létre újra**, ha a **vezérlés** a **újra** a **blokkra** kerül.

9.2. A láthatóság kérdése és a deklarációt meghatározó kulcsszavak

A láthatóság kérdését az objektumokkal kapcsolatban már megemlítettük, most nézzük meg hogyan néz ez ki a gyakorlat-ban.

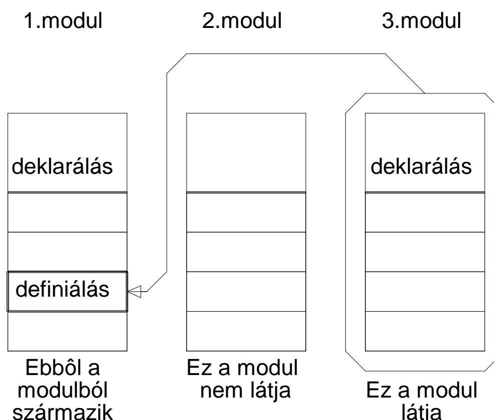
A láthatóság szabályai azt mondják ki, hogy egy blokkból csak a blokkban definiált objektumok, illetve a hierarchiában felette álló blokkok objektumai láthatók. Egyes szakiraodalmak ezt a fogalmat hatáskörnek is nevezik.

Forrás: http://www.doksi.hu

Az **alap C**-ben a **blokkok**, amelyek **tulajdonképpen** a **függvények, teljesen egyenrangúak - a main is -**, ezért az **egyik függvényben deklarált változók** a másikból nem láthatók.

Az egyenrangúság másik következménye, hogy bármelyik függvény, bármelyik függvényt hívhatja, még önmagát is.

Globális függvényt úgy deklarálunk, hogy a **függvényt deklarációját minden blokkon kívül helyezzük el**, így a függvényünk minden modulból látszani fog, amely modulokban a függvényre hivatkozunk, **ezekben** a **modulokban** a függvényt **mindig deklarálni kell**, természetesen **abban** is, ahol **definiáltuk**.



Ha egy modulban nem kívánjuk a globális függvényt használni, akkor ott nem kell deklarálni, mert nincs értelme.

Globális változót hasonlóan deklarálunk, mint függvényt, vagyis **minden blokkon kívül** helyezzük a deklarációt. Azonban van egy **fontos különbség**, hogy a változó **elé** az **alapmodulon kívül ki kell tenni az extern kulcsszót**. Kell választanunk egy **alapmodult**, ami a változó eredeti helye, ebben a modulban nem adható ki az extern kulcsszó.

(Az extern kulcsszót ki lehet adni függvények esetén, de csak olyan modulban, ahol a függvény nincs definiálva.)

Modulra lokális függvény és **változó** esetén a deklarációnak **minden blokkon kívül kell lennie**, (függvények esetén egyébként máshol nem is lehetne, egyenrangúság miatt), de ekkor a **deklarációs sor elé** a **static** kulcsszót ki kell **tenni**. A **static hatására** kizárólag az **adott modulból** lesz **látható** a kérdéses **objektum**.

Blokkra lokális objektum csak **változó lehet**, a **deklaráció** a **blokk belsejében történik**. Ezt a **változót csak** a **blokk**, vagyis C esetben csak a függvény, **belsejéből lehet látni**, emlékezzük a maximum kereső függvény példájára.

```
int max3(int a,int b, int c)
      {
      int   i,j;        /* Blokkra lokálisok */
      i=(a>b?a:b);
      j=(i>c?i:c);
      return(j);
      }
```

Blokkra lokális dinamikus objektumok deklarációja nagyon egyszerű, gyakorlatilag, ha a **blokkon belül deklarálunk** egy változót, a **változó dinamikus lesz**. **Két kulcsszó létezik** a dinamikus változók deklarációjának módosítására. Az egyik a **register**, amely utasítja a fordítót, ha lehetséges, a **változót processzor** valamelyik **regiszterében helyezze** el. A másik kulcsszó az **auto**, amely a **fordító számára** jelzi, hogy a változót **sokszor kívánjuk használni**, ezért a tárolás elérési szempontból **optimális legyen**.

Blokkra lokális statikus változó deklarálása szintén a **blokkon belül történik** csak a **static** kulcsszót kell a **deklarációs sor elejére** tenni. Ez azonban lehetetlenné teszi, hogy rekurzív függvényhívást alkalmazzunk, mert ez a belső változókat az újrahívás felülírja.

Forrás: http://www.doksi.hu

A dinamikusan deklarált változók esetén a változó számára lefoglalt terület a stacken helyezkedik el, a visszatérési cím felett. Ha egy rekurzív függvényhívás történik, a lokális változó terület újra felépül a stack-en, és ez az új terület nem fedi át a régit.

Az alábbi táblázat összefoglalja a változók élettartamát és láthatóságát:

Élettartam	Láthatóság	Deklarálás helye	Kulcsszó
Statikus	Globális	Minden modulban	extern
Statikus	Modulra lokális	Adott modulban	static
Statikus	Blokkra lokális	Blokkban	static
Dinamikus	Blokkra lokális	Blokkban	auto
Dinamikus	Blokkra lokális	Blokkban	register

Mi történik akkor, ha azonos néven neveztünk el egy blokkra lokális, illetve egy modulra lokális és egy globális változót?

Egy lehetőség rögtön kiesik, modulra lokális és globális változót egyazon néven egy modulban nem lehet deklarálni, hiszen egymás mellet (alatt) lenne két azonos nevű változó. Azonban, ha van a teljes programban azonos nevű globális és modulra lokális változó, akkor az adott modulban a modulra lokális fog látszani.

Hasonló a helyzet a blokkra lokális változók esetén, akár globális, akár modulra lokális változó létezik a blokkra lokális változóval azonos néven, az adott blokkon belül csak a blokkra lokális fog látszani.

Nézzünk egy példát, amelyben lesz azonos nevű modulra lokális és globális függvény, lesz azonos nevű blokkra lokális és globális változó:

1.modul

```
#include    <stdio.h>

/* globális változók */

char  glbtxt[25]="\tEz a globális szöveg\n";
void  fgv2(void);                    /* globális függvények */
void  fgv3(void);

main()                  /* Csak egyetlen main lehet  !!!! */
      {                 /* egy programban */
      fgv2();
      fgv3();
      }
```

Forrás: http://www.doksi.hu

2.modul

```
#include    <stdio.h>

extern      char  glbtxt[25];

void  fgv2(void); /* globális függvények */
static void fgv3(void); /* modulra lokális */

void  fgv2(void)
      {
      printf("2.modulban járunk\n");
      printf("%s\n",&glbtxt[0]);
      fgv3();
      }

void  fgv3(void)
      {
      printf("2.modulban modulra ");
      printf("lokális vagyok, fgv3\n\n");
      }
```

3.modul

```
#include    <stdio.h>

extern      char  glbtxt[25];

void  fgv3(void); /* globális függvények */
void  fgv4(void); /* globalis de csak itt hívjak */

void  fgv3(void)
      {
      printf("3.modulban járunk\n");
      printf("%s\n",&glbtxt[0]);
      fgv4();
      }

void  fgv4(void)
      {
      char  glbtxt[]="úgy se láttok\n";
      printf("\n3.modulban fgv4-ben \n");
      printf("blokkra lokális szöveg vagyok\n");
      printf("%s",&glbtxt[0]);
      }
```

Futás után képernyőn

```
2.modulban járunk
     Ez a globális szöveg

2.modulban modulra lokális vagyok, fgv3

3.modulban járunk
     Ez a globális szöveg

3.modulban fgv4-ben
blokkra lokális szöveg vagyok
úgy se láttok
```

Forrás: http://www.doksi.hu

10. A legfontosabb C függvények

Ebben a fejezetben a legfontosabb alap C fogalmakat és az ehhez tartozó függvényeket fogjuk ismertetni. Kihangsúlyozzuk, hogy ezek a módszerek minden C implementációban igazak, ha tehát ezeket használjuk programunk hordozható lesz. Természetesen nem törekedhetünk teljességre, csak a legfontosabb elveket és a legfontosabb függvényeket ismertetjük. Előfordulhat, hogy egyes függvényeknek lesznek olyan opciói, amelyek a Borland C++-hoz tartoznak, ezeket külön ki fogjuk hangsúlyozni.

10.1. Be- és kivitel

Ebben az esetben a **be-** és **kivitelt** az u.n. **standard input** és a **standard output**-ra vonatkoztatjuk. A standard input (ezentúl **stdin**) a billentyűzet, a standard output (ezentúl **stdout**) a képernyő. Megjegyezzük, hogy **ez a beállítás változhat** ha ezeket a perifériákat **átirányítjuk**.

10.1.1. A karakteres be- és kivitel

Az egyenkénti karakteres bevitelre a **getchar()** függvény (a Borland C++-ban makro) szolgál, amely hívása után karakterenként kiolvassa az **stdin** "tartalmát". Ha meghívjuk a **getchar()-t** és a standard input puffere üres, addig fog várni, míg legalább egy karakter nem érkezik.

A függvény deklarációja az stdio.h-ban található és a következő:

```
#include <stdio.h>

int  getchar(void);
 |            |
 |         Nincs paramétere
 |
Int-et ad ugyan vissza,
de csak karakterként
használjuk
```

A getchar() párja a **putcar()**, amely az **stdout-ra** küld **karaktert**. A putchar() deklarációja szintén az stdio.h-ban található.

```
#include <stdio.h>

int  putchar(int c);
 |            |
 |         A kiírandó karakter
 |         int formában, de
 |         csak karakterként
 |         használjuk.
Ha sikeres a kiíratás a
karakter kódja, egyébként EOF.
```

Nézzünk egy rövidke példaprogramot a getchar() és a putchar() használatára:

Forrás: http://www.doksi.hu

```
#include    <stdio.h>
char  c;                          /* munka karakter */
while(1)                          /* végtelen ciklus */
      {
      c=getchar();                /* c beolvasása */
      putchar(c);                 /* kiírása */
      if(c=='\n') break;          /* itt a vége */
      }
```

10.1.2. A formázott kiíratás

A formázott, vagy más néven formátumozott kiíratás függvénye a **printf(),** amelyről nem túlzás, ha azt állítjuk, hogy a C nyelv egyik, talán a legfontosabb függvénye. A printf() szintén a **stdout-ot** használja és mint ilyen a deklarációja is természetesen az stdio.h header file-ban található. (Az stdio annyit tesz, hogy standard input-output.) Nézzük a deklarációt:

```
#include  <stdio.h>

int printf(cons char *format,...);
  |                 |            |
  |                 |            |
  |                 |            |
```

A sikeresen kiírt byte-ok száma, hiba esetén EOF.	A formátum string.	A függvény argumentumai.

A printf a formátum stringnek megfelelően konvertálja az argumentumait és kiírja az stdout-ra. A formátum stringben karakterek és **formátum specifikátorok** fordulhatnak elő. A karakterek azonnal kiíródnak, a formátum specifikátorok helyére az általuk kijelölt argumentum elemek kerülnek ki az stdout-ra.

Hogyan épül fel egy formátum specifikátor:

%[*flags*][*width*][*.prec*][*FNhlL*]*type*

-A **%** jel jelzi a függvény számára, hogy formátum specifikáció következik.

-A **flags** használata nem kötelező, ezért is tettük szögletes zárójelbe. A flags alapvetően a kiíratás formáját befolyásolja. A flags részben a következő karakterek szerepelhetnek:

-	Az eredmény balra igazítását írja elő.
+	Előjeles számok esetén kiírja a + előjelet is.
szóköz	Előjeles számok esetén a + helyett szóközt tesz.
#	Speciális kiíratást ír elő. Ami hexadecimális és oktális esetben annyit jelent, hogy kiírja a vezető 0x-et, illetve a 0-t, lebegőpontos konverzióknál mindenképpen kiírja tizedes-pontot.

-A **width** rész a minimális mezőszélességet jelöli ki. A width egy decimális szám, amely ha 0-val kezdődik a mező fennmaradó részét 0-val, egyébként szóközökkel tölti ki. Ha a width helyén ***** található, a függvény a mező szélességet argumentumként olvassa be. A width használata nem kötelező.

-A **.prec** egy decimális számjegy, amely a kijelzés pontosságát határozza meg. Egész típusok esetén a minimálisan kijelzendő számjegyek számát, lebegőpontos esetben a tizedespont után kiírt számjegyek számát adja meg. Ha a szám rövidebb, mint amit a .prec

Forrás: http://www.doksi.hu

előír, akkor nullákkal egészíti ki. Ha a **"."** után ***-ot** írunk, a prec értékét az argumentumban adhatjuk meg. A prec használata nem kötelező.

-Az **FNhlL** jelzők segítségével lehet megadni azt, ha a formátum specifikátort nem alapértelmezésű konverzióra adtuk ki. A jelzők jelentése a következő:

F Az argumentumban far pointer van. IBM implementáció!

N Az argumentumban near pointer van. IBM implementáció!

h Az argumentum short int.

l Az argumentum long int, vagy double.

L Az argumentum long double.

Az FNhlL nem kötelező.

A **type** adja meg a tulajdonképpeni konverziós műveletet. A type formátumkarakterek:

Formátum-karakter	Típus	Kimeneti formátum
d,i	Egész	előjeles decimális
o	Egész	előjeltelen oktális
u	Egész	előjeltelen decimális
x,X	Egész	előjeltelen hexadecimális
f	Lebegőpontos	előjeles fixpontos
e,E	Lebegőpontos	előjeles normál alak
g,G	lebegőpontos	legrövidebb ábrázolás
c	karakter	egyetlen karakter
s	karakter mutató	Karakter sorozat a \0-ig.
p	pointer	pointer értéke

Nézzünk néhány példát a printf használatára:

1.A formatum stringben szöveg van:

```
printf("Szöveg\n");
```

Futás után:

```
Szöveg
! cursor
```

2.Külső string kiíratása:

```
char str[]="Ez egy string\n";

printf("%s",str);
```

Futás után:

```
Ez egy string
!
```

Forrás: http://www.doksi.hu

3.Formátum string és külső egész típusú változók keverése:

```
int  a=12,b=23,c=6; /* egész változók */

printf("a=%d\t,b=%d\t,c=%d\n",a,b,c);
```

Futás után:

```
a=12 b=23 c=6
!
```

4.Hosszú egész kiíratása:

```
long he=69000;

printf("%ld\n",he);
```

Futás után:

```
69000
!
```

10.1.3. Formátumozott beolvasás

A formátumozott beolvasás függvénye a **scanf().**Ez a függvény karaktereket olvas az **stdin-ről** és ezeket, a printf()-hez hasonlóan, formátum specifikátorok alapján értelmezi. A **scanf()** éppúgy, mint a printf() nemcsak egyetlen adatot tud beolvasni, hanem a formátum specifikátoroknak megfelelően többet. A scanf() deklarációja a következő:

```
#include  <stdio.h>

int scanf(const char *format,...);
```

int	const char *format	...
A visszatérési érték, amely a sikeresen beolvasott elemek számát adja meg, hiba esetén EOF.	A formátum string.	Az argumentumok, amelyek most mutatók.

A formátum string a következő elemeket tartalmazhatja:

-**Szóközjellegű karakter** (szóköz, újsor, tabulátor), melynek hatására a scanf() beolvassa, de nem tárolja szóköz karaktereket az első ezektől különböző karakterekig.

-**Egyéb karakter**, amelynek illeszkednie kell a bemenetre érkező karakterhez.

-**Formátum specifikátor**, amely meghatározza a beolvasandó adat milyen típusban kerüljön eltárolásra.

A scanf() mindaddig olvassa a bemenetét, amíg a beolvasott karakter nem felel meg a formátum stringben lévő nem szóközjellegű karakternek, vagy file vége jelzés jött a bemenetre (a file kezelésnél erről még beszélünk), vagy teljesen feldolgozta a formátum stringet.

A formátum specifikátor a következő elemekből áll:

%[*][width][FNhlL]type

Forrás: http://www.doksi.hu

A **%** jel, a **width**, és az **FNhlL** jelentése megegyezik a printf()-nél leírtakkal. A **type** is típust jelöl, de van egy apró értelmezésbeli eltérés, erről később részletesen be-szélünk. Ha a **%** jelet * követi, a scanf() végigolvassa az adott mezőt, de nem tárolja el.

A scanf() az argumentumában **minden esetben mutatót kér**, eltérően a printf()-től, erre érdemes odafigyelni, mert sok kellemetlenségtől óvhatjuk meg magunkat.

A type formátumkarakterek a scanf() esetén:

Formátum karakter	Típus	Argumnetum
i, I, d, D	decimális egész	int *
o, O	oktális egész	int *
u, U	előjeltelen decimális egész	int *
x, X	hexadecimális egész	int *
e, E, f, g, G	lebegőpontos érték	float *
s	string	char *
c	karakter	char *
n	Az idáig beolvasott karakterek száma	int *
p	Pointer Szegmens:offset	void **

A táblázat utolsó elemében szereplő **void **** azt jelenti, hogy egy mutatót kell az argumentumba megadni, amely egy olyan mutatóra mutat, amely egyenlőre nem rendelkezik típussal. Azt, hogy ezt a Borland C++ ilyen formátumban kéri az IBM PC-s környezet következménye.

10.2. File kezelés

A C nyelv egyik nagy erőssége a file-kezelés. Ebben az esetben azonban nem csak a lemezegységen található állományokról beszélhetünk, hanem egyéb ki- és bemeneti eszközökről is. Vehetjük példának rögtön az elõzőekben említett stdin-t, vagy stdout-ot, bármilyen meglepő a rendszer ezt is file-ként kezeli.

A file-kezelésnek két szintje van: az alacsonyszintű és a magaszintű file-kezelés.

A file-kezeléssel kapcsolatban néhány egyszerű fogalmat előre tisztáznunk kell. Ezek a következők:

-**Megnyitási mód**, amely azt jelenti, hogy a file-t milyen módon kívánjuk felhasználni. E szerint egy file lehet:

-csak olvasható,
-csak írható,
-olvasható és írható,
-bővíthető (append),
-ezek kombinációi.

-**File-pozíció**, amely megmutatja, hogy a file hányadik byte-jánál tart az éppen aktuális file művelet. A file-t úgy képzeljük el, mintha az őt alkotó byte-ok sorban egymás után helyezkednének el, így ezek beszámozhatóak. Az első byte száma 0 lesz. Miután byte-ban "mérjük" a file-pozíciót mozoghatunk a file-n belül abszolút és relatív módon is. Az abszolút

Forrás: http://www.doksi.hu

file-pozíciónálás a file elejétől történik, a relatív pozicionálás az éppen aktuális helyzettől előre (negatív irányban), vagy hátra (pozitív irányban).

-Text és bináris file-kezelés, amely azt jelenti, hogy az általunk használt állományt szövegfileként kezeljük, vagy adatállományként. Ha szöveges üzemmódban dolgozunk a file-kezelés rutinjai minden egyes kocsi vissza karakter után egy soremelés karaktert tesznek ki az állományba, ha bináris módon dolgozunk, amit kiírunk, az lesz a file-ban.

10.2.1. Az alacsonyszintű file-kezelés

Az alacsonyszintű file-kezelés arra épül, hogy a UNIX operációs rendszer, a file-okat egydimenziós byte-tömbökként kezeli, amelyek általában valamely lemezegységen találhatók. Ahhoz, hogy a file-ban tárolt adatokhoz hozzá tudjunk jutni a file-t meg kell nyitni. A megnyitás során a file kap egy számot, amellyel később hivatkozni tudunk rá. Ez a szám tulajdonképpen egy file-leíró struktúratömb index értéke, amelyet az angol nyelvű irodalom **handle** néven említ. Az általa azonosított struktúra tartalmazza az adott file-unk adatait. Azt, hogy hány file-t tudunk megnyitni, a struktúratömb mérete határozza meg.

Az alacsonyszintű file-kezelés legfontosabb függvényei:

open,close,creat,read,write,lseek,tell

Az **open** függvény a file-ok megnyitására szolgál.

```
#include  <io.h>
int open(const char *name,int mode,unsigned attrib);
```

A handle értéke hiba esetén -1

A file elérési útja és neve

A file megnyitási módja

A file attributuma

A file megnyitási módja a következő lehet:

- **O_RDONLY** csak olvasásra,
- **O_WRONLY** csak írásra,
- **O_RDWR** írásra, olvasásra,
- **O_APPEND** hozzáfűzésre,
- **O_CREAT** létre kell hozni,
- **O_EXCL** ha volt és létrehozzuk hibajelzést ad,
- **O_TRUNC** ha volt és létrehozzuk a régi elvész,
- **O_BINARY** bináris nyitás,
- **O_TEXT** szöveges nyitás.

Ezek a szimbólumok az **fcntl.h** header file-ban találhatók.

A file attributuma a következő lehet:

- **S_IREAD** olvasható,
- **S_IWRITE** írható,
- **S_IREAD|S_IWRITE** írható és olvasható.

Ezek a szimbólumok a **sys\stat.h** header file-ban találhatók.

A **close** függvény megnyitott file-t zár le.

```
#include  <io.h>
int close(int handle);
```

A file handle.

Forrás: http://www.doksi.hu

Hiba esetén -1.

A **creat** függvény file-ok létrehozását teszi lehetővé, estleg **felülír** egy meglévő file-t.

```
#include <io.h>
int creat(const char *path,unsigned attrib);
```

A paraméterek jelentését lásd az open függvénynél.

A **read** lehetővé teszi, hogy a file-ból adatokat olvassunk. Az olvasást az aktuális file-pozíciótól kezdi és az adatokat szekvenciálisan egy bufferba tölti. Az olvasás során a file-pozíció a beolvasott byte-ok számával növekszik.

```
#include <io.h>
int read(int handle,void *buff,unsigned len);
```

- int: A beolvasott byte-ok száma, vagy 0, ha file vége, -1, ha hibás.
- handle: A file
- buff: A buffer kezdőcíme, ahova olvas
- len: A beolvasandó byte-ok száma.

A **write** függvény a file-ba való olvasást teszi lehetővé egy kijelölt bufferből. Az írás az éppen aktuális file-pozíciótól történik. A file-pozíció a kiírt byteok számával növekszik.

```
#include <io.h>
int write(int handle,void *buff,unsigned len);
```

- int: A kiírt byte-ok száma, vagy -1, ha hibás.
- handle: A file
- buff: A buffer kezdőcíme, ahonnan ír.
- len: A kiírandó byte-ok száma.

Az **lseek** függvény file-pozíciót állít be.

```
#include <io.h>
long lseek(int handle,long offset,int origin);
```

- long: A file-poíció a file elejétől, hiba esetén -1.
- handle: A file handle.
- offset: A módosítás nagysága
- origin: A módosítás honnan történik:
  - SEEK_SET A file elejéről.
  - SEEK_CUR Az aktuális helyről.
  - SEEK_END A file végéről.

Az **tell** függvény lekérdezi az aktuális file-pozíciót.

```
#include <io.h>
long tell(int handle);
```

- long: Az aktuális file pozíció
- handle: A file handle

Természetesen az alacsonyszintű file-kezelésnek még számos függvénye van, azonban ezeket hely hiányában, legnagyobb sajnálatunkra nem ismertethetjük.

10.2.2. Magasszintű file-kezelés

Forrás: http://www.doksi.hu

A magasszintű file-kezelés magasabb szintű szolgáltatásaiban és bonyolultabb függvényeiben tér el az előzőekben ismertetett kezelési módtol. Sajnos a több szolgáltatás mellet az adatforgalom sebessége kissé lassabb lett, mint azt az előző módszernél tapasztalhattuk. Ha nem csak felhasználói szempontból vizsgáljuk ezt a szintet, akkor láthatjuk, hogy ez un. bufferelt a file-kezelés, amely azt jelenti, hogy a beolvasott adatok nem közvetlenül jutnak a felhasználóhoz, hanem egy bufferbe olvasódnak be, ahonnan már a felhasználó hozzájuk juthat. (Írásnál ez afolyamat fordítva történik.) Ezt a file-kezelést gyakran nevezik **folyam** jellegű file-kezelésnek is.

A kettes szinten a legfontosabb fogalom az úgynevezett **FILE** struktúra, amely típusként van definiálva az **stdio.h-**ban. Minden file-hoz, amelyet használni kívánunk tartozni fog egy ilyen struktúra. A file-ra egy olyan pointer segítségével hivatkozunk, amely a file saját struktúrájára fog mutatni.

A legfontosabb függvényei:

fopen,fclose,getc,putc,fprintf,fscanf,fseek,ftell fread,fwrite

Az **fopen** függvény magasszintű file-kezelésre nyit meg egy file-t.

```
#include  <stdio.h>
FILE *fopen(const char *path,const char *mode);
```

- A FILE struktúra kezdőcíme, hiba esetén NULL pointer.
- A file neve teljes elérési úttal is lehet.
- A megnyitás módja

A megnyitás módja:
- **r** létező file megnyitása csak olvasásra.
- **w** új file létrehozása, vagy létező felülírása.
- **a** létező file megnyitása hozzáfűzésre.
- **r+** létező file megnyitása olvasásra, írásra.
- **w+** új file létrehozása, vagy új file megnyitása olvasásra, írásra.
- **a+** megnyitás olvasásra és hozzáfűzésre. Ha nincs file létrehozza.
- **t** szöveges megnyitás. (Pl.r+t)
- **b** bináris megnyitás. (Pl.w+b)

Nézzünk egy egyszerű példát a megnyitásra:

```
FILE  *fp;        /* struktúra mutató deklarációja */

fp=fopen("c:\\word\\akrmi.doc","rt");
```

Ne feledkezzünk el a két \-rõl lásd a karakter-konstansok

Az **fclose** függvény megnyitott file-t zár le.

```
#include  <stdio.h>
int flcose(FILE *stream);
```

- Hiba esetén EOF.
- Struktúra mutató.

Forrás: http://www.doksi.hu

A **getc** függvény egy karaktert olvas a file-ból

```
#include  <stdio.n>
int getc(FILE *stream);
```

A beolvasott karakter, file vég, vagy hiba esetén EOF. | Struktúra mutató.

A getc függvény int típusú függvény, bár karaktert ad vissza. Ennek az az oka, hogy a transzparens átvitelbe, vagyis a 0-255 ábrázolási tartományba, az EOF karakter nem fér bele. Ezért az EOF -1 értékű egész lesz.

A **putc** függvény egy karaktert ír ki a file-ba

```
#include  <stdio.h>
int putc(int c,FILE *stream);
```

A kiírt karakter, vagy hiba esetén EOF. | Struktúra mutató. | A kiírandó karakter.

Az **fprintf** függvény formázott kiírást biztosít a file-ba teljesen úgy, mintha az stdout-ra írnánk.

```
#include  <stdio.h>
int fprintf(FILE *stream,const char *format,...);
```

Lásd printf. | Struktúr a mutató | Lásd printf.

Az **fscanf** függvény formázott beolvasást biztosít file-ból.

```
#include  <stdio.h>
int fscanf(FILE *stream,const char *format,...);
```

Lásd scanf | Struktúr a mutató | Lásd scanf

Az **fseek** függvény a file-pozíciót állítja be magasszintű file-kezelés esetén.

```
#include  <stdio.h>
int fseek(FILE *stream,long offset,int origin);
```

Ha sikeres 0 | Struktúr a mutató | Lásd seek | Honnan

Az **ftell** függvény az aktuális file-pozíciót szolgáltatja.

```
#include  <stdio.h>
long ftell(FILE *stream);
```

File pozíció, hiba esetén -1 | Struktúr a mutató

Az **fread** függvény file-ból tömböt olvas be.

Forrás: http://www.doksi.hu

```c
#include <stdio.h>
size_t fread(void *ptr,size_t size,size_t n,FILE *stream);
```

A beolvasott elemek száma, nem byte-ok száma!

A buffer, ahova olvasunk

A beolvasandó elemek száma

Struktúra mutató

A beolvasandó elemek mérete.

A **size**_t típus az **stdio.h**-ban van definiálva **valójában** egy **unsigned int**.

Az **fwrite** függvény tömböt ír ki a file-ba.

```c
#include <stdio.h>
size_t fwrite(void *ptr,size_t size,size_t n,FILE *stream);
```

A kiírt elemek száma, nem byte szám!

A buffer, ahonnan írunk

A kiírandó elemek száma

Struktúr a mutató

A kiírandó elemek mérete.

Forrás: http://www.doksi.hu

Példaprogramok

1. példa
Az első programunk az 'Ez aztan a nem semmi!' szöveget irja a képernyőre, majd sort emel. Feltétlenül nézzük át a példa kapcsán a printf() függvény használatát.

```
/* 1. példa */

#include        <stdio.h>

void main (void)  /* a void elhagyható */
{
        printf( "Ez aztan a nem semmi!\n" );
} /* main */
```

2. példa
Példa változók deklarálására és használatára. Kezdeti értéket adhatunk rögtön a deklarációban.

```
/* 2. példa */

#include        <stdio.h>

main()
{
int             a, b=-2, c;
                /* változók deklarálása, kezdeti érték adás */
int     osszeg, szorzat;

   a = 1;
   c = 3;
   osszeg = a + b + c;/* aritmetikai kifejezések */
   szorzat = a * b * c;
   printf( "Az osszeg = %d\n", osszeg );
   printf( "A szorzat = %d\n", szorzat );
} /* main */
```

3. példa
Példa konstansok deklarálására, használatára és megszün-tetésére. A konstansneveket nagybetüvel szokás irni!

```
/* 3. példa */

#include        <stdio.h>

#define         TUCAT           12
#define         UZENET      "Szoveg konstans vagyok\n"

main()
{
int             w;

                w = TUCAT;
                printf( "w = %d\n", w );
#undef TUCAT           /* innentől TUCAT már nem használható */
                printf( UZENET );
} /* main */
```

Forrás: http://www.doksi.hu

4. példa

Példa az egész számábrázolás veszélyeire. Próbáljuk megin-dokolni, hogy a túlcsordult eredmények miért azt az értéket adják mint amit a képernyőn látunk!

```
/* 4. példa */

#include        <stdio.h>

main()
{
int             a, b, c;
unsigned int    aa, bb, cc;

        a = 15000;
        b = 15000;
        c = a + b;              /* sikeres az összeadás          */
        printf( " a =  %6d,  b = %6d,  c = %6d\n", a, b, c );
        a = 20000;
        b = 20000;
        c = a + b;              /* túlcsordulás lesz     */
        printf( " a =  %6d,  b = %6d,  c = %6d\n", a, b, c );
        aa = 30000;
        bb = 30000;
        cc = aa + bb;           /* ez rendben van...     */
        printf( "aa =  %6u, bb = %6u, cc = %6u\n", aa, bb, cc);
        aa = 40000;
        bb = 40000;
        cc = aa + bb;           /* ez megint túlcsordul */
        printf( "aa =  %6u, bb = %6u, cc = %6u\n", aa, bb, cc);
} /* main */
```

5. példa

Egy újabb, immár részletesebb példa a printf használatára. A formátumstring alapos ismerete nélkül ne lépjünk túl ezen a példán!

```
/* 5. példa */

#include        <stdio.h>

main()
{
char            x;
int             a, b;
float   c, d;
        x = 'w';
        a = -57;
        b = 197;
        c = 2.0897;
        d = -3.67e8;
        printf("A valtozok ertekenek kiiratasa\n");
        printf("    a = %d          b = %d\n", a, b);
        printf("    c = %f     d = %e\n", c, d);
        printf("    x = %c\n",x);
} /* main */
```

Forrás: http://www.doksi.hu

6. példa

Példa az aritmetikai operátorokra. Nézzük át az elméletben az operátorok precedenciáját!

```
/* 6. példa */

#include        <stdio.h>

main()
{
int     x, y, mar;
float   a = 5.2, b = -2;
float   ossz, kul, szor, oszt;

        x = 6;
        y = 15;
        ossz = a + b;
        kul = a - b;
        szor = a * b;
        oszt = a / b;
        mar = x % y;
        printf("osszeg = %+.2f kulonbseg = %+.2f\n",ossz,kul);
        printf("szorzat= %+.2f hanyados  = %+.2f\n",szor,oszt);
        printf("maradek= %.3d\n",mar);
} /* main */
```

7. példa

Példa a bit operátorokra. A kizáró vagy illetve az egyes komplemens képző operátort is próbáljuk ki!

```
/* 7. példa */

#include        <stdio.h>

main()
{
int             a, b, c, d, e;

        a = 16;
        b = a << 2;     /* kettővel balra léptet */
        c = a >> 4;     /* néggyel jobbra léptet */
        d = a & 0x1;    /* bitenkénti és kapcsolat hexa 1-el */
        e = a | o07;    /* bitenkénti vagy kapcsolat okt.7-el */
        printf("a= %d b= %d c= %d d= %d  e = %d\n",a,b,c,d,e);
} /* main */
```

Forrás: http://www.doksi.hu

8. példa

Példa a speciális operátorokra. A háromoperandusú operátor későbbi példában szerepel.

```
/* 8. példa */

#include        <stdio.h>

main()
{
int             a, b, c, d, e, *mutato;

        mutato = &a;            /* mutató a-ra mutat     */
        a = 10;
        b = a++;                                /* a b-be, a növelése    */
        c = ++a;                                /* a növelése, a c-be    */
        d = *mutato;            /* d-be a mutatott érték */
        e = sizeof(mutato);             /* mutato mérete e-be    */
        printf( "a = %d, b = %d, c = %d\n", a, b, c );
        printf( "d = %d, e = %d\n", d, e );
} /* main */
```

9. példa

Példa a kevert típusú kifejezésekre.

```
/* 9. példa */

#include  <stdio.h>

main()
{
int             i;
char            j;
float     x;

        i = -5; j = 3;
        x = 1.0;
        printf( "  -5/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "  -4/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "  -3/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "  -2/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "  -1/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   0/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   1/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   2/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   3/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   4/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   5/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   6/3 +   1.0  is %f\n", i/j + x ); i++;
        printf( "   7/3 +   1.0  is %f\n", i/j + x ); i++;

        i = -5; j = 3;
        x = 1.0;
        printf( "  1.0  + (-5/3) is %f\n", x + i/j ); i++;
        printf( "  1.0  + (-4/3) is %f\n", x + i/j ); i++;
        printf( "  1.0  + (-3/3) is %f\n", x + i/j ); i++;
        printf( "  1.0  + (-2/3) is %f\n", x + i/j ); i++;
        printf( "  1.0  + (-1/3) is %f\n", x + i/j ); i++;
        printf( "  1.0  +   0/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   1/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   2/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   3/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   4/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   5/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   6/3  is %f\n", x + i/j ); i++;
        printf( "  1.0  +   7/3  is %f\n", x + i/j ); i++;
} /* main */
```

Forrás: http://www.doksi.hu

10. példa
Példa a for() ciklus használata. Faktoriális számolása. A for() ciklus utáni utasítás zárójel tulajdonképpen felesleges.

```
/* 10. példa */

#include        <stdio.h>

main()
{
int             i, x;
double          y;

        x =5;
        y=1;
        for ( i = 1; i <= x; i++ )
          {
           y = y * i;
          } /* for */
        printf ( "%d faktorialisa = %f\n",x,y);
} /* main */
```

11. példa
Példa a while() ciklus használatára. Gondoljuk át, hogy milyen egész érték felel meg logikai nullának!

```
/* 11. példa */

#include        <stdio.h>

main()
{
int             p=10;

        while ( p )  printf( "p = %d\n", p-- );
} /* main */
```

12. példa
Példa a do...while ciklus használatára. Nézzük meg azt az esetet is amikor x integer típusú változó!

```
/* 12. példa */


#include        <stdio.h>

main()
{
float   x;

        printf( "Nekiallok varakozni !" );
        x = 0;
        do
          {
           x++;
          }
        while ( x < 10000 );
        printf( "\n  Befejeztem.  \n" );
} /* main */
```

Forrás: http://www.doksi.hu

13. példa
Példa az if elágazásra. Parancsértelmezés. Gondolkodjunk el azon, hogy mely utasitás zárójelek feleslegesek!

```
/* 13. példa */

#include        <stdio.h>

main()
{
char            c;

        printf( "Usse be a parancskodot (a, b, c ): " );
        c = getchar();          /* a kod beolvasása */
        if ( c == 'a' )
          {
           printf( "Az a parancs vegrehajtva\n" );
          } /* if */
        else
        {
           if ( c == 'b' )
             {
                printf( "A b parancs vegrehajtva\n" );
             } /* if */
           else
           {
                if ( c == 'c')
                  {
                   printf( "A c parancs vegrahajtva\n" );
                  } /* if */
                else
                  {
                   printf( "Illegalis parancs: '%c'\n", c );
                  } /* else */
           } /* else */
        } /* else */
} /* main */
```

14. példa
Az előző feladat megoldása switch-el. Mi történik ha el-hagyom a break utasitásokat?

```
/* 14. példa */
#include        <stdio.h>
main()
{
char            c;
        printf( "Usse be a parancskodot (a, b, c ): " );
        c = getchar();          /* a kod beolvasása */
        switch ( c )
          {
           case 'a': printf( "Az a parancs vegrehajtva\n" );
                     break;
           case 'b': printf( "A b parancs vegrehajtva\n" );
                     break;
           case 'c': printf( "A c parancs vegrehajtva\n" );
                     break;
           default : printf( "Illegalis parancs\n" );
          }
} /* main */
```

Forrás: http://www.doksi.hu

15. példa
Példa a getchar() és a putchar() függvények használatára. Az eredmény csak az ENTER lenyomása után válik láthatóvá.
Miért?

```
/* 15. példa */

#include    <stdio.h>

main()
{
char            c;

        c = getchar();
        putchar( c );
} /* main */
```

16. példa
Példa a getchar() és a putchar() függvények ciklusban töténő használatára. Feltétlenül jegyezzük meg az EOF kódját a program inditása elött.

```
/* 16. példa */

#include        <stdio.h>

main()
{
char            c;

        c = getchar();
        while ( c != EOF )
                        /* amíg a beolvasott karakter nem EOF */
        {               /*     EOF = CTRL-Z     */
          putchar( c );/* vissza stdout-ra      */
          c = getchar();
        } /* while */
} /* main */
```

17. példa
Példa nagybetű-kisbetű konverzióra az if segítségével. Csak a nagybetűket alakítja át a többit változatlanul hagyja.

```
/* 17. példa */

#include        <stdio.h>

main()
{
char            c;

        while ( ( c = getchar() ) != EOF )
        {
          if ( 'A' <= c && c <= 'Z' )
          {
                c = c + 'a' - 'A';
          } /* if */
          putchar( c );
        } /* while */
} /* main */
```

Forrás: http://www.doksi.hu

18. példa
Példa kisbetű-nagybetű felcserélésre feltételes kifejezéssel. Itt található példa a háromoperandusú kifejezés használatára.

```
/* 18. példa */
#include    <stdio.h>

main()
{
char            c;

        while ( ( c = getchar() ) != EOF )
        {
          putchar(('A'<=c && c<='Z') ? c-'A'+'a' : c+'A'-'a');
        } /* while */
} /* main */
```

19. példa
Példa számsorozat beolvasására, a szóközöket átlépve. Gondoljuk át pontosan a continue működését!

```
/* 19. példa */

#include        <stdio.h>

main()
{
char            c;
float   i;

        i = 0;
        while ( ( c = getchar() ) != EOF )
          {
           if ( c >= '0' && c <= '9' ) i = i*10 + c - '0';
            else if ( c == ' ' ) continue;
                else break;
          } /* while */
        printf( "Az atalakitott ertek %f\n",i );
} /* main */
```

Forrás: http://www.doksi.hu

20. példa

Példa integer konverziójára ASCII kódsorozattá. Ez a program is tartalmaz olyan utasítás zárójeleket amelyek csak az olvashatóság megkönnyítésére kerültek bele.

```
/* 20. példa */

#include    <stdio.h>

main()
{
unsigned   int    num;          /* konvertált szám      */
char               c;                      /* beolvasott karakter */
char               bit;             /* kicsorgó bit értéke  */
int                i;                      /* bitszámláló           */

        num = 0;
        printf( "Uss be egy pozitiv szamot: " );
        while ( ( c = getchar() ) != EOF )
        {
          if ( c == '\n' || ( c < '0' || c > '9' ) )
          {
                printf( "%u a kettes szamrendszerben ", num );
                for ( i = 16; i > 0; i-- )
                {
                  bit = !( !( num & 0x8000 ) );
                  bit = bit + '0';    /* ASCII lesz   */
                  putchar( bit );
                  num = num << 1;   /* num balra lép*/
                } /* for */
                putchar( '\n' );
                printf( "Uss be egy pozitiv szamot: " );
                num = 0;
          } /* if */
          else
          {                      /* ASCII-bináris konv.          */
                num = num * 10 + c - '0';
          } /* else */
        } /* while */
} /* main */
```

21. példa

Példa függvénydefinícióra és függvényhívásra. Miért nem kell deklarálni az add függvényt?

```
/* 21. példa */
main()
{
int                 sum,a1,a2,a3;
        a1 = 5;
        a2 = -7;
        a3 = 30;
        sum = add( a1, a2, a3 );
        printf( "Az összeg = %d\n", sum );
} /* main */
int add( a, b, c )
int       a;              /* a függvény ...           */
int       b;              /*  argumentumainak ...           */
int       c;              /*  deklarációja            */
{
int                 sum;      /* belső változó definíciója   */

        sum = a + b + c;
        return( sum );
} /* add */
```

Forrás: http://www.doksi.hu

22. példa
Példa több return utasítást tartalmazó függvényre. Kell-e feltétlenül jelezni a függvény típusát?

```
/* 22. példa */

main()
{
int             a;

        a = maximum( 5, -3 );
        printf( "A maximum %d\n", a );
} /* main */

int             maximum( a, b )
int             a, b;
{
        if ( a < b )
          {
            return( b );
          } /* if */
        else  return( a );
} /* maximum */
```

23. példa
Az előző függvény egy másfajta realizációja. Egy másik példa a háromoperandusú operátor használatára.

```
/* 23. példa */

#include        <stdio.h>

main()
{
int             a;

        a = maximum( 5, -3 );
        printf( "A maximum %d\n", a );
} /* main */

int             maximum( a, b )
int             a, b;
{
        return( a < b ? b : a );
} /* maximum */
```

Forrás: http://www.doksi.hu

24. példa
Példa a hatványozás megvalósítására. (Tudjuk, hogy a math.h-ban pow néven már realizálták.)

```
/* 24. példa */

#include        <stdio.h>

int             power( int, int )


main()
{
int             kitevo, alap, ered;

        printf( "Positiv alap, novekvo kitevok\n" );
        for ( kitevo = 0, alap = 4; kitevo < 5; kitevo++ )
          {
           ered = power( alap, kitevo );
           printf( "%d a(z) %d.-on: %d\n",alap,kitevo,ered );
          } /* for */
        printf( "Negativ alap, novekvo kitevok\n" );
        for ( kitevo = 0, alap =-2; kitevo < 5; kitevo++ )
          {
           ered = power( alap, kitevo );
           printf( "%d a(z) %d.-on: %d\n",alap,kitevo,ered );
          } /* for */
        printf( "0 hatvanyai\n" );
        for ( kitevo = 0, alap = 0; kitevo < 3; kitevo++ )
          {
           ered = power( alap, kitevo );
           printf( "%d a(z) %d.-on: %d\n",alap,kitevo,ered );
          } /* for */
        printf( "Nehany ertek negativ hatvanya\n" );
        for ( kitevo = -2, alap = -2; alap < 3; alap++ )
          {
           ered = power( alap, kitevo );
           printf( "%d a(z) %d.-on: %d\n",alap,kitevo,ered );
          } /* for */
        kitevo = -1;
        alap =  5;
        ered = power( alap, kitevo );
} /* main */
int             power( base, pwr )
        int     base, pwr;
{
int     val;

        if ( pwr < 0 )
          {                                     /* negatív kitevő */
           if ( base == 1 )  return( 1 );
                                                /* 1 minden hatványa 1 */
           return( 0 );
          } /* if */
        val = 1;        /* ez jó 0 kitevőre is ! */
        while ( pwr-- > 0 ) val *=base;
                                        /* ismételt szorzás, míg kell */
        return( val );
} /* power */
```

Forrás: http://www.doksi.hu

25. példa
Példa függvényhívásra eltérő paraméterszámmal. Gondolkodjunk el, hogy y értéke az első kiiratáskor miért a képernyőn látható érték!

```
/* 25. példa */
#include        <stdio.h>

int             fuggv( int, int )

main()
{
int             i;

        i = 200;
        printf( "<< main  >>-ben    i = %d\n", i );
        fuggv( i );             /* egy paraméter        */
        printf( "<< main  >>-ben    i = %d\n", i );
} /* main */

int             fuggv( x, y )
int             x;                      /* a függvény két       */
int             y;                      /* argumentumot vár  */
{
        printf( "<< fuggveny >>-ben x = %d, y = %d\n", x, y );
        x = y = 100;
        printf( "<< fuggveny >>-ben x = %d, y = %d\n", x, y );
} /* fuggv */
```

26. példa
Példa nem egész típusú függvény deklarációjára, hívására és definíciójára. Mikor kell feltétlenül függvény deklaráció? Mit mond ez a függvényről?

```
/* 26. példa */

float   replusz(float,float);   /* deklaráció */

main()
{
float   x, y;
float   ret;

        x = 10.0; y = 20.0;
        ret = replusz( x, y );
        printf( "Az eredmeny = %f\n", ret );
} /* main */

float   replusz( a, b )
        float   a, b;
  {
        float   ered;

        ered = ( a * b ) / ( a + b );
        return( ered );
  } /* replusz */
```

Forrás: http://www.doksi.hu

27. példa
Példa tömbökre - definíció, felhasználás, előkészítés futásidőben. Gondoljuk át a tömb indexváltozójának tartományát!

```
/* 27. példa */

float   arr[ 20 ];                  /* 20 elemű tömb */

#include        <stdio.h>

main()
{
int             i;

        i = 0;
        while ( i < 20 )
          {
           arr[ i ] =     i / 10.0;
           printf( "arr[ %d ] = %.4f\n", i, arr[ i ] );
           i++;
          } /* while */
} /* main */
```

28. példa
Példa kétdimenziós tömb definiálása, futásidőben történő előkészítése és felhasználása.

```
/* 28. példa */
#include        <stdio.h>

#define         OSZLOP      6
#define         SOR         8

int             arr[ OSZLOP ][ SOR ];       /* tömbdefiníció        */

main()
{
int             i, j;                       /* indexváltozók        */

        for ( i = 0; i < OSZLOP; i++ )
        {
           for ( j = 0; j < SOR; j++ ) arr[ i ][ j ] = i * j;
        } /* for */
        for ( i = 0; i < OSZLOP; i++ )
        {                           /* tömb felhasználása */
           printf( "arr[ %d ][*]:\t", i );
           for ( j = 0; j < SOR; j++ )
           {
                printf( "%5d", arr[ i ][ j ] );
           } /* for */
           printf( "\n" );
        } /* for */
} /* main */
```

Forrás: http://www.doksi.hu

29. példa
Példa string beolvasására karakteres tömbbe. Mire való és melyik a string vége (EOS) karakter? Hogyan adunk karaktertömbnek kezdeti értéket?

```
/* 29. példa */

#include        <stdio.h>

char            szoveg[ 100 ] = "Ez a kezdeti szoveg";

main()
{
int             i, meret;
char            c;

        printf( "'%s'\n", &szoveg[ 0 ]);
        printf( "Uss be egy uj szoveget ENTER-el lezarva: " );
        for(i=0; (c = getchar()) != EOF && c != '\n'; i++ )
        {
           szoveg[ i ] = c;
        } /* for */
        szoveg[ i ] = 0;          /* EOS generálása */
        for(i = 0,meret = 0;szoveg[i] != 0;i++)        meret++;
        printf( "Az uj szoveg hossza = %d\n", meret );
} /* main */
```

30. példa
Példa karaktertömb egyszerű másolására. Figyeljük meg, hogy a karaktertömb neve az első elem címével egyezik meg!

```
/* 30. példa */

#include        <stdio.h>

char            tombbol[] = "Ezt a tombot masoljuk";
char            tombbe[ 20 ];

main()
{
int             i;

        i = 0;
        while ( tombbol[ i ] != '\0' )
        {
           tombbe[ i ] = tombbol[ i ];
           i++;
        } /* while */
        tombbe[ i ] = '\0';     /* EOS generálása */
        printf( "A másolandó string: %s\n", tombbol );
        printf( "A másolt string   : %s\n", tombbe );
} /* main */
```

Forrás: http://www.doksi.hu

31. példa

Az előző példa C-szerű ciklusban.

```
/* 31. példa */
#include        <stdio.h>

char            tombbol[] = "Ezt a tombot masoljuk";
char            tombbe[ 20 ];

main()
{
int             i;
        i = 0;
        while ( ( tombbe[ i ] = tombbol[ i ]) != '\0' )  i++;
        printf( "A másolandó string: %s\n", tombbol );
        printf( "A másolt string   : %s\n", tombbe );
} /* main */
```

32. példa

Példa stringek méretének összehasonlítására. Nézzük át, hogy a string.h header file milyen függvényeket tartalmaz még!

```
/* 32. példa */
#include    <stdio.h>
#include    <string.h>
char            buff1[ 200 ];
char            buff2[ 200 ];
main()
{
int             ered, meret1, meret2;
        printf( "Uss be ket stringet!\n" );
        gets( buff1 );          /* első sor beolvasása    */
        gets( buff2 );          /* második sor beolvasása */
        ered = strcmp( buff1, buff2 );
        if ( ered == 0)         /* eredmény értékelése */
        {
          printf( "A ket string teljesen egyforma.\n" );
          printf("Hoszusaguk: %d karakter.\n",strlen(buff1));
        } /* if */
        else
        {
          meret1 = strlen( buff1 );
          meret2 = strlen( buff2 );
          if ( meret1 == meret2 )
          {
                printf( "Csak a hosszusaguk egyezik meg.\n" );
                printf( "Hosszusaguk: %d karakter.\n",meret1 );
          }
          else if ( meret1 < meret2 )
          {
          printf( "Az elso rovidebb mint a masodik.\n" );
          printf( "Hosszusaguk: %d es %d karakter.\n", meret1,meret2);
          }
          else
          {
          printf( "Az elso hosszabb mint a masodik.\n" );
          printf( "Hosszusaguk: %d es %d karakter.\n",meret1, meret2 );
          }
        } /* else */
} /* main */
```

Forrás: http://www.doksi.hu

33. példa
Példa a címaritmetikára. Gondoljuk át, hogy milyen műveletek végezhetők mutatókkal!

```
/* 33. példa */

#include        <stdio.h>

int             tomb[] = { 0, 10, 20, 30, 40, 50, 60 };

main()
{
int             i, *tombmut;

        tombmut = tomb;
        for ( i = 0; i < 7; i++ )
        {
          printf( "%d. tomb elem: %4d\t", i, *tombmut++ );
          printf( "cime: %4x\n", tombmut );
        } /* for */
} /* main */
```

34. példa
Példa többszörösen indirekt pointerek definiálására, előké-szítésére és használatára.

```
/* 34. példa */
#include        <stdio.h>

main()
{
float   alap;
float   *mut1;          /* pointer az alapra   */
float   **mut2;          /* pointer a pointerre */
float   ***mut3;         /* stb...       */
float   ****mut4;

        mut1 = &alap;/* pointerek előkészitése */
        mut2 = &mut1;
        mut3 = &mut2;
        mut4 = &mut3;

        alap = 0.0;
        printf( "Az alap = %f\n", alap );
        *mut1 = *mut1 + 10.0;
        printf( "Az alap = %f\n", *mut1 );
        **mut2 = **mut2 + 10.0;
        printf( "Az alap = %f\n", **mut2 );
        ***mut3 = ***mut3 + 10.0;
        printf( "Az alap = %f\n", ***mut3 );
        ****mut4 = ****mut4 + 10.0;
        printf( "Az alap = %f\n", ****mut4 );
} /* main */
```

Forrás: http://www.doksi.hu

35. példa

Példa struktúrára. A strukturák felépítésének, kezdeti érték adásának, használatának tanulmányozása mellett alaposan nézzük át a scanf() függvény használatát!

```
/* 35. példa */

#include        <stdio.h>

struct  szemely
{
        char    nev[ 30 ];
        char    cim[ 50 ];
        char    szemigszam[ 12 ];
        long    telefon;
};

struct  szemely valaki          = { "GABOR GABOR",
                                    "BP. GAZDAGRET TUZKO U. 8.",
                                    "AU-1 182347",
                                    1334512 };

main()
{

        printf( "\nValaki parameterei:\n\n" );
        printf( "neve:\t\t%s\n",valaki.nev );
        printf( "cime:\t\t%s\n",valaki.cim );
        printf( "szemig. szama:\t%s\n",valaki.szemigszam );
        printf( "telefon szama:\t%ld\n",valaki.telefon );

        printf( "\nkerem %s uj telefonszamat: ",valaki.nev );
        scanf( "%ld", &valaki.telefon );

        printf( "\nValaki uj parameterei:\n\n" );
        printf( "neve:\t\t%s\n",valaki.nev );
        printf( "cime:\t\t%s\n",valaki.cim );
        printf( "szemig. szama:\t%s\n",valaki.szemigszam );
        printf( "telefon szama:\t%ld\n",valaki.telefon );

} /* main */
```

36. példa

```
/* 36. példa */

#include        <stdio.h>

struct  szemszam
{
        char            sex;
        int             szulev;
        int             szulhonap;
        int             szulnap;
        char            chsum[ 4 ];
};
char                    ktomb[ 10 ];
int                     itomb[ 10 ];
float           ftomb[ 10 ];
double          dtomb[ 10 ];
```

Forrás: http://www.doksi.hu

```
struct szemszam         stomb[ 10 ];

main()
{
        printf( "Kulonbozo tipusok merete byte-ban:\n" );
        printf( "\tChar merete     = %d\n", sizeof(  char  ) );
        printf( "\tInt merete      = %d\n", sizeof(  int   ) );
        printf( "\tFloat merete    = %d\n", sizeof( float  ) );
        printf( "\tDouble merete   = %d\n", sizeof( double ) );
        printf( "\tSzemszam merete = %d\n",
sizeof( struct szemszam));

        printf( "Tombok merete byte-ban:\n" );
        printf( "\tKtomb merete    = %d\n", sizeof( ktomb ) );
        printf( "\tItomb merete    = %d\n", sizeof( itomb ) );
        printf( "\tFtomb merete    = %d\n", sizeof( ftomb ) );
        printf( "\tDtomb merete    = %d\n", sizeof( dtomb ) );
        printf( "\tStomb merete    = %d\n", sizeof( stomb ) );

        printf( "Tombok elemszama:\n" );
        printf( "\tKtomb: %d\n", sizeof(ktomb)/sizeof(char) );
        printf( "\tItomb: %d\n", sizeof(itomb)/sizeof(int) );
        printf( "\tFtomb: %d\n", sizeof(ftomb)/sizeof(float) );
        printf( "\tDtomb: %d\n",
 sizeof( dtomb ) / sizeof( double ) );
        printf( "\tStomb: %d\n",
 sizeof( stomb ) / sizeof( struct szemszam));
} /* main */
```

37. példa
Példa a text üzemmód lekérdezésére. Toldjuk meg a programot egy olyan résszel ami a text_info struktura további elemeit is a képernyőre irja!

```
/* 37. példa */
#include        <stdio.h>
#include        <conio.h>

struct text_info         mod;  /* conio.h-ban dekl. struktura */

main()
{
        gettextinfo( &mod );
        printf( "Text mod informaciok:\n" );
        printf( "\ttext mod kodja   : %u\n",mod.currmode );
        printf( "\tablak szelessege : %u\n",mod.screenwidth );
        printf( "\tablak magassaga  : %u\n",mod.screenheight );
        printf( "\tszoveg attributum: %u\n",mod.attribute );
} /* main */
```

Forrás: http://www.doksi.hu

38. példa
Példa a text üzemmódban használható előtér szinekre. Miért nem látjuk a 0 kodú előtérszint?

```
/* 38. példa */
#include        <conio.h>
#include        <stdio.h>

main()
{
int             i;

        clrscr();       /* aktuális ablak törlése */
        for ( i = 0; i < 16; i++ )
        {
                textcolor( i );         /* előtérszin állítása */
                cprintf( "A(z) %d. eloterszin vagyok ",i );
                printf( "\n" );
        } /* for */
} /* main */
```

39. példa
Példa a text üzemmódban használható háttér szinekre.

```
/* 39. példa */

#include        <conio.h>

main()
{
int             i;

        for ( i = 0; i < 8; i++ )
        {
                textbackground( i );    /* háttérszin állítása */
                window( 10 * i + 1, 1, 10 * ( i + 1 ), 25 );
                clrscr();
        } /* for */
} /* main */
```

40.példa
Példa egy text üzemmódú ablak tartalmának memóriába mentésére majd visszatöltésére. Figyeljük meg, hogy melyik függvény használ relatív illetve abszolút koordinátákat!

```
/* 40. példa */

#include        <conio.h>
#include        <dos.h>
#include        <stdio.h>
#include        <stdlib.h>

main()
{
char            memhely[ 11 * 6 * 2 ];
int             ered;

        clrscr();
        window( 35, 1, 45, 6 );         /* ablak létrehozása */
        textbackground( RED );
        textcolor( GREEN );
        clrscr();
```

Forrás: http://www.doksi.hu

```
        gotoxy( 4, 3 );
        cprintf( "abcd" );
        window( 1, 1, 80, 25 );
        textbackground (BLACK );
        textcolor( WHITE );
        gotoxy( 1, 25 );
        printf( "Gombnyomasra varok !" );
        getch();        /* billentyűnyomásra várakozás */
        ered = gettext( 35, 1, 45, 6, memhely );   /* mentés */
        if ( ered == 0 )
        {
                printf( "Sikertelen mentes !" );
                exit(1);
        } /* if */
        clrscr();
        sound( 1000 );/* hanggenerálás              */
        delay (100 );   /* késleltetés              */
        nosound();      /* hanggenerátor kikapcsolása */
        puttext( 1, 10, 11, 15, memhely );     /* visszatöltés */
        if ( ered == 0 )
        {
                printf( "Sikertelen visszatoltes !" );
                exit(1);
        } /* if */
} /* main */
```

41.példa

Példa a grafikus üzemmód inicializálására.

```
/* 41. példa */
#include        <graphics.h>
#include        <stdio.h>
#include        <stdlib.h>
#include        <conio.h>
main()
{
int             gd, gm, hiba;

        detectgraph( &gd, &gm );      /* hardware vizsgálat */
        if ( gd < 0 )
        {
                printf( "Nincs grafikus meghajto !" );
                exit(1);
        } /* if */
        printf( "Grafikus meghajto kod:%d, mod kod:%d\n",
gd, gm );
        getch();
        initgraph( &gd, &gm, "" );    /* inicializálás */
        hiba = graphresult();         /* eredmény lekérdezés */
        if (hiba < 0 )
        {
                printf( "initgraph hiba: %s.\n",
                grapherrormsg( hiba ) );
                exit(1);
        } /* if */
        bar( 0, 0, getmaxx()/2, getmaxy() );
        getch();
        closegraph();  /* grafikus rendszer lezárása */
} /* main */
```

Forrás: http://www.doksi.hu

42.példa
Példa a kitöltési paraméterek használatára a képernyőtar-talom elmentésére valamint a visszatöltés megvalósítására. Nézzünk utána hogyan lehet saját mintázatot használni!

```
/* 42. példa */
#include        <stdio.h>
#include        <alloc.h>
#include        <graphics.h>
#include        <stdlib.h>
#include        <conio.h>

main()
{
int             gd = 1, gm = 4, hiba, x, xrel, yrel, a, b;
void            *memhely;
unsigned        meret;

        initgraph( &gd, &gm, "" );
        hiba = graphresult();
        if (hiba < 0 )
        {
                printf( "initgraph hiba: %s.\n",
                grapherrormsg( hiba ) );
                exit(1);
        } /* if */
        a = getmaxx();
        b = getmaxy();
        for ( x = 0; x < 12; x++)
        {
                setfillstyle( x, 1 );
                xrel = 20 * x;
                yrel = 8 * x;
                bar( 0 + xrel, 0 + yrel, a - xrel, b - yrel );
        } /* for */
        getch();
        line( 50, 25, 320, 25 );
        line( 50, 25, 50, 100 );
        line( 50, 100, 320, 100 );
        line( 320, 100, 320, 25 );
        meret = imagesize( 50, 25, 320, 100 );
                                                /* méret meghatározás */
        memhely = malloc( meret );  /* helyfoglalás       */
        getimage( 50, 25, 320, 100, memhely );
                                                /* memóriába mentés   */
        getch();
        cleardevice();
        putimage( 320, 100, memhely, COPY_PUT );
                                        /* visszatöltés  */
        free( memhely );            /* memória felszabadítás */
        getch();
        closegraph();
} /* main */
```

Forrás: http://www.doksi.hu

43. példa

Példa alacsony szintű file kezelésre. Alaposan tanul-mányozzuk az ide tartozó függvényeket! Milyen előnyökkel és hátrányokkal rendelkezik az alacsony szintű file kezelés a magas szintűhöz képest?

```
/* 43. példa */
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <math.h>
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
#define    MERET 100
#define    PI              3.1415926
extern int errno;
char                       fnev[] = "a:\\adat.dat";
main()
{
int            fh, x, ered, gd, gm, hiba;
long           poz;
double   adat;
         fh = open(fnev, O_RDWR | O_CREAT | O_BINARY, S_IWRITE);
                           /* file megnyitása */
         if ( fh == -1 )
         {
                  fprintf( stderr, "File hiba '%s'\n", fnev );
                  exit(1);
         } /* if */
         poz = sizeof( double ) * MERET;
         lseek( fh, poz, SEEK_SET );
         ered = write( fh, NULL, NULL );
         if ( ered == -1 )
         {
                  printf( stderr, "Helyproblema adodott, %d\n", errno );
                  exit(1);
         } /* if */
         for ( x = 0; x < MERET; x++ )
         {
                  adat = sin( 2 * PI * x / MERET );
                  poz = x * sizeof( double );
                  lseek( fh, poz, SEEK_SET ); /* file pozicio szamitas */
                  write( fh, &adat, sizeof( double ) );  /* file mutato allitas   */
                                                          /* file-ba iras  */
         } /* for */
         gd = 1;
         gm = 3;
         initgraph( &gd, &gm, "" );
         hiba = graphresult();
         if ( hiba < 0 )
         {
                  printf( "initgraph hiba: %s.\n",
                  grapherrormsg( hiba ) );
                  exit(1);
         } /* if */
         for ( x = 0; x < MERET; x++ )
         {
                  poz = x * sizeof( double );
                  lseek( fh, poz, SEEK_SET );
                  read( fh, &adat, sizeof( double ) );
                  putpixel( 3 * x, 100 - (int) (100 * adat), 2 );  /* file-ból olvasás */
         } /* for */
         getch();
         closegraph();
         close( fh );                                     /* file lezárása */
} /* main */
```

Forrás: http://www.doksi.hu

44. példa

Az előző példa realizálása magas szintű file kezeléssel. Miért célszerű a buffert kiüríteni az írás és olvasás között? Az irás és olvasást végző függvények állítják-e a file mutatót?

```
/* 44. példa */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
#include<math.h>
#define         MERET 100
#define         PI              3.1415926
char fnev[] = "a:\\adat.dat";
main()
{
FILE            *fp;
double  adat, tomb[ MERET ];
int             x, gd, gm, hiba;

        if ( ( fp = fopen( fnev, "w+b" ) ) == NULL )
                                                        /* file nyitás */
        {
                fprintf(stderr,"File nyitasi hiba'%s'\n",fnev );
                exit(1);
        } /* if */
        for ( x = 0; x < MERET; x++ )
        {
                tomb[ x ] = sin( 2 * PI * x / MERET );
        } /* for */
        x = fwrite( tomb, sizeof(double), MERET, fp );
                                                /* file-ba irás */
        if ( x != MERET )
        {
                fprintf( stderr, "Iras hiba '%s'\n",fnev );
                exit(1);
        } /* if */
        fseek( fp, 0, SEEK_SET );
                                                /* buffer kiürítés */
        x = fread( tomb, sizeof(double), MERET, fp );
        if ( x != MERET )
        {
                fprintf( stderr, "Olvasasi hiba '%s'\n",fnev );
                exit();
        } /* if */
        gd = 1; gm = 2;
        initgraph( &gd, &gm, "" );
        hiba = graphresult();
        if ( hiba < 0 )
        {
                printf( "initgraph hiba: %s.\n",
                grapherrormsg( hiba ) );
                exit(1);
        } /* if */
        for ( x = 0; x < MERET; x++ )
        {
                putpixel(3 * x, 100 - (int) (100 * tomb[ x ]), 2);
        } /* for */
        getch();
        closegraph();
        fclose( fp );                           /* file lezárása */
} /* main */
```

Forrás: http://www.doksi.hu

45. példa

Példa DOS parancsok C programból való végrehajtására. Fejlesszük tovább úgy a programot, hogy file-okat másoljon egyik könyvtárból a másikba!

```
/* 45. példa */
#include        <process.h>
#include        <conio.h>
#include        <stdio.h>
#include        <stdlib.h>
main()
{
int             ered;
        ered = system( "dir *.EXE" );
        if ( ered == -1 )
        {
          printf( "Hibas vegrehajtas" );
          exit( 1 );
        } /* if */
        getch();
        ered = system( "tree" );
        if ( ered == -1 )
        {
          printf( "Hibas vegrehajtas" );
          exit( 1 );
        } /* if */
        getch();
        ered = system ( "time" );
        if ( ered == -1 )
        {
          printf( "Hibas vegrehajtas" );
          exit( 1 );
        } /* if */
} /* main */
```

Forrás: http://www.doksi.hu

46. példa
Ez a példa a 9-es megszakítást irányítja át egy saját megszakítási rutinra. A rutin teszõleges billentyű lenyomására sípolni kezd, majd ha elengedjük a billentyűt a sípolás megszünik.

```
/* 46. példa */

#include <stdio.h>
#include <conio.h>
#include <dos.h>

void  interrupt   (*old9)(void); /* A régi IT vektor */
void  interrupt   new9(void);    /* Az új IT függvény */

char  sc,c;              /* A scan kód és a karakter kód*/

main()
  {
  clrscr();              /* Képernyő törlés */
  old9=getvect(9);       /* A régi IT vektor mentése */
  setvect(9,new9);       /* Az új IT vektor feltöltése */

  while(1)               /* Ez egy végtelen ciklus */
    {
    if(kbhit()) c=getche(); /* A karakter kivétele */
    if(c==27)   break;      /* Ha a karakter ESC kilép*/
    }
  setvect(9,old9);       /* A régi vektor visszaállítása */
  nosound();             /* A hang kikapcsolása */
  }

void  interrupt   new9(void)  /* A megszakítási rutin */
  {
  sc=inportb(0x60);      /* A klaviatúra port olvasása */
  if(sc<128)  sound(50);  /* Lenyomási scan kód megszólal*/
  else        nosound();  /* Felengedési kód elhallgat */

asm   {  pushf             /* A régi IT rutin meghívása*/
         call  old9
      }
  }
```

Forrás: http://www.doksi.hu

47. példa

Ebben a példában egy kritikus hibakezelést valósítunk meg.

```
/* 47. példa */

#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define IGNORE  0
#define RETRY   1
#define ABORT   2

int buf[500];

int error_win(char *msg);           /* Hibaüzenet kiírása */
int handler(int,int,int,int);       /* A hibakezelõ rutin */

/* Hibaüzenetek */

  static char *err_msg[] = {
      "írás védett",
      "ismeretlen egység",
      "a meghajtó nem kész",
      "ilyen parancs nincs",
      "CRC hiba",
      "hibás kérés",
      "seek hiba",
      "ismeretelen hordozó",
      "szektor hiba",
      "kifogyott a papír",
      "írás hiba",
      "olvasás hiba",
      "általános hiba",
      "tiltott",
      "tiltott",
      "érvénytelen diszk csere"
  };

int main(void)
   {
   harderr(handler);            /* A hibakezelõ megadása */
   clrscr();
   printf("Ne tegyél lemezt az A-ba");
   printf("Nyomj egy gombot\n");
   getch();
   printf("Nézzük csak A:-t\n");
   printf("fopen %p-vel tér vissza\n",
                  fopen("A:temp.dat", "w"));
   return 0;
   }

int error_win(char *msg)
   {
   int retval;

   cputs(msg);
   while(1)
```

Forrás: http://www.doksi.hu

```
    {
    retval= getch();
    switch(retval)
      {

      case 'a': case 'A': retval=ABORT;break;
      case 'r': case 'R': retval=RETRY;break;
      case 'i': case 'I': retval=IGNORE;break;
      }
    }
  return(retval);
  }
```

/*
A következõ **#pragma** elõfeldolgozó utasítás megszünteti azt a warnig üzenetet, amely azért keletkezik, mert nem használjuk az errval változót */

```
#pragma warn -par

int handler(int errval,int ax,int bp,int si)
{
  static char msg[80];
  unsigned di;
  int drive;
  int errorno;

  di=_DI;
  if(ax < 0)                            /* Diszk hiba-e */
    {
    error_win("Készülék hiba");      /* Nem */
    hardretn(ABORT);                    /* Kilépni */
    }
  drive = ax & 0x00FF;                  /* Diszk hiba */
  errorno = di & 0x00FF;
/* Milyen hiba */
  sprintf(msg,"%s iba a %c megh.-ón\r\nA,R,I:",
         err_msg[errorno],'A'+drive);
  hardresume(error_win(msg));
  return(ABORT);
  }
#pragma warn +par
```

Forrás: http://www.doksi.hu

48. példa

Ez a példa szintén a 9-es megszakítást irányítja át, de rezidensként betmarad a tárban. Ha a programot x paraméterrel hívjuk meg, és már rezidens módon bent volt a tárban kitörli magát. A program nem engedi azt sem, hogy többször töltsük be ugy, hogy a F2h megszakításba, amelyet a rendszer men használ betölt egy jelzést.

Az átirányított IT vektorokat szintén nemhasznált IT vektorokba mentjük.

Ha a DEBUG értéke 1 a program az IDE-ben futtatható, ha 0, az IDE-ben csak fordítani szabad, mert a fordítás során már TSR-ré válik.

```
/* 48. példa */

#include <dos.h>
#include <conio.h>
#include <stdlib.h>

#define  DEBUG 0

#define  FGV   (void interrupt (*)())
#define  TSR   0x55aa                   /* Ez csak segítség, hogy */
                                              /* ne kelljen sokat írni */
void  interrupt   new9();

void  interrupt   (*old9)();     /* Elõzõ példa */
void  interrupt   (*PSP)();      /* Segéd változó */

main(argc,argv)                         /* A main függvény és */
  int   argc;                           /* argumentumai */
  char  **argv;
  {
  if(argc>1 && *argv[1]=='x') /* Argumentum vizsgálat */
    {                                   /* Ha volt */
    if(MK_FP(0x55aa,0x55aa)!=getvect(0xf2))  exit(1);
    PSP=getvect(0xf3);                  /* A régi psp vissza */
    old9=getvect(0xf4);           /* A régi IT9 vissza */
    setvect(0xf2,NULL);           /* A jelzést törölni */
    setvect(0x9,old9);            /* Régi vektort vissza */
    _ES=FP_SEG(PSP);                    /* A psp betöltése ES-be */
    _AH=0x49;                           /* 49h szolgáltatás */
    geninterrupt(0x21);           /* INT 21h */
    exit(0);                            /* Kilépés */
    }

  old9=getvect(0x9);              /* Régi vektort elteszi */
  setvect(0x9,new9);            /* Új vektort beállítja */
  setvect(0xf2,FGV MK_FP(TSR,TSR));     /* Jelzés */
  setvect(0xf3,FGV MK_FP(_psp,0));      /* psp */
  setvect(0xf4,old9);                   /* Régi vektor */

#if DEBUG                               /* IDE mód */

  while(!kbhit())   if(27==getch())   break;
```

Forrás: http://www.doksi.hu

```
  setvect(0x9,old9);              /* Kilépés az IDE módból */
  nosound();

#else
  keep(0,_SS+(_SP+480)/16-_psp);  /* Tárrezidens kilépés */
#endif
  }

void  interrupt new9()                    /* Az új IT rutin
  {
  int   i;
  if(!(128 & inportb(0x60))) sound(100);      /* Hang */
  asm   {  pushf                                  /* Régi rutin hívása */
         call  old9
       }
  for(i=0;i<400;i++);                      /* Kis idõzítés */
  nosound();                                    /* Hang ki */
  }
```

49. példa

Ez a példa az EMS kezelését mutatja be az IDE-ben.
Ne felejtsük el, hogy az EMS-t még az IDE futtatása elõtt le kell foglani.

```
/* 49. példa */

#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <process.h>
#include <string.h>
#include <sys\stat.h>
#include <io.h>

void interrupt  (*ems)();
static char                buff[80];

main()
        {
        FILE           *fp;              // File pointer
        union   REGS r;                  // Processzor regiszterek
        long           seg;              // Szegnes mutató
        char           *cp;
        int            hnd,pages,pages2;
        int            i,k;
        char           ch;

        ems=getvect(0x67);   // A vektor címe
        seg=FP_SEG(ems);              // A szegmens megszerzése
        cp=MK_FP(seg,10);             // Mutatót csinálunk
        if(strncmp("EMMXXXX0",cp,8))    return(1);
```

Forrás: http://www.doksi.hu

```
                                        // Ha nincs EMS
r.h.ah=0x41;              // A keret szegmens címének lekérd
int86(0x67,&r,&r);
seg=r.x.bx;                            // BX-ben volt
printf("\n\nSzegmens cim=%X\n",seg);
r.h.ah=0x42;              // Szabad lapok száma
int86(0x67,&r,&r);
pages=r.x.bx;             // BX-ben volt
pages2=-1;
printf("\nSzabad lapok száma=%i",pages);
printf("\nHányat foglalhatok le ? : ");
if (pages)                             // ha van szabad lap
  {
  while(pages2>pages || pages2<0)
    {
    scanf("%s",buff);
    if (buff[0]==(char)0) pages2=0;    // ha nem írt be semmit
    sscanf(&buff[0],"%i",&pages2);
    }
  if (pages2)             // ha 0, akkor nem kell lefoglani
    {
    r.h.ah=0x43;                       // EMS lapok lefoglalása
    r.x.bx=pages2;
    int86(0x67,&r,&r);
    hnd=r.x.dx;           // Az un handler a DX-ben
    printf("\nA handler : %i",hnd);
    }
  else hnd=-1;
  }
else     hnd=-1;
r.h.ah=0x42;              // Nézz feljebb
int86(0x67,&r,&r);
pages=r.x.bx;
printf("\nSzabad lapok száma=%i",pages);
fp=fopen("EMS_HND.HND","w"); // A handler és a
                             //lefoglalt lapok, valamint
                             //a keretsegmens címe
fprintf(fp,"%i %i %X",hnd,pages2,seg);
                             // értékének kiírása az EMS_HND.HND
fclose(fp);                  // nevű file- ba
spawnl(P_WAIT,"bc.exe",NULL);
chmod("EMS_HND.HND",S_IREAD|S_IWRITE);        // A file kitörlése
unlink("EMS_HND.HND");                        // Kitörölni !!!
printf("\nFelszabadítok, renben ? (i/n)");
do{
  ch=getch();
  }
while (ch!='i' && ch !='n' && ch!=(char)27 && ch!=(char)13 );
if ( (ch=='i' || ch==(char)13) && pages>0)
  {                                    // ha nem volt szabad lap akkor ne tedd
```

Forrás: http://www.doksi.hu

```
r.h.ah=0x45;            // Felszabadítok
r.x.dx=hnd;
int86(0x67,&r,&r);
r.h.ah=0x42;            // Nézz feljebb
int86(0x67,&r,&r);
pages=r.x.bx;
printf("\nSzabad lapok száma=%i\n",pages);
}
}
```

Az IDE-n belül ezt egy #define #if #else #endif szerkezettel kezelhetjük le. A következõ módon.

1. Definiáljunk egy IDE pszeudó változót.

#define IDE 1 vagy #define IDE 0

2. Ha az IDE értéke 1, akkor keresse a file-t, ha az érték 0, akkor kezelje az EMS-t hagyományos módon.