C programozási nyelv
munkapéldány

Dr. Schuster György

2014. szeptember 18.

Tartalomjegyzék

1. Bevezetés

A C programozási nyelvet Dennis Ritchie és Ken Thompson fejlesztette ki az 1970-es évek elején. Céljuk az volt, hogy a UNIX operációs rendszert könnyen hordozhatóvá tegyék különböző számítógépek között.

Mivel az ”ős” UNIX assembly nyelven készült a hordozhatóság nehézkes volt, ezért a fejlesztők olyan magasszintű programozási nyelvet kerestek, amely egyszerű, elég hatékony rendszer és hardver programozásra és ezek mellett alkalmas a operációs rendszerek írására.

Mivel olyan nyelvet, amely mindezen követelményeket kielégítette nem találtak, ezért Ritchie Thompson segítségével létrehozta a C programozási nyelvet. Ezzel a UNIX operációs rendszer lett az első olyan operációs rendszer, amely magjának jeletős része magasszintű programozási rendszerben íródott.

1978-ban megjelent a Dennis Ritchie és Brian Kernigham által írt *A C programozási nyelv* című könyv, amelyben az úgynevezett K& R C nyelvet írják le. A fordító programok nagy többsége ezt a C verziót támogatja.

1983-ban az ANSI (American National Standards Institute) létrehozott egy bizottságot, amelynek az volt a célja, hogy szabványosítsa és szabványos módon bővítse a C-t. Ezt a C verziót minden mai C fordító támogatja.

A C nyelv továbbfejlesztése a C++ nyelv, mely a C obejktum orientált ”kiterjesztése”.

Manapság UNIX és UNIX-szerű operációs rendszerek magját ma már szinte kizárólag C-ben írják. A beágyazott rendszerek firmware-i nagyrészt ebben készülnek. Így ez a programozási nyelv szinte mindenütt jelen van.

Ez a jegyzet az ANSI C-vel foglalkozik. A példákban alkalmazott fordítóprogram a gcc. Ez a jegyzet nem tér ki a nyelv nem szabványos könyvtáraira, illetve azokra a könyvtárakra, amelyek nem közvetlenül szükségesek az alap működés megértéséhez.

2. Szintaktikai elemek

Minden programozási nyelv szintaktikai[1] elemekből áll. Ezek általában jellemzők a nyelvre. A C nyelv a következő szintaktikai elemeket tartalmazza:

- változók, változók módosítói és mutatók,
- operátorok,
- utasítások,
- függvények,
- összetett adatszerkezetek,
- modulok.

A jegyzetben ezeket a fenti sorrend szerint ismertetjük[2].

2.1. Kommentek

Mielőtt a tényleges szintaktikai elemekbe belevágnánk beszéljünk egy pár szót a megjegyzésekről, vagy kommentektől.

A C nyelv egy nagyon könnyen írható, de elég nehezen olvasható nyelv. Ezért programunkat célszerű megjegyzésekkel ellátni, amelyek a későbbiekben segíthetik a programlistában a tájékozódást.

Az eredeti C nyelvben a kommentek `/* */` párok közé kerülhettek. Ezek a fajta megjegyzések több soron keresztül is tarthattak és a fordító program teljesen figyelmen kívül hagyta őket.

Az ANSI C már megengedi az un. egysoros kommentet ez a `//` jellel kezdődik és a sor végéig tart. Tehát csak egy sorra vonatkozik.

2.2. Változók

A programok általában valamilyen adatokon hajtanak végre műveleteket, illetve ezen műveletek eredményeit is változókba helyezik. A legtöbb úgynevezett fordított[3] programozási nyelvben a változóknak típusa van.

[1] Szintaktikai - helyesírási, vagy nyelvi elemek.

[2] Sajnos ez azt is jelenti, hogy néha olyan programrészleteket vagy programokat kell áttekintenünk, amelyek elemei még nem kerülhettek ismertetésre

[3] A programozási nyelvek lehetnek un. interpreteres nyelvek, mint pl. a shell szkript, ahol a futtató program sorról sorra fordítja és értelmezi az utasításokat a futás során. Továbbá lehetnek fordított (compiled) nyelvek, ahol a programot a futtatás előtt egy futtatható kóddá fordítják le, majd ebben a formában is tárolják.

2.2.1. Változók típusai

Az ANSI C 5 öt alaptípust ismer, ezek:

- egész típus: csak egészek tárolására szolgál, mérete implementáció függő (általában 16 - 32 bit), az alkalmazott számábrázolás kettes komplemens. A típus az un. egész jellegű változók csoportjába tartozik, deklarációs megnevezése `int`, számábrázolási tartománya: 16 bit esetén -32768..32767, 32 bit esetén -2147483648..2147483647.
- karakter típus: alapvetően karakterek tárolására szolgál, de kis egész számokat is tárolhatunk bennük, mérete 8 bites, az egész jellegű változók csoportjába tartozik, deklarációs elnevezése `char`,
- lebegőpontos típus: ANSI szabvány szerinti egyszeres pontosságú lebegőpontos számok tárolására szolgál, mérete 4 bájt, a lebegőpontos jellegű változók csoportjába tartozik, deklarációs elnevezése `float`, számábrázolási tartománya: $\pm 3.4 * 10^{\pm 38}$,
- ”dupla” lebegőpontos típus: ANSI szabvány szerinti kétszeres pontosságú lebegőpontos számok tárolására szolgál, mérete 8 bájt, a lebegőpontos jellegű változók csoportjába tartozik, deklarációs elnevezése `double`, számábrázolási tartománya: $\pm 1.7 * 10^{\pm 308}$,
- felsorolás típus: deklarációs elnevezése `enum`

2.2.2. Változók deklarálása

Ahhoz, hogy bármilyen változót a programunkban használni tudjunk, azt előbb deklarálni kell. Ez fizikailag annyit jelent, hogy a változó számára helyet foglalunk a számítógép memóriájában és az adott tárterületnek megadjuk a típusát is. A deklarációnak mindig meg kell előznie a felhasználás helyét. Ez a C nyelvnél azt jelenti, hogy a deklarációnak meg kell előznie a működő program sorokat, akár minden függvényen kívül, akár egy adott függvényen belül deklarálunk változót[4].

A változó neve csak betűvel és aláhúzás karakterrel (_) kezdődhet. A név tartalmazhat kisbetűt, nagybetűt, aláhúzást és számot. A kis és nagybetűk között a C különbséget tesz[5]. A változó neve nem lehet foglalt kifejezés.

Változó deklarálása formálisan a következőképpen történik:

```
típus  változóneve;
```

Az ANSI szabvány megengedi, egyetlen kifejezésben egyszerre, több (elvileg akárhány) azonos típusú változót is deklaráljunk:

```
típus  változóneve1,változóneve2,változóneve3;
```

[4] A deklaráció helye a későbbiekben még szóba kerül.
[5] Ez az un. Case Sensitive tulajdonság.

A következő példa egy egész változó függvényen belüli deklarációját mutatja be[6].

```
1  #include <stdio.h>
2  int main(void)
3    {
4    int i;
5    i=5;
6    return i;
7    }
```

Az `i` nevű változó deklarációja a `main` függvényen belül a 4. sorban történik.

Tehát:

```
int     i;    // egész változó deklarációja
char    c;    // karakter változó deklarációja
float   f;    // lebegőpontos változó deklarációja
double  d;    // dupla pontosságú lebegőpontos változó deklarációja
```

2.2.3. Enumerity

Az enumerity az úgynevezett felsorolás típus. Ezt akkor célszerű alkalmazni, ha valamilyen növekvő számsor egyes eleminek nevet szeretnénk adni. Ezek a számok csak egészek lehetnek. Vegyük példának a hét napjait:

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

Ekkor a `hetfo` értéket 0, a kedd értéke 1, stb. lesz.

Ha valamilyen okból a lista számait változtatni szeretnénk, akkor ezt a megtehetjük.

```
enum NAPOK{hetfo=1,kedd, szerda,csutortok,pentek,szombat=0,vasarnap=0};
```

Ebben a példában a `hetfo` értéke 1, a `kedd` értéke 2, a `szombat` és a `vasarnap` értéke 0 lesz.

2.2.4. Típus módosítók

Más programozási nyelvekben járatos olvasóknak valószínűleg feltűnt, hogy a C nyelvben kevés változó típus van[7].

[6] Ez a program működőképes, lefordítható és futtatható.
[7] A sztring típus is hiányzik, de arról majd később beszélünk.

Ezt az ellentmondást a C készítői úgy oldották fel, hogy az alaptípusokat úgynevezett módosító prefixumokkal módosítani lehet.

Az **`int`** módosítói:

`unsigned` ebben az esetben az `int` típus ”előjeltelen” lesz. Tehát a típus számábrázolási tartománya a megvalósítástól függően 16 biten 0..65535, illetve 32 biten 0..4294967295. A deklaráció a következő:

```
unsigned int ui;
```

`long` az így deklarált változó mérete nem lehet rövidebb, mint az eredeti `int` típus mérete. Ezért például 32 bites gcc esetén a `long` szintén 32 bites. 16 bites gcc esetén a `long` 32 bit szélességű.

`short` az így deklarált változó mérete nem lehet hosszabb, mint az eredeti `int` típus mérete. Ezért például 32 bites gcc esetén a `short` 16 bites. 16 bites gcc esetén a `short` 16 bit szélességű.

`long long` ez a módosítás nem szabványos a gcc fordítók azonban ismerik. Ez egy 64 bit szélességű változót ír elő.

Az `int` prefixumainál nem szükséges kiírni deklarációkor az `int` kulcsszót. Vagyis:

`unsigned int ui;` kifejezésnek teljesen megfelel a `unsigned ui;`[8].

A C++ nem is engedi meg ilyen esetekben az `int` használatát.

A prefixumok keverhetők, tehát nyugodtan írhatjuk, hogy

```
unsigned long ui;
```

Természetesen értelemszerűen. A `long short auu;` deklaráció nem helyes.

A **`char`** módosítói:

`unsigned` a karakter típust előjeltelen 8 bites egészre változtatja, számábrázolási tartománya 0..255,

`signed` a karakter típust előjeles 8 bites egészre változtatja, számábrázolási tartománya -128..127.

A karakter típus esetén a `char` kulcsszó nem hagyható el. Ennek az az oka, hogy ha csak az `unsigned` prefixumot használnánk, akkor a fordító program ez `unsigned int`-nek értelmezné. A `char` kötelező a `signed` esetben is.

[8] Ennek az az oka, hogy a Kernigham és Ritchie által definiált ”ős” C nyelv csak az `int` típust ismerte, így fölösleges volt a típus kiírása.

A karakter típus alapértelmezése a fordító program beállításától függ, és ez általában előjeles. Ha azonban egy 8 bites előjeles egészre van szükségünk és a beállítás esetleg a megszokottól eltérhet célszerű használni a `signed` kulcsszót[9].

A **`float`** típusnak nincs prefixuma.

A **`double`** egyetlen prefixummal rendelkezik.

`long` egyes megvalósításokban 10, másokban, például a gcc 12 bájt méretű lebegőpontos számábrázolást ír elő.

2.2.5. Futásidejű konstans `const`

A `const` jelző bármilyen típus és módosított típus esetén használható csak akkor célszerű a használata, ha a változó értéket kap a deklarációnál. Példa:

```
const double PI=3.1415;
```

A fordító program minden módosítási kísérletünkre hibaüzenetet küld.

2.2.6. ”Hagyd békén” `volatile`

A fordító programok valamilyen szempontból optimalizálni próbálnak[10]. Ha fölöslegesnek tartott változó területeket találnak a kódban, azt egyszerűen kihagyják. Előfordul olyan eset, hogy mi ezt nem szeretnénk. Ekkor használjuk a `volatile` módosítót. Példa:

```
volatile int a;
```

Ez főleg moduláris programozás esetén lehet fontos (lásd 56. oldal.).

2.2.7. Típus definiálás

A C lehetővé teszi, hogy új típusokat definiáljunk valamely alaptípus[11] segítségével.

Ha például szeretnénk egy előjeltelen 8 bites egész változó típust mondjuk `u8` néven, akkor a következőt tehetjük:

```
typedef unsigned char u8;
```

[9] Igen használjuk, nem PC-re írt programoknál, de amikor mikrokontrollereket használunk, ahol olyan kevés a memória, hogy minden bit számít sokszor tárolunk adatot `char` típusban is.

[10] Ez egy kicsit korai, de itt a helye (sajnos).

[11] - és összetett adatszerkezet, illetve mutató -

A `typedef` kulcsszó mondja meg a fordítónak, hogy ha valahol egy `u8` típusú változó deklarációt talál, akkor az tulajdonképpen egy `unsigned char` típusú deklaráció[12].

Ezután egy változó deklarálása a következő módon történik:

```
u8 a;
```

2.2.8. Változók inicializálása

A változóknak adhatunk közvetlenül értéket.

Egész jellegű változók értékadása[13]:

decimális szám `i=65;`

oktális szám `i=0105;`

hexadecimális szám `i=0x41;`

bináris szám `i=0b01000101;`

karakter kód `i='A';`

Lebegőpontos változók esetén egyszerűen értéket adunk a változónak. Néhány régebbi C fordító megkövetelte, hogy `double` és `long double` változó esetén a szám után egy `L` karaktert kellett írni.

2.3. Operátorok

Az operátorok olyan szintaktikai elemek, amelyek egyszerű műveleteket írnak elő, vagy kijelöléseket végeznek. A C nyelvben az operátoroknak hét csoportja van, ezek:

1. elválasztó operátorok,
2. értékadó,
3. aritmetikai operátortok,
4. relációs operátorok,
5. logikai operátorok,
6. bit operátorok,
7. egyéb operátorok.

[12] Akkor mire is jó nekünk ez a típus definiálás? A válasz annyi, hogy a program jobban olvasható, követhető.
[13] Ez az eredetileg egész jellegű változókat ezek módosított és az ezekből definiált típusokat jelenti.

2.3.1. Elválasztó operátorok

`{}, (), [], ,, ;`

`{}` a kapcsos zárójel pár egy logikai programrészletet zár magába. Ez lehet valamilyen szerkezet, pl. `switch-case`, lehet ciklus törzs, elágazás esetén igaz, vagy hamis ág és lehet függvény törzs.

`()` a megszokott aritmetikai jelentéssel bíró zárójel precedenciát, vagyis művelet végrehajtási sorrendet változtat meg. Például:

`a=b+c*d;` nem azonos a `a=(b+c)*d;` kifejezéssel.

`[]` tömb indexelő operátor. Lásd tömbök 42. oldal.

`,` vessző operátor. Részkifejezések elválasztására szolgál, de a kifejezéseket nem zárja le. Például:

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

Ezt az operátort a makróknál fogjuk használni. Lásd 36. oldal.

`;` lezáró operátor egy adott kifejezést zár le.

2.3.2. Értékadó operátorok

Az értékadó operátorok enyhén eltérnek más programozási nyelvek értékadó operátoraitól, nevezetesen tetszőleges típusú változó értékét tetszőleges típusú változónak adhatjuk át.

`=`, **`?:`**, **`rekurzio`**

`=` egyszerű értékadás operátor. A baloldalon álló kifejezésnek adja át a jobb oldalon lévő kifejezés értékét. Példa:

```
a=3;
```

FIGYELEM ez a kifejezés nem helyes: `3=a;`

`?:` a feltételes értékadás operátora. Formálisan három kifejezésből áll a következő módon:

```
kifejezés1?kifejezés2:kifejezés3
```

ha a `kifejezés1` igaz, akkor a `kifejezés2`, ha hamis a `kifejezés3` hajtódik végre. Például:

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

Vagyis, ha `a` nagyobb, mint `b`, akkor `c` értéke legyen az `a`-ban tárolt érték. Ellenkező esetben `c` kapja meg `b` értékét.

`rekurzio` ez az operátor család minden olyan binér[14] operátoron alkalmazható, ahol valamilyen érték keletkezik. A célja az, hogy az adott kifejezés rövidebb legyen. Példa:

[14] Két operandusú operátor.

`a=a+b;` kifejezés helyett írható a `a+=b;` kifejezés.

FIGYELEM gyakori a következő hiba:

`a=b-a;` kifejezés helyett a `a-=b;` kifejezést írják.

Ez hiba, mert az `a-=b;` az `a=a-b;` -nek felel meg és ne feledjük, hogy a kivonás nem kommutatív.

Az operátor csoport tipikusan az aritmetikai és bit operátoroknál használatos. Bizonyos megkötésekkel a logikai operátorok esetén is lehet használni a rekurzív formát, de ez igen ritka.

2.3.3. Aritmetikai operátorok

Az aritmetikai operátoroknak van egy érdekes tulajdonsága. Ha aritmetikai műveletet hajtunk végre két különböző típusú változón, akkor a művelet a két típus közül abban típusban hajtódik végre, amelyben az eredmény pontosabb[15]. Tehát ha egy `int` és egy `double` típusú változót adunk össze, akkor a művelet `double` típusban történik.

+, **-**, *****, **/**, **%**, **-**, **++**, **--**

+ az összeadás operátora,

- a kivonás operátora (binér operátor),

***** a szorzás operátora,

/ az osztás operátora. Ez az operátor kíván némi figyelmet, mert ha két egész számot osztunk, akkor az eredmény is egész lesz. Példa: a következő programrészletben két egész változó hányadosát adjuk át egy lebegőpontos változónak.

```
int a,b;
double d;

a=8;
b=5;
d=a/b;
```

Első pillanatra az ember azt gondolná, hogy a `d` változó értéke 1.6 lesz, **de nem**. Mert az `a/b` művelet egész típusban hajtódik végre, aminek az eredménye 1, majd ez az 1 egész érték konvertálódik át lebegőpontos 1 -é és kerül be a `d` változóba.

% a maradékképzés operátora. Csak egész jellegű változókra alkalmazható. Lásd a következő programrészletet!

[15] Szakszerűen a művelet a magasabb tárolási osztályban hajtódik végre.

```
int a,b,c;

a=8;
b=5;
c=a%b;
```

A `c` változó értéke 3 lesz, mert az 5 egyszer van meg a 8-ban és maradt 3.

- egyoperandusú mínusz Az operátor előjelváltást ír elő. Példa:

```
b=-a;
```

Vagyis `b` értéke `a` értékének a -1 szerese lesz.

++ inkrmentáló operátor[16]. Az operátor egy változó értékét egyel megnöveli. Az operátor lehet posztfix, vagyis állhat a változó neve után, vagy lehet prefix, vagyis állhat a változó neve előtt.

`a++;` Ez az operátor posztfix alkalmazása

`++a;` Ez az operátor prefix alkalmazása

A változó szempontjából a két mód azonos. Azonban ha az operátort összetett kifejezésben használjuk a kifejezés teljes egészére az eredmény már különböző. Vegyük példának a következő két programrészletet!

```
int a,b;

a=5;
b=a++;
```

```
int a,b;

a=5;
b=++a;
```

Az egyértelmű, hogy mindkét programrészlet lefutása után az `a` értéke 6 lesz. A kérdés az, hogy a `b` értéke mekkora

A baloldali programrészletben a `b` értéke 5 lett. A jobboldaliban pedig 6.

A magyarázat a `++` operátor un. kiértékelési irányából indul ki.

Nagyon egyszerűen magyarázva a bal oszlopban lévő programrészletben a `b=a++;` sora a következőképpen működik. A program megtalálja az `=` operátort ez azt jelenti neki, hogy a bal oldali változónak át kell adnia a jobb oldalon lévő kifejezés értékét. Ez a kifejezés most az `a`, tehát a `b` értéke 5 lesz. Viszont a következő lépésben a program megtalálja a `++` operátort, ami `a` változóra vonatkozik, tehát inkrementálja azt.

A jobb oszlopban lévő programrészlet először a `++` operátort találja meg. Ezután keresi azt a változót, amire ez vonatkozhat. A bal oldalán az `=` van, tehát a jobb oldalon keres. Itt megtalálja az `a` változót ezt inkrementálja és ennek az új értéket adja át a `b`-nek.

-- dekrementáló operátor. Az operátor egyel csökkenti a kérdéses változó értékét. Ezen kívül működése mindenben megegyezik a `++` működésével.

[16] az inkrementálás növelést jelent.

2.3.4. Relációs operátorok

A relációs operátorok a klasszikus összehasonlító műveleteket jelölik ki. Ezeket az operátorokat az elágazás (19. oldal) és a ciklus utasításokban (22. oldal) , illetőleg a feltételes értékadás operátor esetén használjuk.

<, **<=**, **==**, **!=**, **>=**, **>**

< kisebb,

<= kisebb, egyenlő,

== egyenlő. Eltérően más programozási nyelvekben ez az operátor a relációs operátor. Tipikus hiba, hogy a programozó[17] csak egyet írnak. Erre az esetre a C fordító egy figyelmeztetést küld, de nem veszi hibának.

!= nem egyenlő,

>= nagyobb, egyenlő,

> nagyobb.

2.3.5. Logikai operátorok

A relációs kifejezések alapvetően relációs kifejezések összefűzésére szolgálnak. Érdekes tulajdonságuk az, hogy csak addig dolgozza fel a program az összetett kifejezéseket, amíg az egész kifejezés értéke el nem dől.

Később erre mutatunk példát.

&&, **||**, **!**

&& és operátor. Akkor igaz, ha az operátor mindkét oldalán álló kifejezés igaz. Például:

```
a>=5 && a<=10
```

vagyis a kifejezés akkor igaz, ha `a` értéke 5 és 10 zárt intervallumba esik.

Ha három kifejezést fűzünk össze, akkor a kifejezés addig megy míg van esély arra, hogy az igaz legyen.

```
kif1 && kif2 && kif3
```

A `kif2` kifejezés csak akkor hajtódik végre, ha a `kif1` igaz volt. A `kif3`-ra csak akkor kerül a vezérlés, ha a `kif1` és `kif2` igaz volt.

[17] ...palánták...

|| vagy operátor. Akkor igaz, ha az operátor valamelyik, vagy mindkét oldalán lévő kifejezés igaz. Példa:

```
a<=5 || a>=10
```

Ha az a változó értéke 5, vagy annál kisebb, akkor a második kifejezésre nem kerül rá a vezérlés, mert az eredmény már biztos. Ha a nagyobb, mint 5, akkor a jobboldali relációs művelet is elvégzésre kerül.

! nem operátor. Az igaz kifejezés értékét hamisra, a hamist igazra változtatja.

MEGJEGYZÉS: a C nyelv igaznak vesz minden olyan értéket, amely nem 0 és hamisnak, amely 0 (lásd a példát a 20. oldalon).

2.3.6. Bit operátorok

A bit operátorok bitmanipulációhoz használatosak. Csak és kizárólag egész jellegű típusokon lehet használni őket.

~, **&**, **|**, **^**, **<<**, **>>**

Az operátorok magyarázatához bemutatott példáknál a következő változókat használjuk inicializálva.

```
char a=0b00110101;
char b=0b00001111;
char c;
```

~ egyes komplemens képző operátor. A művelet az adott pozíciókon lévő bitek közt hajtódik végre. Példa:

```
c=~a;
```

A c változó értéke a művelet után 11001010_b.

& bitenkénti és kapcsolat operátora. A művelet az adott pozíciókon lévő bitek közt hajtódik végre. Példa:

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

A c változó értéke a művelet után 00000101_b.

| bitenkénti vagy kapcsolat operátora. A művelet az adott pozíciókon lévő bitek közt hajtódik végre. Példa:

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

A c változó értéke a művelet után 00111111_b.

`^` bitenkénti kizáró vagy kapcsolat operátora. A művelet az adott pozíciókon lévő bitek közt hajtódik végre. Példa:

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

A `c` változó értéke a művelet után 00111010_b.

`<<` balra eltolás operátora. Az adott változó értékét tolja el balra (az MSB felé) adott számban. A LSB-be 0 értékű bitek kerülnek. A kilépő bitek elvesznek. Példa:

```
c=a<<2;
```

A `c` változó értéke a művelet után 11010100_b.

`>>` jobbra eltolás operátora. Az adott változó értékét tolja el jobbra (az LSB felé) adott számban. A MSB-be 0 értékű bitek kerülnek. A kilépő bitek elvesznek. Példa:

```
c=a>>2;
```

A `c` változó értéke a művelet után 00001101_b.

2.3.7. Egyéb operátorok

Ezek az operátorok egyetlen más csoportba sem sorolhatók. Nagy részüket nem is itt tárgyaljuk, hanem azokon a helyeken, ahol már a szükséges szintaktikai elemek rendelkezésre állnak.

`(cast)`, **`sizof()`**, **`&`**, **`*`**, **`.`**, **`->`**

`(cast)` kikényszerített típuskonverzió. Használata a következő programrészleten látható:

```
int a=6,b=5;
double d;

d=(double)a/b;
```

Ha a `(double)` konverzió nem lenne előírva az `a` változóra, akkor a `d` értéke a művelet után 1 lenne. A `(double)a` hatására az `a` változó a műveletben `double` típusú változóként szerepel. Ekkor a művelet `double` típusban hajtódik végre és ezért `d` értéke 1.2 lesz.

`sizeof()` méret operátor Az operátor egy változó, vagy egy típus méretét adja meg bájt méretben. Példa:

```
a=sizeof(int);
```
vagy
```
a=sizeof(b);
```

Szerepe az `int` méretének meghatározásánál, illetőleg a tömböknél (42. oldal) és a struktúráknál (47. oldal) lényeges.

`&` címképző operátor. Részletesen tárgyaljuk a mutatók fejezetben (39. oldal).

`*` indirekciós operátor. Részletesen tárgyaljuk a mutatók fejezetben (39. oldal).

`.` mezőhozzáférés operátor. Részletesen tárgyaljuk a struktúrák fejezetben (47. oldal).

`->` indirekt mezőhozzáférés operátor. Részletesen tárgyaljuk a struktúrák fejezetben (47. oldal).

2.3.8. Operátorok precedenciája

Az operátorok precedenciája fogalom azt jelenti, hogy az operátorok végrehajtási sorrendje milyen, ha többszörös alkalmazás esetén nem teszünk zárójelet.

A kiértékelési iránya azt mondja meg, hogy az alkalmazott operátor a kifejezés mely részére vonatkozik. Mint láthatjuk az unáris operátorok kiértékelési iránya a jobbról balra. Ez azt jelenti, hogy a kérdéses művelet az operátorhoz viszonyítva jobboldalt található operanduson hajtódik végre.

Például: a `b=~a;` kifejezésben a `~` operátor a hozzá képest jobbra lévő `a` változón fejti ki hatását.

legerősebb	{} **() [] -> .**	balról jobbra
	! ++ -- + - (cast) * & sizeof()	jobbról balra unáris operátorok
	*** / %**	balról jobbra bináris operátorok
	+ -	balról jobbra
	>> <<	balról jobbra
	> < < = > =	balról jobbra
	== !=	balról jobbra
	&	balról jobbra
	^	balról jobbra
	|	balról jobbra
	&&	balról jobbra
	||	balról jobbra
	?:	jobbról balra
	= += -= ...	jobbról balra és a többi hasonló rekurzív
leggyengébb	**,**	balról jobbra

2.4. Futásidejű utasítások

2.4.1. `if`

Az `if` utasítás feltételes elágazást tesz lehetővé. Ha az utasítás argumentumában szereplő kifejezés igaz a program az un. igaz ágat végrehajtja, ha nem az igaz ág a program futása során figyelmen kívül lesz hagyva.

Az igaz ág lehet egyetlen kifejezés, amely az első `;` operátorig tart, vagy lehet egy logikai blokk, amelyet a {} operátor pár zár közre.

A {} megoldást akkor kell használni, ha több `;`-vel lezárt kifejezést szeretnénk az igaz ágba elhelyezni. Természetesen, ha a program olvashatósága megkívánja akkor is alkalmazhatjuk, ha csak egyetlen ilyen kifejezésünk van.

Példa az egyetlen kifejezésre:

```
if(a>5) printf("Nagyobb");
```

Ha `a` értéke nagyobb, mint 5, akkor kiírja, hogy `Nagyobb`[18].

Példa a több kifejezést tartalmazó igaz ágra:

```
if(a>5)
  {
  b=a;
  printf("Nagyobb");
  }
```

Láthatjuk, hogy itt már két kifejezés szerepel, ezért szükséges a kapcsos zárójelpár alkalmazása.

FIGYELEM ne tegyél `;` a záró kapcsos zárójel mögé!

Az `if` argumentumába általában relációs kifejezés kerül, de nem okvetlenül. A kiértékelés mindenképpen relációs jellegű. Ha egy kifejezés értéke 0, akkor az hamisnak minősül, ha nem igaznak.

Tehát egy változó 0 értékének vizsgálata C programban általában így néz ki:

```
if(!a)
```

Ha `a` értéke 0, akkor hamis a `!` operátor miatt ez igaz lesz, tehát a program belép az igaz ágba.

2.4.2. `if-else`

Az `if` használata csak azt tette lehetővé, hogy a program egy igaz ágat hajtson végre. Az `else` alkalmazása az `if` után lehetővé teszi a hamis ág alkalmazását.

Az `else` ág is egyetlen `;`-vel lezárt kifejezésből, vagy egy {} által határolt logikai blokkból áll éppúgy, mint az `if` esetén.

FIGYELEM az `else` utasításnak közvetlenül kell az `if` utasítást követnie. Nem lehet közöttük semmilyen kifejezés.

Példa:

```
if(a>5)
  {
  b=a;
```

[18] a `printf` függvénnyel most ne foglalkozzunk, ezt később tárgyaljuk részletesen.

```
    printf("Nagyobb");
    }
  else
    {
    b=-a;
    printf("Nem nagyobb");
    }
```

Ha tehát az `a` változó értéke nagyobb, mint 5, akkor `b` felveszi `a` értékét és a `printf` kiírja, hogy `Nagyobb`. Ha nem teljesül a feltétel, a program az `else`-re ugrik és `b` felveszi `a` mínusz egyszeresét és a `printf` kiírja, hogy `Nem nagyobb`.

2.4.3. `switch-case`

Gyakran előforduló probléma, hogy egy változó értékétől függően a program különböző feladatokat lát el. Ez a probléma megoldható ugyan az előzőekben megtanult `if-else` szerkezetekkel, de nem kényelmes és elég nehezen követhető. Az ilyen ”szétugrási” feladatokban a kérdéses változó általában egész jellegű.

Ezt a feladatot látja el a `switch-case` szerkezet.

A következő programrészlet bemutatja a `switch-case` felépítését

```
1    switch(a)
2       {
3       case 1: printf("Egy");
4               break;
5       case 2: printf("Ketto");
6               break;
7       case 3: printf("Harom");
8               break;
9       default:printf("Egyik sem");
10      }
```

Az 1. sorban a `switch` argumentumába kerül az `a` változó, ami az elágazás alapja.

Ha az `a` értéke 1, akkor a program a 3. sorra ugrik. A `printf` kiírja `Egy`, majd a program a 4. sorban található `break` utasításra ugrik és elhagyja a szerkezetet.

Az `a` 2 esetén a program az 5. sorra ugrik. 3 esetén a 7. sorra kerül feldolgozásra.

Ha `a` értéke se nem 1, 2, vagy 3, akkor a program a `default` során folyatódik.

A `default` nem kötelező. Ha nincs `default`, akkor ilyen esetben a program kilép a `switch-case` szerkezetből.

Nézzük a következő esetet!

```
1      switch(a)
2         {
3         case 1: printf("Egy");
4         case 2: printf("Ketto");
5         case 3: printf("Harom");
6                 break;
7         default:printf("Egyik sem");
8         }
```

Látható, hogy az 1 és a 2 esetből hiányoznak a `break` utasítások. Ekkor `a` 1 esetén kiírásra kerül `EgyKettoHarom`, vagyis az adott `case` után a program nem hagyja el a szerkezetet, hanem fut tovább egészen addig, amíg nem talál `break`-et, vagy nem ér véget a szerkezet.

Tehát ha `a` értéke 2, akkor a képernyőre a `KettoHarom` kiírás kerül.

2.4.4. `for`

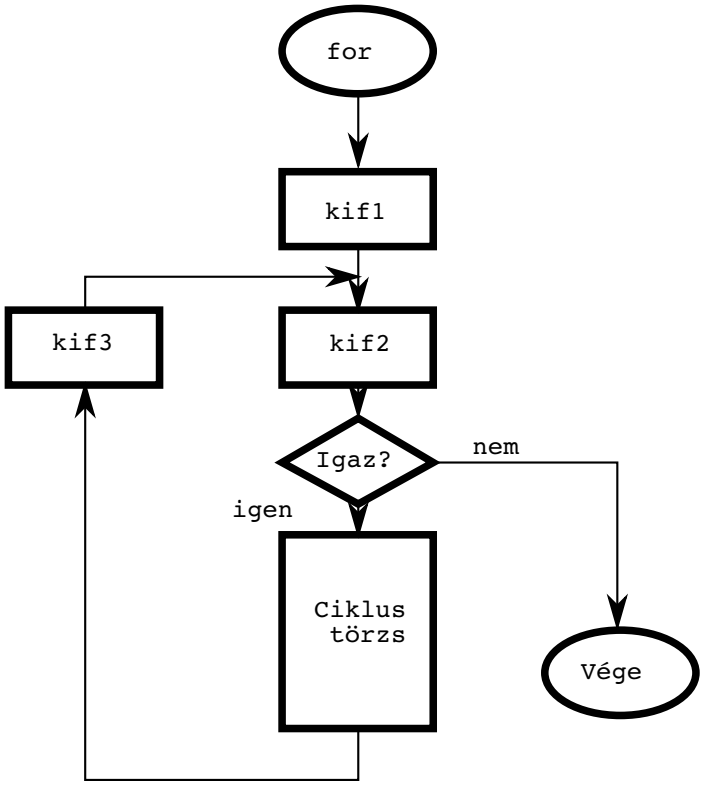
A `for` egy általános elöltesztelő ciklust létrehozó utasítás. Argumentuma három kifejezésből áll.

A ciklustörzs hasonlóan az `if` utasításhoz vagy egy `;`-vel lezárt kifejezés, vagy egy {} operátorok közé zárt logikai blokk.

A `for` formális alakja:

```
for(kif1;kif2;kif3) ciklustörzs
```

A működését a következő folyamatábra szemlélteti.



1. ábra: A `for` működése

A folyamatábrán látható, hogy az `kif1` kifejezés csak egyszer hajtódik végre, ezért jól használható a ciklus inicializálására

A második kifejezés (`kif2`) kiértékelése relációs jellegű és a ciklustörzs előtt található. Tehát a `for`-ral létrehozott ciklus elöltesztelő. Ha a `kif2` igaz a ciklus törzse lefut, ha hamis, akkor kilépünk a ciklusból.

A ciklus törzs után a harmadik kifejezés hajtódik végre.

A következő program 1-től 10-ig összeadja a számokat.

```
1     #include <stdio.h>
2     int main(void)
3        {
4        int i,j;
5        j=0;
6        for(i=1;i<=10;i++)
7           {
8           j=j+i;
9           }
10       printf("%d",j);
11       return 0;
12       }
```

A 6. sorban látható a `for` utasítás. Az első kifejezés az `i` változónak adja meg a kezdő értéket. A második kifejezés egy összehasonlítás. Ez addig igaz amíg az `i` változó értéke túl nem lépi az 10-et. A harmadik kifejezés az `i` értékét növeli.

A 8. sorban képezzük az összeget.

Az 10. sorban íratjuk ki az eredményt (lásd részletesen 64. oldalon).

Láthatjuk, hogy a ciklustörzshöz a {} megoldást választottuk, pedig itt csak egy kifejezés szerepel. Az ok így jobban olvasható a program.

Ez a program csak C guruknak szól:

```
1     #include <stdio.h>
2     int main(void)
3        {
4        int i,j;
5        for(i=1,j=0;i<=10;i++) j+=i;
6        printf("%d",j);
7        return 0;
8        }
```

Azt láthatjuk, hogy ez a program jóval tömörebb. A tulajdonképpeni program csak az ötödik sor. Kérdés az, hogy melyik program jobb. Talán a második megoldás egy hajszállal gyorsabb a += használata miatt, de - és ez nagyon lényeges - az első programlista lényegesen jobban olvasható.

Ez már csak a teljesen elvadult guruknak:

```
1     #include <stdio.h>
2     int main(void)
3        {
4        int i,j;
5        for(i=1,j=0;i<=10;j=j+i++);
6        printf("%d",j);
7        return 0;
8        }
```

Na ez a lista már tényleg szörnyű. A működése gyakorlott programozónak triviális, azonban kezdőnek csapdákat rejt magában.

Amit itt meg kell magyarázni az a `j=j+i++;`. Az `j` változóba a `j` előző értéke és az `i` aktuális értékének összege kerül, majd ezután inkrementálódik az `i`.

Az lássuk be, hogy a ciklustörzs üres, az egész lényegi rész tulajdonképpen a 3. kifejezésben történik.

A `for` utasítás esetén az összes belső kifejezés tetszőleges lehet. Nincs semmilyen megkötés arra, hogy milyen típusú változót használjunk, milyen aritmetikai, vagy logikai kifejezést használjunk.

Tipikus hiba a for zárójele után pontosvessző kerül. Ekkor a ciklustörzs a `;` operátor lesz.

```
5         j=0;
6         for(i=1;i<=10;i++);
7            {
8            j=j+i;
9            }
```

EZ EGY HIBÁS LISTA!!!! a 6. sorban a `;` lezárja a ciklust, ezért a végeredmény a várt 55 helyett 11 lesz. A ciklus ekkor csak az `i` értékét növeli 11-ig, majd ezt adja át `j`-nek.

2.4.5. `while`

A `while` utasítás elöltesztelő ciklus létrehozására szolgál. Az utasítás argumentumában lévő kifejezés kiértékelése relációs jellegű. A program mindaddig belép a ciklustörzsbe, amíg a kifejezés igaz.

A ciklustörzs itt is vagy egy `;`-vel lezárt kifejezésig, vagy az utasítást közvetlenül követő {} párba zárt logikai blokk.

Készítsük el az első 10 egész szám összegét a `while` segítségével! A következő program futtatható.

```
1      #include <stdio.h>
2      int main(void)
3         {
4         int i,j;
5         i=1;
6         j=0;
7         while(i<=10)
8            {
9            j=j+i;
10           i++;
11           }
12        printf("%d",j);
13        return 0;
14        }
```

Itt az 5. és a 6. sorban külön kell a változókat inicializálni. a 7. sorban van a `while`, aminek az

argumentumában egy relációs kifejezés van. Ha ez igaz, akkor a program belép a ciklustörzsbe.

A 10. sorban történik meg az összegzés.

A 11. sorban nekünk kell gondoskodnunk az `i` változó növeléséről.

Tipikus hiba itt is mint a `for`-nál a `;` lerakása a `while` zárójele után.

EZ EGY HIBÁS LISTA!!!!

```
5       i=1;
6       j=0;
7       while(i<=10);
8         {
9         j=j+i;
10        i++;
11        }
```

A probléma ezzel a programmal az, hogy nem kívánt végtelen ciklusba kerül. Ennek oka az, hogy `i` változó értéke nem fog változni, mert a program soha nem lép tovább a 7. sorról és a feltétel igaz marad.

2.4.6. `do-while`

A `do-while` szerkezet hátultesztelő ciklus létrehozását teszi lehetővé.

Hátultesztelő ciklus ciklus esetén az ciklustörzs legalább egyszer biztosan lefut, mert a ciklusban maradás feltétele - röviden ciklus feltétel - a ciklustörzs végrehajtása után kerül feldolgozásra.

A ciklustörzs vagy egyetlen `;`-vel lezárt kifejezés, vagy összetettebb esetben {} közé zárt logikai blokk.

Az egyszerű esetre példa:

```
1       int main(void)
2           {
3           int i=0;
4           do
5             i++;
6           while(i<10);
7           return 0;
8           }
```

A ciklustörzs itt az 5. sorban a `i++;` kifejezés.

A ciklus addig fut, amíg a `while` argumentuma igaz.

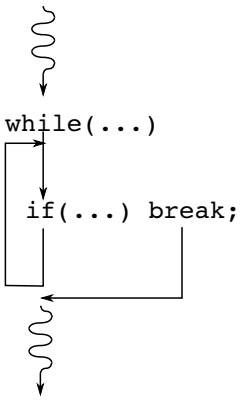
Figyeljük meg, hogy a 6. sorban a `while ;`-vel le van zárva.

Az összetett esetre vegyük a megszokott példát a számok összegét 1-10-ig.

```
1     #include <stdio.h>
2     int main(void)
3       {
4       int i,j;
5       i=1;
6       j=0;
7       do
8         {
9         j=j+i;
10        i++;
11        }
12      while(i<=10);
13      printf("%d",j);
14      return 0;
15      }
```

2.4.7. `break`

A `break` utasítással már találkoztunk a `switch-case` szerkezetben. Itt is hasonló a szerepe, amikor egy cikluson belül a program találkozik egy `break` utasítással, akkor a program az aktuális ciklust elhagyja. Ezt a futási gráfon lehet követni.

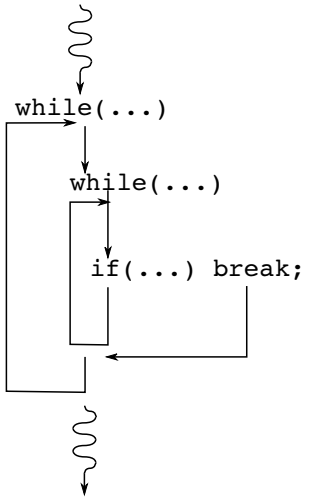


2. ábra: A `break` hatása

Láthatjuk, hogy a `break` hatására a program kiugrik a ciklusból[19].

[19] Az `if(...)` azért szerepel a gráfban, mert ciklusba csak úgy nem írunk `break` utasítást. Nem sok értelme lenne.

Nézzük azt az esetet, ha két ciklus van egymásba ágyazva és a belső ciklusban van a `break` utasítás.



3. ábra: A `break` hatása egymásba ágyazott ciklusok esetén

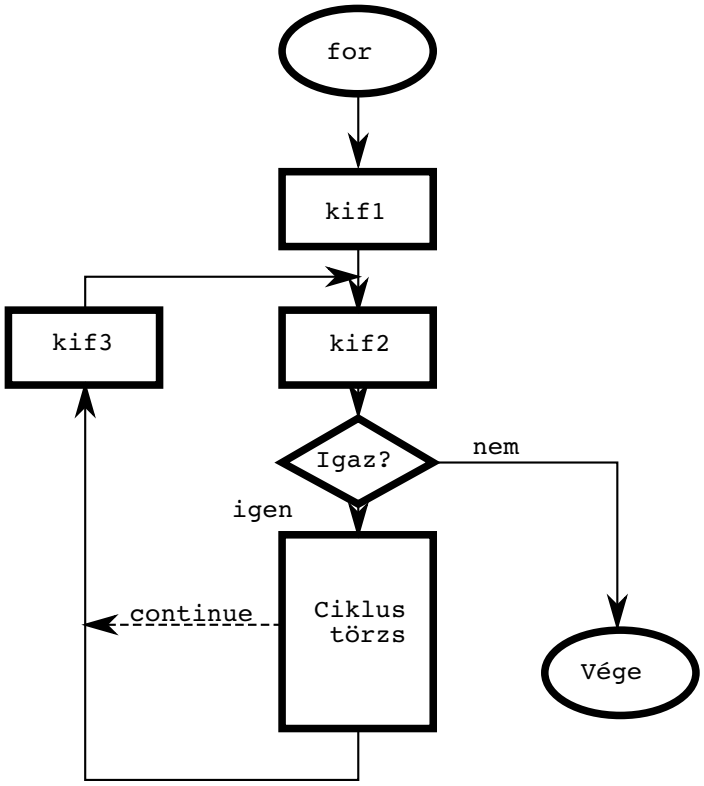
A `break` szempontjából teljesen mindegy melyik ciklus utasítást, illetőleg szerkezetet használjuk, a működése azonos minden esetben.

2.4.8. `continue`

Ez az utasítás is a ciklus szerkezetek futását módosítja. Hatására a ciklustörzs hátralévő részét a program figyelmen kívül hagyja és a ciklus feltétel, illetve `for` esetén a 3. kifejezéssel folytatja futását[20].

A `for` működése `continue` esetén a következő folyamatábrán látható:

[20] Régebbi szakkönyvek és jegyzetek azt írják, hogy a ciklus újra kezdődik, ez nem igaz.



4. ábra: A `continue` hatása egymásba `for` ciklus esetén

A következő példa a `continue` működését mutatja be.

```
1     #include <stdio.h>
2     int main(void)
3        {
4        int i;
5        for(i=0;i<10;i++)
6           {
7           if(i==5) continue;
8           printf("%d ",i);
9           }
10       return 0;
11       }
```

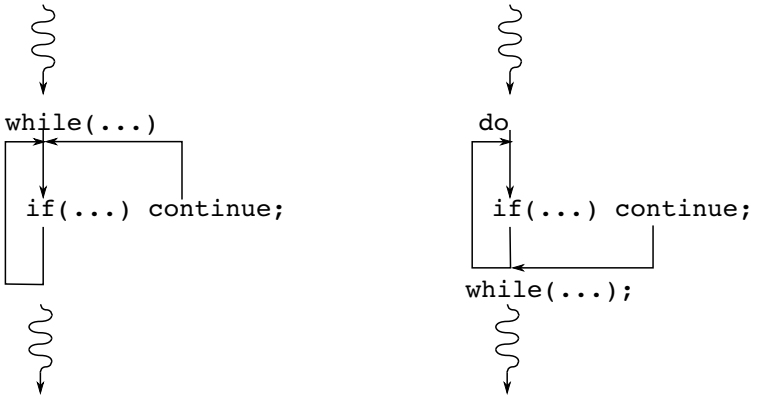
A program 6. sorában lévő `continue` kihagyja a ciklus további részét és a 3. kifejezésre ugrik. Ezért a program kimenete a következő:

```
0 1 2 3 4 6 7 8 9
```

Mint látjuk a `printf` függvény nem írta ki az 5-ös értéket, mert nem került rá a vezérlés.

A `printf` ebben az esetben az `i` aktuális értékét írja a képernyőre.

A következő két futási gráf bemutatja a `continue` hatását `while` és `do-while` esetén.



5. ábra: A `continue` hatása `while` és `do-while` ciklusok esetén

2.4.9. `goto`

A `goto` utasítás feltétel nélküli ugrás ír elő. Valahol a programba elhelyezünk egy címkét és a `goto`-val erre a címkére ugorhatunk.

A címke a következő, például:

```
CIMKE:
```

A `goto`-val az ugrás:

```
goto CIMKE;
```

A címkére ugyanazok a szabályok vonatkoznak, mint a változónevekre, de azért szoktunk[21] nagy betűs címkéket használni, mert nincsenek nagybetűs kulcsszavak és jól láthatók.

Figyelem a `goto` használata nagyon nem ajánlott. Átgondolatlan ugrással a programot tökéletesen ki lehet fagyasztani. Csak akkor indokolt a használata, ha időzavarban van a program. De ez csak nagyon ritkán fordul elő.

2.4.10. `return`

Ez az utasítás arra szolgál, hogy visszatérjünk egy függvényből, illetőleg a visszatérés mellett értéket is adjunk vissza a hívó félnek.

Mivel ez az utasítás szervesen kapcsolódik a függvényekhez, ezért ezt ott tárgyaljuk részletesen. Lásd lejjebb.

[21] Nem szoktunk egyáltalán nem szoktunk `goto`-t használni.

2.5. Függvények

A függvények tulajdonképpen szubrutinok.

A szubrutin egy olyan alprogram vagy programrész, amely a program bármely részéről többször meghívható.

A program a szubrutin hívás helyéről a szubrutin soraira ugrik. Amikor ezeket végrehajtotta a hívás utáni soron folytatja futását.

A programozási nyelvek - beleértve az assemblereket is - kevés kivétellel ismerik a szubrutint.

A függvény tehát egy önálló programrész, amely valamilyen feladatot végez[22].

Minden C program tulajdonképpen egy függvény. Eddig minden C programunkban található egy `int main(void)` sor. Ez az úgynevezett vezető függvény fejléce.

A C egy teljesen ”demokratikus” programozási nyelv minden függvény szintaktikailag mindenhonnan hívható - még a `main` is.

A függvények ismertetését egy példán keresztül mutatjuk be. A függvény három egész változó közül választja ki a legnagyobbat és azt adja vissza.

A következő programrészlet a függvény **definíciója**.

```
1     int max3(int a,int b,int c)
2        {
3        int i;
4        if(a>b) i=a;
5        else    i=b;
6        if(c>i) i=c;
7        return i;
8        }
```

Az lista 1. sorában látható a függvény fejléce. Az első kifejezés `int` azt mondja meg, hogy milyen típusú változót fog visszaadni a függvény. Ezt nevezzük a **függvény típusának**.

A következő kifejezés a függvény neve `maxi` ennek a névnek a segítségével fogunk hivatkozni függvényre.

A név utáni zárójelpárban a paraméterlista található. Itt deklaráljuk az úgynevezett átadott paramétereket. Ezek a változók fogják a függvény bemeneti paramétereit tartalmazni.

A 2. sorban a függvény törzsét megkezdő kapcsos zárójelet találjuk.

[22] Néhány programozási nyelv, mint például a Pascal a szubrutinokat két csoportra osztja: eljárásokra, amelyek nem adnak vissza értéket és függvényekre, amelyek értéket adnak vissza. A C esetében minden szubrutin függvény.

A 3. sorban egy változót deklarálunk, amely átmeneti tárolóként fog szerepelni. Ez a változó egy úgynevezett blokkra lokális változó (lásd 59. oldal), amely csak és kizárólag ebben a függvényben látható és érhető el.

A 4. és 5. sorral most nem foglalkozunk[23], annak ellenére, hogy ez függvény működő része.

A 6. sorban láthatunk egy `return` utasítást. Ennek az utasításnak az a szerepe, hogy a programot kiléptesse a függvényből és az argumentumában lévő változó értékét visszaadja a hívó függvénynek.

Figyelem a definíció végén nincs pontosvessző!

Nézzük ennek a függvénynek a használatát egy programban!

```
1     #include <stdio.h>
2     int max3(int,int,int);
3     int main(void)
4        {
5        int x,y,z;
6        int r;
7        scanf("%i%i%i",&x,&y,&z);
8        r=max3(x,y,z);
9        printf("%d",r);
10       return 0;
11       }
12    int max3(int a,int b,int c)
13       {
14       int i;
15       if(a>b) i=a;
16       if(c>i) i=c;
17       return i;
18       }
```

A 2. sorban láthatjuk a `int max3(int,int,int);` kifejezést. Ezt nevezzük a függvény **deklaráció**-jának

Itt megadjuk a függvény típusát, a nevét és a paraméterlistában szereplő változók listáját sorrendben. A deklarációt lezárjuk pontosvesszővel.

Az 5. és 6. sorban a `main` belső változóit deklaráljuk. Ezek a változók csak a `main`-ban ”léteznek”.

A 7. sorban a beolvassuk az `x`, `y` és `z` változókat (lásd 69 oldalon.).

A 8. sorban történik meg a `max3` függvény meghívása. A meghívás során a `main` `x` változó értéke

[23] Ezt már az olvasónak értenie kell.

belemásolódik a `max3` `a` változójába, a `y` a `b`-be és a `z` a `c`-be.

A függvény végén a 17. sorban található `return i;` hatására a `max3` `i` változó értéke a `main` `r` változójába másolódik.

A `printf` kiírja `r` értékét.

A 10. sorban lévő `return` utasítás az operációs rendszernek ad át egy úgynevezett errorkódot.

Mégegyszer!

- A függvény deklarációjánál megadjuk a függvény fejlécét pontosvesszővel lezárva Ekkor a fordító programnak mondjuk meg, hogy a függvényt a hívásokkor hogyan kell értelmezni.
- A függvény definíciója a függvény tulajdonképpeni megírása.
- A függvény típusa annak a változónak a típusa, amit a függvény visszaad.

Megjegyzés: a függvényeket néhány esetben nem kötelező deklarálni. Ezek az esetek, ha a függvény definíciója megelőzi a függvény hívását (nem mindig teljesíthető) és ha a függvény típusa `int`. De a kellemetlen figyelmeztetések és rejtélyes hibák elkerülésére erősen ajánlott a deklaráció. Ha valaki C-ről áttér C++-ra és nem deklarál a figyelmeztetés helyett már hibaüzeneteket kap.

Felmerül a kérdés, hogy mi van azokkal a függvényekkel, amelyek nem adnak vissza értéket[24]. Ezek az úgynevezett `void` típusú függvények.

Abban az esetben, ha a függvénynek nincsenek átadott paraméterei, akkor célszerű ezt is `void`-ként megadni. Tehát például az előbb említett képernyőtörlő függvény deklarációja:

```
void clrscr(void);
```

A `void` típusú függvényeknél nem kötelező a `return` utasítás használata, a függvény egyszerűen csak véget ér Azonban lehet használni. Ha egy `void` függvényből nem a végén akarunk kilépni, hanem valahol ”útközben”, akkor egyszerűen használhatjuk a `return;` kifejezést.

Ez igaz akkor is, ha a függvény nem `void` mert a függvény bármely részéről visszatérhetünk egy `return`-al - persze értelemszerűen.

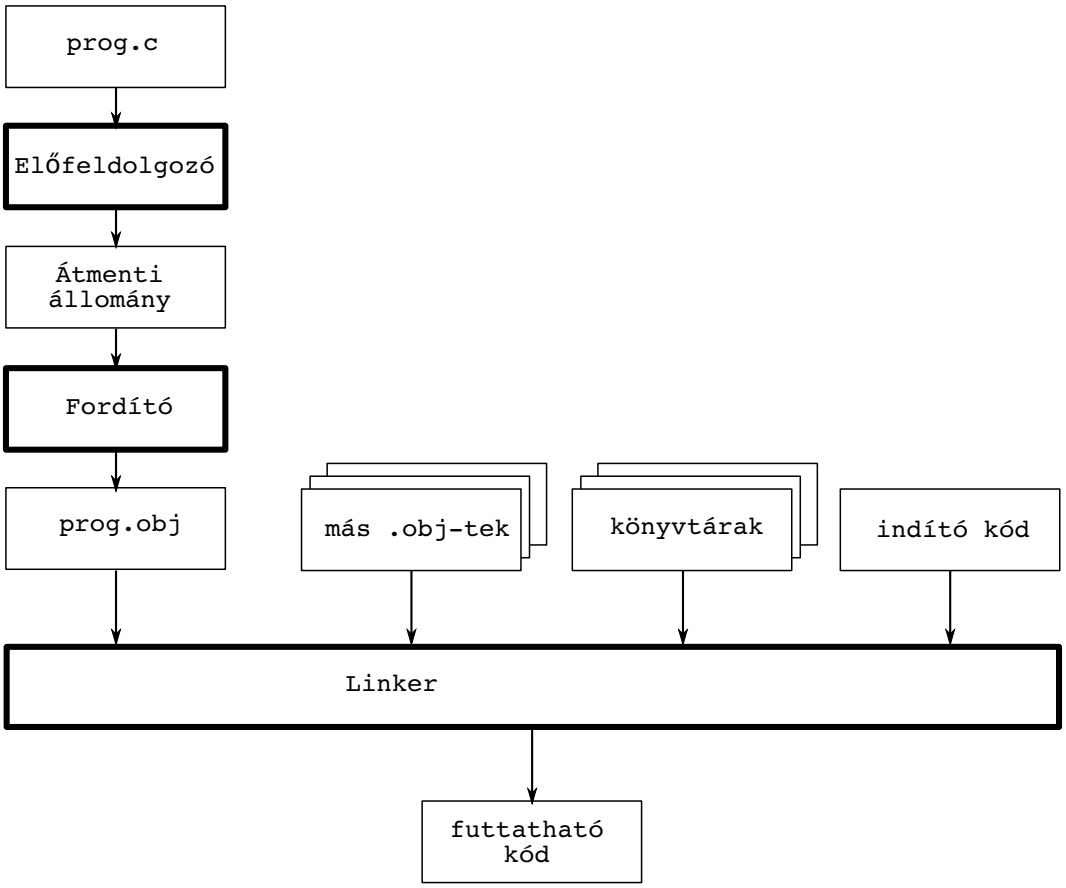
A C nyelv nagyon széles függvénykönyvtárral rendelkezik. Alapesetben is egy gcc környezet több ezer függvényt biztosít a felhasználó számára. Ezek a könyvtárak a fordítás során épülnek be a futtatható kódba (lásd következő fejezet).

[24] Igen vannak ilyenek. Egy képernyő törlésnek miért kellene értéket visszaadnia.

2.6. A fordítás folyamata

Igazából ez a témakör nem szigorúan tartozik a szintaktikai elemek közé, de ha gyorsan megbeszéljük sokkal könnyebb az ezt követő elemek magyarázata

A fordítás folyamata a következő ábrán követhető.



6. ábra: A fordítás vázlata

Kiindulunk egy egyszerű `prog.c` nevű forráskódból. Ebből szeretnénk futó programot készíteni.

A fordítás első lépéseként a forráskódot egy előfeldolgozó (precompiler) kapja meg. Ennek feladata az hogy a forrásból a fordítás számára lényegtelen dolgokat eltávolítsa - megjegyzéseket, fölösleges szóközöket és zárójeleket - és hajtsa végre az előfeldolgozó számára szóló utasításokat.

Ekkor képződik egy átmeneti állomány, amit mi nem is látunk.

Az átmeneti állományból a fordító program (compiler) készít egy úgynevezett `prog.obj` állományt. Ez már egy lefordított kód, de a bizonyos címek és hivatkozások még nincsenek kitöltve.

A végleges fordítást a linker végzi, ami minden szükséges elemet hozzácsatol a kódunkhoz és az előzőleg kitöltetlen adatokat is kitölti és ezáltal egy futtatható kódot állít elő.

Megjegyzés: ez azért egy nagyon vázlatos leírás.

2.7. Előfeldolgozó utasítások

Az előfeldolgozó utasításoknak nincs közvetlen hatásuk a program futására. Ezek a fordítás folyamatát befolyásolják.

Minden előfeldolgozó utasítás # karakterrel kezdődik és nincs a végén pontosvessző.

2.7.1. `#define`

A `#define` szöveghelyettesítésre szolgál. Ezért fordítás idejű konstansokat és makrókat lehet vele definiálni.

Elsőként nézzünk példát a fordítás idejű konstansra!

```
#define ZERO 0
```

Attól a ponttól kezdve ahonnan a `ZERO` szövegkonstanst definiáltuk, a program ismeri ezt. tehát ha a programunkba leírjuk például, hogy:

```
a=ZERO;
```

akkor az előfordítás után az optimalizált kódba az kerül, hogy

```
a=0;
```

Ekkor csak egy egyszerű szöveghelyettesítés történik.

A makró is tulajdonképpen szöveghelyettesítés, de kicsit intelligensebb módon. Nézzünk erre is egy példát!

```
#define MAX(a,b) (a>b)?(a):(b)
```

Ha a programunkba leírjuk a következő sort:

```
z=MAX(x,y);
```

akkor a következő szöveg kerül be a kódba előfeldolgozás után:

```
z=x>y?x:y;
```

a fölösleges zárójelek eltűntek és az `a` és `b` formális paraméterek helyére az `x` és az `y` változók kerültek.

Figyelem! ne feledjük a makró nem függvény, hanem csak szöveghelyettesítés. Nagyon szerencsétlen hibát okozhat például egy ilyen makró hívás.

```
z=MAX(x++,y);
```

Ennek elemzését az olvasóra bízzuk. De ha lehet, akor kerüljük az ilyen eseteket.

Szintaktikailag akár konstans, akár makró definiálás estén a `#define`-t legalább egy szóköz értékű karakternek kell lennie, aztán jöhet a szimbólum rész, amiben nem lehet szóköz jellegű karakter, majd a helyettesített rész, amiben lehet szóköz és vagy a sor végéig tart, vagy valamelyik megjegyzés mezőhöz.

Ha nem férünk el a helyettesített résszel az adott sorban, akkor azt elválaszthatjuk egy \karakterrel. Erre mutat példát a következő programrészlet:

```
#define RNG() (mx>x && mx<x+w &&\
               my>y && my<y+h &&\
               btn)
```

Így sokkal jobban olvasható a program és akár soronként megjegyzések is fűzhetők a kódba.

Ne tedd! Ha egy mód van egy makró legyen egyetlen kifejezés és a belsejében ne legyen pontosvessző és ne legyen benne logikai blokk. Persze lehet, de azt nagyon fejben kell tartani, mert nagyon sok kellemetlenséget okozhat.

Ha mégis: mert nem tudjuk elkerülni, hogy a makrónk több kifejezést tartalmazzon, akkor használjuk a következő példában alkalmazott trükköt:

```
#define PLD() do {\
                printf("Ez az elso sor\n");\
                printf("Ez a m\'asodik\");\
                printf("Es igy tovabb\n");\
                }\
              while(0)
```

Az utolsó sorban nincs pontosvessző, mert azt majd a programozó fogja a programban letenni.

Ezzel a megoldással akármekkora makrót írhatunk és szintaktikailag a programunk ”szilárd” marad. Azt viszont ne feledjük, hogy ez csak a záró pontosvessző problémáját oldja meg.

Néhány esetben kisegíthet a vessző operátor is, példa erre a következő makró:

```
#define    ERR(a,b)    printf(a),exit(b)
```

Ekkor a `printf` függvény után nincs pontosvessző, tehát a kifejezés nincs lezárva, hanem az `exit` függvény következik, amely után a lezáró pontosvesszőt a programozó fogja letenni.

2.7.2. `#undef`

Ha valamilyen okból akár a makrónkat, akár a fordítás idejű konstansunkat meg akarjuk szüntetni, mert például felül akarjuk írni, akkor az `#undef`utasítást kell használni. Tehát például:

```
#undef ZERO
```

sor után a `ZERO` szövegkonstans nem létezik.

2.7.3. `#include`

`#include` utasítás arra utasítja az előfeldolgozót, hogy az utasítás helyére szerkessze be az utasítás argumentumában szereplő fájl tartalmát.

Ezeket a fájlokat nevezzük header fájloknak.

Ennek két eltérő formája van. Történelmi okokból a gyári header fájlokat egy `<>` jelpár közé, a mi általunk írt header állományokat pedig `""` pár közé tesszük.

Vannak olyan környezetek, amelyek ezen jelölés alapján különbséget tesznek fordításkor, de nem mindegyik. Célszerű ezt a konvenciót betartani.

A header fájlok függvény deklarációkat, makrókat, fordításidejű konstansokat és típusdefiníciókat tartalmaznak. Tartalmazhatnak ugyan függvény definíciókat, de ez erősen kerülendő.

Példa a gyári header fájl behívására:

```
#include <stdio.h>
```

Példa a saját header behívására:

```
#include "enyem.h"
```

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

Az alfejezet címében szereplő utasítások a feltételes fordítás utasításai. A feltételes fordítást általában akkor használjuk, ha a megírt programnak különböző környezetekben is jól kell működnie, vagy esetleg hibakeresésé céljából speciális kódrészleteket kell beletenni a programba, amelyek később szükségtelenek, stb.

A feltételes blokkok tetszőleges számban egymásba ágyazhatók.

Kezdjük a végén! Minden egyes feltételes fordítás blokk az `#endif` utasítással ér véget.

Az `#if` utasítás a kezdete egy feltételes blokknak. Ha az argumentumában lévő kifejezés igaz az `#if`-et

követő blokk lefordul, vagy az `#endif`, vagy a `#else`, vagy az `#elif` utasításig, vagy esetleg egy újabb feltételes blokk kezdetéig.

```
#if DEBUG==1
      |
    fordul
      |
#endif
```

Ha tehát a `DEBUG` paraméter értéke 1, akkor az `#if` és `#endif` közötti rész lefordul.

Ha a kifejezés nem igaz, akkor viszont egyáltalán rá se kerül a fordítóra, mert az előfeldolgozó nem ”engedte” tovább a kérdéses kódrészletet.

Az `#else` ág - ha van - akkor érvényesül, ha az `#if`-ben lévő kifejezés hamis.

```
#if 2==1
      |
    nem fordul
      |
#else
      |
    fordul
      |
#endif
```

Mivel a példában 2 nem egyenlő 1-el ezért a kifejezés hamis, tehát most az `#else` ág fordul be a végső kódba.

Az `#elif` tulajdonképpen egy `#else` és egy `#if`. Ha az argumentumában lévő kifejezés igaz, akkor egy új fordítási blokkot indít, viszont nincs szükség újabb `#endif`-re.

```
#if 2==1
      |
    nem fordul
      |
#elif 2==2
      |
    fordul
      |
#else
```

```
        |
      nem fordul
        |
#endif
```

Az `#ifdef` egy olyan ”`#if`”, amely akkor ”igaz”, ha az argumentumában szereplő kifejezés már előzőleg egy ”`#define` utasítással már definiálásra került. Az értéke teljesen mindegy.

`#ifndef` akkor igaz, ha a kérdéses szimbólum nem definiált.

Mind az `#ifdef`, mind az `#ifndef` esetén a többi utasítást azonos módon lehet - kell használni

2.7.5. `#pragma`, `#warning`, `##`

A `#pragma` előfeldolgozó utasítás arra szolgál, hogy a fordító programnak különböző üzeneteket küldjünk. Ezzel az utasítással például egyes figyelmeztetéseket ki lehet kapcsolni.

A `#warning` utasítás az argumentumában megtalálható szöveget fordítás során kiírja a képernyőre. Ez nem szabványos C utasítás.

A ##utasítás két szövegrészt fűz össze a forráskódban. Például:

```
value##a=a;
```

Ekkor a fordításra kerülő szöveg:

```
valuea=a;
```

2.8. Mutatók

A mutatók, vagy pointerek olyan speciális változók, amelyek nem értéket, hanem valamilyen más objektumnak[25] a címét tartalmazza.

2.8.1. Alaptípusok mutatói, címképzés

Azt mondtunk, hogy a mutató egy speciális típus, amely valaminek címét tartalmazza. A legegyszerűbben az alaptípusok mutatóit értelmezhetjük.

Vegyük példának egy egész típusú változó pointerét. A deklarációja a következő:

```
int *ip;
```

[25] Ezt most nagyon nehezen írtam le. Ezen a helyen az objektum kifejezés zavart kelthet. Az objektumokról a 56. oldalon beszélünk.

Láthatjuk, hogy meg kell adni a típust, majd a csillag jelzi, hogy ez nem egy változó, hanem egy pointer, aztán jön a pointer neve.

Hogyan kap értéket egy ilyen pointer? Nézzük a következő program részletet!

```
int *ip;
int i;
   :
   :
ip=&i;
```

Láthatjuk, hogy van egy `*ip` pointerünk és van egy `i` változónk aztán az `ip=&i;` sorban az `ip` értéket kap a `&` operátor segítségével. Ez a címképző operátor.

Szó szerint az adott sor azt jelenti, hogy:

Az `ip` érteke legyen egyenlő `i` címével.

Ez az operátor nem keverendő a bitenkénti és operátorával. A címképző operátor egyoperandusú, míg a bitenkénti és kétoperandusú.

Bármelyik alap- és definiált típusnak van mutatója, de a mutatóknak is lehet mutatója. Például egy egész mutató mutató deklarációja a következő:

```
int **p;
```

A `p` természetesen csak a név[26].

2.8.2. Indirekció

Az indirekció egy igen fontos fejezete a C nyelvnek. Ezt rögtön egy programrészleten mutatjuk be.

```
1     int i,j;
2     int *ip;

       :

3     i=5;
4     ip=&i;
5     j=*ip;
```

[26]Kipróbáltam meddig megy, de 15 csillagnál abba hagytam. Még egy megjegyzés háromszoros pointert már használtam.

Az 1. sorban két `int` változót deklaráltunk és a másodikban egy `int` pointert.

A 3. sorban az `i` változó értéket kapott.

A 4. sorban az `ip` pointer értéke az `i` változó címe lett.

Az 5. sor egy úgynevezett indirekt értekeadás A 5. sor szó szerint a következőt csinálja:

A `j` változó értéke legyen egyenlő az `ip` pointer által mutatott változó értékével.

Mivel az `ip` ebben az esetben az `i` címét tartalmazza `j` értéke `i` értékével lesz egyenlő. De nem közvetlenül, hanem a pointeren keresztül, ezért nevezzük ezt indirekt értékadásnak.

Persze fordítva is lehetséges, ha most az írjuk, hogy:

```
*ip=9;
```

akkor az `i` változó értéke 9-re változik, mert - mint eddig azt láttuk - `ip` `i` címét tartalmazza.

Az indirekció operátora a `*` operátor, amely nem tévesztendő össze a szorzás operátorával. Az indirekció operátora egyoperandusú, míg a szorzás operátora kétoperandusú operátor.

2.8.3. Függvény mutatók

A függvény is egy objektum, meghatározott helye van a számítógép memóriájában, ezért létezik rá mutató is.

Nézzünk egy példaprogramot a függvénymutató deklarálására és használatára.

```
1     #include <stdio.h>
2     int fgv(int);
3     int main(void)
4        {
5        int r;
6        int (*fgv_pt)(int);
7        fgv_pt=fgv;
8        r=fgv_pt(42);
9        printf("%i",r);
10       return 0;
11       }
12    int fgv(int x)
13       {
14       return x;
15       }
```

A 2. sorban deklarálunk egy függvényt, amelyet 12. sortól írunk meg. A függvény csak annyit csinál, hogy az átadott paramétert visszaadja a hívó félnek.

Az 5. sorban deklarált változóban kapjuk meg az eredményt.

A 6. sorban egy függvény pointert deklarálunk. A pointer neve `fgv_pt`. Láthatóan van típusa és paraméter listája. Ez a pointer egy olyan függvényre fog mutatni, amely `int` típusú és egyetlen `int` paramétere van. Az `fgv` függvény pont ilyen[27].

A 7. sorban a pointernek átadjuk az `fgv` függvény címét. Figyeljük meg, hogy a függvény címét a függvény neve képviseli a zárójelek nélkül.

A 8. sorban a pointeren keresztül - vagyis indirekt módon - meghívjuk a függvényt. A visszatérési értéket az `r` változóban kapjuk meg. Az egész indirekt meghívás olyan, mintha a függvényt közvetlenül hívtuk volna.

2.8.4. A `void` pointer

A `void` pointer egy általános pointer semmilyen típushoz sem kötődik. Ezt a pointer típust jó néhány könyvtári függvény használja főleg azért, mert nem lehet tudni előre, hogy milyen típusú adatokon fog a függvény dolgozni.

A `void` pointer eltér a normál típusokra vonatkozó pointerektől abban, hogy nem igaz rá a pointer aritmetika.

A pointer aritmetikát a 45. oldalon ismertetjük.

2.9. Összetett adatszerkezetek

2.9.1. Tömbök

A tömb egy olyan összetett adatszerkezet, amely azonos típusú adatokat rendez egy összefüggő memória területen.

Egy egydimenziós `int` tömb deklarációja a következő módon történik:

```
int t[10];
```

Ez a tömb 10 elemű, tehát 10 darab `int` típusú változót tartalmaz. A tömb elemeire a tömb neve és az úgynevezett index segítségével hivatkozunk.

A példában egy tíz elemű tömb van ennek indexelési tartománya 0..9. Tehát a tömb első elemének indexe 0, az utolsó elemének indexe 9.

[27] Vajon miért?

Egy tömb elemre a következőképpen hivatkozhatunk:

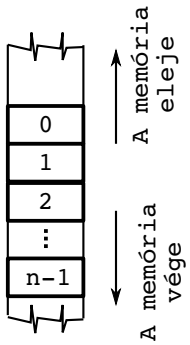
```
t[5]=8;
```

Tehát a `t[5]` egy tömb elem, amelyet pontosan ugyanúgy használhatunk, mint bármilyen más `int` típusú változót.

Az index lehet pozitív egész szám, vagy egész jellegű változó.

Figyelem! A C nem ellenőrzi futásidőben az indexelési tartományt. Tehát könnyedén túlindexelhetünk. Vagyis olyan index tartományt adunk meg, ami már nem a tömb területére hivatkozik.

A definícióban említettük, hogy a tömb összefüggő memóriaterületen foglal helyet. A legalacsonyabb indexű elem a legalacsonyabb memória címen található, a következő elem közvetlenül mögötte és így tovább a növekvő memória címek felé.



7. ábra: A tömb a memóriában

A tömb deklarációkor inicializálható[28].

```
int t[5]={1,3,-2,9,123};
```

Ilyenkor nem szükséges megadni a tömb méretét.

```
int t[]={1,3,-2,9,123};
```

A tömb mérete ilyenkor is 5 lesz.

Ha megadjuk a tömb méretét, de több elemet akarunk a tömbbe beletenni, mint a méret, akkor fordításkor hibaüzenetet kapunk. Ha kevesebbet a tömböt a fordító program az elejétől kezdve feltölti, a hiányzó elemeket általában 0-val tölti fel[29].

A tömb kezdőcíme úgy határozható meg, hogy egyszerűen a tömb nevét használjuk hasonlóan a függvény pointerekhez.

A következő program részlet erre mutat rá.

[28] Kezdő értéket lehet neki adni.
[29] Ezt fordító programja és a tömb helye határozza meg.

```
int t[10];
int *ip;
  :
  :
 ip=t;
```

Az `ip=t;` kifejezés teljesen megegyezik a `ip=&t[0];` kifejezéssel.

Erről még esik szó a pointer aritmetikánál.

2.9.2. Többdimenziós tömbök

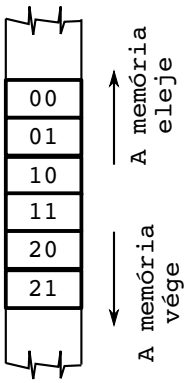
A C - hasonlóan más nyelvekhez - ismeri a többdimenziós tömböket. A dimenziók számában nincs korlátozás, azonban ne feledjük, hogy a lefoglalt memóriaterület mérete a dimenziók számával exponenciálisan növekszik.

Deklaráljunk egy kétdimenziós `int` tömböt!

```
int t[3][2];
```

Az indexelési tartomány dimenziónként 0..n-1 tartományban van. Tehát a példa tömbünk első eleme `t[0][0]`, az utolsó eleme pedig a `t[2][1]`

Ekkor is a tömbelemek egymást követve helyezkednek el a memóriában úgy, hogy a leghátsó index változik a leggyorsabban.



8. ábra: A kétdimenziós tömb a memóriában

A többdimenziós tömbök is inicializálhatók deklarációkor (szépen).

```
int t[3][2]={ { 1,2 },
              { 3,4 },
              { 5,6 } };
```

(és nem szépen.)

```
int t[3][2]={1,2,3,4,5,6};
```

Az első módszer azért jobb, mert sokkal jobban követhető, ha bármilyen hibát vétünk az jól javítható. A második módszer esetén egy elírás nagyon nehézzé teszi a javítást[30].

A tömb mérete lekérdezhető a `sizeof` operátor segítségével. Figyelem a tömb méretét a bájtban adja meg a `sizeof` és csak abban a modulban, ahol a tömböt deklaráltuk.

2.9.3. Pointer aritmetika

Ennek a fejezetnek logikailag nem itt van a helye, de a pointer aritmetika működése sokkal könnyebben követhető az egydimenziós tömbök segítségével, mint önállóan.

A pointereken öt művelet hajtható végre, ezek:

- pointer inkrementálása,
- pointer dekrementálása,
- egész szám hozzáadása a pointerhez,
- egész szám kivonása a pointerből,
- két pointer kivonása egymásból.

A pointer inkrementálásánál a kiindulási memóriacím annyi bájttal növekszik, amekkora a pointer által mutatott típus mérete.

Ha tehát a kérdéses típus `char` az inkrementálás során a cím a következő bájtra fog mutatni. Ha a kérdéses típus `double` akkor cím nyolc bájttal fog nőni.

A pointer dekrementálásakor a cím csökken a típusnak megfelelő mérettel (bájtban értelmezve).

Nézzük a következő programrészletet:

```
double t[10];
double *p;
   :
p=t;
p++;
```

[30] Na ja nem 6db elem esetén

A példában `double` típusú tömb kezdőcímét átadjuk a pointernek, vagyis az a 0-ás indexű elem címét tartalmazza. Az inkrementálás után a pointer az 1-es indexű elemre fog mutatni.

Nagyon jól kihasználható tulajdonság, az hogy a tömbök és a pointerek ”átjárhatók”. Nézzük a következő példát.

```
double t[10];
double *p;
   :
p=t;
p[2]=3.1415;
```

A pointer megkapja a tömb kezdőcímét, ezután a pointert használjuk úgy, mint a tömböt. Vagyis a `p[2]=3.1415;` kifejezés valójában a `t` tömb 2-es indexű elemébe helyezi el a PI közelítő értékét.

Ez a gondolat rögtön átvezet a pointerek értékének egész számmal történő módosítására.

Ha egy pointer értékét egész számmal módosítjuk (hozzáadunk, kivonunk), akkor a pointer értéke annyi bájtszor típusmérettel módosul, amekkora a kérdéses egész szám mérete.

A következő programrészlet ezt mutatja be.

```
double t[10];
double *p;
   :
p=t;
*(p+2)=3.1415;
```

Ekkor a `*(p+2)=3.1415;` kifejezés megint a `t[t]` tömb 2. indexű elemébe teszi a PI értékét.

Két pointer kivonása esetén az adott típus méretében megkapjuk a két pointer távolságának egész részét. Tehát:

```
double t[10];
double *p1,*p2;
int i;
   :
p1=&t[2];
p2=&t[4];
i=p2-p1;
```

Az `i` értéke ebben az esetben 2.

2.9.4. Sztringek

Más programozási nyelvekben járatos olvasó esetleg hiányolhatta a típusok ismertetésénél a sztringek hiányát. A C nem tekinti típusnak a sztringeket, hanem karakter tömbként kezeli őket.

A sztring végét egy 0 értékű[31] karakter jelzi, függetlenül a tömb méretétől.

A sztring deklarálásnál inicializálható, hasonlóan más változókhoz és tömbökhöz. A következő példa ezt mutatja be.

```
char nev[]="Nokedli";
```

A `nev` nevű tömb 8 elemű, mert ebben az esetben a fordító program gondoskodik a lezáró 0-ról. Tehát a 7-es indexű elem értéke 0.

Természetesen deklaráláskor itt is megadhatjuk a tömb méretét.

A sztingekre való hivatkozáskor nem a sztringre, mint értékre hivatkozunk, hanem a sztringet tartalmazó tömb kezdőcímével, tehát a **következő programrészlet teljesen rossz!**

```
char a[]="Hello!";
char b[80];

b=a;
```

Figyelem az előző programrészlet HIBÁS!

Az ilyen problémák megoldására a `string.h` header fájlban deklarált függvények szolgálnak.

2.9.5. Struktúrák

A tömbök szigorúan csak azonos típusú változók tárolására szolgálnak. Számos esetben azonban szükségünk lehet logikailag egybe tartozó, de különböző típusú adatok kezelésére. Erre szolgálnak a C-ben a struktúrák.

A struktúra bármilyen típusú adatelemet, tömböt, más struktúrát és tetszőleges pointert tartalmazhat kivéve pontosan olyan struktúrát mint önmaga. Ezt a megszorítás igaz közvetlen és közvetett tartalmazásra is[32].

Ha struktúrát szeretnénk alkalmazni, először azt definiálni kell. Ez esetleg lehet gyári definíció, ekkor általában egy header fájlban ezt a rendszer gyártója megteszi. Ha saját struktúrát készítünk akkor nekünk kell a definíciót elkészíteni.

[31] És nem kódú.

[32] Tehát ha ez tartalmaz egy olyan struktúrát, amely valahol az ”alap” struktúra példányát tartalmazza, esetleg több áttétellel is.

Ha a definíció kész, utána készíthetünk belőle példányt. Ekkor tulajdonképpen egy adott struktúra ”típusú[33]” változót hozunk létre.

A következő példa a struktúrák használatát mutatja be.

```
1     #include     <stdio.h>
2     #include     <string.h>
3     struct ADAT
4       {
5       char           nev[80];
6       unsigned char  kor;
7       double         suly;
8       long long      sszam;
9       };
10
11    int main(void)
12      {
13      struct ADAT ember;
14      strcpy(ember.nev,"Tojas Tobias");
15      ember.kor=22;
16      ember.suly=72.93;
17      ember.sszam=99813342134;
18      return 0;
19      }
```

A program semmi mást nem csinál, mint definiál egy struktúrát, majd létrehoz egy adott struktúra példányt és azt feltölti adatokkal.

Nézzük részletesen!

a 3.-tól a 9. sorig láthatjuk az ADAT ”típusú” struktúrát. A 3. sorban adjuk meg a struktúra ”típus” nevét a `struct` kulcsszó után.

A 4. sorban láthatjuk a struktúra belsejét megnyitó kapcsos zárójelet.

Az 5. sortól deklaráljuk a struktúra mezőit. Jelen esetben ezek a mezők egy 80 karakteres tömb, egy előjeltelen karakter, egy dupla pontosságú lebegőpontos és egy `long long` típusú változók.

A definíciós részt a záró zárójel és a pontosvessző zárja le[34].

A 13. sorban létrehozunk egy példányt az ADAT ”típusú” struktúrából.

[33] Ne igazán tudok jobb kifejezést erre, mint a típus, mert tulajdonképpen ez az, de a későbbiekben látni fogjuk, hogy vannak típusként definiált struktúrák is.

[34] A definíciós részt, vagy más néven struktúra definíciót úgy tekinthetjük, mint az ilyen ”típusba” tartozó struktúrák tervrajzát.

A 14. sort picit tegyük félre.

A 15. sorban az `ember` struktúra `kor` mezőjét töltjük fel. Ekkor a '**.**' mezőhozzáférés, vagy pont operátort használjuk. A feltöltéshez tehát a `ember.kor=22;` kifejezést alkalmazzuk, ami szó szerint azt jelenti, hogy az `ember` struktúra `kor` mezője legyen egyenlő 22-vel.

Egy struktúra egy adott mezője ugyanúgy kezelhető, mint egy a mezővel azonos típusú változó. Tehát lehet neki értéket adni és az értékét felhasználni.

A 16. és 17. sorban a többi mezőt töltjük fel.

A 14. sorban egy függvényt kell használnunk, mert a sztringeknek nem lehet közvetlenül értéket adni. A sztringre való hivatkozás nem a sztring értékével, hanem címével történik. A `strcpy` függvény az argumentumának első tagjában megadott címre másolja a második argumentumban megadott sztringet. Vagyis feltölti a `nev` mezőt.

Ezután a struktúra mezőit pontosan ugyanúgy használhatjuk, mint a "közönséges" változókat.

A struktúra inicializálható a példány deklarálásánál, hasonlóan a tömbökhöz Ez a példánkban a 13. sorban lehetséges a

```
13      struct ADAT ember={"Tojas Tobias",22,72.99,99813342134};
```

módon. Ne feledjük az ilyen jellegű feltöltés csak itt a deklarációnál lehetséges.

Mi történik akkor, ha a struktúra címe áll rendelkezésre és nem a példány. A 14. sortól szúrjuk be a következő sorokat:

```
14      struct ADAT *ept;
15      ept=&ember;
```

Tehát az `ept` egy olyan pointer, amely az `ember` struktúra példányra mutat.

Ekkor használhatjuk az úgynevezett `->` operátort. Ekkor egy mező elérése az alábbi módon történhet:

```
ept->kor=22;
```

Ez a sor szó szerint a következőt teszi:

Az `ept` mutató által címzett struktúra `kor` mezője legyen 22.

A következő kérdés a struktúra mérete. Első közelítésben azt mondhatnánk, hogy a struktúra akkora területet foglal le, mint a benne lévő mezők méretének összege. Ez az állítás nem biztos, hogy igaz, mert a különböző processzorok és környezetek az adatok tárbeli elhelyezkedését optimalizálják adott szempontok szerint és így próbálják a program futását gyorsítani.

Ezért a struktúra vagy egy struktúra példány méretét célszerűen a `sizeof` operátorral kell meghatározni. Vagyis jelen esetben a

`s=sizeof(struct ADAT);`, vagy `s=sizeof(ember);`

kifejezés használata vezet helyes eredményre.

2.9.6. Típusként definiált struktúra

A struktúrák definiálhatók típusként is a `typedef` kulcsszó segítségével. Nézzük erre az előző példát!

```
1   #include    <stdio.h>
2   #include    <string.h>
3   typedef struct
4     {
5     char          nev[80];
6     unsigned char kor;
7     double        suly;
8     long long     sszam;
9     } ADAT;
10
11  int main(void)
12    {
13    ADAT ember;
14    strcpy(ember.nev,"Tojas Tobias");
15    ember.kor=22;
16    ember.suly=72.93;
17    ember.sszam=99813342134;
18    return 0;
19    }
```

Láthatjuk, hogy a struktúra felépítése nem változott meg, de a típus azonosítója az `ADAT` a definíció végére került.

Egy másik eltérés, hogy a 13. sorban lévő deklaráció egyszerűbb (és szebb) lett[35].

A példányt ezután teljesen úgy kezeljük, mint az előzőekben látott ”hagyományos” definícióval megadott struktúrát[36].

[35] Azt, hogy a programozó melyik módszert használja, az az ő egyéni ízlésétől függ.

[36] Van egy apró eltérés ezt nemsokára a rekurzív struktúráknál látjuk majd.

2.9.7. Bitmező struktúrák

A bitmező struktúrák lehetővé teszik, hogy egész jellegű mezőkből úgynevezett bitmezőt hozzunk létre.

A bitmezőt olymódon kell elképzelni, hogy a kívánt (és szigorúan egész jellegű) típusú változóból hány bites mezőt szeretnénk létrehozni. A mező hossza nem lehet nagyobb, mint az eredeti típus hossza[37]

Egy bitmező struktúra definíciója a következő:

```
1     struct bitfield
2        {
3        unsigned short a:  3;
4        unsigned short b:  8;
5        unsigned short c:  4;
6        unsigned short d:  1;
7        };
```

A 3. sorban létrehozunk egy 3 bit szélességű mezőt. A 4. sorban egy 8-bites, az ötödik sorban egy 4-bites és a 6. sorban egy egybites mezőt.

Ha ezt most összeadjuk, akkor pontosan 16 bitet kapunk, ami az `unsigned short` típus mérete. Tehát egy ilyen struktúra két bájtot foglal le a memóriában.

Nem kötelező a mezőknek kiadni az alaptípus méretét, lehet ez rövidebb, vagy hosszabb.

Ha mondjuk a `b` mező most csak 6 bites lenne, akkor az ilyen ”típusú” struktúra még mindig két bájtot foglalna le.

Most feltételezzük azt, hogy a `d` mező 2 bit méretű lenne. Ekkor a bitmezők szélességének összege 17 lenne, ami nagyobb, mint az `unsigned short` mérete. Ha most deklarálunk egy ilyen elemet, akkor annak mérete 4 bájt lenne, mert a fordító ”megkezdett” egy új `unsigned short`-ot.

Az ilyen struktúrák mezőt pontosan ugyanúgy lehet elérni, mint egy hagyományos struktúráét, csak nagyon figyelni kell a számábrázolási tartományra.

Nézzük példának a következő esetet:

```
int i:  1;
```

Kérdés: mi az `i` mező számábrázolási tartománya. A válasz:{`0,-1`}. A válasz indoklását az olvasóra bízzuk.

A bitmezők keverhetők a hagyományos struktúra elemekkel is. Lásd a következő példát:

[37] Valamikor ezt a megoldást arra találták ki, hogy memória helyet takarítsanak meg. Manapság a PC-k világában ez nem ennyire éles, hiszen a memória - akár fizikailag és átvitt értelemben is - nagyon olcsó. A bitmezőket inkább a mikrokontrollerek esetén használjuk egyrészt valóban helytakarékossági szempontból, másrészt a hardver programozásához.

```
1     struct mix
2        {
3        unsigned char a:  3;
4        unsigned char b:  5;
5        float         f;
6        };
```

A példa 5. sorában lévő `float` típusú elem egy ”normál” struktúra mező.

Megjegyzés: lehet, hogy mikrokontrolleres esetben helyet takarítunk meg a bitmezők alkalmazásával, azonban amikor ezeket az elemeket használjuk a processzornak az adatokat tologatnia kell. Ez bizony időbe kerül.

A bitmező struktúrák típusként is definiálhatók.

2.9.8. Rekurzív struktúrák

Az előzőekben láthattuk, hogy egy struktúra bármilyen elemet tartalmazhat, csak olyan struktúrát semmiképpen sem, mint amilyen önmaga.

Viszont tartalmazhat olyan típusú pointert, mint önmaga. Nézzünk erre egy példát:

```
1  struct ITEM
2     {
3     int v;
4     struct ITEM *next;
5     };
```

Ez a struktúra tehát két mezőt tartalmaz a 3. sorban lévő `v` nevű egészt és egy olyan típusú pointert, mint önmaga.

Megjegyzés: ez a struktúra az úgynevezett lista adatszerkezet egy elemét írja le. A lista adatszerkezet felépítése olyan, hogy van egy adatrésze (nem okvetlenül egy elemű), és van egy mutatója, amely a következő elemre mutat.

A lista csak elölről olvasható, visszafelé nem és végét a C nyelvben úgy jelezzük, hogy az utolsó elem `next` pointere egy úgynevezett `NULL` pointer[38].

A rekurzív struktúrák nem definiálhatók típusként, ennek oka, hogy a típus a definíció végén derül ki. Így fordítás során a fordító program a mutató típusát nem fogja felismerni.

[38] A `NULL` pointert például így lehet előállítani: `#define NULL ((void *)0)`

2.10. Unionok

A union az egyik legfurcsább szintaktikai elem. Formailag nagyon hasonlóak a struktúrához, de teljesen más célra használjuk.

Míg a struktúra arra szolgál, hogy különböző elemeket egy egységként tudjunk kezelni, a union célja az, hogy egy adott memória területet különböző típusként érhessünk el. Nézzünk erre egy példát!

A union definíciója:

```
1     union SAMPLE
2       {
3       float a;
4       int b;
5       };
```

Mint látjuk a union definíciója nagyon hasonlít egy struktúra definíciójához.

A union típusa jelen esetben SAMPLE , lásd 1. sor.

A 3. sorban egy float mezőt definiálunk a néven.

a 4. sorban egy int típusú mező definiálása látható.

A definíció után deklaráljunk egy union SAMPLE típusú változót.

```
union SAMPLE minta;
```

Ekkor már van egy valós változónk. Ha ez struktúra lenne, akkor a minta mérete legalább akkora lenne, mint a `float` és az `int` változó méretének összege[39].

Azonban jelen esetben a `sizeof(minta)` csak 4-es értéket ad. A union két mezője egyazon területet definiál. Ha a program során az írjuk, hogy

```
...  minta.a ... ,
```

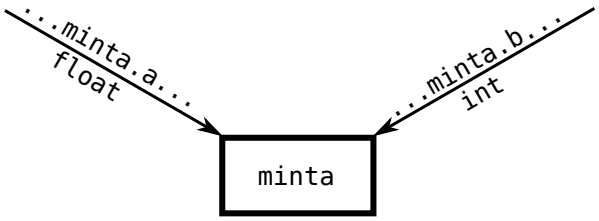
akkor az adott memória területhez `float`-ként férünk hozzá.

A következő kifejezés esetén

```
...  minta.b ... ,
```

[39] Ez a gcc esetén 4+4 bájt, vagyis 8 bájt lenne.

a minta változó területét `int` típusként használjuk[40].



9. ábra: A `minta` példa grafikusan

Sokkal életszerűbb az a megvalósítás, amikor egy `union`-t arra használunk fel, hogy egy nagyméretű változót bájtokra bontsunk adott cél érdekében.

Ez az előző `float` estén így nézhet ki:

```
union convert
  {
  float f;
  char  b[4];
  };
```

Mint láthatjuk a unionba `f` mezőként elhelyezett `float` típusú változó bájtokra bontva a `b` tömbben kapható meg.

Felmerül a kérdés, hogy mi történik abban az esetben, ha a két, vagy több mező mérete nem azonos. Ekkor:

- a legnagyobb mező határozza meg a union méretét,
- a mezőket a union kezdőcímétől számítjuk.

Definiáljunk egy ilyen uniont!

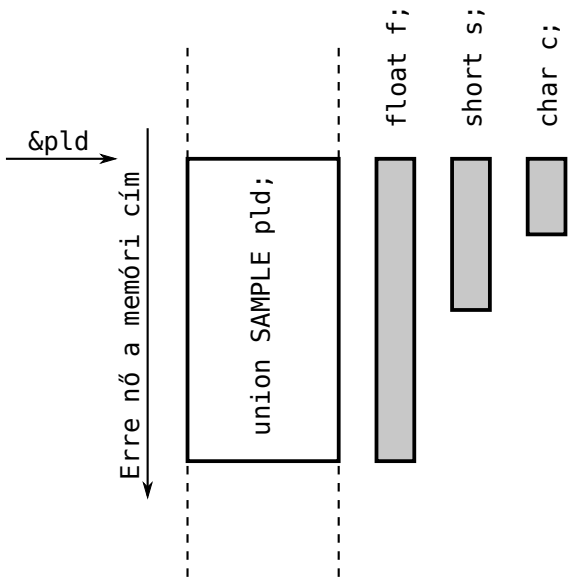
```
union SAMPLE
  {
  float f;
  short s;
  char c;
  };
```

[40] Ez a példa igazából nem túl értelmes, csak a union használatát mutatja be

Majd deklaráljunk egy példányt belőle!

```
union SAMPLE pld;
```

A következő ábra azt mutatja, ahogyan a különböző mezők ”látják” a memóriát.



10. ábra: A eltérő méretű mezők helye a unionban

2.10.1. Típusként definiált unionok

A unionok is definiálhatók típusként. Nézzük példaként a `SAMPLE union` definiálását és egy példány deklarálását ilyen módon!

```
typedef union
   {
   float a;
   int b;
   } SAMPLE;
```

és

```
SAMPLE minta
```

Láthatjuk, hogy ez is hasonlóan struktúrákhoz kicsit egyszerűbb lett. A mezők hozzáférése teljesen azonos, mint a hagyományos definíció esetén

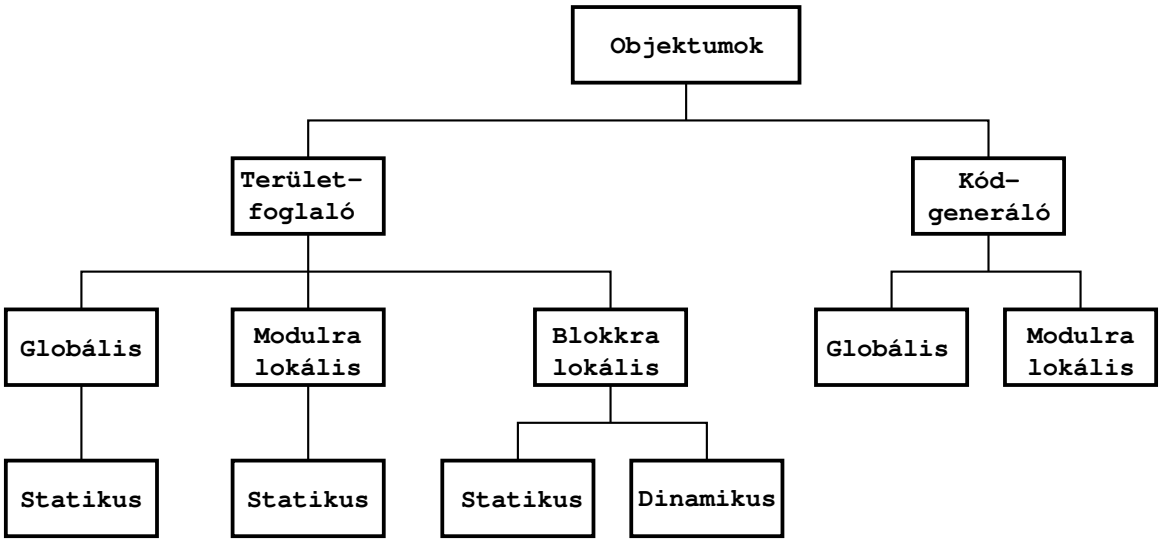
2.11. Modulok, moduláris programozás

A c programozási nyelvet eredetileg azzal a céllal hozták létre, hogy operációs rendszereket fejlesszenek benne. Azonban egy operációs rendszer forrása több 10, esetleg 100 ezer programsorból áll. Ezt nyilvánvalóan nem lehet egyetlen forrásban tárolni, mert akkor ez fájl kezelhetetlen lenne. Ez ért a programot külön forrás részletekre bontjuk. Ezeket a különálló programrészeket nevezzük modulnak.

FIGYELEM! A modul nem tévesztendő össze az include fájllal. A modul önállóan fordul forrásállományból az úgynevezett object állományba Lásd 6. ábra!

A modulok lehetővé tették azt, hogy a programozó jól definiált és alaposan letesztelt előzőleg már megírt program részleteket használjon.

Mivel várhatóan egy nagyobb programot több programozó készít, gondoskodni kell arról, hogy a változó és kódnevek fölöslegesen ne keveredjenek. A modulok ennek a problémának a megoldását is lehetővé teszik. Lehetőség van egyes függvények és változók elrejtésére.



11. ábra: A C objektumai

A 11. ábra a C nyelv objektumait ábrázolja.

Definíció: a területfoglaló objektumok a változó jellegű objetumok, úgymint:

változók,

- mutatók,
- tömbök,
- struktúrák,
- unionok.

Definíció: a kódgeneráló objektumok a függvények.

Definíció: egy változó **statikus**, ha a változó területe fordításkor jön létre. Amennyiben egy statikus változót a deklarációnál inicializálunk az a program során egyszer és csakis egyszer kap kezdeti értéket[41].

MEGJEGYZÉS: természetesen a változó értéket kaphat. Ez csak a kezdeti inicializálásra vonatkozik.

Definíció: egy változó **dinamikus**, ha a változó területe futásidőben jön létre és futásidőben is szűnik meg.

Amennyiben egy dinamikus változót a deklarációnál inicializálunk az annyiszor kap kezdeti értéket, ahányszor a változó létrejön.

A következő példa segít a további fogalmak megértésében. A program három modulból áll, ezek az `1.c`, a `2.c` és a `3.c` források.

[41] Az esetlegesen még nem definiált fogalmakat ezen szakasz után tisztázzuk.

```
1 #include <stdio.h>
2 void fgv1(void);
3 void fgv2(void);
4 char txt[]="G text\n";
5 int main(void)
6   {
7   fgv1();
8   fgv2();
9   return 0;
10  }
```

1.c

```
1 #include <stdio.h>
2 void fgv1(void);
3 static void fgv2(void);
4 extern char txt[];
5 void fgv1(void)
6   {
7   printf("%s",txt);
8   fgv2();
9   }
10 void fgv2(void)
11   {
12   printf("ML fgv2\n");
13   }
```

2.c

```
1 #include <stdio.h>
2 void fgv2(void);
3 void fgv3(void);
4 static char txt[]="N text\n";
5 void fgv2(void)
6   {
7   printf("G fgv2\n");
8   printf("%s",text);
9   fgv3();
10  }
11 void fgv3(void)
12   {
13   char txt[]="B text\n";
14   printf("%s",txt);
15   }
```

3.c

Az program az 1.c modulban található `main` függvény meghívásával indul. A `main` először meghívja az `fgv1` függvényt (7. sor), amely a `2.c` modulban található.

Figyeljük meg, hogy az 1.c modulban az `fgv1` deklarálásra került (2. sor).

A 2.c modulban található `fgv1` függvény definíciója (5-9. sor). Ez először kiírja a `txt` nevű sztringet. Ez a sztring **globális** változóként került deklarálva az `1.c` modulban (4. sor).

Azonban ezt a sztringet deklarálni kell abban a modulban is, ahol használni szeretnénk. Tehát egy dek-

larációra a 2.c modulban is szükség van, különben a fordító olyan hibajelzést küld, hogy nem deklarált változót használunk. Ez a deklaráció viszont nem történhet ugyan úgy, mint azt az `1.c`-ben láttuk, mert ez egy új változó terület lefoglalását jelentené.

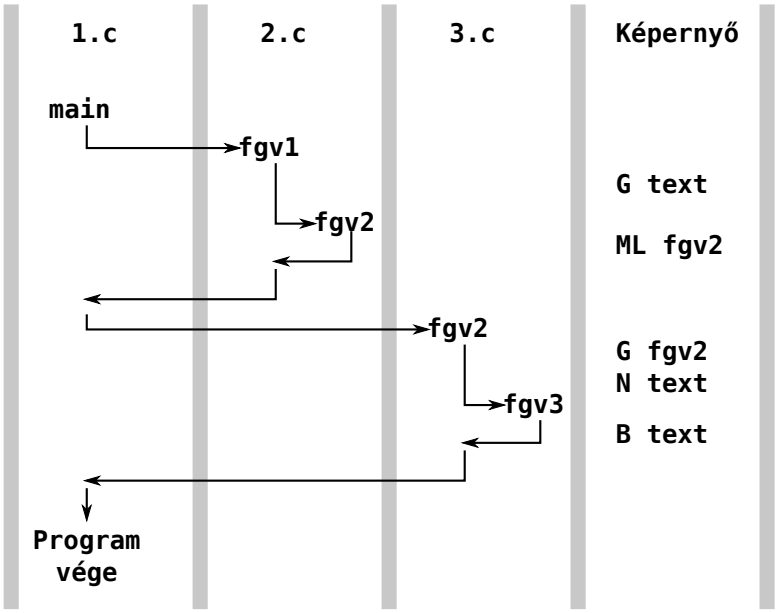
Ezért a `txt` deklarációja a `2.c` modulban egy **`extern`** előtaggal történik. Ez azt mondja a fordító programnak, hogy van már ilyen nevű változó terület deklarálva, de nem az aktuális modulban, hanem valahol egy másikban.

Ezután a `2.c`-ben az `fgv1` meghívja a `2.c`-ben található `fgv2` függvényt(10. sor). Az ért ezt hívja meg és nem a `3.c`-ben lévőt, mert a `2.c`-ben lévő `fgv2` függvény **`static`** előtaggal deklaráltuk. Ezáltal ez a függvény modulra lokális lett.

Ezután a program visszatér az `1.c` modulba, ahonnan meghívja a `3.c` modulban található `fgv2` függvényt.

A `3.c` modulban az `fgv2` függvény kiírja, hogy `G fgv2` és a `txt` változót. Azonban ez a `txt` az adott modulban egy **`static`** előtaggal rendelkezik (4. sor). Ez azt jelenti, hogy ez egy **modulra lokális** változó. és a globális `txt` sztringtől különböző változó területet foglal le.

Ezután a `3.c`, `fgv2` meghívja az `fgv3`-t, amely kiír újra egy `txt` sztringet, amely viszont az `fgv3`-ban lett deklarálva (12. sor). Ekkor ez a `txt` egy **blokkra lokális** (és dinamikus) változó.



12. ábra: A program működése

Definíció: Egy változó `globális`, ha minden modulból megvan a lehetősége, hogy elérhető legyen.

- ezt a változót a teljes program folyamán egyszer és csakis egyszer egy modulban, minden függvényen kívül deklarálni kell úgy, hogy a deklaráció nem kap semmilyen előtagot. Ekkor ebben a modulban történik meg a változó helyének lefoglalása.

- amennyiben a kérdéses változót egy másik modulból el akarjuk érni, akkor az adott modulban a változó deklarációja elé egy `extern` kulcsszót kell tenni. Ekkor új memória lefoglalás nem történik.
- ezt a változót csak abban a modulban lehet deklarációkor kezdőértékkel ellátni, amelyben a változó terület lefoglalásra kerül[42].

  a globális változók statikusak.

Definíció: egy változó **modulra lokális**, ha a kérdéses változó a modul összes függvényéből elérhető (kivételt lásd lejjebb), de más modulokból ez a változó nem látható.

- a modulra lokális változók deklarációja esetén a deklaráció kap egy **`static`** előtagot,
- a modulra lokális változók statikusak.

Definíció: egy változó **blokkra lokális**, ha érvényességi köre csak egy függvényre korlátozódik,

- a blokkra lokális változók deklarációja az adott függvény törzsének az elején található,
- a blokkra lokális változók lehetnek dinamikusak és statikusak.

Felmerül a kérdés, hogy mi történik akkor, ha azonos néven létezik egy globális, modulra lokális és egy blokkra lokális változó. Melyiket látja a kérdéses függvény.

A válasz az, hogy mindig azt a változót látjuk, amely közelebb van. Tehát sorrendben a blokkra lokális a modulra lokális és a globális a láthatósági sorrend[43]

Definíció: egy függvény **globális**, ha meg van annak a lehetősége, hogy tetszőleges modulból elérhető legyen.

- a globális függvényt célszerű mindazon modulokban deklarálni, ahol használni szeretnénk.
- a globális függvényt egy és csakis egy modulban szabad és kell is definiálni,
- a globális függvény deklarációjánál semmilyen előtagot sem alkalmazunk.

Definíció: egy függvény **modulra lokális**, ha az az adott modul minden függvényéből meghívható, de más modulokból nem.

- a modulra lokális függvény deklarációja elé egy **static** kulcsszót kell tenni,

[42] Tehát az `extern int a=5` **Nem helyes**!
[43] Persze. Miért is hoznánk létre olyan változót, amit nem látunk.

- a modulra lokális függvényt az adott modulban definiálni kell.

Abban az esetben, ha egy modulban szerepel egy olyan nevű modulra lokális függvény, amely néven a programban azonos néven globális függvény is szerepel, akkor a modul függvényeiből csak a modulra lokális érhető el[44].

Lehetőség van **blokkra lokális statikus változó** deklarálására is. Ekkor a következő módon kell a változót deklarálni az adott függvényben:

```
void fgv(void)
  {
  static int i;
      :
      :
```

Vagyis a változót ugyanott a függvényen belül deklaráljuk, de elé tesszük a **`static`** kulcsszót. Ekkor a példában szereplő `i` változó statikus lesz, tehát fordítás időben jön létre.

Fontos!! Ha egy blokkra lokális statikus változót hozunk létre, akkor ez a változó épp úgy, mint a blokkra lokális dinamikus változó csak és kizárólag az adott függvényben látható. De a statikus változó egyszer és csakis egyszer jön létre - mégpedig fordítás időben - akárhányszor is hívjuk meg a függvényt.

A blokkra lokális dinamikus változók esetén a változó terület mindig újra létrejön és megsemmisül a hívások és kilépések során.

A fentieknek két következménye van:

- a blokkra lokális statikus változó nem veszti el az értékét két függvényhívás között, míg a blokkra lokális dinamikus igen,
- a blokkra lokális statikus változó deklarációkor történő értékadása csak egyszer és csakis egyszer történik meg, míg a blokkra lokális dinamikus változónál ez a művelet minden függvényhívásnál végrehajtódik.

```
void fgv(void)
   {
   static int i=5;
         :
         :
```

Statikus eset

```
void fgv(void)
   {
   int i=5;
        :
        :
```

Dinamikus eset

[44] Ez van közelebb.

A változók működésének bemutatására írjunk egy olyan rekurzív faktoriális számoló függvényt, amely kiírja, hogy mekkora a rekurzió mélysége[45].

Definíció: rekurzívnak nevezzük azt a függvényt, amely önmagát hívja meg.

```
1  unsigned fact(unsigned n)
2    {
3    static int cnt=0;         // Ez a melyseg szamlalo
4    unsigned r;               // Seged valtozo
5    if(n<=1)
6      {
7      cnt++;
8      printf("%i\n",cnt);
9      return 1;
10     }
11   cnt++;
12   r=n*fact(n-1);            // A tenyleges rekurziv hivas
13   return r;
14   }
```

A 3. sorban a `static int cnt=0;` kifejezés egy blokkra lokális statikus változót hoz létre, amelyet `0` értékre inicializál Ez egyszer történik meg a program során.

A 4. sorban az `r` változó egy blokkra lokális dinamikus változó.

Az 5. sorban megvizsgáljuk, hogy az `n` változó értéke `1`, vagy esetleg `0`. Ha igen, akkor a függvény `1` értekkel tér vissza.

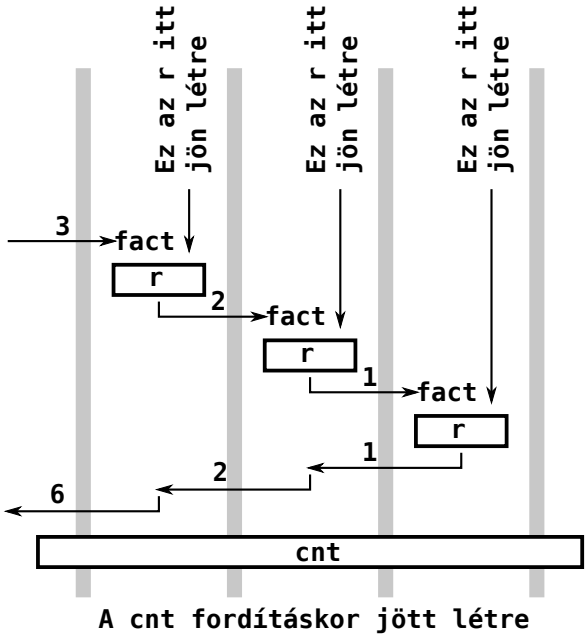
Ha `n` értéke 1-nél nagyobb, akkor a függvény inkrementálja a `cnt` változót és a 12. sorban meghívja önmagát eggyel csökkentett értékkel. A függvény visszatérési értékét az `n` értékkel megszorozzuk.

Ez mindaddig így megy, amíg az átadott paraméter értéke nem lesz `1`. Ekkor a rekurzívan meghívott függvények elkezdenek visszatérni.

Tegyük fel, hogy `3!` szeretnénk kiszámolni[46].

Láthatjuk, hogy a `cnt` változó területe minden hívásnál ugyanaz. Az `r` változó viszont minden hívásnál újra létrejön.

[45] Azn faktoriálisa `n!` a következő: $n! = 1 * 2 * \ * n$ definíció szerűen: $0! = 1,\ 1! = 1,,\ 2! = 2 * 1,,\ n! = n * (n-1)!$.
[46] Ez nem programlista, hanem egyfajta működési leírás.



13. ábra: A változók a rekurzív függvényhívásban

A jobbra mutató nyilakon lévő számok az átadott paraméter értéket, a balra mutató nyilakon lévő számok a függvény aktuális hívásából származó visszaadott értéket mutatják.

A vastag szürke vonalak közötti sáv azt mutatja, hogy a függvény belső, aktuális `r` - tehát az aktuális függvényhíváskori - változója mikor és hol látszik a program futása során.

3. I/O kezelés

Az I/O kezelés a UNIX rendszerekben alapvetően fájl kezelést jelent. Mivel a C és a UNIX mondhatni szimbiózisban van, ezért mi is ezt a fogalomkört használjuk.

A UNIX jellegű rendszerekben bármilyen eszközt szeretnénk elérni, azt fájl elérésként tehetjük meg[47].

Ebbe most részletesebben nem megyünk bele, mert a jegyzet tárgya nem a UNIX jellegű rendszerek kezelése, hanem a C programozási nyelv.

3.1. Standard I/O

A UNIX jellegű rendszerek három standard fájlt használnak, ezek:

- standard input, amely általában a billentyűzet (`stdin`),
- standard output, amely általában a képernyő (`stdout`),
- standard error, amely általában szintén a képernyő (`stderr`).

Az `stdin`-t és az `stdout`-ot speciálisan erre a célra írt függvényekkel kezelhetjük. Az `stderr` kezelésére a később ismertetésre kerülő fájlkezelő függvények szolgálnak.

3.1.1. `printf`

A printf függvény az `stdout`-ra ír ki formátumozottan. Deklarációja a `stdio.h` header fájlban található és egyszerűsített formában a következő:

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

A függvény első paramétere az úgynevezett formátum sztring . Ebben adjuk meg azt, hogy a következő paraméter listát a `printf` hogyan értelmezze.

A formátum sztring tartalmazhat formátum specifikátorokat és egyéb szövegelemeket. A formátum specifikátort a függvény értelmezi. Azokat a részeket, amelyeket nem lehet specifikátorként értelmezni a `printf` kiírja.

A következő argumentum a deklarációban a `...`. Ez azt jelenti, hogy a függvény ezután tetszőleges számú és típusú argumentum elemet tartalmazhat. Ezek értelmezése a kiíratásnál a formátum sztring alapján történik.

A formátum specifikátor szerkezete:

[47] Bármilyen ...X UNIX, Linux, QNX, stb.

```
%[flags][width][.prec][length]type
```

A nem kötelező elemeket `[ ]` pár közé tettük[48].

Mint láthatjuk a formátum specifikátor minden esetben egy `%` jellel kezdődik. És a típus (`type`) kötelező, a többi megadható paraméter opcionális.

Elsőnek nézzük a legfontosabb típusokat!

`d,i` decimális egész kiíratás előjellel,

`u` decimális előjeltelen egész kiíratás,

`o` oktális előjeltelen egész kiíratás,

`x,X` hexadecimális előjeltelen egész kiíratás,

`f,F` egyszeres pontosságú lebegőpontos kiíratás 123.456 formában,

`e,E` egyszeres pontosságú lebegőpontos kiíratás 1.23456e2 , vagy 1.23456E2 formában,

`g,G` egyszeres pontosságú lebegőpontos kiíratás az előző két forma közül abban, amelyik a rövidebb[49],

`c` egy karakter kiíratása,

`s` az argumentumban megadott című sztring kiíratása a lezáró nulla értékű karakterig,

`p` a megadott mutató értékét írja ki hexadecimális formában.

Tehát, ha például egy karaktert akarunk kiíratni, akkor ez a következő módon történik:

```
printf("%c",chr);
```

ahol a karakter változó neve `chr`.

A **flag** karakterek a következők:

`#` típustól és formátumtól függő speciális kiíratást ír elő:

- **`o`** az oktális szám elé kiírja a `0` karaktert,
- **`x, X`** a hexadecimális szám elé kiírja a `x`, vagy `X` karaktert,
- **`f, F, e, E, g, G`** esetén mindenképpen lesz tizedesjegy.

más esetekben ez a flag hatástalan.

[48] Csak a leírásban, amikor alkalmazzuk ezeket, akkor nem kell a `[ ]` pár.
[49] Első pillantásra az `f,F` a rövidebb, de ha a 0.0000000001 számot vesszük, akkor az `e,E` a rövidebb (1e-10).

0 a kiíratási méretnek megfelelően a kiírandó számokat feltölti 0 -kal. Az egészeket a bal oldalról a lebegő pontosakat a jobb oldalról,

szóköz pozitív számok elé egy szóközt szúr be,

- a kiirtást balra rendezi. Az alapértelmezett a jobbra rendezés.

+ a pozitív számok esetén a kiírásnál egy + jelet tesz a szám elé.

A `width` mező megadja a minimális kiíratás szélességét. Ha a kiíratás valamilyen okból nagyobb, mint amit előírtunk, akkor a kiíratás felülírja ezt az értéket.

A **.prec** mező hatása szintén típusfüggő:

f, F, e, E, g, G esetén a kiírandó tizedesjegyek számát adja meg[50],

d, i, u, o, x, X a kiírandó számjegyek számát adja meg úgy, hogy balról a számot nullákkal tölti fel,

s, S a kiírandó sztring maximális hosszát adja meg.

A **length** mező a kiíratás pontosságát határozza meg:

h előjeles, vagy előjeltelen egészek esetén short értéket ír ki,

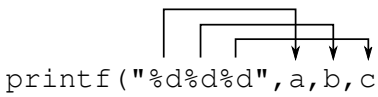
hh előjeles, vagy előjeltelen egészek esetén egy bájtos (egész) értéket ír ki (signed char , vagy unsigned char),

l előjeles, vagy előjeltelen egészek esetén long érteket ír ki,
lebegőpontos változok esetén double érteket ír ki,

ll előjeles, vagy előjeltelen egészek esetén long long értéket ír ki,

L lebegőpontos változok esetén (f,F,e,E,g,G) long double értéket ír ki.

A `printf` függvény egyszerre több értéket is kiírhat. A kiírás módját a formátum sztring határozza meg. A formátum specifikátorok sorban meghatározzák a kiírandó értékeket a következőképpen:

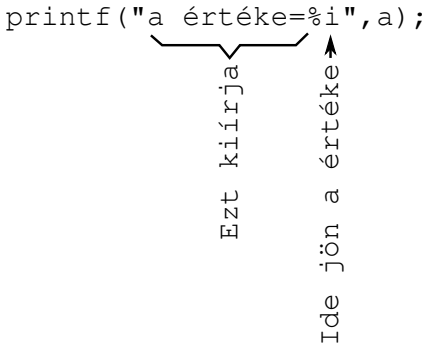


A függvény először kiírja `a`, majd `b` és végül `c` értékét.

Természetesen a függvény egyszerre nem csak egyfajta típust tud kiírni.

[50] Lebegőpontos számok esetén a kiírandó tizedesjegyek alapértelmezett száma hat. Természetesen ennél pontosabb értékkel számol a gép.

Ha a formátum sztringben specifikátornak nem értelmezhető karakterek vannak, azokat a függvény az adott helyen kiírja. Ahol formátum specifikátort talál, azon a pozíción a megfelelő változó értékét írja ki:



Megjegyzés: a `printf` a formátum sztringben található formátum specifikátorból tudja, hogy az adott változót hogyan kell kezelni, és nem törődik azzal, hogy az argumentumban mi a valójában a változó típusa.

Bár néhány okosabb fordító esetleg figyelmeztetést küld (lásd gcc), de ezzel már futás időben nincs mit tenni. Ekkor a kiíratás természetesen hibás lesz.

A fentiek alapján nézzünk néhány példát! Minden példa esetén a kiíratás a képernyő szélétől történik. A szóközöket a □ karakterek, a cursort a ■ karakter, a terminál szélét a || karakter jelzi. Legyen a kiírandó érték 123 és egész!

Legyen a formátum specifikátor: `%5i`

```
printf("%5i",123);                    ||□□123■
```

Legyen a formátum specifikátor: `%-5i`

```
printf("%-5i",123);                   ||123□□■
```

Cseréljük ki a `-` flag-et `+` -ra!

```
printf("%+5i",123);                   ||□+123■
```

Használjuk a szóköz flag -et!

```
printf("% i",123);                    ||□123■
```

Ugyanezt -123 -ra!

```
printf("% i",-123);                   ||-123■
```

Most használjuk a `.prec` mezőt!

```
printf("%5.4i",123);                  ||□0123■
```

Látható, hogy a szám elé egy 0 át odaír, mert a `.prec` egész jellegű változó esetén a minimálisan kiírandó számjegyek számát adja meg.

Legyen a formátum specifikátor újra: `%5i`

```
printf("%5i",123456789);                    ||123456789■
```

Látható, hogy a megadott szélességet, az 5-öt, a kiíratás szélessége meghaladta. Ekkor nem csonkul a kiírás, a teljes számot kiírja.

Nézzünk példákat a lebegőpontos kiíratásra! A szám 123.456.

Legyen a formátum specifikátor: `%f`

```
printf("%f",123.456);                       ||123.456■
```

Cseréljük le az `f` -et `e` -re, majd `E` -re!

```
printf("%e",123.456);                       ||1.23456e2■

printf("%E",123.456);                       ||1.23456E2■
```

Legyen a formátum specifikátor: `%.2f`

```
printf("%.2f",123.456);                     ||123.45■
```

Láthatjuk, hogy a hatos már lemaradt, mert lebegőpontos számok esetén a `.prec` mező a kiírandó tizedesjegyek számát adja meg.

Használjuk a `0` flag-et!

```
printf("%0f",123.456);                      ||123.456000■
```

Ekkor a tizedesjegyek száma hat lesz, mert ez a default kiíratási méret. A program természetesen ennél pontosabban számol, ez csak kiíratás.

Természetesen a kiíratási pontosság felülírható a `.prec` mezővel.

Most írassunk ki egy sztringet és használjuk a `.prec` mezőt!

```
printf("%.5s","abcdefghij");                ||abcde■
```

Láthatjuk, hogy csak az első öt kartert írta ki, mert sztringek esetén a a maximálisan kiírandó karakterek számát adja meg a `.prec`.

Gyakorlati tanács: a `printf` függvény (igazából a standard kimenet) alapvetően úgy működik, hogy vagy összevár egy adott mennyiségű karaktert, hogy ezeket egyben kiírja, vagy egy adott ideig vár, ha ez a mennyiség nem jött össze és akkor írja ki az információt. Ez általában vagy 4096 karakter, vagy 30

másodperc.

Azért, hogy ezt az időt ne kelljen kivárni - mondjuk hibakeresés esetén - használhatjuk az `fflush` függvényt. Az `fflush` a kérdéses fájlt, esetünkben a standard kimenetet kényszeríti a puffer azonnali kiírására Ez azt jelenti, hogy az információ azonnal megjelenik a képernyőn. Használata:

```
fflush(stdout);
```

3.1.2. `scanf`

A `scanf` függvény a standard bemenetről olvas elemeket. A standard bemenet az esetek többségében a billentyűzet.

A függvény deklarációja az `stdio.h` header fájlban található és egyszerűsített formája a következő:

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

A függvény egész típusú. A visszatérési értéke azt adja meg, hogy hány elemet olvasott be sikeresen.

Figyelem nem azt hogy hány karaktert olvasott be. Ha egy ezer karaktert tartalmazó egyetlen sztringet olvasunk be sikeresen, akkor 1-et ad vissza.

Hasonlóan a `printf` függvényhez a függvény első argumentuma a formátum sztring. Ez határozza meg, hogy az argumentum további részében megadott változók címeit hogyan használja a `scanf`.

Nagyon lényeges a `scanf` függvény esetén az argumentum további részében nem a változókat, hanem a változók címeit kell megadni.

A formátum sztring tartalmazhat formátum specifikátort és tetszőleges karaktereket. A formátum specifikátort értelmezi, az egyéb szöveg elemeket figyelmen kívül hagyja.

A formátum specifikátor felépítése a következő:

```
%[*][width][length]type
```

A `*` mező azt jelenti, hogy a `scanf` beolvassa az adatot a standard bemenetről, de nem konvertálja és nem adja vissza[51].

A `width` mező meghatározza a maximálisan beolvasható karakterek számát.

A `length` mező teljesen megegyezik a `printf` függvénynél látottakkal.

[51] Tulajdonképpen ezt a beolvasandó mezőt átlépi.

A típusok - `type` - gyakorlatilag megegyeznek a `printf` függvénynél ismertetettekkel (lásd 64. oldal). Az eltérés az, hogy a lebegőpontos számok beolvasásánál az `f,e,E,g,G` csak azt jelenti, hogy valamilyen formában lebegőpontos számot olvasunk be.

A `scanf` használata azonban nem egyszerű. Vegyük példának a következő esetet!

Adott egy karakter tömb, amelyben egy teljes (vezeték és kereszt) nevet szeretnénk tárolni. Kérjük be a nevet!

A jelölésrendszer azonos a `printf` függvénynél látottakkal. Az `ENTER` billentyű leütését a ↩ karakter szimbolizálja.

```
scanf("%s",name);
```

Ekkor beírjuk, hogy

```
||Tojás␣Tóbiás↩
```

Látjuk, hogy a név belsejében egy szóköz karaktert ütöttünk. Ha az eredményt kiíratjuk csak azt láthatjuk, hogy

```
||Tojás■
```

Ez azért következett be, mert a szóköz karakter egy adott mező beolvasását lezárja.

Megjegyzés: a sor beolvasása csak az `ENTER` billentyű leütésére következik be, de a mezőt a szóköz és a tabulátor karakter is lezárja[52].

Olvassunk be három egész értékű változót! A beolvasás:

```
scanf("%i%i%i",&i,&j,&k);
```

Ne feledkezzünk el a címképző operátorokról a változók előtt.

Ekkor beírhatjuk a számainkat többféleképpen is.

[52] Sajnos.

a) ||1␣2␣3↩

b) ||1␣2↩
||3↩

c) ||1↩
||2␣3↩

d) ||1↩
||2↩
||3↩

A szóközök tabulátor karakterrel is helyettesíthetők.

A példából látható, hogy legalább négyféleképpen bevihetjük a számokat a beolvasás során. Ez komolyan zavaró lehet.

A beolvasás formátuma a formátum specifikátortól függ. Ha a változó típusa, amelynek címét megadjuk a kérdéses helyen nem felel meg a formátum specifikátorban megadott típusnak, akkor esetleg a fordítás során kapunk egy figyelmeztetést. A futás során viszont a `scanf` azt fogja csinálni, amit megadtunk neki. Például ha azt írtuk elő, hogy olvasson be egy `double` változót, de egy karakter típusú változó címét adjuk meg, akkor arra címre olvas be egy 8 bájt hosszúságú adatot. Ekkor nagy valószínűséggel a program hibával kilép, vagy esetleg lefagy.

Tipikus másik veszély a sztringek beolvasásánál a sztring túlírása. Ez azt jelenti, hogy több karaktert akarunk beírni az adott helyre, mint amekkora hely rendelkezésre áll, ezen a problémán segíthet a `width` mező alkalmazása.

3.2. Fájl kezelés

Definíció: fáj az a logikailag és fizikailag összefüggő állomány, amely valamely háttértárolón foglal helyet.

A C nyelv a UNIX operációs rendszerből örökölten a fájlokat egy dimenziós bájt tömböknek - ha jobban tetszik - karakter tömböknek tekinti.

Definíció: fájl pozíció mutatónak nevezzük azt az index értéket, amely megmutatja, hogy a fájl elejétől számítva melyik pozíción hajtódik végre a következő művelet.

A fájl pozíció mutató alapműveletek (írás - olvasás) automatikusan csak növekedhet az érintett bájtok számától függően. Egyéb mozgatására speciális függvényeket kell alkalmazni.

A fájl használatához először a kérdéses fájlt meg kell nyitni. A megnyitás alapvetően történhet:

olvasásra Ekkor az állományból csak olvashatunk.

írásra Ekkor az állományba csak írhatunk.

írásra - olvasására Ekkor mindkét műveletet végezhetjük.

hozzáfűzésre Ekkor tulajdonképpen az állományt írásra nyitjuk meg, de a fájl pozíció mutató a fájl végére mutat.

Néhány környezet különbséget tesz úgynevezett text és bináris fájlkezelés között.

A text jellegű fájl kezelés esetén a fájlban található `CR LF` szekvenciákból a beolvasás után csak az `LF` marad meg, illetve a kiíráskor egy `LF` karakter esetén `CR LF` karakterek kerülnek az állományba

A bináris jellegű fájl kezelésnél a beolvasás és a kiírás esetén a rendszer mindent beolvas és kiír a tartalomtól függetlenül.

A C programozási nyelv kétféle fájlkezelést valósít meg, ezek: az alacsony szintű és a magas szintű fájlkezelés. A mindkét mód az alap fájlkezelő műveleteket ismeri, azonban szolgáltatásaik különböznek.

A két fájlkezelési mód átjárható. Vagyis egy alacsony szinten megnyitott fájl egy függvényhívás után magas szinten is kezelhető és viszont.

3.3. Alacsony szintű fájlkezelés

Az alacsony szintű fájlkezelés esetén a fájlt egy úgynevezett fájl leíró azonosítja. Ez a fájl leíró egy int típusú változó[53].

Az alacsony szintű fájlkezeléshez szükséges header állományok:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Windows-os környezetben még szükséges lehet a

```
#include <io.h>
```

Nem ismertetünk minden függvényt, csak a legfontosabbakat. Ezek viszont bőven elegendők a fájlkezeléshez.

3.3.1. `open` függvény

Az `open`, mint azt neve is mutatja egy fájl megnyitására szolgál, deklarációja a következő:

[53] Ez tulajdonképpen egy index változó, amely az rendszer adott területén található struktúrákból álló tömb egy adott elemének az indexe. A kiválasztott struktúra tartalmazza a fájl kezelésével kapcsolatos összes információt.

```
int open(const char *path,int flags,mode_t mode);
```
vagy:

```
int open(const char *path,int flags);
```

A függvény első paramétere `path` a függvény elérési útja. Itt adjuk meg szövegként, hogy melyik fájlt szeretnénk megnyitni[54].

A második paraméter a `flags`. Ezzel állítjuk be, hogy a megnyitás milyen módon történjen A legfontosabb flag-ek a következők:

`O_RDONLY` a kérdéses állományt csak olvasásra nyitjuk meg,

`O_WRONLY` a kérdéses állomány csak írásra nyitjuk meg,

`O_RDWR` a kérdéses állományt írásra és olvasásra nyitjuk meg,

`O_APPEND` a kérdéses állományt hozzáfűzésre nyitjuk meg,

`O_CREAT` ha az állomány nem létezik, akkor az `open` függvény létrehozza,

`O_DIRECT` a fájl megnyitása után minimalizálja az átmeneti tár méretét és a lehető leggyorsabban küldi ki a fájlba írt adatokat,

`O_EXCL` ha már egy olyan fájlt szeretnénk létrehozni, amely létezik, akkor hibaüzenetet kapunk,

`O_NONBLOCK` ha a olvasás történik, de a fájl esetleg üres az olvasó függvény nem fog rajta várakozni,

`O_TRUNC` ha írásra (és olvasására) nyitunk meg egy fájlt, akkor 0 -ra csökkenti a méretét,

`O_BINARY` a fájlt bináris módban nyitja meg,

`O_TEXT` a fájlt text módban nyitja meg.

Ezek a paraméterek valóban flag bitek. Abban az esetben, ha összetett paraméterezést szeretnénk megadni, azt bitenkénti vagy kapcsolattal tehetjük meg. Például hozzunk létre egy állományt, nyissuk meg írásra és ha már volt előzőleg állomány, akkor vágjuk azt 0 hosszúságúra. Ekkor a `flag` mező:

```
O_CERAT | O_WRONLY | O_TRUNC
```

A harmadik mező a `mode` mező. Használata nem kötelező. Itt állíthatjuk be a megnyitott, illetőleg létrehozott fájl attribútumát. Ez a mező operációs rendszer függő.

Linux esetén, ezek:

`S_IRWXU` a fájl tulajdonosa olvasási, írási és végrehajtási joggal rendelkezik,

[54] Ne feledkezzünk el azonban arról, hogy windows-os környezetben a könyvtárakat \karakterrel választjuk el. A \a C-ben azt jelenti, hogy speciális karakter következik. Ekkor ha konstansként adjuk meg a fájl nevét, akkor a \helyet \\írjunk.

S_IRUSR a fájl tulajdonosa olvasási joggal rendelkezik,

S_IWUSR a fájl tulajdonosa írási joggal rendelkezik,

S_IXUSR a fájl tulajdonosa végrehajtási joggal rendelkezik,

S_IRWXG a fájl tulajdonosának csoportja olvasási, írási és végrehajtási joggal rendelkezik,

S_IRGRP a fájl tulajdonosának csoportja olvasási joggal rendelkezik,

S_IWGRP a fájl tulajdonosának csoportja írási joggal rendelkezik,

S_IXGRP a fájl tulajdonosának csoportja végrehajtási joggal rendelkezik,

S_IRWXO a többiek olvasási, írási és végrehajtási joggal rendelkeznek,

S_IROTH a többiek olvasási joggal rendelkeznek,

S_IWOTH a többiek írási joggal rendelkeznek,

S_IXOTH a többiek végrehajtási joggal rendelkeznek.

A beállítások alapértéke a rendszer `umask` paraméterétől függ.

Windows-os környezetben a két leggyakoribb `mode`:

S_IWRITE a fájl írható,

S_IREAD a fájl olvasható.

Ezek a paraméterek is biteket jelentenek, tehát ha több jogot szeretnénk összeszerkeszteni, akkor a bitenkénti vagy kapcsolatot kell használnunk - hasonlóan a `flag` mezőnél látottakkal.

A függvény típusa `int`, a függvény itt adja meg a fájl azonosítóját, amelyet a továbbiakban a fájl azonosítására használunk.

Ha a megnyitás esetén **bármilyen hiba lép fel**, akkor a visszaadott érték -1.

A fentiek alapján nyissunk meg windows-os környezetben egy text.txt állományt a gyökér könyvtárban olvasásra, binárisan.

```
{
int hnd;
    :
hnd=open("c:\\text.txt",O_RDONLY | O_BINARY);
    :
```

3.3.2. `close` függvény

Ha megnyitottunk egy fájlt az le is kell tudni zárni. Erre a célra szolgál a `close` függvény, melynek deklarációja a következő:

```
int close(int hnd);
```

A függvény paraméterként megkapja a fájl azonosítót.

Ha a lezárás sikeres, akkor a függvény 0 értékkel, ha hiba történt -1 értékkel tér vissza.

3.3.3. `read` függvény

A `read` függvény a fájlból olvas be adatokat a memóriába.

Deklarációja a következő:

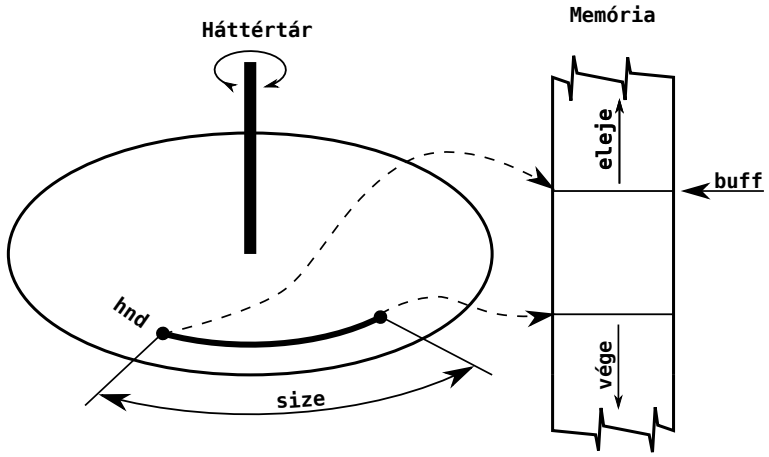
```
ssizet_t read(int hnd,void *buff,size_t size);
```

Az első paraméter a `int hnd`. Ezzel azonosítjuk a fájlt, amelyből olvasni szeretnénk.

A `void *buff` paraméter annak a memória területnek a kezdőcíme, ahova a beolvasás történik[55].

A `size_t size` az mondja meg a függvénynek, hogy hány bájtnyi adatot szeretnénk a fájlból olvasni[56].

A `read ssizet_t` típusú függvény. Sikeres beolvasás esetén a visszatérési érték a beolvasott bájtok száma, sikertelen művelet esetén a visszatérési érték -1.



14. ábra: A `read` működése

[55] Azért `void *`, mert nem tudjuk pontosan, hogy milyen típusú adatot fogunk beolvasni.
[56] Ha `text` módban dolgozunk a `size` a fájlban tárolt adatok mennyisége.

Eljött annak az ideje, hogy az eddig tanult függvényekkel írjuk egy egyszerű programot, amely a képernyőre listázza a `text.txt` szövegfájlt[57].

```
1     #include <stdio.h>
2     #include <unistd.h>
3     #include <sys/types.h>
4     #include <sys/stat.h>
5     int main(void)
6        {
7        int hnd;
8        int r;
9        char c;
10       hnd=open("text.txt",O_RDONLY);
11       while(1)
12          {
13          r=read(hnd,&c,1);
14          if(r<1) break;
15          printf("%c",c);
16          }
17       close(hnd);
18       return 0;
19       }
```

A program 1. - 4. sorában a szükséges header fájlokat olvassuk be (Linux verzió).

A 7. sorban a fájl azonosításához használt hnd nevű fáj azonosítót deklaráljuk.

A 10. sorban megnyitjuk a fájlt olvasására (nem ellenőrizzük a hibát).

A 11. sorban egy végtelen ciklust kezdünk.

A 13. sorban egyetlen bájtot olvasunk a fájlból. A `read` függvény visszaadott értékét az `r` változóba tesszük.

Ha a 14. sorban az `r` változó értéke kisebb, mint 1, akkor egy `break` utasítással kilépünk a ciklusból.

A 15. sorban kiíratjuk a beolvasott karaktert (bájtot).

Amennyiben kiléptünk a ciklusból a 17. sorban lezárjuk a fájlt.

A 18. sorban kilépünk a programból.

[57] A `text.txt` -t előzőleg valakinek létre kell hoznia az aktuális könyvtárban.

3.3.4. `write` függvény

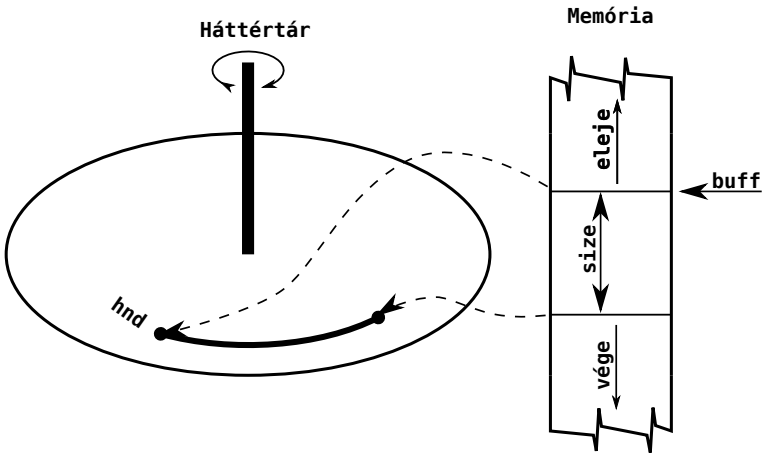
A `write` függvény a memóriából ír adatokat ki a megnyitott fájlba. Deklarációja a következő:

```
ssizet_t write(int hnd,void *buff,size_t size);
```

A függvény első paramétere a fájl leíró.

A második paraméter az a memória cím, ahonnan az adatokat kiírjuk a háttértárra.

A harmadik paraméter a kiírandó bájtok száma a memóriában.



15. ábra: A `write` működése

3.3.5. A `tell` és az `lseek` függvények

A `tell` függvény Linux alatt nem implementált függvény. Windows rendszerek alatt a függvény deklarációja:

```
long tell(int hnd);
```

A függvény a `hnd` által azonosított fájl pozíció mutató értékét adja vissza. Hiba esetén a visszaadott érték -1.

Az `lseek` feladata a fájl pozíció beállítása. Deklarációja:

```
off_t lseek(int hnd,off_t offset,int whence);
```

Az első paraméter a `hnd`, amellyel a fájlt azonosítjuk.

A második paraméter az `offset`. Ez mondja meg, hogy hogyan változtassuk meg a fájl pozíció mutatót.

A harmadik paraméter a `whence`. Ez azt mutatja meg, hogy a módosítást honnan értelmezzük. Ez lehet:

`SEEK_SET` a módosítást a fájl elejétől számítva végzi. A 0-ás fájl pozíció értékhez adja hozzá az offset értékét,

`SEEK_CUR` a módosítást az aktuális pozíciótól számítva végzi,

`SEEK_END` a módosítást a fájl végétől számítva végzi a függvény.

A fejezet elején említettük, hogy a `tell` függvény a Linux rendszerek esetén nincs implementálva. Az `lseek` lehetővé teszi, hogy megkapjuk a fájl pozíció mutató aktuális értékét.

```
long position;
     :
position=lseek(hnd,0,SEEK_CUR);
     :
```

Vagyis állítsuk a fájl pozíció mutatót az aktuális pozíciótól 0 offsettel. Ekkor a visszatérési érték a fájl pozíció mutató értéke (amely nem változott).

Az `lseek` abban is segít, hogy egy fájl méretét meghatározzuk. Erre mutat példát a következő függvény definíció.

```
1     long filelength(int hnd)
2        {
3        long pos;
4        long lgt;
5        pos=lseek(hnd,0,SEEK_CUR);
6        if(pos==-1) return -1;
7        lgt=lseek(hnd,0,SEEK_END);
8        lseek(hnd,pos,SEEK_SET);
9        return lgt;
10       }
```

3.3.6. `fcntl` függvény

Az `fcntl` függvény arra szolgál, hogy egy megnyitott fájl beállításait utólag megváltoztathassuk. Deklarációja a következő:

```
int fcntl(int hnd,int cmd,...);
```

Az függvény első paramétere a fájl leíró.

A második paraméter a `cmd` parancs.

A parancsok a teljesség igénye nélkül:

`F_DUPFL` a megnyitott fájl leíróját megismétli, ha az a harmadik paraméterben megadott értékeknél nagyobb, vagy egyenlő. A függvény visszatérési értéke az új fájl leíró, vagy hiba esetén -1.

`F_GETFL` a kérdéses fájl flag -jeit olvassa vissza (a flag-ek határozzák meg a fájl megnyitási módját lásd open). A harmadik paramétert figyelmen kívül hagyja. A kimeneti érték a visszaolvasott flag-ek.

`F_SETFL` a kérdéses fájl flag -jeit írja át. A harmadik paraméter az írandó flag-ek.

A harmadik paramétert lásd az előző pontnál.

Tipikus alkalmazás az, hogy egy normál megnyitott állományt nemblokkolóra állítunk át (a windows nem tudja).

```
int flags;
   :
flags=fcntl(hnd,F_GETFL);
flags|=O_NONBLOCK;
fcntl(hnd,F_SETFL,flags);
   :
```

3.4. Magas szintű fájlkezelés

A magas szintű fájlkezelés esetén az azonosítás az `stdio.h` fájlban definiált `FILE` típusú struktúra mutatóval történik.

A magas szintű fájlkezeléshez csak az `stdio.h` header fájlra van szükség. Ráadásul ez platformfüggetlen, tehát mindegy milyen környezetben dolgozunk.

3.4.1. `fopen` függvény

Magas szint esetén a fájl megnyitását végzi. Deklarációja:

```
FILE *fopen(const char *path,const char *mode);
```

A függvény első paramétere a fájl elérési útja (lásd open függvény 72. oldal).

A második paraméter az úgynevezett mód sztring. Ez a következő lehet:

"r" a fájlt csak olvasásra nyitjuk meg.

"w" a fájlt írásra és létrehozásra nyitjuk meg. Ha már előzőleg létezett a fájl azt a megnyitás 0 hosszúságúra vágja le.

"a" a fájlt hozzáfűzésre nyitjuk meg.

"r+" a fájlt olvasásra és írásra nyitjuk meg. Ha előzőleg létezett a fájl, az a megnyitáskor nem változik.

"w+" a fájlt órásra, létrehozásra és olvasásra nyitjuk meg. Ha már előzőleg létezett a fájl azt a megnyitás 0 hosszúságúra vágja le

"a+" a fájlt hozzáfűzésre és olvasásra nyitjuk meg

Amennyiben a rendszer különbséget tesz a bináris és text jellegű fájlkezelés között, akkor a mód sztringhez hozzáfűzhető bináris esetben egy `b`, text esetben egy `t` karakter. Például egy olvasásra, írásra megnyitott bináris állomány mód sztringje: `"r+b"`.

A függvény visszatérési értéke sikeres megnyitás esetén a kérdéses állományt leíró `FILE` struktúra címe. Hiba esetén a függvény `NULL` pointerrel tér vissza.

3.4.2. `fclose` függvény

Ha megnyitottuk az állományt le is kell tudni zárni. Ezt végzi el magas szinten az `fclose`. Deklarációja:

```
int fclose(FILE *fp);
```

A függvény paramétere a fájlt azonosító `FILE` mutató.

A visszatérési érték sikeres lezárás esetén 0, hiba esetén -1.

3.4.3. `getc` és `putc` függvény

A `getc` függvény egyetlen bájt beolvasását végzi a megnyitott fájlból. Deklarációja:

```
int getc(FILE *fp);
```

Érdekes megfigyelni, hogy ugyan egyetlen bájtot olvasunk a kérdéses fájlból, de a függvény típusa `int`. Ennek oka az, hogy a beolvasás teljesen transzparens kell, hogy legyen. Ez azt jelenti, hogy az egy bájton ábrázolható 0, 1, . . . , 255 értékeket be kell tudnia olvasni. Ezt `int` változón ábrázoljuk, szóval ezzel nincs gond, azonban ezt a karakter típus is tudná. Karakter típus használatakor viszont nem maradna érték a hibajelzésre.

A hibajelzést úgy oldja meg a függvény, hogy hiba esetén -1 értéket ad vissza. Ez az oka, hogy a függvény típusa `int`.

A `putc` függvény egy bájt kiírását végzi a megnyitott fájlba. Deklarációja:

```
int putc(int chr,FILE *fp);
```

érdekes, hogy a függvény első paramétere, amely a kiírandó bájt mégis `int` típusú. Ennek történelmi okai vannak. Az ős C csak az `int` típust ismerte. Ez a függvény tehát az őskorból származik.

A függvény sikeres kiírás esetén a kiírt bájt értékével tér vissza. Hiba esetén a visszatérési érték -1.

Nézzünk egy példát, amely egy fájlt másol át bájtról bájtra, hibakezeléssel!

```
1     #include <stdio.h> 1
2     int main(void) 2
3       {
4       FILE *fp;
5       FILE *cp;
6       int chr;
7       fp=fopen("source.txt","r");
8       if(fp==NULL) return 1;
9       cp=fopen("destination.txt","w");
10      if(cp==NULL) return 1;
11      while(1)
12        {
13        chr=getc(fp);
14        if(chr==-1) break;
15        putc(chr,cp);
16        }
17     fclose(fp);
18     fclose(cp);
19     return 0;
20     }
```

A 4. sorban deklaráljuk annak a fájlnak az azonosítóját, amely a másolás forrása lesz.

Az 5. sorban a c él fájl azonosítója kerül deklarálásra.

A 6. sorban az az egész típusú változó kerül deklarálásra, amely majd a beolvasott, illetve kiírt bájt értékét tartalmazza.

A 7. sorban megnyitjuk a forrásállományt csak olvasásra.

A 8. sorban megvizsgáljuk, hogy a megnyitás sikertelen volt-e. Ha sikertelen 1-es értékkel kilépünk a programból.

A 9. és 10. sorban szintén ezt tesszük azzal a különbséggel, hogy a célállományt írásra és létrehozásra nyitjuk meg.

A 11. sorban egy végtelen ciklust indítunk.

A 13. sorban beolvassuk a következő karaktert.

A 14. sorban megvizsgáljuk, hogy a beolvasott érték hibát jelez-e. Ha igen, akkor elhagyjuk a ciklust.

A 15, sorba akkor jut a program, ha a `getc` sikeres beolvasást hajtott végre. Ekkor a kérdéses értéket kiírjuk a cél állományba.

A 16. és 17. sorban lezárjuk a fájlokat.

A 18. sorban elhagyjuk a programot 0 értékkel.

3.4.4. `fprintf` függvény

Az `fprintf` formátumozott kiírást tesz lehetővé egy fájlba éppúgy, mint a 64 oldalon leírt `printf` függvény teszi a standard kimenetre. Deklarációja:

```
int fprintf(FILE *fp,const char *format, ... );
```

A függvény első paramétere a fájl azonosítója

A többi paraméter és a visszatérési érték minden pontban megegyezik a 64. oldalon leírt `printf` függvény paramétereivel.

Amennyiben hibaüzenetet szeretnénk kiírni, célszerű az `fprintf` -et használni úgy, hogy az `stderr` -re írunk, például:

```
fprintf(stderr,"Ez egy hibauzenet\n");
```

Ne feledjük, hogy az stderr állományt nem kell megnyitnunk, mert alapértelmezetten nyitott.

3.4.5. `scanf` függvény

A `fscanf` függvény a 69. oldalon ismertetett `scanf` függvényhez hasonlóan formátumozott beolvasást tesz lehetővé a kiválasztott fájlból. Deklarációja a következő:

```
int fscanf(FILE *fp,const char *format, ... );
```

Az `fscanf` esetén figyelnünk kell, hogy a fájl pozíció a megfelelő pozícióban álljon, különben a függvény rossz értéket olvas be. Ekkor érdemes a függvény visszatérési értékét figyelni, mert ha nem olyan típust talál a fájlban, mint a várt, akkor a beolvasás hibás.

A függvény első paramétere a fájl azonosítója. A többi paraméter és a visszatérési érték megegyezik a 69 oldalon ismertetett scanf függvénnyel.

3.4.6. `fflush` függvény

A korszerű operációs rendszerek próbálnak a számítógép erőforrásával takarékoskodni. Ez fájlba íráskor az jelenti, hogy nem bájtonként írják ki az információt, hanem egy adott adatmennyiségig várnak és csak akkor fordulnak a kérdéses perifériához. Ez a méret rendszerfüggő, de a legelterjedtebb méret 4096 bájt.

Ha adott idő belül nem jön össze a fent említett méret, akkor a rendszer mindenképpen kiírja az adatokat. Ez az idő szintén rendszerfüggő. Egy sűrűn használt időkorlát a 30 másodperc.

Megjegyzés: természetesen a fájl lezárásakor az adatok kiíródnak.

Ez a tulajdonság néhány esetben nagyon kellemetlen. Például akkor, ha a programunk futását a képernyőre írással próbáljuk követni. Ekkor célszerű a `fflush` használata.

Az `fflush` a kérdéses fájlba írást a meghívásakor azonnal megkezdi. Deklarációja:

```
int fflush(FILE *fp);
```

Sikeres végrehajtás esetén a visszatérési érték 0, egyébként -1.

Ha az `fflush` argumentumába `NULL` pointer írunk, vagy üresen hagyjuk, akkor az összes írásra nyitott fájlra végrehajtja a kiírást.

Ha tehát a képernyőre ki akarjuk íratni azonnal az információt, akkor a következőt célszerű tenni:

```
printf("Szoveg\n");fflush(stdout);
```

3.4.7. `fread` és `fwrite` függvények

Az `fread` függvény - hasonlóan a `read` függvényhez - a megnyitott fájlból olvas be a memóriába. Deklarációja a következő:

```
size_t fread(const void *ptr,size_t size,size_t n,FILE *ptr);
```

A függvény első paramétere a kérdéses memória cím, ahova olvasni szeretnénk.

A második paraméter megadja, hogy a mekkora egy beolvasandó elem mérete.

A harmadik paraméter megmondja, hogy a kérdéses elemekből hány darabot akarunk beolvasni.

A negyedik paraméter a fájl azonosítója.

A függvény visszatérési értéke sikeres beolvasás esetén a beolvasott elemek száma. Hiba esetén 0 (néhány rendszer esetén -1, ekkor a függvény típusa int).

```
size_t fwrite(const void *ptr,size_t size,size_t n,FILE *ptr);
```

A függvény első paramétere a kérdéses memória cím, ahonnan ki akarjuk az adatokat írni.

A második paraméter megadja, hogy a mekkora egy kiírandó elem mérete.

A harmadik paraméter megmondja, hogy a kérdéses elemekből hány darabot akarunk kiírni.

A negyedik paraméter a fájl azonosítója.

A függvény visszatérési értéke a sikeresen kiírt elemek száma. Hiba esetén 0 (néhány rendszer esetén -1, ekkor a függvény típusa int).

3.4.8. `ftell` és `fseek` függvények

Az `ftell` függvény megmondja, hogy a fájl pozíció mutató hol áll a kérdéses fájlban. Deklarációja:

```
long ftell(FILE *fp);
```

Sikeres függvényhívás esetén a visszatérési érték a fájl pozíció mutató értéke. Hiba esetén -1.

Az `fseek` függvény a 77. oldalon ismertetett `lseek` -hez hasonlóan a fájl pozíció mutató értékét állítja be.

```
long fseek(FILE *fp,long offset,int whence);
```

Az első paraméter a fájl azonosítója. A többi paraméter és a visszatérési érték teljesen megegyezik a `lseek` függvénynél leírtakkal (lásd 77. oldal).

Vegyük példának azt feladatot, hogy a `text.txt` fájlban az `e` karaktereket `E` karakterekre cseréljük[58].

[58] Ez egy iskolapélda, sokkal optimálisabban is megírható.

```
1     #include <stdio.h>
2     int main(void)
3       {
4       FILE *fp;
5       int chr;
6       long pos;
7       fp=fopen("text.txt","r+");
8       if(fp==NULL)
9         {
10        fprintf(stderr,"Nyitasi hiba\n"); return 1;
11        }
12      while(1)
13        {
14        pos=ftell(fp);
15        chr=getc(fp);
16        if(chr==-1) break;
17        if(chr==e)
18          {
19          fseek(fp,pos,SEEK_SET);
20          putc(fp,E);
21          }
22        }
23      fclose(fp);
24      return 0;
25      }
```

A program eleje már az előző mintaprogramokból ismert.

A program 6. sorában egy `long` típusú változót deklarálunk, amelybe a fájl pozíció mutatót fogjuk elhelyezni menden beolvasás előtt.

A 7. sorban a `text.txt` allományt nyitjuk meg írásra és olvasásra `"r+"`.

Ha sikertelen a megnyitás, akkor 8. sorban az `if` argumentuma igaz.

A 10. sorban a standard hiba fájlra kiírjuk a hibát és elhagyjuk a programot egy 1-es értékkel.

A 12. sorban egy végtelen ciklust kezdünk.

A 14. sorban az éppen aktuális beolvasandó karakter pozícióját eltesszük a `pos` változóba.

A 15. sorban beolvassuk a karaktert.

A 16. sorban megvizsgáljuk, hogy a beolvasás sikeres volt-e. Ha nem elhagyjuk a ciklust.

A 17. sorban megvizsgáljuk, hogy a beolvasott karakter `e` -e.

Ha igen a program a 19. sorban folytatódik, ahol a beolvasás előtti pozícióra állítjuk a fájl pozíció mutatót.

A 20. sorban kiórjuk az `E` karaktert.

A 22. sorban zárjuk le a végtelen ciklust.

Innetől a program működése már ismert.

3.5. Átjárás a két fájlkezelés között

Előzőleg már említettük, hogy az alacson és a magas szintű fájl kezelés között van átjárás mindkét irányban. Azt is láthattuk, hogy a mindkét módszernek megvannak az előnyei és a hátrányai. Ezért érdemes a ismernünk a következő függvényeket azért, hogy a lehető legjobban ki tudjuk a két módszer tulajdonságait.

3.5.1. `fileno` függvény

A `fileno` függvény egy magas szinten megnyitott fájl fájl leíróját adja vissza. Deklarációja, amely az `stdio.h` -ban található, a következő:

```
int fileno(FILE *fp);
```

Ha az `fp` -vel azonosított fájl létezik a visszatérési értéke a fájl leírója, ha nemlétezik, akkor -1.

A következő példában az `stdin` fájlt állítjuk át nem blokkolóvá.

```
1     #include <stdio.h>
2     #include <unistd.h>
3     #include <fcntl.h>
4     int main(void)
5       {
6       int hnd;
7       int r;
8       int f;
9       char buff[80];
10      hnd=fileno(stdin);
11      f=fcntl(hnd,F_GETFL);
12      f|=O_NONBLOCK;
13      fcntl(hnd,F_SETFL,f);
```

```
14        while(1)
15          {
16          r=scanf("%s",buff);
17          if(r==1)
18            {
19            printf("%s",buff);
20            break;
21            }
22          printf(".");
23          fflush(stdout);
24          }
25        return 0;
26        }
```

A program eleje a 9. sorig ismert.

A 10. sorban az alapértelmezetten nyitott standard bemenet fájl leíróját (hnd) kapjuk meg a `fileno` függvénnyel.

A 11. sorban az fcntl segítségével lekérdezzük a fájl flag -jeit az `f` változóba.

A 12. sorban az `O_NONBLOCK` flag-et rámásoljuk a flag -ekre

A 13. sorban az átalakított flag -eket beállítjuk

A 14. sorban egy végtelen ciklust indítunk.

A 15. sorban hívjuk a `scanf` függvényt. A függvény visszaadott értékét az `r` változóban tároljuk.

Ha az `r` változó értéke 1, akkor a `scanf` olvasott be értékelhető inputot és az igaz ág végrehajtódik (lásd sor).

Ha történt beolvasás, akkor azt kiírjuk a képernyőre - 19. sor és elhagyjuk a végtelen ciklust - 20. sor.

Abban az esetben, ha nem történt értékelhető beolvasás, a program a 22. sorban folytatódik, ahol egy `.` karaktert ír ki a képernyőre.

A 23. sorban a kiíratást kényszerítjük az `fflush` függvénnyel.

A program további részei ismertek.

Megjegyzés: ez a program windows alatt nem megy. Linux-on és BSD UNIX-okon problémamentes a futtatása.

3.5.2. `fdopen` függvény

Az `fdopen` függvény lehetővé teszi, hogy egy alacsony szinten megnyitott fájl konvertálható legyen magas szintre. A függvény deklarációja:

```
FILE *fdopen(int hnd,const char *mode);
```

A függvény első paramétere a fájl leíró.

A második paraméter a az úgynevezett mód sztring. Ez teljesen megegyezik az `fopen` függvényben ismertetett mód sztringgel (lásd 79. oldal).

A függvény visszatérési értéke a kívánt `FILE` pointer. Ha a konverzió sikertelen a visszatérési érték `NULL` pointe

3.6. Amire figyelni kell (intelmek a fájlkezeléssel kapcsoltaban)

A fájl kezelés C alatt nem bonyolult dolog, azonban néhány dologra célszerű figyelni. A szempontok a következő:

- a megnyitott fájlt minden esetben zárjuk le,
- kerüljük a ciklusokban fájlok megnyitását. Ha mégis szükséges, akkor annyiszor zárjuk le, ahányszor megnyitottuk. Erre különösen figyeljun a `break` és `continue` utasítások esetén.
- ha egy függvénybe belépünk és ott nyitunk fájlt, akkor a kilépés előtt zárjuk is le minden kilépési pont előtt.

4. Dinamikus memória kezelés

A gyakorlati életben sokszor előforduló probléma, hogy nagy mennyiségű memóriára van szükség egy adott feladat elvégzéséhez. Ebben az esetben nem célszerű ezt a memória mennyiséget változóterületként deklarálni, mert vagy valamilyen a rendszer által felállított korlátozás nem engedi meg, vagy egyszerűen fölöslegesen nagy programot készítünk.

Ilyen esetekre a dinamikus memória kezelés a helyes megoldás[59]

4.1. A memória lefoglalása `malloc` , `calloc` , `realloc`

Első lépés a dinamikus memória kezelésnél a memória lefoglalása. Sikeres lefoglalás esetén kapunk egy összefüggő memória tartományt, amelyet a kezdőcímével azonosít a program[60]

Erre a célra a legegyszerűbb a `malloc` függvény használata. Deklarációja az `stdlib.h` header fájlban található meg.

```
void *malloc(size_t size);
```

A függvény egyetlen paramétere azt mondja meg, hogy hány bájtot szeretnénk a memóriában lefoglalni.

A visszatérési érték a lefoglalt memória kezdőcíme. Ha a lefoglalás sikertelen, akkor `NULL` pointer.

Ne feledkezzünk meg arról, hogy a visszaadott pointer `void` típusú. Ennek oka, hogy nem tudjuk, hogy milyen típusú a lefoglalt memória.

A memória lefoglalásra használható a `calloc` függvény is. Ez annyiban különbözik, hogy a lefoglalt memória méretét nem bájtban, hanem a kívánt típus méretében kell megadni. Deklarációja:

```
void *calloc(size_t n,size_t size);
```

Az első paraméter a lefoglalandó elemek száma, a második paraméter a a kérdéses típus mérete.

A visszatérési érték teljesen megegyezik a `malloc` esetén leírtakkal.

Ilyenkor felmerül a kérdés, hogy miért nem ad vissza olyan pointert, mint, aminek a méretét megadjuk a függvény argumentumában. A válasz egyszerű, a függvény nem tudhatja, hogy milyen típust adtunk meg, mert csak a méretet adtuk meg. Arra nincs lehetőség, hogy a típust adjuk meg.

Ha szükség van a kérdéses típushoz rendelt pointerre, akkor erről a programozónak kell gondoskodnia a programban Például:

[59] Azt azért jegyezzük meg, hogy vannak olyan területek, ahol a dinamikus memória kezelés nem megengedett. Ilyen például a biztonságkritikus szoftverek területe.

[60] Igazából az operációs rendszer.

```
long *p;
  :
p=(long *)calloc(1000,sizeof(long));
  :
```

A visszaadott típus `void *`, de nekünk `long *`-ra van szükségünk, ezért a függvény kimeneti értékét ”castoljuk” kívánt típusra[61].

Gyakran előforduló probléma, hogy a lefoglalt memória mérete nem megfelelő. Ekkor használhatjuk a `realloc` függvényt.

```
void *realloc(void *ptr,size_t size);
```

Az átméretezendő memóriát a `ptr` pointer azonosítja. Az igényelt új memória méretet a `size` változó adja meg. Ha sikeres a lefoglalás, akkor az újonnan lefoglalt memória kezdőcímét kapjuk meg, ha sikertelen a függvény `NULL` poiterrel tér vissza.

Az átméretezett memória eredeti tartalmát semmi nem garantálja. Elképzelhető, hogy az újonnan lefoglalt terület teljesen máshol helyezkedik el, mint a régi.

4.2. A lefoglalt memória felszabadítása `free`

Amennyiben dinamikusa memóriát foglaltunk le, azt fel is kell szabadítani. Ezt a műveletet a `free` függvény végzi.

```
void free(void *ptr);
```

A `free` bemeneti paramétere a memóriát azonosító pointer a `ptr`.

4.3. Amire figyelni kell (intelmek dinamikus memóriakezelés esetén)

A dinamikus memória kezeléssel kapcsolatban a következő szabályokra kell figyelni:

- a lefoglalt memóriát minden esetben szabadítsuk fel,
- kerüljük a dinamikus memória foglalást ciklusokban. Ha ez nem lehetséges, akkor gondoskodjunk arról, hogy minden kilépési pontban legyen megfelelő felszabadítás.
- ha egy függvény belsejében foglalunk le dinamikusan memóriát, akkor a kilépéskor szabadítsuk fel[62].

[61] Ez igazából inkább C++ probléma. A C erre nem érzékeny.
[62] Hacsak nem speciális céllal tesszük, de ha lehet ezt az esetet kerüljük.