

---

Pere László  
UNIX – GNU/Linux  
Programozás C nyelven

---

Pécs, 2003



Pere László: UNIX – GNU/Linux  
Programozás C nyelven  
© 2003 Pere László

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója  
© 2003 Kiskapu Kft.

1081 Budapest, Népszínház u. 29.  
<http://kiado.kiskapu.hu/>  
e-mail: [kiado@kiskapu.hu](mailto:kiado@kiskapu.hu)

Szakmailag ellenőrizte: Csizmazia Balázs  
Nyelvileg ellenőrizte: Rézműves László  
Borítóterv: Zsédely Teréz

ISBN szám: 963 9301 68 x

Készült a debreceni Kinizsi Nyomdában  
Felelős vezető: Bördős János

# Tartalomjegyzék

Előszó	5
1. A C programozási nyelv	7
A munkamenet	8
Változók és függvények	10
Megjegyzések a szövegben	26
Elemi adattípusok	27
Összetett típusok	31
A mutatók	33
Struktúrák	49
Új típusok létrehozása	53
Összetett típusokból készült típusok	55
Ajánlott irodalom	64
2. A GNU C könyvtár	65
Hibák és hibaüzenetek	66
Nyelvi beállítások	68
Karakterláncok kezelése	73
Karaktervizsgálat és átalakítás	80
Rendezés és keresés	81
Mintaillesztés	85
Matematikai függvények	89
Hatványozás, gyökvonás és logaritmus	96
A környezeti változók kezelése	100
Memóriafigoglalás	103
Műveletek a memóriában	106
Csatornák megnyitása és lezárása	108
Formázott kimenet	119
A felhasználói adatbázis	129
Állománykezelés	137
Időt kezelő függvények	172
Folyamatok indítása	179
Naplóbejegyzések elhelyezése	186
Visszaugrás függvényből	188

---

Jelzések	190
Hálózati kapcsolatok kezelése	200
Megosztott memória kezelése	227
Szemaforok	236
Ajánlott irodalom	247

## Előszó

A C programozási nyelv nem tartozik az egyszerűen megtanulható nyelvek közé. A nyelv szerkezete első pillantásra nem tűnik ugyan bonyolultnak, a megtanulandó eszközök listája sem túlságosan hosszú, jó C nyelvű programot írni mégsem könnyű. A nyelv elsajátítása sok gyakorlást, tanulást igényel. Ebben a tanulásban igyekszünk segítséget nyújtani az olvasó számára.

Hiszen érdemes megismerni e nyelvet. Nem csak azért, mert a mai nap is kere-ssett programozási nyelvről van szó, amelyet igen sok programozó és szoftverfejlesztő vállalat használ, nem csak azért, mert magát a Linux operációs rendszert is C nyelven írták, hanem azért is, mert az elmúlt harminc év meghatározó programozási nyelve. A C programozási nyelv a rendszermérnökök, rendszerprogramozók és az alkalmazásfejlesztők közös eszköze, amely a számítástechnika bizonyos területeit szinte születése óta uralja. Nyilvánvalóan nem véletlen, hogy a felsőoktatásban minden számítástechnikai vagy informatikai szakon szerepel a C nyelv.

A könyv elkészítésekor egyik legfontosabb célunk az egyszerűség, a közérthetőség volt. Olyan olvasók számára ajánljuk e könyvet, akik rendelkeznek ugyan némi előismerettel a programozást, a számítógéphasználatot illetően, de a lehető legkevesebb előismeretet feltételező könyvet keresik. Ha az olvasó rendelkezik némi Unix felhasználói tapasztalattal, tudja mi a szabványos bemenet és kimenet, egy szövegszerkesztő program alapvető használatát ismeri, és képes eligazodni az állományrendszerben, e könyv segítségével önállóan is megismerkedhet a C programozási nyelv és a C programkönyvtár legfontosabb elemeivel. Előtanulmányként a Kis-kapu Kiadónál megjelent *Linux: felhasználói ismeretek* első két kötetét, esetleg más bevezető jellegű Unix- vagy Linux-könyvet javasoljuk.

A könyv a C programozási nyelv alapjait és a szabványos eszközök készletét elérhető C programkönyvtár lényegesebb elemeit mutatja be. Nem teljes referenciáról van szó tehát, csak egy rövid bemutatóról. A teljes leírás sokkal hosszabb művet eredményezett volna, amely nem volna túlságosan hasznos a nyelvvel, a programozással ismerkedők számára. Az anyag egyszerűsítése során sok nyelvi szerkezet, eszköz, függvény kimaradt, de a legfontosabbak remélhetőleg helyet kaptak. A könyv megírásakor igen fontos cél volt az, hogy az ismertetett eszközök készlet a minden nap életben előforduló programozási feladatok lehető legszélesebb körét fedjék le. A könyvben bemutatott nyelvi elemek „majdnem minden feladat” elvégzésére elegendőek.

Kik forgathatják haszonnal e könyvet? Mindazok, akik a C programozási nyelvet ismerkednek, akik úgy gondolják, hogy közelebb szeretnének kerülni a számítógéphez, meg kívánják érteni, hogyan épülnek fel azok az alkalmazások, segédprogramok, amelyeket minden napjaik során használnak és akik úgy gondolják, hogy egyszerűbb programok készítésével szeretnék kiegészíteni a Linux adta eszközöket.

A könyvben több mint száz önállóan is működőképes példaprogram található. Ezek a példaprogramok letölthetők a <http://kiado.kiskapu.hu/60> címről. A könyvben a példaprogramok felett található könyvtárnevek a letöltött állományban való eligazodást segítik.

A szerző ez úton is szeretne köszönetet mondani mindenkit, akik a könyv elkészítésében segítségére voltak, különösen Csizmazia Balázsnak és Rézműves Lászlónak. Az ő munkájuk nélkül e könyv nem jelenhetett volna meg.

# 1. fejezet

## A C PROGRAMOZÁSI NYELV

*A C volt a jó programnyelv, a megfelelő időben.*

*A mai napig uralja a rendszerprogramozást.*

*A. TANENBAUM: Modern Operating Systems*

*Hümmögött, hümmögött, de lefordította!*

GALAMB

A C programozási nyelv eredetileg a Unix számára készült, az első olyan magasszintű programozási nyelvként, amely alkalmas operációs rendszer írására. Ebből az egyszerű tényből következik, hogy a C nyelv olyan hatékony, olyan rugalmas és olyan sikeres a programozók körében. A Unix kezdeti sikerének egyik fontos oka a hordozhatósága volt, amelyet egyértelműen annak köszönhetett, hogy majdnem teljes egészében C nyelven írták.

Természetesen, mint a legtöbb programozási nyelv, a C sem a semmiből született. A programozási nyelveket tervező szakemberek más nyelvekből vesznek át ötleteket, hasznosnak tartott tulajdonságokat. A tervezőkre – bevallottan vagy be nem vallottan – az alkotás során hatnak más nyelvek, más alkotók munkái. A C nyelv igen kiterjedt és ősi családfával rendelkezik. Rokonsága egészen az 1954-ben, első programozási nyelvként kifejlesztett – és a mai napig is használt – Fortran nyelvig nyúlik vissza.

A C nyelv elődjének tekinthető a Martin Richards által kifejlesztett BCPL programozási nyelv, melyből Ken Thompson alkotta meg a B programozási nyelvet. Ennek a nyelvnek a továbbfejlesztésével készítette Dennis Richie a C nyelvet, amelyet aztán Brian Kernighan barátjával írtak le és tártak a nagyközönség elé az „*A C programozási nyelv*” c. könyükben. A nyelv később a K&R C (Kernighan és Richie C nyelv) néven vált közismertté.

A nyelv az azóta eltelt évtizedekben apró változásokon ment át, több szabványügyi hivatal is szabványosította. A nyelv szabványos változatáról ma ANSI C nyelv-

ként beszélhetünk, utalva az amerikai szabványügyi hivatal X3.159-1989-„ANSI C” szabványára, vagy nevezhetjük ISO C nyelvnek is, az ISO/IEC 9899:1990 *Programozási nyelvek*: C nemzetközi szabvány szerint. Bármelyik szabványt vegyük is alapul tudnunk kell, hogy a C programozási nyelv a többi programozási nyelvhez képest igen keveset változott születése óta. Ez minden bizonnal annak köszönhető, hogy a nyelv már megszületésekor jól átgondolt, hatékony és sikeres volt.

Mivel a Unix operációs rendszer család minden tagja C nyelven íródott, minden Unix rendszerhez tartozik egy C fordító, amelynek segítségével a C nyelven írt programjainkat lefordíthatjuk futtatható állománnyá. Nincs ez másképpen a Linux rendszerekkel sem. Linux rendszereken általában a GNU C fordítót használjuk, amely a Linux operációs rendszer készítése előtt született, Richard Stallman munkájaként. A legenda szerint Stallman a világ elől elbújva, 18 hónapnyi csendes magányban készítette a GNU C fordító első változatát.

A GNU C fordító képes C, C++, Objective-C, Fortran, Java, és Ada programok fordítására. C program fordítása közben a GNU C fordító a vonatkozó ANSI és ISO szabványokat veszi alapul, bár rendelkezik jónéhány bővítéssel, amelyeket ki kell kapcsolnunk, ha azt akarjuk, hogy mindenben a szabvány szerint járjon el. A fordító az 1989-ben, 1990-ben, valamint 1995-ben életbe lépett szabványokat maradéktalanul teljesíti, az 1999-es módosításokat pedig részben valósítja meg. Az érdeklődők a szabványokhoz tartozó kapcsolókat a fordító dokumentációjában találhatják meg.

## A munkamenet

Brian Kernighan és Dennis Richie a C programozási nyelvről született első könyükben a „helló világ” programmal kezdik a nyelv bemutatását. Ez a program – amely csak egy üzenetet ír a szabványos kimenetre –, azóta legendássá vált, ezért mi is ezt a programot mutatjuk be elsőként.

Készítsük el kedvenc szövegszerkesztő programunkkal (amely a vi vagy az emacs, esetleg az emacs vagy a vi, tehát szövegszerkesztő program) a következő C nyelvű programot:

	<i>alap/hellovilag.c</i>
1	#include <stdio.h>
2	
3	main(){
4	printf("helló világ\n");
5	}

A programot mentük egy állományba, amelynek neve a .c karakterekre végződik. Unix rendszereken minden a .c végződést adjuk a C programokat tartalmazó

állományok neveinek. A forrásprogramot tartalmazó állomány neve lehet például `hello.c` vagy `elso.c`.

Az elkészült C nyelvű programot Unix rendszereken a `cc` nevű C fordítóval fordíthatjuk le. Linux rendszereken általában a GNU C fordítót használjuk, amely indítható a `gcc` parancs segítségével is. A programot tartalmazó forrásállomány nevét a `gcc` után írva a fordítás megtörténik, a futtatható – lefordított – program az `a.out` állományba kerül. Az elkészült állományt a következőképpen próbálhatjuk ki:

```
Bash$ gcc hello.c
Bash$ ./a.out
helló világ
Bash$
```

Amint a példából látható, az indítandó program neve előtt szerepelnie kell a `./` kifejezésnek, amely a munkakönyvtárat jelenti. Amint az már nyilvánvalóan sokak előtt ismeretes, erre azért van szükség, mert a Unix rendszerek biztonsági okokból nem keresik az indítandó programot a munkakönyvtárban, hacsak erre – a `./` előtaggal – külön utasítást nem kapnak.

Természetesen a C nyelven megírt és lefordított programjainkat is elhelyezhetjük a saját könyvtárunkban létrehozott `bin/` könyvtárban, így a héj bármikor megtalálja – anélkül, hogy a program pontos elérési helyét begépeelnénk.

A fordító által készített állomány nevéből nem szabad arra következtetnünk, hogy a futtatható állomány az egyébként már elavultnak tekinthető `a.out` formátumot használná. Amint a következő példa mutatja, a bináris állomány ELF formátumban készült:

```
Bash$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), not stripped
Bash$
```

A `gcc -o` kapcsolója segítségével meghatározhatjuk a fordítás eredményeképpen született futtatható állomány nevét. A kapcsolót az elkészítendő állomány nevének kell követnie.

```
Bash$ gcc hello.c -o hello
Bash$ ./hello
helló világ
Bash$
```

A példából látható, hogy most a `hello` nevű állományban helyezte el a fordító a futtatható programot.

# Változók és függvények

Valószínűleg a világ egyik legegyszerűbb C nyelvű programja a következő példa-program, amely kiszámítja  $1 + 1$  értékét:

```

1 main(){
2     int a, b;
3
4     a=1;
5     b=a+1;
6 }
```

alap/osszeadas.c

A program valójában a `main` nevű függvény leírása. Az első sorban látható a `main` neve, utána egy `()` zárójelpár, amely jelzi a fordító számára, hogy egy függvényt szeretnénk létrehozni. A név és a zárójelek után `{}` kapcsos zárójelpárt találunk, amely között a függvény megvalósítását, a függvényt alkotó utasításokat olvashatjuk. A C függvények olyan programrészletek, amelyek névvel rendelkeznek s amelyeket a példában bemutatott módon hozhatunk létre.

Valójában a `main` egy különleges függvény. A C nyelv a `main` szót tartja fenn a program belépési pontjának jelölésére, azaz az elkészített program végrehajtása a `main` függvény végrehajtását jelenti. minden C programban kell lennie egy – és pontosan egy – függvénynek, amelynek a `main` nevet adtuk. A program a `main` függvény végrehajtásával kezdődik és a `main` végrehajtása után azonnal be is fejeződik.

Valójában a `main()` függvényt a példaprogramban szabálytalanul használtuk. Készíthetünk ugyan olyan `main()` függvényt, amely nem fogad paramétert, de visszatérési értékkel rendelkeznie kell minden esetben, amely kötelezően `int` típusú. A legegyszerűbb C nyelvű program tehát nem a következő:

```
main(){ }
```

Helyesen írva a „semmit sem csináló” programot, a következő formát kapjuk:

```
int main(){
    return(0);
}
```

Szerencsére azonban a legtöbb C fordító – így a GNU C fordító is – elfogadja az egyszerűsített, szabálytalan formát.

A példaprogramunk tehát valójában a 2. sorral kezdődik és az 5. sorral véget is ér. Található egy üres sor – a 3. sorban nincs semmi. Tudnunk kell, hogy a nyelv szabad kezet ad a programozónak abban, hogy a forrásprogramot számára olvasható formában rendezze el. Szinte bárhol kezdhetünk új sort, hagyhatunk ki sorokat vagy

rendezhetünk beljebb szövegrészleteket szóközök vagy tabulátorkarakterek elhelyezésével. Ez nagymértékben megkönnyíti a munkát például azzal, hogy az egymásba ágyazást jelezhetjük beljebb írással.

Itt az üres sorral a változók létrehozását választottuk el a programot alkotó parancsoktól. A 2. sorban ugyanis nem végrehajtandó parancsok vannak, amelyek bekerülnek a futtatható programba, hanem két változó létrehozása. A változók típusa int, neve pedig a és b. Változókat tehát úgy hozhatunk létre, hogy leírjuk a változó típusát, majd a változó – vagy változók – nevét. A változó létrehozása után egy pontosvesszőnek kell következnie, ahogyan azt a példában is látjuk.

A példában használt változók int (*integer*, egész) típusúak, amelyek egész számok tárolására alkalmasak. A C nyelvben használható változótípusokra a későbbiekben visszatérünk és részletesen tárgyaljuk őket.

Valójában több sorban is létrehozhattuk volna a változókat, nem kellett volna az összes változót egy sorban felsorolni. Írhattuk volna például azt, hogy

```
1 int a;
2 int b;
```

és így két külön lépésben hozhattuk volna létre a változókat. Általában javasolható, hogy a logikailag együvé tartozó változókat egyazon sorban hozzuk létre, de külön azoktól, amelyek nem tartoznak szorosan hozzájuk.

A példaprogram 4. és 5. sorában két elemi utasítás – két értékkadás – található. Amint látható, az értékkedások ugyanolyan formában írhatóak, mint ahogyan azt más programozási nyelvek esetében már megszokhattuk. Talán az egyetlen különlegesség, hogy az értékkedás után mindenkor mindenkor megjelenjen a pontosvessző, ahogyan azt a változók létrehozásakor már láthattuk. Ha egyetlen pontosvesszőt is elhagyunk, a program fordítása sikertelen lesz.

A példaprogram kapcsán meg kell jegyeznünk, hogy a változók létrehozásának mindenkor mindenkor előtt kell lennie, azaz amint leírtuk az első utasítást, már nem hozhatunk létre újabb változót. Nem írhatjuk például azt, hogy

```
1 main(){
2     int a;
3
4     a=1;
5     int b;
6
7     b=a+1;
8 }
```

még akkor sem, ha a józan ész azt súgja, hogy a program helyes, hiszen minden változót létrehoztunk, mielőtt használtuk volna. Ha ezt a programot megpróbáljuk lefordítani, a fordítás hibaüzenethez vezet és nem készül el a futtatható állomány:

```
Bash$ gcc hibas.c
hibas.c: In function ‘main’:
hibas.c:5: parse error before ‘int’
hibas.c:7: ‘b’ undeclared (first use in this function)
hibas.c:7: (Each undeclared identifier is reported only once
hibas.c:7: for each function it appears in.)
Bash$
```

A hibaüzenetből kiolvashatjuk, hogy az 5. sorban található int kulcsszó előtt nyelvtani hiba található. Amint látjuk, C a nyelv nem engedi meg a változók létrehozását utasítások után.

Ha a példában is látható módon hibaüzeneteket kapunk a fordítás során, mindenkor legelső hibaüzenetre kell figyelnünk, hiszen lehetséges, hogy a többi hiba már csak következménye az elsőnek. A C fordító is fentről lefelé olvassa a programot, ezért a hibákat is ilyen sorrendben fedez fel.

A legjobb, ha hibakereséskor az első hiba kijavítása után újra megpróbáljuk lefordítani a programot, hiszen lehet, hogy a hiba kijavítása minden problémát megold, mint az előbbi példában is. Ne lepődjünk meg azonban azon sem, ha a hiba javítása újabb hibaüzenetek megjelenését eredményezi, ha ugyanis a hiba megszüntetése miatt a fordító tovább jut az állományban, esetleg újabb hibákat fedez fel. Reménykedhetünk abban, hogy az egyre több hibaüzenet a gyógyulás jele és nem azt jelzi, hogy végül sikerült minden teljesen összekutulnunk.

A következő példaprogramunk egy új függvényt hoz létre és alkalmaz. Talán furcsa lehet, hogy a nyelvvel való ismerkedésnek az elején, az első lépések között foglalkozunk a függvények létrehozásával, a függvények azonban a C nyelvű programozás igen lényeges eszközei, ezért már most meg kell velük ismerkednünk.

Hozzunk létre függvényt a matematikai átlag kiszámítására. A programban használt változónevek legyenek hosszabbak és beszédesebbek, mint az előző programunkban. (Az ilyen hosszabb változónevek használata mindenkorban hasznos, hiszen olvashatóbbá és könnyebben megérthetővé teszik a programot.)

Bátran használhatunk kis- és nagybetűket a változónevekben – bár a Linux világában inkább csak kisbetűket használunk –, valamint az aláhúzás karaktert – akár első betüként. A magyar ékezetes karaktereket nem használhatjuk és tartózkodnunk kell minden írásjel használatától, de általában nem zavaró.

A matematikai átlag számítását végző program a következő:

	<i>alap/atlag.c</i>
1	<code>int atlag( int a, int b ) {</code>
2	<code>    return( (a+b)/2 );</code>
3	<code>}</code>
4	
5	<code>main(){</code>

```

6   int elso;
7   int masodik;
8   int matematikai_atlag;
9
10  elso=42; masodik=43;
11  matematikai_atlag=atlag(elso, masodik);
12 }
```

Figyeljük meg a program 1–3. sorát, ahol az `atlag` függvényt hozzuk létre. A függvény létrehozása mindenkor a visszatérési érték típusának megadásával, a függvény nevénél, valamint a zárójelek közt beírt paraméterlistával történik. A paraméterlistát a változók létrehozásának és a függvényt megvalósító utasításoknak kell követniük. Ezt a részt a `{ }` kapcsos zárójelek közé kell írnunk.

A program 11. sorában láthatjuk a függvény felhasználását, vagyis hívását. Itt egyszerűen a függvény neve szerepel, amely után az átadott paraméterekek láthatók zárójelek közt, vesszővel elválasztva. Miután a függvényt elkészítettük, bármilyen kifejezésbe beépíthetjük.

Összpontosítsuk most figyelmünket a függvénynek átadott és az onnan visszaadott értékekre! A program első sorában – a zárójelek közt található paraméterlistában szerepelnek a `a` és `b` változók. Ezeket a változókat a függvényen belül szabadon használhatjuk, értéküket a függvénynek átadott értékek határozzák meg. A 2. sorban használjuk is ezeknek a változóknak az értékét egy matematikai átlag kiszámítását végző kifejezésben.

A 2. sorban található a `return` utasítás. Ez nem beépített függvény – mint amilyen a `main()` –, de sok tekintetben hasonlóképpen viselkedik. A `return` utasítás befejezi a függvény futását és az utána zárójelek közt szereplő értéket visszaadja a hívó függvénynek. Bármit írunk tehát a `return` utasítás után, ha végrehajtja a program, a vezérlés visszatér a függvény hívójához a visszatérési érték átadásával. A példaprogramban a 11. sorban hívtuk az átlagot számító függvényt, annak elvégzése után tehát a visszatérési érték ide, az értékadás bal oldalára helyettesítődik be. A hívott függvény visszatérése után a hívó futása zavartalanul fut tovább a következő utasítás végrehajtásával.

A függvények készítésekor egy dologra mindenkor ügyelnünk kell: a függvényt mindenkor az előtt kell létrehoznunk, mielőtt használnánk, vagyis előbb létrehozunk, utána tudjuk hívni. Ez első hallásra természetesnek tűnik, hiszen a C fordító a programot fentről lefelé olvassa, így előbb jut el a függvény létrehozásához – előbb tanulja meg, mit akarunk a függvénytől – és utána alkalmazza, azaz hívja.

Valójában azonban a C fordító nem futtatja a programot, csak lefordítja! Akkor viszont felmerül a kérdés, hogy miért nem fordítja le a programot lépésről lépésre, a függvények sorrendjétől függetlenül, hiszen úgyis csak akkor lesz rájuk szükségünk, amikor már a teljes program le van fordítva. Ez felmentene minket az alól a terhes kötelezettségtől, hogy megkíséreljük a függvények létrehozását és alkalmazását a megfelelő sorrendbe rendezni. A sorrend betartása mégis kötelező és ennek oka

egyszerű: a C fordító szeretné megvizsgálni a függvények paramétereinek és visszatérési értékének típusát, hogy a híváskor egyeztethesse az ott használt állandók és változók típusával. Nem az utasításokra, hanem a függvény bemenő és kimenő típusára van szüksége a fordítónak. Ezt kihasználva a nyelv rendelkezik egy eszközzel, amely felmenti a programozót a függvények használat alapján való sorrendbe rendezésétől.

A következő példaprogramban eltértünk a szokásos létrehoz-használ sorrendtől. A függvénynek előbb meghatározzuk a paraméterlistáját és visszatérési értékét, utána használjuk és csak a program végén hozzuk létre, csak ott írjuk le a függvényt alkotó utasításokat.

<pre> 1 int atlag( int a, int b ); 2 3 main(){ 4     int elseo; 5     int masodik; 6     int matematikai_atlag; 7 8     elseo=42; masodik=43; 9     matematikai_atlag=atlag(elseo, masodik); 10 } 11 12 int atlag( int a, int b ){ 13     return( (a+b)/2 ); 14 }</pre>	<i>alap/fuggveny.c</i>
---	------------------------

A példaprogram 1. sorában láthatjuk a függvény paramétereinek és visszatérési értékének típusát meghatározó sort. Amint látjuk, a függvénynek itt a szokásos módon adjuk meg a leírását, de utána nem a függvény utasításai következnek, hanem egy ; karakter, amely az utasításokat és a változók leírását is követi.

Magát a függvényt ebben a programban csak a `main()` után találjuk, a 12–14. sorban. Amint látjuk, a szokásos módon jártunk el, azaz itt újra leírtuk a függvény típusát és utána a függvényt alkotó utasítások következnek.

## Programok több állományban

Most, hogy sikerült a függvények típusát és a „testét” egymástól különválasztani, szétszedhetjük a forrásprogramot különálló állományokra. A munka során a függvény létrehozása és alkalmazása külön-külön állományokba kerül, mi pedig közelebb jutunk a gyakorlatban használt módszerekhez, amelyek során egy programot több programozó készít, megosztva egymás közt a feladatokat és az állományokat.

Helyezzük a függvény létrehozását a num.c nevű állományba, típusainak létrehozását pedig a num.h állományba!

Ekkor a num.c tartalma a következő lesz:

```
alap/darabol/num.c
```

```
1 int atlag( int a, int b ){
2     return( (a+b)/2 );
3 }
```

A num.h állományba kerül a függvény típusainak leírása:

```
alap/darabol/num.h
```

```
1 int atlag( int a, int b );
```

A fennmaradó részt – a függvény alkalmazását – helyezzük a main.c nevű állományba. Ennek tartalma a következő lesz:

```
alap/darabol/main.c
```

```
1 main(){
2     int also;
3     int masodik;
4     int matematikai_atlag;
5
6     also=42; masodik=43;
7     matematikai_atlag=atlag(also, masodik);
8 }
```

Most egy fontos lépés következik: gondoskodnunk kell az állományok összekapcsolásáról, vagyis rá kell vennünk valahogy a fordítóprogramot, hogy a program fordítása előtt illessze össze a szétdarabolt programot. Erre a feladatra a C fordító előfeldolgozó utasításait használjuk.

Az előfeldolgozó utasításokat a fordítás előtt – külön menetben – végzi el a C fordító, ezért mondhatnánk, hogy ezek nem a C programozási nyelv elemei, de szerencsére minden C fordító szabványosan kezeli ezeket is, hiszen a C nyelvet leíró szabványok az előfeldolgozó utasításait is szabványosították. Aki C nyelven programoz rendszeresen, biztos kézzel kezeli az előfeldolgozó utasításokat is.

Az első lépésben gondoskodjunk arról, hogy a main.c fordításakor a num.h állomány tartalma bekerüljön az állomány elejére. (Emlékezzünk rá, azért van erre szükség, hogy a fordító megtalálja az atlag() nevű, általunk megírt C függvény típusát.) Mivel az előfeldolgozó #include parancsa szolgál az állományok beillesztésére, a main.c a következő formát kapja:

```

1 #include "num.h"
2
3 main(){
4     int elso;
5     int masodik;
6     int matematikai_atlag;
7
8     elso=42; masodik=43;
9     matematikai_atlag=atlag(elso, masodik);
10}

```

A program első sorában található előfeldolgozó utasítás arra utasítja a fordítóprogramot, hogy mielőtt elkezdené fordítani a programot, illessze be a num.h állomány tartalmát. A beillesztést természetesen nem úgy végzi a program, hogy fizikailag bemásolja (nem volna szerencsés, ha hozzájárulna a programozó munkájához), inkább úgy kell elképzelni, hogy az általa használt munkaváltozatban végzi el ezt a módosítást.

A következő lépésben arról gondoskodunk, hogy a num.c állomány elejére kerüljön be a num.h tartalma. (Emlékezzünk rá, enélkül nem rendezhetnénk tetszőleges sorrendbe a függvényeket az állományon belül.) Könnyű kitalálni, hogy a num.c állomány hogyan fog kinézni:

```

1 #include "num.h"
2
3 int atlag( int a, int b ){
4     return( (a+b)/2 );
5 }

```

A program első sorában található előfeldolgozó utasítás arra utasítja a fordítót, hogy másolja be az atlag() függvény típusát leíró állományt a forrásprogram általa használt munkaváltozatába, mielőtt a fordítást elvégezné. Ez most még feleslegesnek tűnik, de ahogy a fejlesztés során egyre több – egymást is hívó – függvény kerül az állományba, igen hasznosnak fog bizonyulni.

A munkával ezen a ponton végeztünk. A num.c állományban található a függvény, minden gond nélkül lefordítható, a main.c állományban használjuk a függvényt, reméljük ezzel sem lesz gondunk. Reményeink hiábavalóak: ha a szokott módon próbáljuk meg lefordítani programunkat, keservesen csalódunk:

```
Bash$ gcc num.c
/usr/lib/crt1.o: In function '_start':
/usr/lib/crt1.o(.text+0x18): undefined reference to 'main'
```

```
collect2: ld returned 1 exit status
Bash$
```

A hibaüzenet szerint a program nem találja a `main()` függvényt, amely a program belépési pontja. Az észrevétel jogos, hiszen amint láttuk, minden C nyelvű programnak tartalmaznia kell egy `main()` függvényt és a `num.c` állomány ilyet bizony nem tartalmaz.

Próbáljuk meg a `main.c` néven elhelyezett állományt lefordítani, hiszen abban megtalálható a `main()`:

```
Bash$ gcc main.c
/tmp/ccUqLswq.o: In function ‘main’:
/tmp/ccUqLswq.o(.text+0x20): undefined reference to ‘atlag’
collect2: ld returned 1 exit status
Bash$
```

Az eredmény várható volt, innen meg az `atlag()` nevű függvény hiányzik.

Megoldást a következő – kissé bonyolultnak tűnő – utasítássorozat adja:

```
Bash$ gcc -c main.c
Bash$ gcc -c num.c
Bash$ gcc main.o num.o
Bash$
```

Az első parancs lefordítja a `main.c` nevű programot, de nem készíti el a program futtatható változatát – láttuk már, az nem is sikerülne –, hanem egy gépi kódú utasításokat tartalmazó, de nem teljes programot készít. Erre az üzemmódra a `gcc`-t a `-c` kapcsoló utasítja. A kapcsoló hatására a fordító csak a fordítást (*compiling*) végzi el, aminek az eredménye egy `main.o` nevű, úgynevezett tárgykódú program (*object code*) lesz. Ez önállóan nem futtatható, de tartalmazza az önállóan futtatható program egy részét.

A második utasítás célja hasonló, elkészíti a `num.c` állományból a `num.o` tárgykódú programot tartalmazó állományt, amely a kész programnak szintén csak egy részét tartalmazza.

A végső simításokat az utolsó parancs hatására végzi el a rendszer; a `main.o` és a `num.o` állományok összeszerkesztésével előállítja a program végső, futtatásra alkalmas változatát.

Valójában az utolsó lépést nem a C fordító végzi. Ha a `gcc` azt tapasztalja, hogy a neki átadott minden állomány tárgykódú programot – nem pedig C nyelven megírt szöveget – tartalmaz, elindítja az `ld` nevű programot és azzal végezeti el a munkát. Az `ld` a Unix rendszereken a programok szerkesztését végző program, amelyet valójában „titokban” akkor is meghív a fordító, ha egyetlen állományban van a programunk. Ezt könnyen tetten érhetjük, ha megfigyeljük az előző, sikertelen fordítási kísérletünk hibaüzeneteit.

Miután sikerült egy meglehetősen bonyolult folyamat során lefordítanunk a programot, elgondolkodhatunk azon, hogy miért volt érdemes ilyen összetett feladatot vállalnunk annak érdekében, hogy a programunkat három részre vágjuk.

Amint már említettük, mindenre azért van szükség, hogy a programunkon egyszerre több programozó is dolgozhasson. Ha megvizsgáljuk a példaprogramot, akkor látjuk, hogy az egy főprogramból (`main.c`) és egy matematikai problémák megoldására szolgáló programrészletből (`num.c`) áll. Nyilvánvaló, hogy a matematikai programrész nem kell ugyanannak elkészítenie, aki a főprogramot írja, sőt valójában a rendszer bővíthető: sok kis részfeladatra számtalan külön állományt tartunk fenn. Így válik lehetővé, hogy a program egészét számtalan programozó közösen készítse el.

A programozók közt kapcsot a példában a `num.h` állomány adja. A matematikai feladatokat elvégző programozó ebben az állományban írja le, hogy az általa készített függvényeket hogyan kell használni. Még csak az sem szükséges, hogy minden általa írt – és a `num.c` állományban elhelyezett – függvényt leírjon, elegendő, ha csak azokra szorítkozik, amelyeket mások is használnak. Így az ilyen `.h` végződésű (*header*, fejléc) fejállományokba általában csak azoknak a függvényeknek a meghatározása kerül bele, amelyeket mások is használnak. Az apró részfeladatokat elvégző függvények, amelyeknek megismerése feleslegesen terhelné a többi programozót – hiszen úgysem hívniuk kell – nem kerülnek be a fejállományokba.

Valójában sokszor még arra sincs szükség, hogy a `num.c` állományt lefordítsa a programozó, aki a `main()` függvényt készíti. Ha megkapja a másik programozótól a `num.o` és `num.h` állományokat, akkor a `num.c` állomány lefordítása nélkül is elkészítheti a program végső változatát. Ez azért igen fontos, mert – ahogyan azt látni fogjuk – a munkánk során általában sokszorosa az átvett program a saját magunk által írtak és valószínűleg nem szeretnénk, ha több ezer sornyi programkódot kellene lefordítanunk egy pár soros saját munka elkészítéséhez.

## Kiíratás

Bizonyára sokakban felmerül a kérdés, hogy miért kell csoportos munkával, mások által írt C nyelvű programokkal foglalkoznia valakinek, aki még csak most ismerkedik a nyelv elemeivel. Nyilván sokan gondolják úgy, hogy távol állnak attól, hogy másokkal együtt dolgozva készítsenek C nyelven programokat, ezért számukra a több állomány kezelése egyelőre felesleges. Ez egyáltalán nem így van, ahogyan ezt az üzenetek kiíratása kapcsán látni fogjuk.

A C nyelv önmagában meglehetősen kevés eszközt ad a programozó kezébe. Igen rugalmas nyelvről van szó, amely szinte minden lehetővé tesz, amellyel egy tapasztalt programozó gyakorlatilag minden meg tud tenni, de tudnunk kell, hogy önmagában igen kevés eszközt ad a feladatok megoldására. Bármilyen meglepő, a C nyelvben például nincs olyan eszköz, amely alkalmas üzenet kiíratására a szabványos kimenetre. (Ezért van az, hogy az eddigi példaprogramjaink egyetlen betűt

sem írtak a képernyőre.)

Szerencsére azonban nem kell attól tartanunk, hogy magunknak kell megoldani ezt a problémát: természetesen készültek már erre a feladatra függvények, amelyek beépíthetők programjainkba. Ezek a függvények elérhetők számunkra, programjainkban felhasználhatjuk őket, mint ahogyan ezt a már bemutatott klasszikus példa-programunk is mutatja:

```
1 #include <stdio.h>
2
3 main(){
4     printf("helló világ\n");
5 }
```

alap/hellovilag.c

Köszönhetően az előző részeknek, most már értelmezni is tudjuk ezeket a sorokat.

Az első sor egy előfeldolgozó utasítás, amely ahoz szükséges, hogy mások által megírt függvények típusainak meghatározását betöltsük a fordítás előtt. Itt az stdio.h fejállomány betöltése látható. Az állománynév a szabványos bemenet/kimenet rövidítése (*standard input/output*), ebből is látszik, hogy az itt létrehozott függvények a bemenetet és a kimenetet kezelik.

Az #include utasítás után az állománynév nem idézőjelek között van, hanem <> jelek között – amelyeket általában egészen másra szoktunk használni. Ez azt jelenti az előfeldolgozó számára, hogy a beillesztendő állományt nem a munkakönyvtárban találja, hanem a rendszer fejállományok tárolására szolgáló valamelyik könyvtárában. (Ez Unix rendszerek esetében általában a /usr/include/ könyvtárszerkezet, de nem kell foglalkoznunk vele, a fordító elintézi helyettünk.)

A printf() függvény a szabványos kimenetre való formázott kiíratásra szolgál, a segítségével szöveget és a változók értékét írathatjuk ki a szabványos kimenetre. A példaprogramban egyszerűen a helló világ szöveget írattuk ki, de a következő példa bemutatja, hogyan írathatjuk ki a változók értékeit:

```
1 #include <stdio.h>
2
3 main(){
4     int     a, b;
5     a=8;
6     b=123;
7     printf( "a=%d\nb=%d\n", a, b );
8 }
```

alap/printf.c

A program a 6. sorban tartalmazza a printf() hívását, amellyel két változó értékét írattuk ki a szabványos kimenetre. Itt a függvénynek három paramétere van.

Tudnunk kell, hogy a `printf()` függvénynek igen rugalmas a paraméterkezelése: legalább egy paramétert adnunk kell a függvénynek, de adhatunk többet is. Ez meglehetősen szokatlan a C programozási nyelvben, hiszen amint láttuk, a függvény létrehozásakor meg kell adnunk a paraméterek számát. Valójában a rugalmas paraméterszám-kezelés egy viszonylag ritkán használt szolgáltatása a nyelvnek.

A `printf()` függvény első paramétere a formátumot előíró szöveges érték – a formázószöveg `-`, amelyet minden esetben meg kell adnunk. A formázószöveg írja elő, hogy az utána következő paramétereket milyen formában írassa ki a függvény a szabványos kimenetre. A formázószöveg a következő elemeket tartalmazhatja:

**egyszerű karakterek** Az egyszerű karaktereket – amelyeknek nincs különleges jelentése a `printf()` számára – a függvény változtatás nélkül kiírja a szabványos kimenetre.

Majdnem minden karakter ebbe a csoportba tartozik, kivéve a `%` és `\` karaktereket, de a többi karakter is különleges jelentésű lehet, ha a `%` vagy a `\` karakterek után található.

**rövidítések** Bizonyos karaktereket nem tudunk a szövegszerkesztővel beírni a forrásvagyprogramba. Ezeknek a karaktereknek a kiíratásához használhatjuk a kétbetűs rövidítéseket, melyek `\` jellel kezdődnek. A `\t` például a tabulátor karakter kétbetűs rövidítése, amelynek hatására az adott helyre a `printf()` az egybetűs, 9-es ASCII kódú karaktert írja a kimenetre.

**formátumleírók** A formázószöveg után beírt paraméterek értékére a `%` jellel kezdődő formátumleíróval hivatkozhatunk. A formátumleírók legalább két betűből állnak, de több jelet tartalmazó összetettebb kifejezések is lehetnek. Mindenesetre mindenkorral a `%` jellel kezdődnek.

A `printf()` minden formátumleíróhoz hozzárendeli a következő paraméter értékét és azt írja a helyére. A formázószövegen található első formátumleíróhoz a második paraméter, a második formátumleíróhoz a harmadik paraméter tartozik és így tovább.

A példaprogramban a `%d` a következő paraméter értékének kiíratására ad utasítást tízes számrendszerben, míg a `\n` az újsor karakter helyettesítésére szolgál. A példaprogram a következő két sort írja a szabványos kimenetre:

```
Bash$ ./printf
a=8
b=123
Bash$
```

A `printf()` függvényről bővebben a 119. oldalon olvashatunk, a formátumleíró részletes tárgyalását pedig ugyanezen az oldalon találhatjuk.

## Vezérlőszerkezetek

A C nyelvben rendelkezésünkre áll a feltételes utasításvérehajtás, az elől és háltaltesztelő ciklus és még néhány más eszköz az utasítások vérehajtási sorrendjének megváltoztatására. A C nyelv nyelvtana igen hasonlít a bc és az awk nyelvtanához, a vezérlőszerkezetek szinte teljesen ugyanúgy használhatók, ezért sokaknak ismerős lehet ennek a résznek a témaja. A vezérlőszerkezetek megértésében igen nagy segítséget jelenthet az irodalomjegyzékben található *Linux: felhasználói ismeretek* c. könyv második kötete.

Valójában a nyelvcsaládnak a C nyelv egy korábbi változata az ōse; a bc és az awk nyelve azért hasonlít a C nyelvhez, hogy a C nyelvet ismerő felhasználóknak ne kelljen túl sokat tanulniuk. Nem a C nyelv hasonlít tehát az awk és a bc nyelvéhez, hanem éppen fordítva.

A feltételes utasításvérehajtás egy logikai kifejezés igaz vagy hamis értékéhez rendeli hozzá utasítások vérehajtását. A következő példaprogram a feltételes utasításvérehajtást mutatja be:

```
1 #include <stdio.h>
2
3 main(){
4     int      a, b;
5     a=8; b=123;
6     if( a+b<100 ){
7         printf( "Kisebb, mint 100.\n" );
8     }else{
9         printf( "Nem kisebb, mint 100.\n" );
10    }
11 }
```

alap/if1.c

A példaprogram 6. sorában található a feltétel, az  $a+b < 100$  állítás, amely igaz vagy hamis lehet. Ha az állítás igaz, a program a 7. sorban található utasítást – az igaz ágat – hajtja végre, ha hamis, a 9. sorban található utasítást – a hamis ágat. A példában csak egy utasítás szerepel az igaz és a hamis ágakban, de természetesen több utasítást is elhelyezhetünk ezeken a helyeken.

Tudnunk kell, hogy ha a feltételes utasításvérehajtás igaz vagy hamis ágában csak egy utasítás szerepel, a kapcsos zárójelek elhagyhatók. Elhagyható továbbá a teljes hamis ág, ha nincs rá szükség. Ezt mutatja be a következő példaprogram:

```
1 #include <stdio.h>
2
3 main(){
```

alap/if2.c

```

4 int      a, b;
5 a=8; b=123;
6 if( a+b<100 )
7     printf( "Kisebb, mint 100.\n" );
8 else
9     printf( "Nem kisebb, mint 100.\n" );
10
11 if( a+b>120 )
12     printf( "Még 120-nál is nagyobb.\n" );
13 }
```

A feltételes utasításvéghajtás után a leggyakrabban használt vezérlőszerkezet az előtesztelő ciklus, amely szintén a más nyelvekben megszokott módon működik, vagyis a programunk addig ismételgeti a ciklusmagban elhelyezett utasításokat, amíg a feltétel igaz logikai értéket képvisel.

Az előtesztelő ciklus használatát mutatja be a következő példaprogram:

```

1 #include <stdio.h> alap/while.c
2
3 main(){
4     int      a;
5     a=8;
6     while( a<12 ){
7         printf("%d\t%d\n", a, a*a);
8         a=a+1;
9     }
10 }
```

A ciklus a 6. sorban kezdődik és a 9. sorban ér véget. A feltétel szerint addig ismétli a program a ciklust, amíg az a változó értéke kisebb, mint 12. A ciklusmagban található utasítások közül az első kiírja az a és a\*a értékét a szabványos kimenetre, a második pedig növeli a értékét egyvel. A program futtatáskor a következő értékeket írja a szabványos kimenetre:

```
Bash$ ./while
8      64
9      81
10     100
11     121
Bash$
```

Az előtesztelő ciklus elkészítésénél elhagyhatjuk a kapcsos zárójeleket, ha a ciklusmag csak egyetlen utasításból áll.

A C nyelv rendelkezik háultesztelő ciklus létrehozására alkalmas szerkezettel is. A következő program a háultesztelő ciklus használatát mutatja be:

```
1 #include <stdio.h>
2
3 main(){
4     int      a;
5     a=8;
6     do{
7         printf("%d\t%d\n", a, a*a);
8         a=a+1;
9     } while( a<12 );
10 }
```

alap/do.c

A ciklus a 6. sorban kezdődik és a 9. sorban ér véget. Amint látjuk, a feltétel a ciklusmag után helyezkedik el, ezzel is hangsúlyozva, hogy a feltétel vizsgálata a ciklusmag végrehajtása után történik meg. A háultesztelő ciklus addig fut, amíg a while kulcsszó utáni feltétel igaz.

A háultesztelő ciklus készítése során is elhagyhatók a kapcsos zárójelek, ha a ciklusmag csak egyetlen utasításból áll.

A C programozási nyelv is rendelkezik egy egyszerűsített ciklusszervező utasítással, amelynek felhasználásával olvashatóbban készíthetünk egyszerű ciklusokat. A következő példaprogram ezt a szerkezetet mutatja be:

```
1 #include <stdio.h>
2
3 main(){
4     int      n;
5     for( n=0; n<5; n=n+1 ){
6         printf("%d\t%d\n", n, 100-n );
7     }
8 }
```

alap/for.c

A programban a ciklus az 5. sorban kezdődik és a 7. sorban ér véget. A ciklus elején található zárójeles kifejezésben három rész található, melyek pontosvesszővel vannak elválasztva egymástól. Az első rész egy utasítást tartalmaz, amelyet a program a ciklusmag első végrehajtása előtt, egyszer hajt végre, a második rész egy feltétel, amelynek igaz értéke esetén a program végrehajtja a ciklusmagot, a harmadik rész pedig egy utasítás, amelyet a program minden alkalommal a ciklusmag végrehajtása után hajt végre.

A for ciklus előltesztelő ciklus, ezért könnyedén átírható while kulcsszót tartalmazó előltesztelő ciklussá. A példaprogramot átírva a következő programot kapjuk:

```

1 #include <stdio.h>
2
3 main(){
4     int n;
5     n=0;
6     while( n<5 ){
7         printf("%d\t%d\n", n, 100-n );
8         n=n+1;
9     }
10 }
```

alap/forwhile.c

A nyelv rendelkezik egy vezérlőszerkezettel, amely a többszörös elágazás kifejezésére alkalmas. A többszörös elágazás – más néven esetszétválasztás – használatát mutatja be a következő példaprogram:

```

1 #include <stdio.h>
2
3 main(){
4     int x;
5     x=10;
6     switch( x ){
7         case 1:
8             printf("egy\n");
9             break;
10        case 2:
11            printf("kettő\n");
12            break;
13        default:
14            printf("Mit tudom én!\n");
15    }
16 }
```

alap/switch.c

A többszörös elágazás a program 6. sorában kezdődik és a 15. sorában ér véget. Az esetszétválasztás a 6. sorban található x változó értékére épül – mert ez található a switch kulcsszó után – és három ága van. Az esetszétválasztás egyes ágai a 7., 10. és 13. sorokban kezdődnek.

A többszörös elágazás elején mindenkor a switch kulcsszót kell használnunk, amely után – zárójelek között – az esetszétválasztás alapját adó értéket kell megadnunk. Az érték lehet egy változó vagy összetett kifejezés, de a típusának mindenkorban egésznek kell lennie. Az alapot adó érték után egy nyitó kapcsos zárójelnek kell következnie. Az esetszétválasztásnak ezek között a kapcsos zárójelek között kell szerepelnie.

A többszörös elágazáson belül használhatjuk a case kulcsszót. minden ágat ez a kulcsszó vezet be, utána egy értékkal és a kettősponttal. Ha a switch utáni kifejezés értéke megegyezik valamelyik case utáni értékkel, a vezérlés a megfelelő case kulcsszó után folytatódik.

A case után egy vagy több utasítást helyezhetünk el, amelyeket az esetszétválasztás alapját képező kifejezés megfelelő értéke esetén hajt végre a program. A break utasítás elhelyezhető a case kulcsszó utáni utasítások végén. A break hatására a vezérlés a switch szerkezet utáni sorra adódik, vagyis a break az esetszétválasztás egy-egy ágának lezárására használható. Ezt mutatja be a következő példa:

```
switch( m ){
    case 1:
        y=y*2;
        break;
    case 2:
        y=y/2;
        break;
    default:
        printf("Mit tegyek?");
}
printf("Vége");
```

Valójában a break kiírását nem írja elő a C nyelvtana. Ha nem tesszük ki a break kulcsszót, a vezérlés „rácsorog” a következő ágra, de ez nem gond, sokszor ez a célunk. Ezt láthatjuk a következő ábrán:

```
switch( m ){
    case 1:
        y=y*2;
        break;
    case 2:
    case 3:
    case 4:
        y=y/2;
        break;
    default:
        printf("Mit tegyek?");
}
printf("Vége");
```

A programban az `y=y/2;` értékkadás hajtódik végre ha az `m` változó értéke 2, 3 vagy 4.

A `case` kulcsszóhoz hasonlóan használható a `default` kulcsszó is. A `default` után található ágra akkor adódik a vezérlés, ha a többszörös elágazást vezérlő kifejezés értéke egyetlen `case` utáni értékkel sem egyezik meg.

A többszörös elágazás kapcsán ki kell emelnünk, hogy az elágazást vezérlő kifejezés értékének és az egyes ágakhoz rendelt értékeknek mindenkorban egész típusúnak kell lenniük, valamint azt, hogy minden ághoz egyedi értéknek kell tartoznia – nem szerepelhet egyazon érték két helyen. Igaz fontos az is, hogy a `case` utáni kifejezések értékének már a fordítás során ismert egész típusú értéknek kell lenniük, tehát nem szerepelhetnek bennük például változók.

## Megjegyzések a szövegben

Amint látjuk, a vezérlőszerkezetekhez sokszor tartoznak olyan kapcsos zárójelek, amelyek párra csak jóval a vezérlőszerkezet kezdete után jelenik meg a programban. Néha kissé nehéz megtalálni azt, hogy egy adott zárójel pontosan melyik vezérlőszerkezetet zárja le, még akkor is, ha az olvashatóság érdekében a sorokat a megfelelő mintázat alapján beljebb rendezzük. Ilyen esetekben – és mindenkor a programot olvasó felhasználó számára tartunk valamit fontosnak közölni – használhatjuk a C nyelv megjegyzések kezelésére alkalmas rendszerét.

A C fordító minden figyelmen kívül hagy, amit `/*` és `*/` kifejezések között talál. Ha tehát valamilyen megjegyzést kívánunk a programhoz fűzni, ezt közre kell fogunk ezekkel a jelekkel, ahogyan a következő példaprogram is mutatja:

	<code>alap/megjegyzesek.c</code>
1	<code>#include &lt;stdio.h&gt;</code>
2	
3	<code>main(){</code>
4	<code>int i, j;</code>
5	
6	<code>/* A következő két ciklus egymásba van ágyazva,</code>
7	<code>* mindkettő lefut, de a belső többször.</code>
8	<code>*/</code>
9	<code>for( i=0; i&lt;3; i=i+1 ){</code>
10	<code>    for( j=0; j&lt;3; j=j+1 )</code>
11	<code>        printf( "&lt;%d, %d&gt; ", i, j );</code>
12	<code>        printf( "\n" );</code>
13	<code>    }/*for*/</code>
14	<code>}/*main*/</code>

A program 6–8. sora megjegyzés, amely segíti az olvasót, de nem befolyásolja a program fordítását vagy futását. A program 13. és 14. sorában láthatjuk, hogyan fűzhetünk megjegyzést a zárójelhez, hogy a program olvashatóságát javítsuk.

Igen fontos tudnunk, hogy a megjegyzéseket jelző /\* és \*/ kifejezések nem ágyazhatók egymásba, mert a fordító a /\* után minden megjegyzésnek tekint, kivéve a \*/ jelsorozatot.

## Elemi adattípusok

Az eddigi példaprogramokban az int típust adtuk minden változónak. Természetesen más adattípusok is a rendelkezésünkre állnak. Ezeket vesszük sorra ebben a fejezetben, ahol az elemi, oszthatatlan, beépített adattípusokat vesszük sorra.

A legegyszerűbb beépített típuscsalád az egész számok ábrázolására alkalmas család. Ennek a családnak a legkisebb tagja a char, amely a C nyelv minden megvalósításában kötelezően 8 bites. A következő változótípus az int, amelynek méretét a C nyelv nem írja elő, így minden megvalósításban akkora, amekkora az adott processzoron a leghatékonyabban megvalósítható. Ez a szabadság nyilvánvalóan terheket ró a programozóra – akinek olyan programot kell írnia, amely minden processzortípuson helyesen működik –, de hatékonyabbá teszi a C nyelven írt programokat.

Az egész típusosztályba tartozik még a short int (rövid egész), a long int (hosszú egész) és a GNU C fordító esetében a long long int (nagyon hosszú egész). Ezeknek a méretét írja ki a következő példaprogram:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 main(){
5     printf("char      : %2zd bájt\n", sizeof(char));
6     printf("short int   : %2zd bájt\n", sizeof(short int));
7     printf("int        : %2zd bájt\n", sizeof(int));
8     printf("long int    : %2zd bájt\n", sizeof(long int));
9     printf("long long int : %2zd bájt\n", sizeof(long long int));
10 }
```

Amint látjuk, a C nyelv nem írja elő az int méretét, szerencsére azonban eszközt ad a kezünkbe a méret vizsgálatára. A nyelv sizeof() kulcsszava függvényeszerűen használva a paraméterként átadott típus méretét adja meg bájtban. A példaprogram Intel Pentium IV processzorral szerelt számítógépen a következő sorokat írja ki:

```
Bash$ ./egeszmeretek
char          : 1 bájt
short int     : 2 bájt
int           : 4 bájt
long int      : 4 bájt
long long int : 8 bájt
Bash$
```

A következő példaprogram bemutatja, miképpen kérdezhetjük le az adott számítógépen mekkora legnagyobb és legkisebb érték tartozik az egyes változótípusokhoz.

	<i>alap/valtozok/egeszhatarok.c</i>
--	-------------------------------------

```

1 #include <stdio.h>
2 #include <limits.h>
3
4 main(){
5     printf( "char min.:      %d\n", SCHAR_MIN );
6     printf( "char max:       %d\n", SCHAR_MAX );
7     printf( "short int min.: %d\n", SHRT_MIN );
8     printf( "short int max.: %d\n", SHRT_MAX );
9     printf( "int min.:        %d\n", INT_MIN );
10    printf( "int max.:        %d\n", INT_MAX );
11    printf( "long int min.:  %ld\n", LONG_MIN );
12    printf( "long int max.:  %ld\n", LONG_MAX );
13    printf( "long long min.: %lld\n", LLONG_MIN );
14    printf( "long long max.: %lld\n", LLONG_MAX );
15 }
```

Ha le akarjuk fordítani ezt a programot, akkor a gcc számára a `-std=c99` kapcsolóval jelezünk kell, hogy a C nyelv 1999-es bővítéseit is használni kívánjuk, hiszen a `LLONG_MAX` és a `LLONG_MIN` állandók csak ekkor váltak elérhetővé.

A program a már említett Pentium processzoron futtatva a következő eredményt adja:

```
Bash$ gcc -std=c99 intranges.c -o minmax
Bash$ ./egeszhatarok
char min.:      -128
char max:       127
short int min.: -32768
short int max.: 32767
int min.:       -2147483648
int max.:       2147483647
long int min.: -2147483648
long int max.: 2147483647
```

```
long long min.: -9223372036854775808
long long max.: 9223372036854775807
```

Tudnunk kell, hogy a `sizeof()` kulcsszó nem csak a típusok kulcsszavával használható. A következő példaprogram azt mutatja be, hogy a változók nevével is használható a `sizeof()`:

```
1 #include <stdio.h>
2
3 main(){
4     long long int valtozo;
5
6     printf("long long int méret: %zd bájt\n", sizeof(valtozo));
7 }
```

Tudnunk kell, hogy a `sizeof()` kulcsszó minden változótípussal, még a későbbiekben bemutatott összetett típusokkal is használható.

Az egész típusosztálynak alapértelmezés szerint minden tagja előjeles, azaz képes kezelní a negatív számokat is. Ezt a tulajdonságukat hangsúlyozhatjuk a `signed` kulcsszóval a típusnév előtt – például `signed int` –, de ennek semmilyen hatása nincs. Ha viszont az `unsigned` kulcsszóval előjel nélküli típust kérünk, a változók csak pozitív számok tárolására lesznek alkalmasak. Ekkor felszabadul az előjel tárolására használt bit, így nagyobb pozitív számokat is tárolni tudunk bennük. Ezt mutatja be a következő példaprogram:

```
1 #include <stdio.h>
2
3 main(){
4     char         a;
5     unsigned char b;
6
7     a=100+100;
8     b=100+100;
9     printf("signed char:    %d\n", a );
10    printf("unsigned char: %d\n", b );
11 }
```

A program két `char` típusú változót hoz létre, egy előjelest az 4. sorban és egy előjel nélkülit a 5. sorban. A program minden két változóhoz megkísérel 200-at rendelni értékként, majd kiírja őket a szabványos kimenetre. A program a következő kimenetet hozza létre:

```
Bash$ ./tulcsordulas
signed char: -56
unsigned char: 200
Bash$
```

Amint látjuk, az előjeles típus túlcsordult és negatív értéket hordoz, míg az előjel nélküli típus a helyes értéket hordozza.

Igen fontos tudnunk, hogy a C nyelv fordítói nem ellenőrzik a változók túlcsordulását, így a programozó felelőssége, hogy mindenkor csak akkora értéket helyezzen a változókba, amekkorát azok hordozni képesek.

Tizedestörteket kezeln a C nyelv lebegőpontos típusaival tudunk. Lebegőpontos számábrázolásra használhatjuk a float, double és long double típusokat. Ezek méretét a következő példaprogram írja ki:

	<i>alap/valtozok/lebegomeretek.c</i>
--	--------------------------------------

```
1 #include <stdio.h>
2
3 main(){
4     printf("float      : %zd bájt\n", sizeof(float) );
5     printf("double     : %zd bájt\n", sizeof(double) );
6     printf("long double : %zd bájt\n", sizeof(long double) );
7 }
```

A long double típus mérete – a char méretéhez hasonlóan – a használt processzor típusától függ. A program – a már említett Pentium processzorral szerelt számítógépen – a következő kimenetet adja:

```
Bash$ ./lebegomeretek
float      : 4 bájt
double     : 8 bájt
long double : 12 bájt
Bash$
```

A lebegőpontos típusoknál nem használható az unsigned módosító, a lebegőpontos változótípusok minden ábrázolják az előjelet is.

A 120. oldalon részletesen olvashatunk arról, hogy miképpen tudjuk kiíratni az egyes típusok értékét. Itt most egy példaprogram segítségével mutatjuk be, hogyan jeleníthetjük meg az egyes változók értékét számként, tízes számrendszerrel használva.

	<i>alap/valtozok/printf.c</i>
--	-------------------------------

```
1 #include <stdio.h>
2
3 main(){
4     char          Char=1;
```

```

5 short int      ShortInt=2;
6 int           Int=3;
7 long int      LongInt=4;
8 long long int LongLongInt=5;
9
10 float         Float=1.1;
11 double        Double=1.2;
12 long double   LongDouble=1.3;
13
14     printf( "char          %hd\n", Char );
15     printf( "short int    %hd\n", ShortInt );
16     printf( "int          %d\n", Int );
17     printf( "long int     %ld\n", LongInt );
18     printf( "long long int %lld\n", LongLongInt );
19
20     printf( "float         %f\n", Float );
21     printf( "float         %f\n", Double );
22     printf( "long double   %Lf\n", LongDouble );
23 }/*main*/

```

## Összetett típusok

Az elemi adattípusok felhasználásával könnyen készíthetünk összetett típusokat. Az összetett típusok olyan adattípusok, amelyeket nem a C fordító, hanem a programozó hoz létre a konkrét feladat megoldása érdekében.

## A tömb

A legegyszerűbb összetett adattípus a tömb. A tömb azonos típusú változókból készített összetett típus. A következő program egészkből álló tömböt hoz létre és használ:

	<i>alap/valtozok/tombok/egeszek.c</i>
--	---------------------------------------

```

1 #include <stdio.h>
2
3 main(){
4     int tomb[12];
5     int n;
6     for( n=0; n<12; n=n+1 )
7         tomb[n]=n*n-1;

```

```

8
9   for( n=0; n<12; n=n+1 )
10    printf( "tomb[%d]=%d\n", n, tomb[n] );
11 }
```

A tömb létrehozását az 4. sorban figyelhetjük meg. Tömb létrehozásakor a típus és név után szöglletes zárójelek között a tömb méretét kell megadnunk. A példában az 5. sorban egy int típusú, 12 darab egész számot hordozó tömböt hozunk létre.

Igen fontos tudnunk, hogy bármit is írunk a szöglletes zárójelek közé, annak értékét a fordítónak ismernie kell, hiszen csak így képes meghatározni a tömb tárolásához szükséges memoriaterület nagyságát. A tömb méretének tehát már a fordításkor ismertnek kell lennie.

Mivel a példaprogramban használt tömb minden eleme int típusú, minden egyes elemét használhatjuk egy egész tárolására. A példaprogram 7. és 10. sorában láthatjuk, hogyan adunk értéket egy tömbelemnek és hogyan használjuk fel azt. Egy dologra azonban igen fontos, hogy felhívjuk a figyelmet:

A C nyelvben a tömbelemekek megszámozása – a tömb indexelése – mindig 0-tól indul. Az  $n$  elemű tömb használatakor a legkisebb index 0, a legnagyobb pedig  $n - 1$ . Ez a példaprogramunkban azt jelenti, hogy a tömb indexe legfeljebb 11 lehet.

Ügyelnünk kell, hogy ne lépjük túl a tömb lehetséges legnagyobb indexét, mert a fordító és az általa létrehozott program nem ellenőrzi az indexet, ezért a hiba a programfutás megszakadásához vezethet. Ezt mutatja be a következő – szándékosan hibás – példaprogram:

	alap/valtozok/tombok/hibasindex.c
--	-----------------------------------

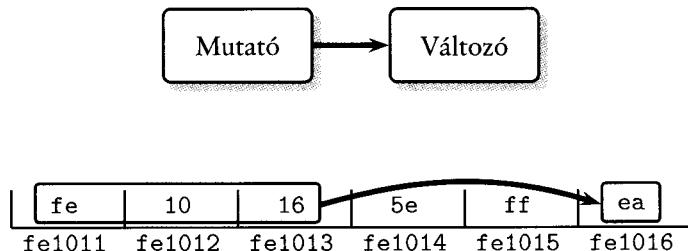
```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 main(){
5     int tomb[3];
6     int n;
7
8     for( n=0; n<5; n=n+1 ){
9         tomb[n]=n*n-1;
10        printf( "tomb[%d]=%d\n", n, tomb[n] );
11    }
12 }
```

A példaprogram jól láthatóan hibás, hiszen a ciklusváltozó a ciklusmag utolsó lefutásakor a 4 értéket veszi fel, amely nem megengedett a tömb indexeként. A program a következő kimenetet adja:

```
Bash$ ./hibasindex
tomb[0]=-1
```

## 1.1. ÁBRA: A mutató



```
tomb[1]=0
tomb[2]=3
tomb[3]=8
tomb[4]=15
Segmentation fault
Bash$
```

A példa futtatásakor tapasztalt **Segmentation fault** (memóriafelosztási hiba) üzenet azt jelzi, hogy a program memóriahibát vétett – vagyis olyan memóriaterülethez próbált meg hozzáférni, amelyhez nem volt jog.

## A mutatók

A C programozási nyelv legfontosabb összetett adattípusa a mutató (*pointer*), amelyet szokás elemi típusnak is tekinteni. A mutatók rugalmasságuk révén igen közkedveltek a tapasztalt programozók körében és igen sok fejfájást okoznak kevésbé tapasztalt kollégáiknak.

A mutató tulajdonképpen egy memóriacím, amely a memória egy bizonyos pontját jelöli ki. Mivel a ma használt számítógépekben a memória alapjában véve bájt szervezésű, a mutatók mindenkor egy bizonyos bájtot jelölnek ki a memóriában.

A fordító ugyan ellenőrizheti – és ellenőrzi is –, hogy a cím, ahová a mutató mutat, milyen típusú adatot tartalmaz, mi a jelölt terület mérete, a mutató azonban nem tartalmaz információt arra nézve, hogy mekkora a mérete a jelölt memóriaterületen található adatnak, így minden mutató mérete azonos. A mutatók mérete nem a kijelölt adattípustól, hanem a futtató számítógép memóriacímzési rendszertől függ. Ha az éppen használt számítógép processzora három bájtot képes használni a memória címzésére, a fordítás során a mutatóink hárombájtosak lesznek. Ez

természetesen nem jelenti azt, hogy az adott számítógépben annyi memória áll a rendelkezésünkre, amennyit a három bájt segítségével címezni tudunk. A mutatók mérete nem a beépített, hanem a processzorral kezelhető memória méretétől függ.

Amint azt már jelezük, a fordító a fordítás során ellenőrzi azt, hogy a mutató a megfelelő típust hordozó területet jelöli-e ki a memóriában. Annak érdekében, hogy ezt megtehesse, a felhasználónak típust kell rendelnie minden mutatóhoz, amikor azokat létrehozza. Léteznek egészeket jelölő mutatók, léteznek lebegőpontos számokat jelölő mutatók és így tovább. Ezt mutatja be a következő program:

```
1 main(){
2     int *mutato;
3     int adat;
4
5     adat=13;
6     mutato=&adat;
7 }/*main*/
```

alap/valtozok/mutatok/mutato1.c

A program 2. sorában egy mutatót hozunk létre, amely int típust képes jelölni a memóriában. Amint látjuk, mutatót úgy hozunk létre, hogy a változó neve elé a \* jelet tessük. Ez nem a mutató nevének része, hanem egy jel, amely a változó létrehozásakor azt jelzi, hogy a változó mutató.

A mutató mellett a 3. sorban egy változót is létrehozunk, amely int típusú. A program 6. sorában adunk értéket a 2. sorban létrehozott mutatónak. A mutató értéke a 6. sor értékadó utasítása után az adat nevű változót jelöli ki a memóriában, vagyis az adat nevű változó legelső bájtjának memóriacímével fog megegyezni. Az értékadásban használt & jel az adat változónév előtt azt jelzi, hogy az adat címét kívánjuk használni, nem pedig az értékét.

Mivel az adat nevű változó int típusú, a mutato nevű változó pedig int típust jelölő mutató, a típusok megegyeznek, a fordító elégedett lehet a munkánkkal. Ilyen esetekben azt mondjuk, hogy a mutató típusa int \*, amely alkalmas int értékeket kijelölni.

Valójában a fordító akkor sem tagadja meg a program fordítását, ha a mutatók típusát nem a megfelelő módon egyeztetjük a kijelölt területek típusával. Ilyenkor a fordító elvégzi a fordítást – hiszen az a mutatók azonos mérete miatt elvégezhető –, de figyelmeztetést ír a szabványos hibacsatornára, ahogyan a következő példa mutatja:

```
1 main(){
2     int *mutato;
3     char adat;
4
5     adat=13;
```

alap/valtozok/mutatok/mutato2.c

```

6     mutato=&adat;
7 }/*main*/

```

A példaprogram hibásan kezeli a mutatókat. A 2. sorban létrehozunk egy int \* típusú mutatót, majd a 6. sorban egy char \* mutató helyett használjuk. A program fordítását a fordító ennek ellenére nem tagadja meg, viszont hibaüzenetet – figyelmeztetést (*warning*) – ír ki a fordítás során:

```

Bash$ gcc mutato1.c
mutato1.c: In function ‘main’:
mutato1.c:6: warning: assignment from incompatible pointer type
Bash$

```

Ha meg szeretnénk szabadulni a figyelmeztetésről, jeleznünk kell a szándékunkat. Erre a típuskényszerítés (*casting*) használható, amely a változó elé zárójelben írt új típust jelenti. Ezt mutatja be a következő példa:

```

alap/valtozok/mutatok/tipusvaltas.c
1 main(){
2     int     *mutato;
3     char    adat;
4
5     adat=13;
6     mutato=(int *) &adat;
7 }/*main*/

```

A példaprogramban a 6. sorban az adat változót jelölő mutatót – &adat – úgy tekintjük, mintha int változóra mutatna, a típusa int \* lenne. Ez átalakítást igazából nem igényel – hiszen minden mutató azonos méretű.

A következő példaprogram újabb adalékkal szolgál a mutatók használatának megértéséhez, amennyiben a mutató által jelölt területet módosítjuk benne:

```

alap/valtozok/mutatok/mutatokovetes.c
1 #include <stdio.h>
2 main(){
3     int     *mutato;
4     int     adat;
5
6     mutato=&adat;
7     *mutato=42;
8     printf( "Új érték: %d\n", adat );
9 }/*main*/

```

A példaprogramban a 6. sorban az adat nevű változó címét a mutato nevű változóba helyezzük, amely egy mutató a megfelelő típussal. A 7. sorban a mutató által

kijelölt területnek megváltoztatjuk az értékét. Itt a `mutató` név előtt elhelyezzük a `*` karaktert, amely azt jelzi a fordító számára, hogy itt nem a mutatóról, hanem az általa jelölt memóriaterületről van szó. Ez a mutató követése (*pointer dereference*), ami nem tévesztendő össze a mutató létrehozásával (3. sor), még akkor sem, ha itt is és ott is a `*` jelet kell használnunk a mutató neve előtt.

A mutató követése igen jól használható a függvényekben történő változóelérésre, amelynek segítségével a függvényeken belül érhetjük el a más függvényekben létrehozott változóinkat.

A függvényeken belül létrehozott változók hatóköre a függvényekre korlátozódik, ezért ezeket helyi változóknak (*local variables*) nevezzük. A függvénynek átadott paraméterek is helyi hatókörűek, ahogyan ezt a következő példa is mutatja:

```
alap/valtozok/helyivaltozo.c
```

```

1 #include <stdio.h>
2
3 int novel( int a ){
4
5     a=a+1;
6     return( a );
7 }/*novel*/
8
9 main(){
10    int adat;
11
12    adat=42;
13    novel( adat );
14    printf( "adat=%d\n", adat );
15 }/*main*/

```

A példaprogram a 2–6. sorokban egy függvényt tartalmaz, amely a paraméterként kapott egész értékét növeli eggyel. A program `main()` függvénye a 12. sorban értéket ad az `adat` nevű változónak, majd átadja a `novel()` függvénynek. A függvény visszatérése után a program a 14. sorban kiírja az `adat` értékét. A program futása a következő eredményt adjja:

```
Bash$ ./helyivaltozo
adat=42
Bash$
```

Amint látjuk, a program a változó eredeti értékét írja a szabványos kimenetre. Ez azért van így, mert a C nyelvben a függvényhíváskor csak a paraméterek értéke adódik át, a C nyelv paraméterátadása érték szerinti paraméterátadás. A fordító olyan kódot hoz létre, amely minden függvényhíváskor másolatot készít az átadandó paraméterek értékéről és azokat hozzárendeli a függvény létrehozásakor megadott nevű

helyi változókhöz. A helyi változók a függvény futása után megsemmisülnek, így a paraméterként kapott értékek bármilyen változtatása elvész a hívó számára.

Ha a függvényeken belül történő változtatásokat meg szeretnénk őrizni a hívó számára, létrehozhatunk általános érvényű változókat. Az általános érvényességi körű változók (*global variables*) létrehozását minden függvényen kívülre kell helyeznünk, ahogyan azt a következő példaprogram bemutatja:

```
1  #include <stdio.h>
2
3  int also;
4
5  int masodik;
6
7
8  novel(){
9      also=also+1;
10     masodik=masodik+1;
11 }
12
13 main(){
14     also=42;
15     masodik=43;
16     novel();
17     printf( "also=%d\nmasodik=%d\n", also, masodik );
18 }
```

Amint láthatjuk, a program 2–3. sorában két általános érvényességi körrel felruházott változót hozunk létre. Ezeknek a változóknak a 11–12. sorban adunk értéket, majd meghívjuk a `novel()` nevű függvényt, amely növeli minden két változó értékét. A program a következő kimenetet hozza létre:

```
Bash$ ./altalanosvaltozo
also=43
masodik=44
Bash$
```

Vegyük észre, hogy az általános érvényességű változók használata lehetővé teszi, hogy egy függvény több értéket is visszaadjon. A példaprogram két változó értékét is módosította. Csak úgy volt erre képes, hogy a használt változók általános érvényessége miatt a hívó és a hívott függvény is elérte azokat.

Sajnos azonban ez a módszer – az általános érvényességű változók használata – nem túl kedvező. Az ilyen változókat minden függvény eléri és ez nem túl szerencsés a feladat részekre bontásának szempontjából. Igazi megoldást a mutatók használata ad, ahogyan ezt a következő példaprogram bemutatja:

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

```

3 novel(int *a, int *b){
4     *a==*a+1;
5     *b==*b+1;
6 }/*novel*/
7
8 main(){
9     int elso;
10    int masodik;
11
12    elso=42;
13    masodik=43;
14    novel( &elso, &masodik );
15    printf( "elso=%d\nmasodik=%d\n", elso, masodik );
16 }/*main*/

```

Láthatjuk, hogy a példaprogram nem a módosítandó változókat – azok értékét – adta át a hívott függvénynek, hanem a módosítandó értékek címét. Így, a címekhez hozzájutva, a hívott függvény közvetlenül elérheti és módosíthatja a hívó függvény helyi változóit. Ez a módszer általánosan elterjedt és igen sokszor használjuk, ezért mindenkorban érdemes megismernünk vele, úgy is mint a mutatók használatának szép példájával.

## Mutatók és tömbök

A mutatók és a tömbök a C nyelvben szoros rokonságban vannak egymással, ahogyan azt a következő példaprogram is mutatja:

	alap/valtozok/mutatok/index.c
--	-------------------------------

```

1 #include <stdio.h>
2
3 kiir( int *tomb ){
4     printf( "tomb[0]=%d\n", tomb[0] );
5     printf( "tomb[1]=%d\n", tomb[1] );
6 }/*kiir*/
7
8 main(){
9     int t[5];
10    t[0]=42;
11    t[1]=43;
12    kiir( t );
13 }/*main*/

```

A példaprogramban egy tömb elemeit írjuk ki a szabványos kimenetre egy függvény segítségével. A függvény a következő kimenetet hozza létre:

```
Bash$ ./index
tomb[0]=42
tomb[1]=43
Bash$
```

Figyeljük meg a program harmadik sorában, hogy az ott létrehozott függvény egy mutatót kap paraméterként, amely int típusú változót jelöl a memóriában. A függvény azonban a mutatót tömbként kezeli: a 4–5. sorokban a mutatót tartalmazó helyi változó neve után [ ] jelek között egy index található. A mutatók tehát minden különösebb előkészület nélkül használhatóak tömbszerűen.

Ami talán még meglepőbb, a program 12. sorában látható függvényhívás. A függvényt int \* típusú mutatóval kell meghívni, mi azonban a 9. sorban létrehozott tömb nevét írtuk ide, ráadásul a [ ] jelek nélkül. E lépés megértéséhez tudnunk kell, hogy a C nyelvben a tömb nevének [ ] jelek nélkül való leírása egyenértékű a legelső – a 0 indexű – elem címével.

A tömb neve tehát a legelső elem címe, az utána található – szögletes zárójelek között elhelyezett – index pedig azt mutatja, mennyit kell az első elem címéhez hozzáadni. A mutató típusából tudja a fordító, hogy mekkora helyet foglalnak a tömb elemei, hány bájt nyit kell a memóriában előrelépni, hogy a következő elemhez jussunk. Az átadott mutató típusa tehát fontosabb, mint azt első pillantásra gondoltuk volna.

Általánosan elterjedt fogás a C nyelvű programozásban, hogy a tömb címével együttes a tömb méretét is átadjuk, hiszen másképpen nem tudhatjuk meg a hívott függvényen belül a tömb méretét. Ennek a módszernek a bemutatására szolgál a következő példaprogram, amely egy tömb legnagyobb elemét választja ki:

```
alap/valtozok/tombok/parameter.c
1 #include <stdio.h>
2
3 int legnagyobb( int *tomb, int meret ){
4     int    n;
5     int    max;
6     max=tomb[0];
7     for( n=1; n<meret; n=n+1 )
8         if( max<tomb[n] )
9             max=tomb[n];
10    return(max);
11 }/*legnagyobb*/
12
13 main(){
14     int    t[5];
```

```

15     t[0]=42;
16     t[1]=40;
17     t[2]=12;
18     printf( "Legnagyobb=%d\n", legnagyobb(t, 3) );
19 }/*main*/

```

Figyeljük meg, hogyan adtuk át a `legnagyobb()` nevű függvénynek a tömb címét és méretét két külön paraméterben. Láthatjuk, hogy bármekkora tömböt átadhatunk ilyen egyszerűen, két paraméter felhasználásával.

Igen fontos érték a C nyelvben a NULL érték, amelyet általában a beállítatlan, valós címet nem hordozó mutatók értékének beállítására használunk. A NULL értéket hordozó mutató nem mutat olyan helyre, amely helyről a programunk olvashat és írhat, és ez fordítva is igaz, vagyis a programunk által olvasható és írható memóriaterületet jelölő mutatók értéke soha nem NULL. Mivel a tiltott memóriaterületre való írás és az onnan való olvasás a folyamatok számára általában egyenértékű a halálos ítéettel, meglehetősen fontos, hogy minden pontosan tudjuk, hogy a programunkban használt mutató mikor jelöl ki használható területet és mikor nem hordoz ilyen információt. Épp erre használjuk a NULL értéket.

Az általánosan elterjedt szokás szerint, amikor létrehozunk egy mutatót, azonnal beállítjuk értékét, hogy az NULL legyen. Mielőtt használnánk a mutatót, minden ellenőrzéssel, hogy értéke NULL érték-e, és csak akkor használjuk, ha nem. Nyilvánvalóan csak akkor lesz a mutató értéke a NULL értéktől különböző, ha a programunk értéket ad neki, valamely használható memóriaterület címét helyezve el a mutatóban. Ezt a folyamatot mutatja be a következő példa:

alap/valtozok/tombok/kezdetierte.c
------------------------------------

```

1 #include <stdio.h>
2
3 int osszeg( int *t, int darab ){
4     int ossz=0;
5     int n;
6     if( t==NULL )
7         return( -1 );
8
9     for( n=0; n<darab; ++n )
10        ossz=ossz+t[n];
11
12    return( ossz );
13 }/*osszeg*/
14
15 main(){
16     int tomb[10]={ 1, 3, 4, 5, 2, 2, 12, 1, 2, 3 };
17     int *t=NULL;
18

```

```

19     printf( "Hibásan: %d\n", osszeg(t, 10) );
20     t=tomb;
21     printf( "Helyesen: %d\n", osszeg(t, 10) );
22 }/*main*/

```

A példaprogramban a 3–13. sorban található egy függvény, amely egészekből képzül tömb elemeit adja össze. A függvény paraméterei közül az első a tömb legelső elemét jelöli ki a memóriában, a második paramétere pedig a tömb elemeinek számát adja meg. A második paraméterre szüksége van a függvénynek, hiszen a tömböt jelölő mutatóból nem lehet kideríteni, hogy mekkora a tömb, vagyis hány elemet tartalmaz.

A függvény a 6–7. sorban megvizsgálja, hogy a mutató értéke nem NULL érték-e. Ha igen, azonnal visszaadja a vezérlést a hívónak, hiszen nem kapott olyan memóriaterületet, amelyen dolgozni tudna. A NULL értékű mutató biztosan nem jelöl ki a folyamat által használható területet a memóriában.

A függvény a 9–10. sorokban egy ciklust tartalmaz, amely a tömb minden elemén végighaladva összeadja azokat az össz változóban. A ciklus futtatása előtt ezt – az összeget tároló – változót természetesen 0-ra kell állítanunk. A függvény ezt a 4. sorban teszi, a változó létrehozásakor. Igen érdekes ez a szerkezet, hiszen a változó létrehozását – típusának meghatározását – és az értékének beállítását is tartalmazza.

A 15–22. sorokban található main() függvény az összeadást végző függvényt hívja próbaadatokkal a működés bemutatásához. A 17. sorban egy mutatót hozunk létre, gondosan ügyelve arra, hogy értékét már a létrehozásakor NULL értékre állítsuk, ezzel is jelezve, hogy a mutató nem a folyamat által birtokolt memóriaterületre mutat. Ha ezzel a NULL értékkel hívjuk meg az összeadást végző függvényt – mint ahogyan meg is tessük a 19. sorban –, a függvény nem követi a mutatót, hiszen első lépésként éppen azt vizsgálja, hogy nem NULL értékű-e.

Ha a 17. sorban nem állítanánk be a mutató értékét, akkor az bármilyen értéket hordozhatna – hiszen a C nyelvben azoknak a változóknak az értéke, amelyeknek nem adtunk értéket, bármi lehet. A „valahova mutató” változót azonban nem tudjuk megkülönböztetni azuktól a változóktól, amelyek hivatalosan a program által birtokolt memóriaterületre mutatnak, és éppen ennek a problémának a megoldására használható a NULL érték.

Figyeljük meg a program 20. és 21. sorában, hogy a mutatónak értéket adunk és újra meghívjuk az összeadást végző függvényt. A mutató most már nem NULL értékű – pontosan a tomb nevű tömb legelső elemét jelöli a memóriában –, ezért az összeadást végző függvény lefut és az összeget adja vissza.

Valójában a NULL állandó nem a C programozási nyelv része, használatához fejállomány betöltésére van szükség, de a C könyvtár oly sok eleme használja, hogy gyakorlatilag bármely fejállomány betöltésével elérhetővé válik.

A program 16. sorában található az összeadandó tömb létrehozása. Érdekes módon itt is már a létrehozáskor megadjuk az értéket, sőt mivel tömbről van szó, az

értékeket. Amint látjuk, a tömbnek a létrehozásakor értéket adni a kapcsos zárójelek segítségével lehet. Ha a tömb létrehozásakor a benne tárolt értékeket is meg akarjuk adni, soha nem szabad megfeledekeznünk a kapcsos zárójelek használatáról.

Könnyítésképpen kihasználhatjuk, hogy a fordító képes kiszámítani a megadott állandók tárolására alkalmas tömb méretét. Amikor tehát tömböt hozunk létre és a benne található elemek értékét is megadjuk, elhagyhatjuk a tömb méretét:

```
1 int tomb[10]={ 1, 3, 4, 5, 2, 2, 12, 1, 2, 3 };
```

Amikor függvényeken belül hozunk létre változókat, meglehetősen szabad kezet kapunk a létrehozásukkor megadott érték tekintetében. A következő példa bemutatja, hogy akár függvényt is hívhatunk a kezdeti érték beállításához:

	<i>alap/valtozok/kezdetiertek.c</i>
--	-------------------------------------

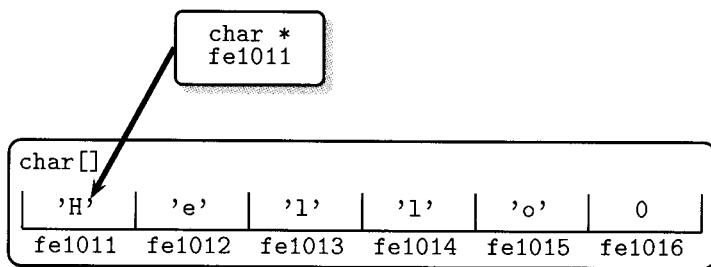
```
1 #include <stdio.h>
2
3 int sum( int a, int b ){
4     return( a+b );
5 }/*sum*/
6
7 main(){
8     int a=21;
9     int b=a;
10    int c=sum(a, b);
11    int d;
12    printf("a=%d, b=%d, c=%d\n", a, b, c);
13 }/*main*/
```

Egészen más azonban a helyzet a függvényeken kívül létrehozott – általános érvényességű – változókkal. Ezek számára a fordító már a fordításkor lefoglalja a helyet a programban, így értéküknek már a fordításkor ismertnek kell lenni. Ha a programot átalakítjuk a következő formára, a fordítás nem fog sikerülni:

	<i>alap/valtozok/kezdetiertek-hibas.c.program</i>
--	---

```
1 #include <stdio.h>
2
3 int a=21;
4 int b=a;
5 int c=sum(a, b);
6 int d;
7
8 int sum( int a, int b ){
9     return( a+b );
10 }/*sum*/
```

## 1.2. ÁBRA: A karakterlánc



```

12  main(){
13      printf("a=%d, b=%d, c=%d\n", a, b, c);
14  }/*main*/

```

A program fordítása során a C fordító megtagadja a fordítást, mivel a 4. és 5. sorban található kezdeti értékkadás a fordítás során nem végezhető el:

```

Bash$ gcc kezdetiertekek-hibas.c
mutato2.c:4: initializer element is not constant
mutato2.c:5: initializer element is not constant
Bash$

```

## Karakterláncok

Már utaltunk rá, hogy a C programozási nyelv nem támogatja a karakterláncok kezelését, azaz nem létezik a C nyelvben olyan beépített változótípus, amely alkalmas lenne karakterláncok kezelésére. Ez valójában csak félígazság, mert a `char` változótípusból készített tömbök segítségével igen egyszerűen tudunk karakterláncokat kezelni.

A C nyelven megírt programok `char` típusból készített tömbök segítségével kezelik a karakterláncokat, így ha a C nyelv kapcsán karakterláncokról beszélünk, minden tömböt értünk ezen a fogalmon.

Igen érdekes az, ahogyan a karakterláncok végét jelzi a nyelv. Megegyezés szerint a karakterláncok végét minden a 0 kódú karakter jelzi, a karakterláncainkat tároló tömbökben tehát egy 0 értékkel kell jelezniünk a szöveg végét. (A 0 kódú karakter nem a 0 számjegy, hanem a 0 értékű bájt.) Ennek bemutatására a következő példa-program szolgál:

```

1   alap/valtozok/karakterlancok/hossz.c
2
3 #include <stdio.h>
4
5 int hossz( char *tomb ){
6     int n;
7     n=0;
8     while( tomb[n] != 0 )
9         n=n+1;
10    return( n );
11 }
12
13 main(){
14     char *szoveg;
15
16     szoveg="Hello";
17     printf( "A 'Hello' hossza: %d\n",
18             szoveg, hossz(szoveg) );
19 }

```

A program kiír a szabványos kimenetre egy szöveget, megszámlálja hány betűből áll és kiírja a hosszát is:

```

Bash$ ./hossz
A 'Hello' hossza: 5
Bash$

```

A programban több érdekes újdonság is van, amelyek nagyon fontosak a munkánk szempontjából. Láthatjuk, hogy a `printf()` függvény (15. sor) első paramétere kérint szereplő formázószövegben egy `%s` kifejezés található. Ez azt jelzi a függvény számára, hogy a következő paraméter egy mutató, amely karaktertömböt jelöl ki a memoriában. A `%s` hatására a függvény sorjában kiírja az itt található karaktereket, egészen az első 0 karakterig. Látható, hogy a szabványok által előírt függvények követik a hagyományt, miszerint a karaktertömbök végét 0 karakterrel kell jelölni. Figyeljük meg, hogy a függvény második paramétere – amelyhez a `%s` tartozik – egy `char *` típusú mutató, így a típusa megfelelő.

Érdekes azonban, ahogyan ezt a mutatót előkészítettük. A program 12. sorában hozzuk létre a szöveg nevű változót, amelynek típusa `char *`. Értéket a mutató a 14. sorban kap, ahol " " jelek között egy szöveget találunk. A kettős idézőjelek a C nyelvben karakterlánc típusú állandók jelzésére szolgálnak. A " " jelek hatására a fordító a közrezárt szöveget elhelyezi a futtatható állományban, a szöveg végére egy 0 karakter helyez – a szöveg lezárására –, a programba pedig, a karakterlánc helyére a memóriacímét teszi, amely a szöveget tartalmazza. C nyelven tehát a " " jelekkel közrezárt szöveges állandók kifejezésben elfoglalt típusa `char *`.

Érdekes a program 3–9. sorában létrehozott függvény is. Ez karakterláncok hosszának megállapítására szolgál. A függvényben a 6–7. sorban található while elöltesztelő ciklus addig vizsgálja a tömb karaktereit, amíg az első 0 karaktert meg nem találja. A ciklus feltételében szereplő != a „nem egyenlő”, akkor ad igaz értéket, ha a két oldalán található kifejezések értéke nem egyezik meg.

A 7. sorban található ciklusmag a szövegen való haladás közben minden lépében növeli n értékét, amely a ciklus végén a szövegben található karakterek számát adja. Ezt a változót adja vissza a függvény a hívónak, amely így a karakterek számát kapja. Figyeljük meg, hogy a ciklus nem számolja bele a hosszba a 0 karaktert, így tulajdonképpen eggyel kevesebb a visszaadott szám a szöveg tárolásához szükséges bájtok számánál.

A példaprogramban elkészített `hossz()` függvényhez igen hasonló módon viselkedik az `strlen()` függvény, amelyről bővebben a 73. oldalon olvashatunk.

Az, hogy a C nyelv nem tartalmaz karakterlánc típusú beépített változót, kissé erőltetett kijelentésnek tűnhet, de igaz. Emiatt tudunk például egyszerű értékkedás segítségével átmásolni egy karakterláncot az egyik változóból a másikba. Ha karakterláncot akarunk másolni, akkor azt – mivel tömbök segítségével kezeljük – lépésről lépésre, betűről betűre tehetjük. Ezt mutatja be a következő példaprogram:

```
alap/valtozok/karakterlancok/masolas.c
```

```

1 #include <stdio.h>
2
3 int masol( char *ide, char *ezt ){
4     int n;
5
6     n=0;
7     do{
8         ide[n]=ezt[n];
9     } while( ezt[n++] != 0 );
10
11    return( n );
12 }/*masol*/
13
14 main(){
15     char tomb[32];
16
17     masol( tomb, "Szöveg..." );
18     printf( "A szöveg: %s\n", tomb );
19 }/*main*/
```

A példaprogramban a 3–12. sorban található `masol()` függvény karakterláncok másolását végzi. A függvény a második paraméterként kapott területről másolja a karaktereket az első paraméterként kapott helyre, mégpedig az első 0 karakterig.

Magát a másolást a 8. sorban található értékkadás végzi, amely a tömb egy elemét helyezi át a másik tömbbe egy értékkadással. A tömb elemeinek típusa char, amely szerepelhet értékkadásokban.

Érdekes a ciklusban található n ciklusváltozó növelése. Ezt az n++ kifejezés végzi a ciklus feltételében. Az n++ kifejezés hatására a változó értéke eggyel nő, de a feltételbe még az eredeti kifejezés helyettesítődik be. (Éppen úgy, mint a bc vagy az awk nyelvében.) Amint látjuk, a 9. sorban található kifejezés megvizsgálja, hogy elértük-e az első 0 karaktert, mellékhatása pedig az, hogy a ciklusváltozó növekszik eggyel nő.

A C nyelvben a megszokott módon használhatjuk a ++ és -- műveleteket, amelyek a változó értékét növelik, illetve csökkentik eggyel. Ha a változó neve előtérben áll, a műveleti jeleket, a kifejezésbe – ahol szerepel – a növelt, illetve csökkentett érték helyettesítődik be, ha a változó neve után, a kifejezésbe a változó eredeti értéke kerül.

Ügynünk kell viszont arra, hogy a C nyelvben csak egész típusú változók kezelésére használhatjuk a ++ és -- műveleteket.

Megfigyelhetjük, hogy a másolást végző függvény a ciklus lefutása után visszaadja a ciklusváltozót, így mintegy „mellékesen” meg is méri a karakterlánc hosszát. A példaprogramunk ezt az értéket nem használja, a függvény hívása során a 17. sorban nem használjuk fel a visszatérési értéket.

A példaprogramban másolás végzésére megvalósított masol() függvényhez igen hasonlóan viselkedik az strcpy() függvény, amelyről bővebben az 74. oldalon olvashatunk.

Habár tudjuk, hogy nem hasonlíthatunk egyszerűen össze karakterláncokat a C nyelv műveleti jeleivel, sokan mégis a következő programrészlethez hasonló módszerrel próbálkoznak:

```

1  char *a;
2
3  a="Hello világ";
4
5  if( a=="Hello világ" )
6  ...

```

Érdemes néhány pillanatig elgondolkodnunk azon, hogy a programot lefordítva, az miért nem úgy működik, ahogyan alkotója szerette volna. (A feltételben nem a karakterláncokat, hanem a két karakterlánc címét hasonlítottuk össze. Ezek akkor is különböznek egymástól, ha az általuk jelölt memóriaterület tartalma azonos.)

Amikor karakterláncokat kezelünk, sokszor előfordul, hogy egyes karaktereket, karakterállandókat helyezünk el a programban. A következő példa azt mutatja be, hogyan készíthetünk char típusú állandókat:

	alap/valtozok/karakterlancok/nagybeture.c
1	#include <stdio.h>
2	#include <string.h>

```

3   char *nagybeture( char *tomb ){
4     int n, N;
5
6     N=strlen( tomb );
7     for( n=0; n<N; ++n )
8       if( tomb[n] >= 'a' && tomb[n] <= 'z' )
9         tomb[n]=tomb[n]+ 'A' - 'a';
10
11    return( tomb );
12  }/*nagybeture*/
13
14 main(){
15   char tomb[32];
16   strcpy( tomb, "Hello világ" );
17   printf( "%s\n", nagybeture(tomb) );
18 }/*main*/
19

```

A programban található `nagybeture()` nevű függvény megkísérli nagybetűsre alakítani a paraméterként kapott mutató által kijelölt karakterláncot, és ezt többszörösen sikeresen el is végzi:

```
Bash$ ./nagybeture
HELLO VILÁG
Bash$
```

Amint látjuk, a programnak gondjai akadnak az ékezetes betűkkel.

A program 4–13. sorában található az a függvény, amely a kisbetűket nagybetűre cseréli a karakterláncban. A függvényben két egész típusú változót hozunk létre az 5. sorban, melyek közül az elsőt a tömb végigjárására használunk, a második pedig a karakterlánc hosszát tartalmazza.

Természetesen elkészíthettük volna a ciklust úgy is, hogy a karakterlánc hosszát megállapító `strlen()` függvényt minden ciklusban meghívja:

```
1  for( n=0; n<strlen(tomb); ++n )
2    ...
```

Ebben az esetben viszont az `strlen()` függvény hívásával feleslegesen terheltük volna a rendszert, hiszen a karakterlánc hossza biztosan nem változik a ciklus végrejártása során.

A betűk nagybetűre történő átalakítását a 9–10. sorban található szerkezet végzi. Ha a feltételes szerkezet úgy találja, hogy a betű az a és z betűk közé esik az ASCII kódtáblában – tehát kisbetű –, hozzáadja az a és A betű kódja közti különbséget. A program tehát kihasználja, az ASCII kódtáblában a betűk sorrendben találhatók

egymás után. (Ez az oka annak, hogy helytelenül kezeli az ékezetes betűket, amelyek szétszórtan helyezkednek el a kódtáblában.)

A programban a 9–10. sorban a betűket nem ASCII kódjukkal, hanem betűként írjuk. A C nyelvben a ' ' jelek között egy karaktert írva egy int típusú állandót kapunk, amely az adott karakter ASCII kódját tartalmazza. Ügyelnünk kell, hogy mindenkor csak egy karaktert írhatunk a ' ' jelek közé, kivéve a két karakterből álló \n, \t stb. rövidítéseket.

A program 9. sorában található a logikai és kapcsolat jele (**&&**). A C nyelvben a szokásos módon használhatjuk a *logikai* és (**&&**), *logikai vagy* (**||**), *bitenkénti* és (**&**), valamint *bitenkénti vagy* (**!**) műveleteket.

Természetesen rendelkezésre állnak azok az eszközök, amelyek lehetővé teszik a magyar ékezetes karakterek nagybetűssé alakítását, de ehhez be kell állítanunk a használni kívánt nyelvi környezetet, ahogyan a következő példa bemutatja:

```
alap/valtozok/karakterlancok/nagybeture-nyelv.c
1 #include <stdio.h>
2 #include <string.h>
3 #include <locale.h>
4
5 main(){
6     char tomb[32];
7     int n, N;
8     setlocale(LC_ALL, "");
9
10    strcpy( tomb, "Hello világ" );
11    N=strlen(tomb);
12
13    for( n=0; n<N; ++n )
14        tomb[n]=(char)toupper( (int)tomb[n] );
15    printf( "%s\n", tomb );
16 }/*main*/
```

A `setlocale()` függvénnyről a 68. oldalon, a `toupper()` függvénnyről pedig a 81. oldalon olvashatunk bővebben.

A karakterláncok tárgyalása során külön kell szólnunk azok kezdeti értékéről. A változók értékét a létrehozásukkor beállíthatjuk, így a karakterláncokat jelölő mutatókat is. Ezt a következő – szándékosan hibás – példaprogram mutatja be:

```
alap/valtozok/karakterlancok/hibas.c
1 #include <stdio.h>
2
3 main(){
4     char *tomb="Hello világ";
5
6     tomb[0]='h' ;
```

```
7   printf("%s\n", tomb);
8 }/*main*/
```

Ha a programot futtatjuk, azonnal látjuk, hogy komoly hibát vétettünk az elkészítésekor:

```
Bash$ ./hibas
Segmentation fault
Bash$
```

A hibát az okozta, hogy egy karakterlánc típusú állandót próbáltunk meg módosítani. A program 4. sorában egy karakterlánc típusú állandót hoztunk létre és a `tomb` nevű mutatóval jelöltük a címét a memóriában. Tudnunk kell azonban, hogy a C fordító a karakterlánc típusú állandókat olyan memóriaterületen helyezi el, amelyeket a programnak nincs joga módosítani, ezért amikor az állandót a 6. sorban megpróbáltuk módosítani, az operációs rendszer megszakította a program futását.

Ha olyan karakterláncot akarunk használni, amelyet módosíthatunk, a következőképpen járunk el:

```
----- alap/valtozok/karakterlancok/allando.c -----
1 #include <stdio.h>
2
3 main(){
4     char tomb[16] = "Hello világ";
5
6     tomb[0] = 'h';
7     printf("%s\n", tomb);
8 }/*main*/
```

Ez a program már tökéletesen működik. Ennek oka az, hogy a 4. sorban itt nem egy mutatót, hanem egy tömböt hozunk létre, amely természetesen módosítható memóriaterületen van.

Ha azt szeretnénk, hogy a gcc által fordított programban a karakterlánc típusú állandókat megváltoztathassuk, használjuk a `-f writable-strings` kapcsolót a fordításkor. Meg kell jegyeznünk, hogy a fordító dokumentációja ezt a lehetőséget csak régebbi programok fordítására ajánlja, nem tartja szerencsés dolognak az állandók módosítását.

## Struktúrák

A struktúrák különféle típusú változók összekapcsolásával készített összetett típusok. A struktúrákat a programot készítő felhasználó hozza létre a feladat megoldásának érdekében.

Struktúrákat a struct kulcsszó segítségével hozhatunk létre, ahogyan azt a következő példaprogram bemutatja:

```
alap/valtozok/strukt/strukt.c
1 struct pont {
2     int x;
3     int y;
4 };
5
6 main(){
7     struct pont a;
8     a.x=0;
9     a.y=0;
10 }/*main*/
```

A program az 1–4. sorokban létrehoz egy új struktúrát, amelynek a neve pont és amely a kétdimenziós síkon képes egy pont adatainak (koordinátáinak) hordozására. A struktúra létrehozása után a programban ezt a struktúrát használhatjuk a pont minden koordinátájának tárolására. A 2–3. sorban látható, hogy a struktúra két azonos típusú változót tartalmaz, mindkettő int típusú, az egyiknek x, a másiknak pedig y a neve.

Látható, hogy a struktúra ugyan különböző típusokból felépített összetett típus, de alkalmas azonos típusok összefogására is. A tömb csak azonos típusú elemeket tartalmazhat, a struktúra azonban akár különböző típusokat is.

A példaprogram 7. sorában látható, hogyan használhatjuk a létrehozott struktúrát konkrét változók készítésére. Az itt létrehozott a nevű változó típusa struct pont, azaz pont struktúra. Fontos, hogy felfigyeljünk arra, hogy a program 1–4. sora nem hoz létre változót – nem foglal memóriaterületet –, csak létrehoz egy új típust, a 7. sor azonban létrehoz egy változót és ezzel memóriát foglal.

A program 8–9. sorában látható, hogyan használhatjuk a struktúra típusú változónkat. Amikor a struktúra elemeire (mezőire) hivatkozunk, le kell írnunk a változó nevét, utána pedig a struktúrában található mező nevét, amelyre hivatkozni szeretnénk. A struktúra létrehozásakor úgy rendelkeztünk, hogy az x nevű elem – az x mező – típusa int, ezért a 7. sorban található a.x típusa int.

Fontos megjegyezni, hogy a struktúrák nem teljes értékű változók, amennyiben nem szerepelhetnek bizonyos kifejezésekben és értékkedásokban. A fordítóprogram nem tudja hogyan kell kezelnie a struktúrákat, nem tudja például hogyan lehet kivonni egymásból két pont koordinátáit. A C nyelvben nincs mód arra, hogy leírjuk az egyes műveletek milyen módon végezhetők el a struktúrákon.

A C nyelv objektumközpontú elemekkel továbbfejlesztett C++ változata alkalmas ennek a kifejezésére. A C++ nyelvnek vannak elemei, amelyekkel műveleteket rendelhetünk a struktúrákhoz (ott osztályok), így lehetőségünk van az általunk létrehozott típusokat „teljes értékű” típusokká tenni.

Az egyetlen művelet, amelyet a struktúrákkal elvégezhetünk, a másolás. Mivel a C fordító ismeri a létrehozott struktúrák méretét, ezért képes másolni őket, hiszen a másoláshoz nem kell ismernie a struktúra belső szerkezetét, elég a méretét tudnia.

A következő példaprogram bemutatja, hogyan használhatjuk a struktúrák másolását a programozás során. A program az előzőekben használt pont struktúra kezelésére létrehoz néhány függvényt.

```
alap/valtozok/strukt/strukthasznalat.c
1 #include <stdio.h>
2
3 struct pont {
4     int x;
5     int y;
6 };
7
8 struct pont kiir( struct pont a ){
9     printf( "x=%d\ty=%d\n", a.x, a.y );
10 }/*kiir*/
11
12 struct pont mozgat( struct pont a,
13                      int x, int y ){
14     a.x=a.x+x;
15     a.y=a.y+y;
16     return(a);
17 }/*mozgat*/
18
19 main(){
20     struct pont a, b;
21     a.x=1;
22     a.y=2;
23     b=a;
24     b=mozgat( b, 5, 5 );
25     kiir(b);
26 }/*main*/
```

A program több helyen is másol struktúrát. A 8. és a 12. sorokban kezdődő függvények struktúrát kapnak paraméterként, ami a C nyelv érték szerinti paramétere-rátadása miatt az érték másolását jelenti. A 16. sorban egy `return()` utasítással struktúrát adunk vissza, amely szintén másolással jár. A 23. sorban egy értékkedásban szerepel a struktúra, ami szintén lehetetlen volna, ha a másolás műveletet nem tudnánk elvégezni struktúrákon.

A struktúrák létrehozásakor ügyelnünk kell arra, hogy ne feledjük el a pontosvesszőt. A következő program teljesen szabályosnak tűnik, pedig végzetes hibát követtünk el a készítésekor:

```

1  struct valtozo{
2      int     ertek;
3      char   nev[16];
4  }
5
6  main(){
7      printf( "Helló világ\n" );
8 }
```

Érdemes néhány másodpercet gondolkodnunk, hogy mi a hiba a programban és mi a hiba következménye. (A struktúra létrehozásának végéről hiányzik a pontosvessző, ezért a fordító úgy értelmezi a programot, hogy a `main()` függvény `valtozo` típusú struktúrát ad vissza.)

## Struktúrát jelölő mutatók

A struktúrákat éppen úgy jelölhetjük mutatókkal a memóriában, mint a beépített, egyszerű változókat. Arra az apróságra kell csak ügyelnünk, hogy struktúramutatók használata esetén nem jelölhetjük a struktúra mezőit a pont segítségével. Pont helyett a kötőjelből és a `>` jelből álló `->` nyíllal kell elválasztanunk a mutatót és a mező nevét. Ezt láthatjuk a következő programban:

	<i>alap/valtozok/strukt/mutato.c</i>
--	--------------------------------------

```

1 #include <stdio.h>
2
3 struct valtozo{
4     char     nev[16];
5     int      ertek;
6 };
7
8 int kiir( struct valtozo *ezt ){
9     printf("%s=%d\n", ezt->nev,
10           ezt->ertek );
11
12     return(1);
13 }/*kiir*/
14
15 main(){
16     struct valtozo  a;
17
18     strcpy( a.nev, "alpha" );
```

```

19     a.ertek=3;
20     kiir( &a );
21 }/*main*/

```

A program egy struktúrát használ változók – névvel és értékkel rendelkező adatszerkezetek – tárolására. A program 3–6. sorában hozzuk létre a valtozo nevű struktúrát, a 8–13. sorban található kiir() nevű függvény pedig az ilyen típusú struktúrák mezőit írja ki. A függvény egy mutatót kap paraméterként, amely a struktúrát jelöli a memoriában. A 9–10. sorban láthatjuk, hogy a mezőkre a -> kifejezéssel hivatkozunk. A main() függvényben a 20. sorban hívjuk a függvényt, ahol a struktúrát jelölő mutatót a szokásos módon – az & jellel – állítjuk elő.

## Új típusok létrehozása

A C programozási nyelv `typedef` kulcsszavával új típusokat hozhatunk létre. A következő példa bemutatja, hogyan használhatjuk ezt a szerkezetet:

```

alap/valtozok/strukt/typedef1.c
1 #include <stdio.h>
2
3 typedef int egesz;
4
5 main()
6 {
7     egesz a=42;
8
9     printf( "a=%d\n", a );
10 }/*main*/

```

A program 3. sorában egy új változótípust hozunk létre, amelynek neve `egesz` és amely közvetlenül az `int` típusból származik. A program 6. sorában láthatjuk, hogy ezt az új típust pontosan olyan formában használhatjuk, ahogyan a beépített változótípusokat. A 6. sorban található változólétrehozásnál pontosan úgy járunk el, mint akkor, amikor a beépített típusokat használjuk.

Valójában a beépített változók közvetlen felhasználásával új típusokat létrehozni nem túl hasznos dolog, hiszen csak azt érhetjük el ily módon, hogy egy újabb névvel hivatkozhatunk a típusra. A `typedef` igazi hasznát akkor tapasztalhatjuk meg, ha összetett típust – például struktúrát – veszünk alapul a típus létrehozásánál. Ezt láthatjuk a következő példaprogramban:

```

alap/valtozok/strukt/typedef2.c
1 #include <stdio.h>
2

```

```

3   struct Pont {
4     int x;
5     int y;
6     int z;
7   };
8
9   typedef struct Pont pont;
10
11  int kiir( pont *p ){
12    printf("x=%d\ny=%d\nz=%d\n",
13      p->x, p->y, p->z );
14 }/*kiir*/
15
16 main(){
17   pont x;
18
19   x.x=1; x.y=42; x.z=84;
20   kiir( &x );
21 }/*main*/

```

A program egy új típust hoz létre a háromdimenziós térben található pontok kezelésére. A 3-7. sorban található a Pont struktúra létrehozása. Ennek a struktúrának a felhasználásával a 9. sorban egy új változót hozunk létre a `typedef` segítségével. A program további soraiban (11., 17.) láthatjuk, hogy a létrehozott új típus a beépített változóknál megszokott módon használható változók létrehozásakor. Más sorokban (13., 19.) azt is megfigyelhetjük, hogy az ilyen típussal rendelkező változók a struktúráknál megszokott módon használhatók.

Szokás a struktúra létrehozását és a `typedef` segítségével felírt változótípus létrehozását egyetlen közös szerkezetbe írni, ami tömörebb formát ad. Ezt mutatja be az előző példaprogram átírt részlete:

```

1  typedef struct Pont {
2    int x;
3    int y;
4    int z;
5  } pont;

```

Figyeljük meg, hogy a szerkezetben megtalálható a `Pont` nevű struktúra és a `pont` nevű új típus létrehozása is. Valójában még az sem szükséges, hogy a struktúra és a típus neve különbözzön egymástól. A részletet átírhatjuk úgy, hogy a két név megegyezzen:

	<i>alap/valtozok/strukt/typedef2.c</i>
--	--

```

1  typedef struct pont {
2    int x;

```

```
3     int y;  
4     int z;  
5 } pont;
```

Ez a forma, amit a gyakorlatban a legtöbbször használunk, ugyanazt a nevet használja a struktúra és a típus létrehozására.

## Összetett típusokból készült típusok

A változók, változótípusok építményének az összetett típusok nem a legösszetettebb elemei. Összetett típusokból is készíthetünk összetett típusokat, azokból újra, és a sort tetszőleges ideig folytatthatjuk. E részben néhány érdekes adatszerkezetet mutatunk be, amelyek a gyakorlatban is hasznosnak bizonyulhatnak.

### Mutatót jelölő mutató

Az eddigieknel összetettebb típust kapunk, ha olyan mutatókat hozunk létre, amelyek mutatókat jelölnek a memóriában. Kissé humorosan azt is mondhatnánk, hogy ilyenkor mutatóra mutatót kapunk.

A következő példaprogram egy karakterláncot jelölő mutató memóriabeli címét adja át egy függvénynek. Mivel a karakterláncot jelölő mutató típusa `char *`, az ilyen változót jelölő mutató típusa `char **` kell, hogy legyen. Ez így is van, ahogyan a példaprogramban láthatjuk:

```
alap/valtozok/mutatok/kozvetett.c  
1 #include <stdio.h>  
2  
3 int szovegre( int szam, char **vissza ){  
4     switch( szam ){  
5         case 1:  
6             *vissza="Egy";  
7             break;  
8         case 2:  
9             *vissza="Kettő";  
10            break;  
11        default:  
12            *vissza="Ismeretlen";  
13            return(1);  
14    }/*switch*/  
15    return(0);
```

```

16 }/*szovegre*/
17
18 main(){
19     char *s=NULL;
20     if( szovegre(1, &s)==0 )
21         printf( "%s\n", s );
22 }/*main*/

```

A programban a 3–16. sorban található függvény egy mutatót jelölő mutatót kap paraméterként. Azért használunk ilyen paramétert, hogy a hívó függvény által előállított mutatót a hívott függvényben meg tudjuk változtatni. A függvény 6., 9. és 12. sorában a mutató követésével keressük meg a memóriában a jelölt változót – amely mutató – és változtatjuk meg. A 18–22. sorban található `main()` függvényben a 20. sorban állítjuk elő a mutatót jelölő mutatót és itt adjuk át a hívott függvénynek.

A C nyelv nem csak mutatókat jelölő mutatók létrehozását teszi lehetővé, hanem tovább is folytathatjuk a sort. Így eljuthatunk például a `char ***` vagy `char ****` típusokhoz, bár ezeknek gyakorlati hasznát csak ritkán vesszük.

## Mutatótömbök

Meglehetősen egyszerű, de az eddigi típusoknál összetettebb típust kapunk, ha mutatókat tartalmazó tömböt készítünk. Ezt mutatja be a következő példa:

```

1 #include <stdlib.h>
2
3 int ir( int ennyit, char **ezt ){
4     int n;
5
6     for( n=0; n<ennyit; ++n )
7         printf( "%s\n", ezt[n] );
8 }/*ir*/
9
10 main(){
11     char *tomb[3];
12     tomb[0]="Nulla";
13     tomb[1]="Egy";
14     tomb[2]="Kettő";
15
16     ir( 3, tomb );
17 }/*main*/

```

A program 11. sorában hozunk létre egy mutatókat tartalmazó tömböt, amelyet a 12–14. sorokban töltünk fel értékekkel. A program 3–8. sorában található egy függvény, amely a mutatókat tartalmazó tömböt kezeli.

Ha mutatókat helyezünk el egy tömbben, akkor erős késztetést érezhetünk arra, hogy a tömb teljes területét 0 értékű bájtokkal kitöltve egy lépében állítsuk az összes mutatót NULL értékre. Ez a gyakorlatban általában működik is, mégsem javasolható módszer. Léteznek ugyanis olyan számítógépek – olyan processzorok – ahol a NULL értékű mutatók nem csupa 0 értékű bájtból állnak.

A mutatókat tartalmazó tömbök tárgyalásakor mindenkorban meg kell említenünk a main() függvényt, amely paramétereinek bemutatását eddig óvatosan kerültük. A main() függvény ugyanis egy egész számot és egy mutatókat tartalmazó tömböt kap a Linuxtól. Ezt mutatja be a következő példaprogram:

```
1 #include <stdio.h>
2
3 int main( int argc, char *argv[] ){
4     int n;
5     for( n=0; n<argc; ++n )
6         printf( "%s ", argv[n] );
7     printf("\n");
8     return(0);
9 }/*main*/
```

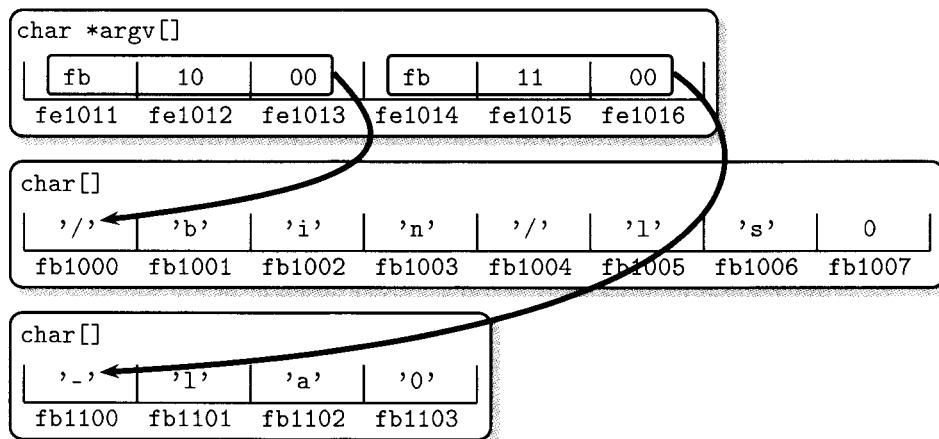
A program 3. sorában látható a main() függvény helyes paraméterezése. A függvény visszatérési értéke int, amely a program visszatérési értékét adja. Mint tudjuk, minden Unixon futtatott program egy egész számot adhat vissza, amely szokás szerint 0 értékkel a hibamentes futást, attól eltérő számokkal pedig a különféle hibákat jelzi.

A main() függvény paraméterei a parancssorban a programnak átadott paramétereket tartalmazzák. Az első paraméter (argc) a programnak adott paraméterek számát adja meg, a második (argv) pedig egy mutató, amely a paramétereket jelölő mutatókat tartalmazó tömb 0 indexű elemét jelölő mutató. A program 5–6. sorában láthatjuk, hogyan járható végig az argv tömb, hogyan férhetünk hozzá a program paramétereire, amelyek karakterláncként vannak elhelyezve a memoriában. A main() függvény argv paraméterének szerkezetét mutatja be a 1.3. ábra.

A main() függvény paramétereit argc és argv néven használjuk a hagyomány szerint. Át lehetne ugyan nevezni a paramétereket, de gyakorlatilag senki sem teszi.

A példaprogram a programnak átadott paramétereket a szabványos kimenetre írja – hasonlóan a BASH echo nevű beépített parancsához. A program futtatásával a következő eredményt kaphatjuk:

## 1.3. ÁBRA: Az argv



```
Bash$ ./echo ek ak
./echo ek ak
Bash$
```

Amint látjuk, a program a saját nevét – a programot tartalmazó állomány nevét – is kiírta. Unix rendszereken – így Linuxon is – a legelső paraméter mindenkor indítására használt parancs.

## Struktúratömbök

Igen praktikus adatszerkezethez juthatunk, ha struktúrákból készítünk tömböt. A struktúratömb leginkább egy adatbázishoz hasonlít, ahol sorok – a tömb elemei – és oszlopok – a tömb mezői – találhatók. A struktúratömb használatát mutatja be a következő példaprogram:

	alap/valtozok/strukt_tomb.c
--	-----------------------------

```

1 #include <stdio.h>
2 #include <string.h>
3
4 typedef struct barat {
5     char nev[32];
6     int   baratja;
7 } barat;
8

```

```
9  barat    baratok[10];
10
11 int kolcsonos( int szemely ){
12     if( baratok[baratok[szemely].baratja].baratja==szemely )
13         return(1);
14     else
15         return(0);
16 }/*kolcsonos*/
17
18 int kiir( int szemely ){
19     if( kolcsonos(szemely)==1 )
20         printf( "%s barátsága kölcsönös\n",
21                 baratok[szemely].nev );
22     else
23         printf( "%s barátsága nem kölcsönös\n",
24                 baratok[szemely].nev );
25     return( kolcsonos(szemely) );
26 }/*kiir*/
27
28 int kiir_osszes( int tomeg ){
29     int n;
30     for(n=0; n<tomeg; ++n )
31         printf( "[%d]\t%s\t%d\n", n, baratok[n].nev,
32                 baratok[n].baratja );
33 }/*kiir_osszes*/
34
35 main(){
36     strcpy( baratok[0].nev, "Gaz Gizi" );
37     baratok[0].baratja=0;
38
39     strcpy( baratok[1].nev, "Gipsz Jakab" );
40     baratok[1].baratja=2;
41
42     strcpy( baratok[2].nev, "Pap Tamás" );
43     baratok[2].baratja=1;
44
45     strcpy( baratok[3].nev, "Kis Valéria" );
46     baratok[3].baratja=0;
47
48     kiir_osszes( 4 );
49     kiir( 1 );
50     kiir( 3 );
```

```
51 }/*main*/
```

A példaprogram egy személyeket tartalmazó kisméretű adatbázist használ, annak sorait írja a szabványos kimenetre és abban keresést végez. A program 4–7. sorában létrehozunk egy struktúratípust, amely egy karakterlánc – a név – és egy szám – a barát kódja – típusú mezőt tartalmaz. A program 9. sorában egy tömböt hozunk létre, amely ilyen struktúrákból képes 10 példányt hordozni.

A program egyik lényeges eleme a 11–16. sorban található függvény, amely vizsgálja, hogy az adott személy barátsága kölcsönös-e. A függvény paraméterként a vizsgálni kívánt személy tömbbeli sorszámát kapja meg. A 12. sorban található viszonylag összetett kifejezés vizsgálja meg, hogy a barátság kölcsönös-e, mégpedig a következő gondolatmenet alapján:

1. A paraméterként átadott sorszám (*szemely*) alapján kikereshetjük az általános érvényességű tömbből a megfelelő elemet, *baratok[szemely]* alakban.
2. A kikeresett struktúra *baratja* mezője az adott személy barátjának kódját adja, amelyet *baratok[szemely].baratja* alakban írhatunk.
3. A tömbből ennek a barátnak az adatai szintén elérhetők, mégpedig *baratok[baratok[szemely].baratja]* formában.
4. Ennek a barátnak is van *barátja*, amely a *baratja* mező jelölésével érhető el, azaz *baratok[baratok[szemely].baratja].baratja* formában hivatkozhatunk rá.
5. A függvényben minket az érdekel, hogy a vizsgált személy adatai közt feltüntetett barát a vizsgált személy-e, hiszen arra vagyunk kíváncsiak, hogy a barátság kölcsönös-e. Erre a kérdésre ad választ a *baratok[baratok[szemely].baratja].baratja ==szemely* kifejezés.

A programot lefordítva és futtatva táblázatos formában kapjuk az adatokat a szabványos kimeneten, valamint néhány személy esetén arra is választ kapunk, hogy kölcsönös barátságban van-e a barátjával:

```
1 \pr \ui{./kolcsonos}
2 [0]      Gaz Gizi      0
3 [1]      Gipsz Jakab   2
4 [2]      Pap Tamás    1
5 [3]      Kis Valéria   0
6 Gipsz Jakab barátsága kölcsönös
7 Kis Valéria barátsága nem kölcsönös
8 \pr
```

Érdemes végiggondolnunk, hogy programunk döntése alapján kölcsönös barátság jellemzi-e a táblázat első sorában található személyt, aki kissé önző módon önmagát jelölte meg legjobb barátjaként.

## Mutatók struktúrákban

A struktúrákban elhelyezhetünk mutatókat is, sőt igen gyakran az adott struktúrát kijelölni képes mutatót helyezünk el struktúrában, ezzel önmagukra hivatkozó adatszerkezeteket hozunk létre. Az önhivatkozás segítségével összetett rendszerek – listák, fák stb. – készíthetők, amelyek igen hatékonyak és érdekesek.

Az önmaguk kijelölésére képes struktúrák létrehozása kapcsán egy fontos apróságot meg kell említenünk: a típus létrehozása közben még nem használhatjuk a típust, csak a struktúrát. Ezt mutatja be a következő programrészlet:

```

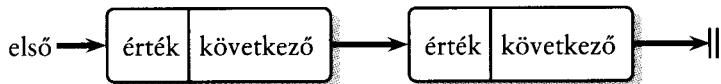
1  typedef struct elem{
2      int          ertek;
3      struct elem *kovetkezo;
4  } elem;
```

A programrészlet 3. sorában egy olyan mutatót készítünk, amely magát a struktúrát képes jelölni a memoriában. Itt azonban még nem használhatjuk az `elem` nevű típust, csak az `elem` nevű struktúrát. A mutató létrehozásakor tehát nem hagyhatjuk el a `struct` kulcsszót, ahogyan a példa is mutatja.

Az önhivatkozó struktúrák használatát mutatja be a következő példaprogram, amely több újdonságot is tartalmaz:

<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt;  3 4 typedef struct elem{ 5     int          ertek; 6     struct elem *kovetkezo; 7 } elem; 8 9 elem *elso=NULL; 10 11 void push( int ertek ){ 12     elem        *uj; 13 14     uj=malloc( sizeof(elem) ); 15     if( uj==NULL ){ 16         printf( "Nincs elegendő memória.\n" ); 17         exit( 1 ); 18    }/*if*/ 19 20     uj-&gt;ertek=ertek; 21     uj-&gt;kovetkezo=elso; 22     elso=uj; </pre>	<i>alap/verem.c</i>
--	---------------------

## 1.4. ÁBRA: Mutatók struktúrában: láncolt lista



```

23 }/*push*/
24
25 int pop( ){
26     int vissza;
27     elem *regi;
28
29     if( elso==NULL ){
30         printf( "Kiürült a verem.\n" );
31         exit( 1 );
32     }/*if*/
33     regi=elso;
34     vissza=elso->ertek;
35     elso=elso->kovetkezo;
36     free(regi);
37     return(vissza);
38 }/*pop*/
39
40 main(){
41     push(10);
42     push(20);
43     push(30);
44     printf("%d\n", pop() );
45     printf("%d\n", pop() );
46     printf("%d\n", pop() );
47 }/*main*/
  
```

A példaprogram egy verem kezelését mutatja be. A verem elemeinek tárolására a 4–7. sorban található új típust vezetjük be, ami struktúraként egy egész szám – a verem által hordozott érték – és egy mutató hordozására képes. A mutató segítségével a verem elemeinek összefűzését végezzük el, azaz a verem egy mutatók segítségével összefűzött láncként van megvalósítva. Figyeljük meg az önmagát jelölni képes adatszerkezet létrehozását a 4–7. sorban!

A verem legfelső elemeinek jelölésére egy általános érvényességi körű változót hozunk létre a 9. sorban. Ez a változó jelöli a memóriában a verem legfelső elemét, ahonnan a többi elem a lánc végigjárásával volna elérhető. Ez a program nem tartal-

maz olyan függvényt, amely a lánc elemeit végigjárja, hiszen a verem kezelése során mindenig csak a legfelső elemet kell elérnünk, de könnyedén készíthetünk ilyet.

Új elemek elhelyezését végzi a vermen a 11–23. sorban található `push()` függvény. A függvény létrehozásakor a `void` visszatérési értéket láthatjuk. A `void` 0 méretű típus, amelyet akkor használunk, ha a függvénynek nem kell értéket visszatámadnia. Amint a példaprogramban láthatjuk, ilyen függvények esetében elhagyhatjuk a `return()` utasítást. A függvény végén a véghajtása anélkül fejeződik be, hogy értéket adnának vissza. Ha mégis szeretnénk használni a `return()` utasítást, akkor azt a zárójelek kiírása nélkül, `return;` formában írhatjuk.

A `push()` függvény a 14. sorban hívja a `malloc()` függvényt, amelynek használatához a `stdlib.h` fejállomány betöltésére van szükség. A `malloc()` memória foglalásra használható, a hívásával a program memóriaterületet kérhet az operációs rendszertől. A `malloc()` hívásakor a Linux lefoglal egy memóriaterületet, engedélyezi annak használatát a programunk számára és visszaadja a memóriaterület első bájtjának címét.

A `malloc()` függvény visszatérési értékének típusa `void *`. Ez nem nulla méretű, mint a `void`, hanem az adott számítógépen használható memóriacím tárolására alkalmas mérettel rendelkezik. A `void *` olyan mutató, amely bármilyen változó címét képes hordozni, tulajdonképpen „ismeretlen típusú elemet jelölő mutató”. A `malloc()` azért `void *` típusú mutatót ad vissza, hogy bármilyen mutatóban tárolni lehessen a visszatérési értéket. A példaprogramunk ezt a 14. sorban használja ki, ahol a visszaadott memóriaterület címét az `uj` nevű változóban helyezi el, amely `elem` típus jelölésére használható. Köszönhetően a `void *` típusnak, ez nem eredményez figyelmeztetést a fordítás során, a mutató pedig a program további részeiben használható struktúrára mutató típusként.

A `malloc()` egyetlen paramétere a lefoglalni kívánt memória méretét adja meg. A példaprogram 14. sorában a híváskor a `sizeof()` kulcsszóval megállapítjuk az `elem` nevű változótípus méretét és ezt adjuk át a memória foglaláshoz. A rendszertől tehát éppen annyi memóriát kérünk, amennyi egy `elem` típusú változó tárolására alkalmas.

A `malloc()` a lefoglalt memóriaterület első bájtjának címét adja vissza – ha a művelet sikeres volt. Ha nem sikerült memóriát foglalni – mert például nincs elegendő szabad memória –, a visszaadott érték `NULL`. Ezt a program a 15–18. sorban ellenőrzi. Ha nem sikerült memóriát foglalni, a program azonnal kilép az `exit()` segítségével. Sok hosszadalmas hibakereséstől óvhatjuk meg magunkat, ha a `malloc()` visszatérési értékét mindenig ellenőrizzük.

A program a 17. és a 31. sorban tartalmazza az `exit()` utasítást. Ezt a parancsot bármely függvényben elhelyezhetjük, a véghajtásakor a program futása befejeződik. Az `exit()` után zárójelben megadhatjuk a program kilépési kódját, amely Unix rendszereken hagyományosan 0 hibamentes futás esetén, illetve attól különböző, ha hiba történt.

A veremből a legfelső elemet eltávolítani a 25–38. sorokban található `pop()` függvénytelével lehet. A függvény eltávolítja a legfelső elemet és visszaadja annak értékét.

A program a 36. sorban a `free()` függvényt használja a `malloc()` segítségével előzőleg lefoglalt memória felszabadítására. A `free()` függvény egyetlen paramétere a felszabadítandó memóriaterületet jelölő mutató, amelyet előzőleg a `malloc()` adott vissza. Ügyelnünk kell a `free()` hívásánál, hogy csak azokat a területeket szabadít-suk fel, amelyeket előzőleg lefoglaltunk.

Tudnunk kell, hogy a program kilépésekor az általa lefoglalt, de fel nem szabadtított memóriaterületeket a Linux felszabadítja, de a program futása során nekünk illik a már nem használt memóriaterületeket felszabadítani, hogy programunk ne terhelje feleslegesen a rendszert.

## Ajánlott irodalom

1. BRIAN W. KERNIGHAN, DENNIS M. RICHIE: *A C programozási nyelv*, Műszaki Könyvkiadó, 1985.
2. PETHŐ ÁDÁM: *abC, C programozási nyelvkönyv*, Számalk Könyvkiadó, 1991.
3. BENKŐ TIBORNÉ, BENKŐ LÁSZLÓ, TÓTH BERTALAN: *Programozzunk C nyelven!*, Computerbooks, 2002.
4. PERE LÁSZLÓ: *Linux: felhasználói ismeretek II.*, Kiskapu, 2002.
5. BJARNE STROUSTRUP *A C++ programozsi nyelv*, Kiskapu, 2002.
6. STEPHEN C. DEWHURST: *C++ hibaelhárító*, Kiskapu, 2003.
7. KYLE LOUDON: *Mastering Algorithms with C*, O'Reilly, 1999.
8. STEVE OUALLINE: *Practical C Programming*, O'Reilly, 1991.
9. DANIEL BARLOW: *The Linux GCC Howto*, Internet.
10. A GNU C fordító honlapja: <http://gcc.gnu.org>

## 2. fejezet

# A GNU C KÖNYVTÁR

*Tanulmányozzad az ő könyvtárait és igyekezz, hogy ne találj ok nélkül semmit fel újra: így leszen a kódod rövid és olvasható, a napjaid pedig örömteliek és gyümölcsözőek.*

HENRY SPENCER: *A C programozók tízparancsolata* (7. parancsolat).

Amint azt már láthattuk, a C programozási nyelv nem tartalmaz eszközt egy sor általánosan jelentkező feladatra. A C nyelvben nincsenek eszközök a szabványos be- és kimenet kezelésére, nincsenek eszközök a karakterláncok kezelésére és így tovább.

Szerencsére az elmúlt évtizedekben a legtöbb általánosan előforduló elemi feladatra elkészültek a C nyelven megírt függvények, amelyek „C programkönyvtár” néven a legtöbb rendszeren elérhetők. Igen fontos tudnunk, hogy ezeknek a munkánk során használható függvényeknek – a C könyvtárnak – a felépítését is leírják az érvényben lévő szabványok, ezért ezeket az eszközöket egységesen kezeli a legtöbb rendszer. Tulajdonképpen mondhatjuk azt is, hogy a C könyvtár függvényei a C programozási nyelv részei, használatuk a C nyelven való programozás részét képezi. A C könyvtár használatát legalább részben mindenkinél kell sajtítania, aki C nyelven kíván programozni és pontosan ebben nyújt segítséget ez a fejezet.

Linux rendszereken a GNU C könyvtár található meg, amely tartalmazza a szabványok által előírt elemeket, így a megismerésével „hordozható” tudásra tehetünk szert. Az a felhasználó, aki megismerekedik a GNU C könyvtár elemeivel, néhány apróságtól eltekintve minden Unix rendszeren képes programozási feladatokat ellátni. Különbségek ugyanis a szabványosítás ellenére vannak. A GNU C könyvtár létrehoz néhány olyan eszközt, amely a szabványokban nem szerepel és amelyeket hiába keresünk más megvalósításokban. A fejezetben kimondottan a GNU C könyvtárat vesszük alapul, célunk az volt, hogy elsősorban Linuxon használható tudást adjunk át.

A C könyvtárról szóló fejezet elején fontos megjegyeznünk, hogy a programkönyvtárnak csak a legfontosabb elemeit találjuk meg e könyvben, az érthetőség és a terjedelmi korlátok miatt jónéhány eszköz kimaradt. Remélhetőleg sikerült a C könyvtár eszköztárának egy olyan részét kiválasztani, amellyel a gyakorlatban előforduló feladatok legtöbbje megoldható.

## Hibák és hibaüzenetek

Mielőtt az összetettebb függvények használatát bemutatnánk, néhány szó erejéig meg kell említenünk a hibák kezelését. Fontos ugyanis, hogy programozáskor és az elkészült program használatakor a lehető leg pontosabban tudjuk, mi okozhatott hibát. A hibaüzenetek segítségével a felhasználó és a programozó igen sok időt és fáradságot takarít meg.

A hibák okának kiderítésére és a hibakezelésre a következő eszközöket használhatjuk:

```
int errno
```

A GNU C programkönyvtár az `errno` általános érvényességű változót használja a hibák okának jelzésére. Ha a programkönyvtár valamely függvénye nem képes elvégezni a feladatát, a hiba tényét általában a visszatérési értékével jelzi, a hiba feltételezhető okát pedig ebben a változóban tárolja.

Tudnunk kell, hogy az `errno` a legutóbb végrehajtott függvény által beállított értéket hordozza, ha tehát a hibát jelző függvény után egy másik függvényt hajtunk végre azok közül, amelyek használják az `errno` változót, a hibajelző érték elvész számunkra.

Az `errno` általános érvényű változó használatához az `errno.h` fejállomány betöltése szükséges.

```
char *strerror(int hibakód);
```

A függvény segítségével a paraméterként átadott hibakód szöveges leírását állíthatjuk elő.

A függvény paramétere a hibakód, amely általában az `errno` általános érvényességű változóból származik.

A függvény visszatérési értéke a hiba szöveges leírását tartalmazó karakterláncot jelöli a memoriában. Ezt a memóriaterületet a folyamat nem módosíthatja. A hiba szöveges változata a nyelvi beállításoknak megfelelő nyelven jelenik meg.

Fontos megemlítenünk, hogy az `strerror()` számára nem ismeretes az, hogy milyen függvény adta a hibakódot, így meglehetősen általános lesz a szöveges

hibaüzenet. Ha tudjuk, melyik függvény adta a hibakódot, sokkal precízebben is megfogalmazhatjuk a hibaüzenetet. Ez az oka annak, hogy e könyvben minden függvény esetében külön megadjuk a hibakódok pontos jelentését.

Az `strerror()` függvény használatához a `string.h` fejállomány betöltése szükséges.

`EXIT_SUCCESS` Ennek az állandónak az értékét használhatja fel programunk a kilépéskor a hibamentes futás jelzésére. Az állandó használatához a `stdlib.h` fejállomány betöltése szükséges.

`EXIT_FAILURE` Ennek az állandónak az értékét használhatja a programunk kilépéskor a hiba jelzésére. Az állandó használatához a `stdlib.h` fejállomány betöltése szükséges.

**1. példa.** A következő program egy függvény hibás futása után kiírja a megfelelő hibaüzenetet, majd kilép.

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <errno.h>
5
6  int main( int argc, char *argv[] ){
FILE *allomany;
7
8
9  allomany=fopen("/nincs/ilyen/fajl", "r" );
10 if( allomany==NULL ){
11     fprintf( stderr, "open(): %s\n", strerror(errno) );
12     exit( EXIT_FAILURE );
13 }/*if*/
14
15 /*
16  * További programrészek helye.
17  */
18 exit( EXIT_SUCCESS );
19 }/*main*/

```

A program a 9. sorban hívja meg az `fopen()` függvényt. Számunkra most mellékes, hogy ez a függvény mire szolgál, a lényeges csak az, hogy a számára átadott állománynéven nem létezik állomány, ezért hibát jelez.

Az `fopen()` függvény – mint a C könyvtár sok más függvénye – a -1 visszatérési értékkel jelzi, hogy a feladatot nem tudta elvégezni. A visszatérési értéket vizsgáljuk meg a 10. sorban, és ennek megfelelően írjuk ki a hibaüzenetet a 11. sorban az `strerror()` függvény segítségével szöveges formában.

*A program futása során a következő kimenetet adja:*

```
Bash$ ./strerror  
open(): No such file or directory  
Bash$
```

## Nyelvi beállítások

Magyar anyanyelvű felhasználó számára különösen fontos lehet, hogy az általa használt programok több nyelvet is támogassanak. Az itt tárgyal függvények a többnyelvű programok készítését támogatják.

Tudnunk kell, hogy többnyelvű programokon nem csak olyan programokat értünk, amelyek az üzeneteket több nyelven is képesek kiírni. Elsősorban ugyanis arra van szükségünk, hogy programjaink a nyelvnek megfelelően viselkedjenek. Ha például egy magyar anyanyelvű felhasználó ki szeretné cserélni az összes betűt egy szövegrészletben, természetesen ki akarja cserélni az „éáőú” betűket is, ami nem mondható el mondjuk egy angol anyanyelvű felhasználóról akinek ezek nem betűk, hanem különleges jelek.

A többnyelvű programok készítésének alapgondolása az, hogy az egyes nyelvek és nyelvjárások jellemző tulajdonságait az operációs rendszer tárolja, így azokat minden alkalmazás eléri. A C könyvtár bizonyos függvényei is úgy készültek, hogy képesek figyelembe venni a nyelvre jellemző beállításokat.

Amikor többnyelvű programot készítünk, a GNU C könyvtár kétféle támogatást ad számunkra. Egyrészt lehetővé teszi, hogy beállítsuk a használandó nyelvet a könyvtár függvényei – és más használt programkönyvtárak – számára, másrészt elérhetővé teszi a nyelvre jellemző tulajdonságokat, hogy a mi általunk készített függvények is a megfelelő módon működhessenek.

Többnyelvű programok készítését a C könyvtár az itt bemutatott függvényekkel támogatja. A függvények használatához szükséges lehet a `locale.h` és a `langinfo.h` fejállományok betöltése.

```
char *setlocale(int mit, const char *nyelv);
```

A függvény segítségével beállíthatjuk a program által használt nyelvet, így módosítva néhány C könyvtábeli függvény viselkedését és elérhetővé tehetjük a beállított nyelv jellemzőit programunk számára.

A függvény első paramétere egy állandó, amely meghatározza, hogy a nyelv jellemzőinek mely csoportját kívánjuk átvenni. Itt a következő állandók állhatnak:

`LC_ALL` minden jellemző beállítása.

**LC\_COLLATE** A szabályos kifejezésekben található karakterosztályok (például [a-z]) és beépített karakterosztályok (például [:alnum:]) nyelvtől függő beállításainak átvétele.

**LC\_CTYPE** A szabályos kifejezésekben található egyéb, nyelvtől függő beállítások átvétele.

**LC\_MESSAGES** A program üzeneteinek átállítása más nyelvre.

**LC\_MONETARY** A pénzösszegek formájának a nyelvre vonatkozó beállításai.

**LC\_NUMERIC** A számok formájának az adott nyelvre jellemző beállítása.

**LC\_TIME** A dátum és idő formájának átvétele.

A függvény második paramétere egy szöveges értéket jelöl, amely a nyelvnek a neve, amelynek jellemzőit át szeretnénk venni. Ennek a névnek meg kell egyeznie a rendszer által támogatott valamely nyelv szabványos nevével. A támogatott nyelvek listáját a `locale` program -a kapcsolója segítségével kérhetjük le.

Ha a függvény második paramétere 0 hosszúságú szöveges érték (""), a függvény az LC\_ALL, majd a LANG környezeti változóból próbálja meg beállítani a használandó nyelvet, így adva módot a felhasználónak arra, hogy beállítsa a nyelvet.

Ha a függvény második paraméterének értéke NULL, a függvény nem módosítja a nyelvi beállításokat, csak visszaadja az aktuális nyelvi beállítás nevét.

A függvény visszatérési értéke a nyelvi beállítás nevét jelölő mutató, ha a művelet sikeres volt, és NULL, ha nem.

```
char *nl_langinfo(nl_item melyik);
```

A függvény segítségével a felhasználó nyelvi beállításai által előírt tulajdonságokat, adatokat kérhetjük le.

A függvény paramétere egy állandó, amely azt határozza meg, hogy a nyelvi beállítások melyik elemét szeretnénk lekérdezni. Ez a következők egyike lehet:

**CODESET** A használt kódlap lekérdezése. A kódlap határozza meg, hogy az egyes kódokhoz milyen alakú betűk, karakterek tartozzanak.

**ABDAY\_1–ABDAY\_7** A hétfő napjainak – legfeljebb hárombetűs – rövidítése. Az első állandó – ABDAY\_1 – a vasárnap, az utolsó – ABDAY\_7 – pedig a szombat rövidítése az adott nyelven.

**DAY\_1–DAY\_7** A hétfő napjainak neve. Az első állandó – DAY\_1 – a vasárnap, az utolsó – DAY\_7 – pedig a szombat neve az adott nyelven.

**ABMON\_1–ABMON\_12** A hónapok neveinek rövidítése az adott nyelven.

**MON\_1–MON\_12** A hónapok nevei az adott nyelven.

**D\_T\_FMT** Az idő és a dátum kiíratására használható formátum az adott nyelven. A visszaadott szöveges jellegű érték használható az `strftime()` függvény segítségével a dátum és idő olvasható formában történő kiíratásához.

**T\_FMT** Az idő kiíratására használható formátum az adott nyelven. A visszaadott szöveges érték használható az `strftime()` függvény felhasználásával az idő kiíratására.

**D\_FMT** A dátum kiíratására használható formátum az adott nyelven. A visszaadott szöveges érték használható az `strftime()` függvény felhasználásával a dátum kiíratására.

**INT\_CURR\_SYMBOL** A fizetőeszköz nevének nemzetközi rövidítése (például HUF).

**CURRENCY\_SYMBOL** A fizetőeszköz nevének rövidítése, jele (például Ft).

**MON\_DECIMAL\_POINT** A pénzösszeg írásakor használt tizedesjel, az egészeket a tizedes helyiértékektől elválasztó írásjel.

**MON\_THOUSANDS\_SEP** A pénzösszeg írásakor – az olvashatóság érdekében használt – tagolójel.

**POSITIVE\_SIGN** A pozitív előjel jelzésére használható szöveg.

**NEGATIVE\_SIGN** A negatív előjel jelzésére használható szöveg.

**FRAC\_DIGITS** Megmutatja, hogy a pénzösszegeket hány tizedes jelig kell kezelnünk.

**P\_CS\_PRECEDES** Értéke 1, ha a fizetőeszköz nevét a szám elé kell írni, 0, ha utána.

**P\_SEP\_BY\_SPACE** A szöveges érték első bájtjának értéke 1, ha a fizetőeszköz nevének rövidítése és a szám közé szóközt kell tennünk, és 0, ha nem.

**DECIMAL\_POINT** A számokban használt tizedesjel.

Az itt található állandók nemelyikéhez az előfeldolgozó számára létre kell hozunk a `_GNU_SOURCE` állandót még a `langinfo.h` fejállomány betöltése előtt, ahogyan a következő példába láthatjuk:

```
1 #define _GNU_SOURCE
2 #include <langinfo.h>
```

A függvény visszatérési értéke egy mutató, amely a nyelvi beállításoknak megfelelő visszaadott szöveges értéket jelöli a memóriában. A mutatóval jelzett területet nem módosíthatjuk.

**2. példa.** Mivel a GNU C könyvtár üzenetei magyar nyelven is elérhetők, a nyelvi beállítások segítségével elérhetjük, hogy a könyvtártól származó hibaüzenetek magyar nyelven jelenjenek meg. A következő példaprogram ezt mutatja be, az `strerror()` függvény használatával.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <locale.h>
4 #include <string.h>
5 #include <errno.h>

6
7 int main( int argc, char *argv[] ){
8 FILE *allomany;

9
10    setlocale(LC_ALL, "");
11    allomany=fopen("/nincs/ilyen/fajl", "r" );
12    if( allomany==NULL ){
13        fprintf( stderr, "open(): %s\n", strerror(errno) );
14        exit( EXIT_FAILURE );
15    }/*if*/

16
17    /*
18     * További programrészek helye.
19     */
20    exit( EXIT_SUCCESS );
21 }/*main*/

```

*Ha a programot lefordítjuk és futtatjuk, a hibaüzenet a környezeti változóknak megfelelően jelenik meg:*

```

Bash$ ./locale
open(): Nincs ilyen fájl vagy könyvtár
Bash$

```

**3. példa.** A következő példaprogram kiírja a szabványos kimenetre a használt nyelv néhány jellemzőjét.

```

1 #define _GNU_SOURCE
2 #include <langinfo.h>
3 #include <locale.h>
4 #include <stdio.h>

5
6 main(){
7     printf( "A nyelv: %s\n", setlocale(LC_ALL, "") );
8
9     printf( "Kódlap:  %s\n\n", nl_langinfo(CODESET) );
10

```

```

11   printf( "Pénzösszegek formája\n");
12   printf( "-----\n");
13   printf( "Nemzetközi jel: %s\n",
14         nl_langinfo(INT_CURR_SYMBOL) );
15   printf( "Rövidítés:      %s\n",
16         nl_langinfo(CURRENCY_SYMBOL) );
17   printf( "Tizedesek száma: %hu\n",
18         *nl_langinfo(FRAC_DIGITS) );
19
20   if( *nl_langinfo(P_CS_PRECEDES)==1 )
21     printf( "A pénznem a szám előtt.\n" );
22   else
23     printf( "A pénznem a szám után.\n" );
24 }/*main*/

```

A példaprogram 2–4. sorában látható, ahogyan betöljük a szükséges állományokat. A 1. sorban létrehozott állandó miatt a `langinfo.h` állomány létrehozza azokat az eszközöket is, amelyek nem szabványosak, amelyeket csak a GNU C könyvtár segítségével használhatunk.

A program a 7. sorban betölti a környezeti változók által előírt nyelv összes jellemzőjét. Ettől a függvényhívástól kezdve a program számára minden nyelvi jellemző a környezeti változókban előírt nyelv szerint áll rendelkezésre. A 7. sor kiírja a `setlocale()` visszatérési értéke által jelzett szöveget, amely az éppen beállított nyelv szabványos neve.

A program a 9. sorban kiírja a beállítások által előírt kódlap nevét. Ez igen fontos információ, hiszen ha nem a megfelelő kódlapot használjuk, programunk kimenete olvashatatlan lesz.

A program a 11–23. sorban kiírja a szabványos kimenetre a pénz jellegű értékek kezelésére előírt néhány jellemzőt. Figyeljük meg a 17–18. sorban, hogy az `nl_langinfo()` által visszaadott értéket hogyan használjuk szám jellegű, bájt méretű adatok átvételére.

A program futásakor kapott eredmények függnek a környezeti változóktól, ahogyan a következő példa is mutatja:

```
Bash$ export LANG=en_US
Bash$ ./langinfo
A nyelv: en_US
Kódlap: ISO-8859-1
```

*Pénzösszegek formája*

*-----*

*Nemzetközi jel: USD*

*Rövidítés: \$*

*Tizedesek száma: 2*

*A pénznem a szám előtt.*

*Bash\$ export LANG=hu\_HU*

*Bash\$ ./langinfo*

*A nyelv: hu\_HU*

*Kólap: ISO-8859-2*

*Pénzösszegek formája*

*Nemzetközi jel: HUF*

*Rövidítés: Ft*

*Tizedesek száma: 2*

*A pénznem a szám után.*

*Bash\$*

## Karakterláncok kezelése

A következő néhány függvény karakterláncok kezelésére szolgál. Valójában a karakterláncok kezelését nem támogatja a C nyelv – azaz nincs karakterlánc típus a nyelvben –, ezért a legegyszerűbb értékadáshoz is az itt tárgyalt függvények valamelyikét kell használnunk.

Ezek a függvények a karakterláncokat egydimenziós char típusú tömbökként ábrázolják, a karakterlánc végét egy 0 értékű bájttal jelzik. Ügyelünk kell a használatukkor, mert ha a karakterlánc végét nem jelezzük a 0 bájt elhelyezésével, a függvények a lefoglalt memóriaterületen túlhaladva idegen memóriaterületre tévedhetnek, és ez a program megszakítását eredményezi.

### Szöveg mérete

A következő függvény a karakterlánc hosszát állapítja meg. A függvény használatához a string.h fejállomány betöltése szükséges.

```
size_t strlen(const char *szöveg);
```

A függvény a paraméterként átadott mutatóval kijelölt szöveg méretét adja vissza. A méretbe nem számít bele a szöveg végét jelző 0 karakter.

A függvény által visszaadott érték a szöveg mérete.

## Másolás

A következő néhány függvény karakterláncok másolására használható. A függvények használatához a `string.h` fejállományt be kell tölteni.

```
char *strcpy(char *cél, const char *forrás);
```

A függvény karakterláncok másolására használható.

A függvény a második paraméterként átadott területről az ott található karakterláncot az első paraméterként jelzett területre másolja. A másolást az első 0 karakterig – a karakterlánc végéig – végzi. A 0 karakter még átmásolódik a célterületre, azaz a célterületen található karakterlánc szabályosan le lesz zárvva.

A forrás és a célterület nem lapolódhat át.

A függvény visszatérési értéke a célterület elejére mutat, vagyis megegyezik a függvény első paraméterével.

```
char *strncpy(char *cél, const char *forrás, size_t méret);
```

A függvény karakterláncok másolására használható, a másolt lánc méretének korlátozásával.

A függvény a második paraméterként átadott területről a karakterláncot az első paraméterként jelzett területre másolja a karakterlánc végét jelző 0 karakterrel együtt. Ha a másolás során a függvény a harmadik paraméter által jelzett számú karaktert átmásolta, a másolást nem folytatja tovább. Ekkor a célterületre másolt karakterlánc vége nem lesz 0 karakterrel lezárvva.

Ha a karakterlánc rövidebb, mint a harmadik paraméterként átadott méret, a függvény a fennmaradó területet 0 karakterekkel tölti fel.

A forrás és célterületek nem lapolódhatnak át.

A függvény visszatérési értéke a célterület elejére mutat, vagyis megegyezik a függvény első paraméterével.

```
char *strdup(const char *szöveg);
```

A függvény másolatot készít a karakterláncról egy általa foglalt memóriaterületre.

A függvény a megfelelő méretű memóriaterületet lefoglalja – a `malloc()` függvény hívásával –, majd a paraméter által jelölt karakterláncot oda másolja.

A visszaadott mutató a függvény által lefoglalt memóriaterület kezdőcíme, vagyis a másolat elejét jelzi.

---

```
char *strndup(const char *szöveg, size_t méret);
```

A függvény másolatot készít a karakterláncról egy általa lefoglalt memóriaterületre, ahol a másolat legfeljebb egy bájttal – a záró 0 karakterrel – hosszabb, mint a második paraméterként átadott méret.

A függvény a megfelelő méretű memóriaterületet lefoglalja – a `malloc()` függvény hívásával –, majd másolatot készít oda az első paraméter által jelzett memóriaterületről. Ha a karakterlánc hosszabb, mint a második paraméterként átadott méret, a függvény csak a méretnek megfelelő számú karaktert másolja át, de a másolat végét akkor is lezárja 0 karakterrel.

A visszaadott mutató a függvény által lefoglalt memóriaterület kezdőcíme, vagyis a másolat elejét jelzi.

```
char *stpcpy(char *cél, const char *forrás);
```

A függvény a második paraméter által jelzett területről az ott található karakterláncot az első paraméter által jelzett területre másolja.

A függvény hasonló az `strcpy()` függvényhez, de a visszaadott mutató nem a célterület első bájtjára mutat, hanem a célterület végén található 0 karakterre. Akkor használhatjuk ezt a függvényt, ha a célterület feltöltését tovább szeretnénk folytatni.

A függvény használatakor a forrás és a célterület nem lapolódhat át.

```
char *stpncpy(char *cél, const char *forrás, size_t méret);
```

A függvény a második paraméter által jelzett területről az ott található karakterláncot az első paraméter által jelzett területre másolja. A függvény legfeljebb a harmadik paraméterként átadott számnak megfelelő mennyiségű karaktert másol.

A függvény a második paraméterként átadott területről az ott található lezáró 0 karakterrel együtt másolja a karakterláncot a célterületre. Ha a karakterlánc hosszabb, mint a harmadik paraméterként átadott méret, a célterület nem lesz lezártva 0 karakterrel.

A függvény használatakor a forrás és a célterület nem lapolódhat át.

A függvény által visszaadott mutató a célterületet lezáró 0 értékre, ha a célterületet a függvény nem zárta le, a másolt utolsó karakterre mutat.

```
char *strcat(char *cél, const char *forrás);
```

A függvény két karakterláncot másol egymás után.

A függvény a második paraméterrel jelzett területen található karakterláncot az első paraméterrel jelzett helyen található karakterlánc végére – az azt záró 0 karaktert felülírva – másolja. A függvény a másolás után a célterületet lezártja egy 0 karakterrel.

A függvény által visszaadott érték a célterület elejét jelző mutató.

```
char *strncat(char *cél, const char *forrás, size_t méret);
```

A függvény két karakterláncot másol egybe.

A függvény a második paraméterrel jelzett területen található karakterláncot az első paraméterrel jelzett helyen található karakterlánc végére – az azt záró 0 karaktert felülírva – másolja. Ha a forrásterület hossza nem nagyobb, mint a harmadik paraméterként megadott méret, a forrásterület végét jelző 0 karakter is át lesz másolva, ha pedig hosszabb, a függvény legfeljebb a méretnek megfelelő számú karaktert másol.

A függvény által visszaadott érték a célterület elejét jelző mutató.

## Összehasonlítás

A GNU C könyvtár néhány függvénye karakterláncok összehasonlítására szolgál. Mivel a karakterláncok valójában tömbök, ezekkel a függvényekkel kell összehasonlítanunk a karakterláncokat, hiszen az összehasonlításra használt műveleti jelek nem használhatók tömbök esetében.

A következő néhány függvény használatához a `string.h` fejállomány betöltése szükséges.

```
int strcmp(const char *a, const char *b);
```

A függvény segítségével két karakterlánc hasonlítható össze.

A függvény első és második paramétere a két összehasonlítandó karakterláncot jelölő mutató.

A függvény visszatérési értéke 0, ha a két karakterlánc megegyezik. A függvény visszatérési értéke negatív, ha az első paraméterként kijelölt karakterlánc kisebb (előrébb található az ábécérendben) mint a második paraméter által kijelölt karakterlánc, és pozitív, ha ez első paraméter által kijelölt karakterlánc nagyobb, mint a második paraméter által jelölt karakterlánc.

```
int strcoll(const char *a, const char *b);
```

A függvény segítségével két karakterláncot hasonlíthatunk össze a nyelvi beállításoknak megfelelő módon. A függvény használata előtt a `setlocale()` függvény segítségével beállíthatjuk a használt nyelvet, amely a karakterláncok sorrendjét is meghatározza.

A függvény első és második paramétere a két összehasonlítandó karakterláncot jelölő mutató.

A függvény visszatérési értéke 0, ha a két karakterlánc megegyezik. A függvény visszatérési értéke negatív, ha az első karakterlánc kisebb, mint a második, és pozitív, ha ez első karakterlánc nagyobb, mint a második.

```
int strcasecmp(const char *a, const char *b);
```

A függvény segítségével két karakterlánc betűit hasonlíthatjuk össze, mégpedig úgy, hogy a kis- és nagybetűk közt nem teszünk különbséget. A függvény a kis- és nagybetűk egymáshoz rendeléséhez felhasználja a nyelvi információkat, amelyeket a `setlocale()` függvény segítségével állíthatunk be.

A függvény első és második paramétere a két összehasonlítandó karakterláncot jelölő mutató.

A függvény visszatérési értéke 0, ha a két karakterlánc megegyezik. A függvény visszatérési értéke negatív, ha az első karakterlánc kisebb, mint a második, és pozitív, ha ez első karakterlánc nagyobb, mint a második.

```
int strncmp(const char *a, const char *b, size_t n);
```

A függvény segítségével két karakterlánc első néhány betűjét hasonlíthatjuk össze egymással.

A függvény első és második paramétere a két összehasonlítandó karakterláncot jelölő mutató. A függvény harmadik paramétere megadja, hogy a karakterláncok elejének hány betűjét kívánjuk vizsgálni az összehasonlítás során.

A függvény visszatérési értéke 0, ha a két karakterlánc megegyezik a vizsgált hosszon. A függvény visszatérési értéke negatív, ha az első karakterlánc kisebb, mint a második, és pozitív, ha ez első karakterlánc nagyobb, mint a második.

```
int strncasecmp(const char *a, const char *b, size_t méret);
```

A függvény segítségével két karakterlánc első néhány betűjét hasonlíthatjuk össze egymással úgy, hogy a kis- és a nagybetűk közt nem teszünk különbséget. A függvény a kis- és nagybetűk párosításához felhasználja a nyelvi információkat, amelyeket a `setlocale()` függvénnyel állíthatunk be.

A függvény első és második paramétere a két összehasonlítandó karakterláncot jelölő mutató. A függvény harmadik paramétere megadja, hogy a karakterláncok elejének hány betűjét kívánjuk vizsgálni az összehasonlítás során.

A függvény visszatérési értéke 0, ha a két karakterlánc megegyezik a vizsgált hosszon. A függvény visszatérési értéke negatív, ha az első karakterlánc kisebb, mint a második. A visszatérési érték pozitív, ha ez első karakterlánc nagyobb, mint a második.

## Keresés

Különösen hasznosak a C könyvtár karakterláncban való keresésre szolgáló függvényei. Ezeknek a függvényeknek a segítségével karakterláncokban kereshetünk részleteket.

A keresésre szolgáló függvények használatához a `string.h` fejállomány betöltése szükséges.

```
char *strchr(const char *szöveg, int karakter);
```

A függvény segítségével karaktert kereshetünk karakterláncban. A függvény a karakter balról számított első előfordulását keresi meg a karakterláncban.

A függvény első paramétere a karakterláncot jelöli, amelyben a karaktert keresük, a második paraméter a karakter, amelyet keresünk.

A függvény visszatérési értéke egy mutató, amely a balról számított első előfordulásra mutat, és NULL, ha a karakter nem található.

```
char * strrchr(const char *s, int c);
```

A függvény segítségével karaktert kereshetünk karakterláncban. A függvény a karakter jobbról számított első előfordulását keresi meg a karakterláncban. A függvény működésében megegyezik az strchr() függvény működésével, azzal a különbséggel, hogy ez a függvény a karakterlánc végétől a karakterlánc eleje felé keres.

A függvény első paramétere a karakterláncot jelöli, amelyben a karaktert keresük, a második paraméter pedig a karakter, amelyet keresünk.

A függvény visszatérési értéke egy mutató, amely a jobbról számított első előfordulásra mutat, és NULL, ha a karakter nem található.

```
char *strstr(const char *szöveg, const char *rész);
```

A függvény segítségével karakterláncot kereshetünk karakterláncban.

A függvény első paramétere a karakterláncot jelöli, amelyben keresni akarunk, a második paramétere pedig a szöveget, amelyet meg akarunk találni benne.

A függvény visszatérési értéke egy mutató, amely az első paraméterben megtalálható részlánc első betűjére mutat, ha a részlánc megtalálható, és NULL ha nem.

```
char *strcasestr(const char *szöveg, const char *rész);
```

A függvény segítségével karakterláncot kereshetünk karakterláncban úgy, hogy a kis- és nagybetűk között nem teszünk különbséget. A függvény a kis- és nagybetűk egymáshoz rendeléséhez felhasználja a nyelvi információkat, amelyeket a setlocale() függvény segítségével állíthatunk be.

A függvény első paramétere a karakterláncot jelöli, amelyben keresni akarunk, a második paramétere pedig a szöveget, amelyet meg akarunk találni benne.

A függvény visszatérési értéke egy mutató, amely az első paraméterben megtalálható részlánc első betűjére mutat, ha a részlánc megtalálható, és NULL ha nem.

```
size_t strspn(const char *szöveg, const char *halmaz);
```

A függvény segítségével karakterlánc elején található részlánc hosszát állapíthatjuk meg, a benne található karakterek meghatározásával.

A függvény első paramétere a szöveget jelöli, amelyben az első részt vizsgáljuk, a második paramétere pedig egy karakterhalmazt határoz meg. A második paraméterként kijelölt karakterláncban a karakterek sorrendje mellékes, az csak a karakterek felsorolása.

A függvény visszatérési értéke megadja, hogy az első paraméterként kijelölt szöveg elejének mekkora része áll a második paraméterként jelölt karakterekből.

```
size_t strcspn(const char *szöveg, const char *halmaz);
```

A függvény segítségével karakterlánc elején található részlánc hosszát állapít-hatjuk meg, a benne található karakterek meghatározásával. A függvény hasonlóképpen viselkedik, mint a strspn(), azzal a különbséggel, hogy második paraméterként nem a karakterhalmazt, hanem annak ellentettjét kell megad-nunk.

A függvény első paramétere a szöveget jelöli, amelyben az első részt vizsgáljuk, a második paramétere pedig egy karakterhalmazt határoz meg. A második paraméterként kijelölt karakterláncban a karakterek sorrendje mellékes, az csak a karakterek felsorolása.

A függvény visszatérési értéke megadja, hogy az első paraméterként kijelölt szöveg elejének mekkora részében nem található meg a második paraméterként jelölt karakterek egyike sem. Más szavakkal azt is mondhatjuk, hogy a függvény visszatérési értéke megadja, hogy a karakterlánc hányadik karakterében szerepel a második paraméter valamelyik betűje elsőként.

```
char *strpbrk(const char *szöveg, const char *halmaz);
```

A függvény segítségével karakterláncban kereshetünk karaktereket egy karakterhalmaz megadásával.

A függvény első paramétere egy karakterláncot jelöl, amelyben karaktereket keresünk, a második paramétere pedig egy karakterláncot, amely a keresendő karaktereket tartalmazza. A második paraméterként átadott karakterláncban a karakterek sorrendje mellékes, az csak a halmazt adja meg az elemek felso-rolásával.

A függvény visszatérési értéke egy mutató, amely az első paraméterként jelölt szövegben jelöli ki a második paraméterrel jelölt karakterhalmaz valamely ele-mének első előfordulását. Ha az első paraméterben nem található meg a ka-rakterhalmaz egyetlen karaktere sem, a visszatérési érték érték NULL.

## Karaktervizsgálat és átalakítás

A C könyvtár rendelkezik néhány függvénytel, amelyeket arra használhatunk, hogy a karaktereket csoportokba soroljuk. Ezek a függvények egy karaktert kapnak paraméterként és a visszatérési értékkel jelzik, hogy az adott karakter bizonyos csoportba tartozik-e.

A függvények paraméterének típusa int, ahogyan ez a karakterek kódolására sok helyen használatos a C nyelvben. A visszatérési érték szintén int, amelynek 0 értéke a hamis, 1 értéke pedig az igaz logikai érték jelzésére szolgál.

A függvények viselkedése nagymértékben függ a nyelvi beállításoktól. Ez természetes, hiszen például az é karakter a magyar nyelvben betű, az angolban egy idegen jel. A nyelvi beállításokat a 68. oldalon leírt `setlocale()` függvénytel állíthatjuk be.

A karakterek osztályozására a `ctype.h` fejállományban létrehozott függvények a következők:

```
int islower(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere kisbetű.

```
int isupper(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere nagybetű.

```
int isalpha(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere betű.

```
int isdigit(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere számjegy.

```
int isalnum(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere betű vagy számjegy.

```
int isxdigit(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere a 16-os számrendszer számjegye, azaz számjegy vagy az abcdefABCDEF betűk valamelyike.

```
int ispunct(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere nyomtatható jel, de nem betű, nem szám és nem a szóköz karakter, azaz a különféle írásjelek és különleges karakterek egyike.

```
int isblank(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere a szóköz vagy a tabulátor karakter.

---

```
int isascii(int c);
```

A függvény visszatérési értéke igaz, ha a paramétere 7 bites ASCII karakter, azaz ha a legfelső bit értéke 0.

Szintén a ctype.h fejállomány betöltésével érhető el a következő néhány függvény, amelyek karakterek átalakítására szolgálnak. A karakterek ábrázolására a C könyvtár ezen függvények esetében is az int típust használja.

```
int tolower(int c);
```

Ha a paraméter nagybetű, a függvény visszatérési értéke a neki megfelelő kisbetű, ha a paraméter nem nagybetű, a visszatérési érték a paraméter értéke, változtatás nélkül.

```
int toupper(int c);
```

Ha a függvény paramétere kisbetű, a visszatérési érték a neki megfelelő nagybetű, ha a paraméter nem kisbetű, a visszatérési érték a paraméter értéke, változtatás nélkül.

```
int toascii(int c);
```

A függvény visszatérési értéke a paraméter 7 bites ASCII formára alakított változata. A függvény egyszerűen 0 értékre állítja a felső biteket, ahol azok értéke 1.

## Rendezés és keresés

A GNU C könyvtár néhány függvénye tömbök sorba rendezésére és tömbökben való keresésre szolgál.

Tömbök rendezésére használhatjuk a következő függvényt, amelynek használatához az stdlib.h fejállomány betöltése szükséges.

```
void qsort(void *tomb, size_t darab, size_t elemméret,
           int(*hasonl)(const void *, const void *));
```

A függvény tömbök elemeit rendez a gyorsrendezés algoritmussal (*quick sort algorithm*).

A függvény első paramétere a rendezni kívánt tömböt jelöli a memóriában, a második paramétere azt adja meg, hogy hány elem található a tömbben, a harmadik pedig egy tömbelem méretét adja.

A függvény negyedik paramétere egy összehasonlító függvény címe, amely két elem összehasonlítását végzi. Ez a függvény két memóriacímet kap a két összehasonlítandó elem címével, a visszatérési értéke pedig 0, ha a két elem

egyenlő, negatív, ha az első kisebb, mint a második, és pozitív, ha az első nagyobb, mint a második. Láthatjuk, hogy ennek a függvénynek az `strcmp` függvényhez hasonlóan kell működnie.

A `qsort()` függvénynek nincs visszatérési értéke, a tömböt mindenképpen sorba rendezzi.

**4. példa.** A következő példaprogram a `qsort()` függvény használatát mutatja be. Neveket és címeket tartalmazó tömböt rendez nevek, majd címek szerint, és kiírja a rendezett tömb elemeit.

	rendezkeres/qsort.c	
--	---------------------	--

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <locale.h>
5
6 typedef struct cim {
7     char    nev[16];
8     char    cim[32];
9 } cim;
10
11 cim  cimek[] = {
12     {"Béna Géza", "Nekeresd u. 6."},
13     {"Bú Botond", "Némancs u. 61."},
14     {"Alak Elek", "Kikerics u. 9."},
15     {"Áram Béla", "Alapos tér 18."}
16 };
17
18 #define MERET 4
19
20 void kiir( void ){
21     int q;
22     printf("\nNÉV           CÍM  \n");
23     for( q=0; q<MERET; ++q )
24         printf( "%-16s %-32s\n",
25                 cimek[q].nev, cimek[q].cim );
26 /*kiir*/
27
28
29 int hasonlit_nev( const cim *a, const cim *b ){
30     return( strcoll(a->nev, b->nev) );
31 /*hasonlit_nev*/
32
33

```

```

34  int hasonlit_cim( const cim *a, const cim *b ){
35      return( strcoll(a->cim, b->cim) );
36  }/*hasonlit_cim*/
37
38  int main( int argc, char *argv[] ){
39      setlocale( LC_ALL, "" );
40
41      qsort( cimek, MERET, sizeof(cim), hasonlit_nev );
42      kiir();
43      qsort( cimek, MERET, sizeof(cim), hasonlit_cim );
44      kiir();
45
46      exit( 0 );
47 }/*main*/

```

A program a 41. és 43. sorokban hívja a qsort() függvényt, egyszer a neveket összehasonlító hasonlit\_nev(), egyszer pedig a címekeket összehasonlító hasonlit\_cim függvényeket adva meg. Figyeljük meg, hogy a függvények neve a () zárójelpár nélkül a függvények címét adja, amely alapján a qsort() képes meghívni a függvényt.

A program futtatáskor a következő eredményt adja:

Bash\$ ./qsort

NÉV	CÍM
Alak Elek	Kikerics u. 9.
Áram Béla	Alapos tér 18.
Béna Géza	Nekeresd u. 6.
Bú Botond	Némancs u. 61.

NÉV	CÍM
Áram Béla	Alapos tér 18.
Alak Elek	Kikerics u. 9.
Béna Géza	Nekeresd u. 6.
Bú Botond	Némancs u. 61.

Bash\$

Figyeljük meg, hogy a setlocale() és strcoll() függvények segítségével a rendezés a magyar nyelv szabályainak megfelelően történt.

Tömbökben való keresésre használhatók a következő függvények. Ezek használatához az stdlib.h és a search.h betöltése lehet szükséges.

```
void *lfind(const void *keresett, const void *tömb, size_t *darab,
            size_t elemméret, int(*hasonl)(const void *, const void *));
```

A függvény segítségével rendezetlen tömbben kereshetünk meg egy bizonyos elemet. A függvény a tömb végigolvasásával keresi meg az elemet, szerencsétlen esetben az összes tömbelemmel összehasonlítva a keresett elemet.

A függvény első paramétere a keresett elemet jelöli a memóriában, a második pedig a tömböt, amelyben a keresését el kívánjuk végezni.

A függvény harmadik paramétere a tömbben található elemek darabszámát tartalmazó változót jelöli, a negyedik pedig az elemek méretét.

A függvény ötödik paramétere az összehasonlításra szolgáló függvény – a `qsort()` függvénynél használt formában –, amely 0 értéket ad vissza, ha a két paraméterként jelölt tömbelem egyezik.

A függvény visszatérési értéke a tömbben megtalált elemet jelöli a memóriában, esetleg `NULL`, ha az elem nem található.

```
void *lsearch(const void *keresett, const void *tömb, size_t
              *darab, size_t elemméret, int(*hasonl)(const void *, const
              void *));
```

A függvény működésében megegyezik a `lfind()` függvénnnyel, azzal a különbséggel, hogy ha nem találja az elemet, azt bemásolja a tömb utolsó eleme után, valamint növeli az elemszámot tartalmazó változó értékét. Ezt a harmadik paraméterként átadott mutató követésével teszi.

```
void *bsearch(const void *keresett, const void *tömb, size_t
              darab, size_t elemméret, int(*hasonl)(const void *, const void
              *));
```

A függvény segítségével keresést hajthatunk végre rendezett tömbökben. A tömbnek az összehasonlító függvény szerint növekvő sorrendben rendezettnak kell lennie, így a függvény a gyors – kevés összehasonlítást alkalmazó – bináris keresőalgoritmusossal dolgozhat. A függvény olyan tömbökön képes keresést végezni, amelyeket előzőleg rendeztünk a `qsort()` függvénnnyel.

A függvény első paramétere a keresett elemet jelöli, a második pedig a tömböt, amelyben keresünk. A harmadik paraméter a tömb elemeinek száma, a negyedik pedig az elemek mérete. Az ötödik paraméter az elemek összehasonlítását végző függvényt jelölő mutató, amely a `qsort()` függvénynél bemutatott módon 0 visszatérési értékkel jelzi, ha a két elem egyenlő, pozitív értékkel, ha az első paraméterként jelölt tömbelem nagyobb, valamint negatív értékkel, ha a második elem a nagyobb.

A függvény visszatérési értéke a tömbben megtalált elemet jelöli a memóriában, ha a keresés eredményes volt, egyébként pedig `NULL`.

## Mintaillesztés

A C könyvtár két eszközt is biztosít a számunkra a karakterláncok segítségével megvalósítható mintaillesztésre. A két eszköztár közül az egyszerűbb állománynevek kezelését hivatott segíteni, a bonyolultabb pedig általánosan használható.

Állománynevek kezelését, a programjainkban használható állománynév-szűrők létrehozását segíti a következő függvény – melynek használatához a `fnmatch.h` be-töltése szükséges:

```
int fnmatch(const char *minta, const char *szöveg, int kapcsoló);
```

A függvény segítségével a héj állománynév-helyettesítő karakter behelyettesítésének megfelelő mintaegyezményt vizsgálhatunk.

A függvény első paramétere egy karakterláncot jelöl, amelyben állománynév-helyettesítő karakterek lehetnek, a második paramétere pedig egy állománynév, amelynek az illeszkedését szeretnénk vizsgálni.

A függvény harmadik paramétere a viselkedést befolyásoló kapcsoló, amely a következő állandóból bitenkénti vagy művelettel képzett kifejezés lehet:

`FNM_FILE_NAME` A / jelek különleges kezelését kapcsolja be, így az állománynevek kezelésekor különösen hasznos. Ha ezt az állandót bekapsoljuk, semmi sem helyettesítheti a / jelet.

`FNM_PATHNAME` Ugyanaz, mint az `FNM_FILE_NAME`.

`FNM_PERIOD` Ezzel az állandóval a Unix rejtett állományainak kezelésére használható viselkedés kapcsolható be. Ha az állandót megadjuk, a `szöveg` első betűjeként álló . karakterre egyetlen minta sem illeszthető.

`FNM_NOESCAPE` A mintában kikapsolja a \ karakter különleges kezelését.

`FNM.LEADING_DIR` Ha ezt az állandót megadjuk, a `szöveg` első részét – az első / jelig – egyeztetjük, a többöt a függvény nem veszi figyelembe.

`FNM_CASEFOLD` A mintaegyeztetés során a függvény nem tesz különbséget a kis- és nagybetűk közt, ha ezt az állandót megadjuk.

A függvény visszatérési értéke 0, ha a második paraméterként jelzett állománynév illeszthető az első paraméterként jelzett karakterláncra, különben `FNM_NOMATCH`.

Általánosan használt mintaillesztő eszközöként a szabályos kifejezések nyelve használható, amelynek támogatására a következő – a `regex.h` fejállományban létrehozott – eszközök állnak a rendelkezésünkre:

```
int regcomp(regex_t *kész, const char *szab, int kapcs);
```

Ezzel a függvénnyel kell előkészítenünk – lefordítanunk – a szabályos kifejezést, mielőtt azt egyezésvizsgálatra használnánk.

A függvény első paramétere a `regex_t` típusú struktúrát jelölő mutató, ahová a függvény a lefordított szabályos kifejezést helyezi.

A függvény második paramétere a szabályos kifejezést tartalmazó karakterláncot jelöli.

A függvény harmadik paramétereként átadott kapcsoló a következő állandók, vagy azokból bitenkénti *vagy* művelettel felépített kifejezés lehet:

`REG_EXTENDED` A minta egy továbbfejlesztett szabályos kifejezés.

A POSIX szabvány írja le az egyszerű és továbbfejlesztett szabályos kifejezések nyelvét, amelyet a C könyvtár használ. Mivel az egyszerű szabályos kifejezések nyelve meglehetősen kevés eszközzel rendelkezik, ezt a kapcsolót valószínűleg be kell kapcsolnunk munkánk során.

`REG_ICASE` A mintaegyeztetés során a nagy- és kisbetűk közti különbséget nem kell figyelembe venni.

`REG_NOSUB` A mintaegyeztetés során az egyes egyező részeket nem kell kigyűjteni.

A mintaegyeztetést vizsgáló – később tárgyalt – függvény képes a mintára illeszkedő részeket kivágni. Ha nem akarunk ezzel a lehetőséggel elni, már a fordítás során jelezhetjük ezzel a kapcsolóval, így gyorsabban kezelhetővé tehetjük a lefordított szabályos kifejezést.

`REG_NEWLINE` Az újsor karakter különleges kezelésének bekapcsolása. Ekkor a `.` nem illeszthető az újsor karakterre, a `$` az újsor karakter elő, a `^` pedig az újsor karakter után illeszthető.

A visszatérési érték 0, ha a szabályos kifejezés lefordítása sikeres volt.

```
int regexec(regex_t *reg, char *szöv, size_t méret, regmatch_t
rész[], int kapcs);
```

A függvény megkíséri illeszteni a lefordított szabályos kifejezést a szövegre.

A függvény utolsó paramétere a mintaegyeztetés során használt kapcsolókat tartalmazza (bitenkénti *vagy* művelettel), amely a következő részekből állhat:

`REG_NOTBOL` A szöveg nem a sor elején található, nem illeszthető rá a sor eleje jel (`^`).

`REG_NOTEOL` A szöveg nem a sor végén található, a végére nem illeszthető a sor vége jel (`$`).

A harmadik és negyedik paraméterek a szabályos kifejezés alapján részkifejezésekre való bontásra szolgálnak. Ha nem kívánjuk a szöveget részeire bontani a szabályos kifejezés alapján, használunk 0-t harmadik és NULL értéket negyedik paraméterként.

Ha részekre kívánjuk bontani a szöveget, készítenünk kell egy `regmatch_t` típusú tömböt. A függvény a tömb nulladik címén elhelyezi a szabályos kifejezésre illeszkedő szövegrészt, a további címeken pedig a szabályos kifejezésben található részkifejezésekre illesztett részek adatait. A `regmatch_t` struktúra mezői a következők:

A <code>regmatch_t</code> struktúra	
<code>int rm_so</code>	Hányadik betűnél kezdődik az illeszthető rész, -1, ha nem található.
<code>int rm_eo</code>	Hányadik betűnél végződik az illeszkedő rész.

A részkifejezésekre vágást a 88. oldalon található 6. példa mutatja be.

A függvény visszatérési értéke jelzi, hogy a mintát illeszteni lehetett-e a szöveg valamely részére. A visszatérési érték a következők egyike lehet:

- 0 Az illesztés sikeres volt, a szöveg – vagy egy része – illeszthető a szabályos kifejezésre.

`REG_NOMATCH` A szöveg nem illeszthető a szabályos kifejezésre.

`REG_ESPACE` A műveletet nem lehetett végrehajtani, mert nem volt elegendő szabad memória.

**5. példa.** A következő példaprogram a szabályos kifejezések használatát mutatja be:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19	<pre>#include &lt;stdio.h&gt; #include &lt;sys/types.h&gt; #include &lt;regex.h&gt;  int main( int argc, char *argv[] ){     regex_t ford;      if( regcomp( &amp;ford, argv[1], 0 ) != 0 ){         printf("Hiba a szabályos kifejezésben.\n");         exit(1);     }/*if*/     if( regexec( &amp;ford, argv[2], 0, NULL, 0 ) == 0 ){         printf("igen\n");         return(0);     }else{         printf("nem\n");         return(1);     }/*if*/ }/*main*/</pre>	szabalyoskif/egyszeru.c
---	---	-------------------------

A program egyszerűen csak megpróbálja az első paraméterét szabályos kifejezésként értelmezve illeszteni a második paramétere. Ha az illesztés sikeres, a program kiírja a szabványos kimenetre az igen szót, ellenkező esetben pedig a nem szó az üzenet:

```
Bash$ ./egyszeru '^[0-9]*$' '8973425'
igen
Bash$
```

A példaprogram 8. sorában fordítjuk le a szabályos kifejezést a `regcomp()` függvény segítségével. Figyeljük meg, hogy a függvény harmadik paramétere 0, semmilyen kapcsolót nem álltottunk be a fordításhoz.

A tulajdonképpeni illeszkedésvizsgálat a 12. sorban a `regexec()` függvény hívásával történik. Az illeszkedésvizsgálat során nem kértük a szöveg részekre vágását, ezért a függvény 3. paramétere 0, a negyedik pedig `NULL`. Az illeszkedésvizsgálat során kapcsolókat nem álltottunk be, ezért a függvény utolsó paramétere 0.

**6. példa.** A következő példaprogram bemutatja, hogyan használható a `regcomp()` és a `regexec()` szöveg szétdarabolására.

szabalyoskif/szetvag.c

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <regex.h>
4
5 void kiir( char *str, int eleje, int vege ){
6     int n;
7
8     if( eleje== -1 )
9         return;
10
11    for( n=eleje; n<vege; n++ )
12        putchar( str[n] );
13        putchar( '\n' );
14    }/*kiir*/
15
16 int main( int argc, char *argv[] ){
17     char      szab [] = "(http|ftp)://([a-zA-Z]+.)*([/a-zA-Z]+.)*";
18     regex_t   ford;
19     regmatch_t tomb [5];
20
21     if( regcomp( &ford, szab, REG_EXTENDED ) != 0 ){
22         printf("Hiba a szabályos kifejezésben. \n");
23         exit(1);
24     }/*if*/
```

```

26     if( regexec( &ford, argv[1], 5, tomb, 0 ) != 0 ){
27         printf("Ismeretlen formátum. \n");
28         return(1);
29     }/*if*/
30
31     printf("Protokoll: ");
32     kiir( argv[1], tomb[1].rm_so, tomb[1].rm_eo );
33
34     printf("Gépnév:      ");
35     kiir( argv[1], tomb[2].rm_so, tomb[2].rm_eo );
36
37     printf("Állomány:    ");
38     kiir( argv[1], tomb[3].rm_so, tomb[3].rm_eo );
39
40     return(0);
41 }/*main*/

```

Figyeljük meg a példaprogramban, hogy a `regmatch_t` struktúra azt tárolja, hányadik betűnél kezdődik és végződik az illeszkedő rész, nem pedig az illeszkedő részt jelöli mutatóval. Látható a példaprogramban az is, hogy a részkifejezésekre illesztett szövegrészek adatai a tömb 1-es címétől kezdve találhatók meg. A 0 címen a teljes illesztett szöveg található.

A program egy futtatásának eredménye a következő:

```
Bash$ ./szetvag http://linux.pte.hu/index.html
Protokoll: http
Gépnév:    linux.pte.hu
Állomány:  /index.html
Bash$
```

## Matematikai függvények

A GNU C könyvtár a vonatkozó szabványoknak megfelelően több eszközt is biztosít matematikai problémák megoldására. Ezek az eszközök a `float` (lebegő), a `double` (dupla) és a `long double` (hosszú dupla) típusokat használják, melyek mindegyike lebegőpontos típus, alkalmazás tizedes törtek kezelésére.

A `float` típus a leggyorsabb – legkevesebb erőforrást igénylő – működést biztosítja, aminek az ára a kisebb pontosság. A `float` a legkisebb pontosságú lebegőpontos típus. Pontosabb – és egyben több erőforrást igénylő – típus a `double`. A három típus közül a leg pontosabb a `long double` típus.

### 2.1. TÁBLÁZAT: Matematikai állandók

Állandó neve	Jelentés
M_E	$e$
M_LOG2E	$\log_2 e$
M_LOG10E	$\lg e$
M_LN2	$\ln 2$
M_LN10	$\ln 10$
M_PI	$\pi$
M_PI_2	$\frac{\pi}{2}$
M_PI_4	$\frac{\pi}{4}$
M_1_PI	$\frac{1}{\pi}$
M_2_pi	$\frac{2}{\pi}$
M_2_SQRTPI	$\frac{2}{\sqrt{\pi}}$
M_SQRT_2	$\sqrt{2}$
M_SQRT1_2	$\frac{1}{\sqrt{2}}$

A C programkönyvtár által létrehozott matematikai állandókat a 2.1. táblázatban találhatjuk. Ezeknek az állandóknak a használatához a \_BSD\_SOURCE előfeldolgozó makrót létre kell hoznunk (akár tartalom nélkül), majd a math.h fejállományt be kell töltenünk.

A matematikai függvények használatához szintén a math.h betöltése szükséges. Az ilyen függvényeket használó programok fordításakor az m könyvtárat a programhoz kell szerkeszteni a gcc -l m kapcsolója segítségével. Ezt mutatja be a következő példa:

```
Bash$ gcc -lm matek.c -o matek
Bash$
```

Található e fejezetben néhány olyan eszköz is, amelyek nem a math.h fejállományban vannak létrehozva és a használatukhoz nem szükséges az m könyvtár használata. Ezek az eszközök azért kerültek ide, mert általában matematikai jellegű problémák megoldásakor bizonyosnak hasznosnak.

## Számok ábrázolása karakterláncként

A számokat gyakran ábrázoljuk karakterláncként, hiszen a felhasználó általában karakterláncként írja be őket és a képernyőn is karakterláncként szeretné elolvasni azokat.

Ha számokat szeretnénk karakterlánc formára alakítani, használhatjuk a szabványos kimenetre író printf() függvényt, vagy a memoriába író sprintf() nevű változatát. Az itt felsorolt függvények segítségével karakterláncként megadott számokat alakíthatunk számmá. A függvények használatához az stdlib.h fejállományt be kell töltenünk.

```
int atoi(const char *szöveg);
```

A függvény segítségével karakterláncként megadott számokat alakíthatunk át int típusúvá.

A függvény első paramétere egy mutató, ami a karakterláncot jelöli a memóriában, amely a számot tartalmazza szövegként.

A függvény visszatérési értéke a számmá alakított szöveg értéke.

Az atoi() függvény helyett a GNU C könyvtár dokumentációja az strtol() függvényt javasolja használni.

```
long int atol(const char *szöveg);
```

A függvény segítségével karakterláncként megadott számokat alakíthatunk át long int típusúvá.

A függvény első paramétere egy mutató, ami a karakterláncot jelöli a memóriában, amely a számot tartalmazza szövegként.

A függvény visszatérési értéke a számmá alakított szöveg értéke.

Az atol() függvény helyett a GNU C könyvtár dokumentációja az strtol() függvényt javasolja használni.

```
long long int atoll(const char *szöveg);
```

A függvény segítségével karakterláncként megadott számokat alakíthatunk át long long int típusúvá.

A függvény első paramétere egy mutató, ami a karakterláncot jelöli a memóriában, amely a számot tartalmazza szövegként.

A függvény visszatérési értéke a számmá alakított szöveg értéke.

Az atoll() függvény helyett a GNU C könyvtár dokumentációja az strtoll() függvényt javasolja.

Ezek a függvények meglehetősen könnyen használhatók, egyszerű működésűek. A GNU C könyvtár nem is javasolja a használatukat, mivel sokak szerint egy ilyen fontos feladatot komolyabb függvényekre kell bíznunk. A következő néhány összetettebb függvény mindenképpen precízebb munkát tesz lehetővé. Ezeknek a függvényeknek a használatához az stdlib.h fejállomány betöltése szükséges.

```
long int strtol(const char *szöveg, char **maradék, int alap);
```

A függvény segítségével karakterláncként megadott számot alakíthatunk át long int típusúvá.

A függvény első paramétere a szöveget jelöli a memóriában, amelyet számmá akarunk alakítani.

A második paraméter egy mutatót jelöl a memóriában, amely a függvény lefutása után a szövegen a szám után található első karakterre mutat. A függvény

megkeresi, hogy a szövegként megadott szám hol végződik és az utána található szövegre állítja a második karakter által kijelölt mutatót. Ha a második paraméter értéke NULL, a függvény nem végzi el a mutató beállítását.

A függvény harmadik paramétere a használt számrendszer jelöli, amelynek értéke 2 és 36 között lehet. Ha ennek a paraméternek 0 az értéke, a függvény maga választ számrendszeret. A választott számrendszer alapértelmezés szerint a 10-es, ha az első paraméter elején a 0x vagy 0X kifejezés található, a 16-os. Ha az első paraméter első értékes – nem szóköz – karaktere a 0, amelyet szám követ, a választott számrendszer a 8-as.

A függvény visszatérési értéke a számmá alakított szöveg, ha a szöveg átalakítása sikeres volt, LONG\_MAX, ha a szám nagyobb, mint a long int típuson ábrázolható, és LONG\_MIN, ha a szám túl kicsi az ábrázoláshoz.

A függvény helyes lefutásának vizsgálatához nem használható a visszatérési érték, de használhatjuk a második paraméterrel jelölt mutatót. Ha a mutatót ugyanis a függvény a számára átadott szöveg elejére helyezi, egyetlen karaktert sem sikerült számként felismernie.

Hiba esetén a függvény beállítja az errno értékét, amely a következők egyike lehet:

**EINVAL** A harmadik paraméterként átadott szám érvénytelen számrendszert határozott meg.

**ERANGE** Az átalakítás során kapott szám túlságosan nagy vagy túlságosan kicsi a long int típusban való tároláshoz.

```
unsigned long int strtoul(const char *szöveg, char **maradék, int
    alap);
```

A függvény segítségével karakterláncként meg adott számot alakíthatunk át unsigned long int típusúvá.

A függvény az strtol() függvényhez hasonlóan működik, de a szövegesen ábrázolt számot előjel nélküli egésszé alakítja.

```
long long int strtoll(const char *szöveg, char **maradék, int
    alap);
```

A függvény segítségével karakterláncként megadott számot alakíthatunk át long long int típusúvá.

A függvény az strtoll() függvényhez hasonlóan működik, de a szövegesen ábrázolt számot long long int típusú számmá alakítja.

```
unsigned long long int strtoull(const char *szöveg, char
    **maradék, int alap);
```

A függvény segítségével karakterláncként megadott számot alakíthatunk át unsigned long long int típusúvá.

A függvény az `strtol()` függvényhez hasonlóan működik, de a szövegesen ábrázolt számot `unsigned long long int` típusú számmá alakítja.

A következő néhány függvény segítségével lebegőpontos számokká alakíthatjuk a karakterláncként ábrázolt számokat. Ezeknek a függvényeknek a használatához is be kell töltenünk az `stdlib.h` fejállományt.

```
double atof(const char *szöveg);
```

A függvény segítségével karakterláncként megadott számot alakíthatunk át `double` formátumban tárolt számmá.

A függvény első paramétere az átalakítandó karakterláncot jelöli a memoriában, a visszatérési értéke pedig az ott található szám `double` formátumban.

```
float strtod(const char *szöveg, char **maradék);
```

A függvény segítségével karakterláncként ábrázolt számot alakíthatunk `float` formátumra.

A függvény hasonlóképpen működik, mint a `strtol()` függvény; az első paramétere jelöli az átalakítandó karakterláncot a memoriában, a második pedig a mutatót, amelyben a függvény elhelyezi, hol található a karakterláncban a szám vége.

A függvény visszatérési értéke a karakterlánc elején tárolt szám `float` formátumban.

```
double strtold(const char *szöveg, char **maradék);
```

A függvény segítségével karakterláncként megadott számot alakíthatunk át `long double` formátumra.

A függvény hasonlóképpen viselkedik, mint a `strtol()` függvény, vagyis az első paramétere jelöli a memoriában az átalakítandó szöveget, a második pedig egy mutatót jelöl, ahová a függvény a karakterláncban tárolt szám végét jelző mutatót helyezi.

A függvény visszatérési értéke az első paraméterként jelölt karakterláncban tárolt szám `long double` formátumban.

```
long double strtold(const char *szöveg, char **maradék);
```

A függvény segítségével karakterláncként tárolt számot alakíthatunk át `long double` formátumra.

A függvény hasonlóképpen viselkedik, mint a `strtol()` függvény, azaz az első paraméter az átalakítandó karakterláncot, a második pedig a mutatót jelöli, amelyben a függvény a szám végét jelző mutatót helyezi el.

A függvény visszatérési értéke az első paraméterként jelölt karakterláncban tárolt szám `long double` formátumban.

**7. példa.** A következő példaprogram az strtod() függvény használatát mutatja be, de az strtod(), strtold() stb. függvényeket is hasonlóképpen kell használni. A program a paraméterként kapott karakterláncot átalakítja double típusú számmá és kiírja az értékét.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( int argc, char *argv[] ){
5     char    *vege;
6     double  erteke;
7
8     erteke=strtod( argv[1], &vege );
9     if( vege==argv[1] )
10        printf( "Nem is szám... \n" );
11     else
12        printf( "Értéke: %f\n", erteke );
13 }/*main*/

```

Figyeljük meg, hogy ha a szám eleje és a vége a függvény szerint egybeesik (9. sor), a hívó program arra következtet, hogy nem sikerült a szöveget számmá alakítani, mert annak formája nem megfelelő.

## Trigonometriai függvények

A C könyvtár által biztosított trigonometrikus függvények radiánban értelmezik a szögeket. A math.h fejállomány betöltésével és az m könyvtár programhoz való szerkesztésével (a gcc -lm kapcsolója) a következő trigonometrikus függvények érhetők el:

```

float sinf(float x);
double sin(double x);

long double sinl(long double x);

```

A szinusz függvény ( $\sin x$ ) kiszámítására szolgáló függvények.

A függvények paramétere a szám, amelynek szinuszát ki kívánjuk számítani, a visszatérési értékük pedig a szám szinuszát adja.

```

float cosf(float x);
double cos(double x);

```

```
long double cosl(long double x);
```

A koszinusz függvény ( $\cos x$ ) értékének kiszámítására szolgáló függvények.

A függvények paramétere a szám, amelynek koszinuszát ki kívánjuk számítani, a visszatérési értékük pedig a szám koszinuszát adja.

```
void sincosf(float x, float *s, float *c);
```

```
void sincos(double x, double *s, double *c);
```

```
void sincosl(long double x, long double *s, long double *c);
```

A függvények segítségével egyszerre számíthatjuk ki számok szinuszát és koszinuszát. Ez a módszer gyorsabb megoldást jelent, mintha a szinusz és koszinusz értéket külön függvényhívással számítanánk ki.

A függvények első paramétere a szám, melynek szinuszát és koszinuszát ki szeretnénk számítani. A függvények második paramétere egy mutató, amely azt a területet jelöli, ahová a szám szinuszát helyezi a függvény. A függvények harmadik paramétere azt a területet jelöli, ahová a szám koszinusza kerül.

```
float tanf(float x);
```

```
double tan(double x);
```

```
long double tanl(long double x);
```

A tangens függvény ( $\operatorname{tg} x$ ) értékének kiszámítására szolgáló függvények.

A függvény paramétere a szám, amelynek tangensét ki kívánjuk számítani, a visszatérési értéke pedig a szám tangense.

A tangens függvénynek  $\frac{\pi}{2}$  páratlan számú többszöröseinél szakadása van – ezeken a pontokon nem értelmezzük a függvényt. A `tan()`, `tanf()` és `tanl()` függvények az ilyen pontok közelében túlcordulást okoznak, amely egy jelzés (*signal*) küldését eredményezi a futó folyamat számára. Ha a jelzés fogadását a folyamat nem végzi el, a rendszer a folyamatot megszakítja.

```
float asinf(float x);
```

```
double asin(double x);
```

```
long double asinl(long double x);
```

Az arkusz szinusz ( $\sin^{-1} x$ ) értékének kiszámítására szolgáló függvények.

A függvények visszatérési értéke megadja, hogy az első paraméterként átadott szám melyik szám szinuszsa. Mivel a szinusz függvény periodikus, végtelen sok ilyen szám létezhet, melyek közül ezek a függvények minden a  $-\frac{\pi}{2}$  és  $\frac{\pi}{2}$  közé esőt adják vissza.

Az arkusz szinusz függvény értelmezési tartománya  $-1$  és  $+1$  közti értékekre korlátozódik ( $-1 \leq x \leq 1$ ); ha a függvények ezen a tartományon kívüli értéket kapnak, a hívó folyamat jelzést kap, ami a folyamat megszakítását eredményezheti.

```
float acosf(float x);
double acos(double x);
long double acosl(long double x);
```

Az arkusz koszinusz ( $\cos^{-1} x$ ) értékének kiszámítására szolgáló függvények.

A függvények visszatérési értéke megadja, hogy az első paraméterként átadott szám melyik szám koszinusza. Mivel a koszinusz függvény periodikus, végtelen sok ilyen szám létezhet, melyek közül ezek a függvények minden a  $0$  és  $\pi$  közé esőt adják vissza.

Az arkusz koszinusz függvény értelmezési tartománya  $-1$  és  $+1$  közti értékekre korlátozódik ( $-1 \leq x \leq 1$ ), ha a függvények ezen a tartományon kívüli értéket kapnak, a hívó folyamat jelzést (*signal*) kap, amely a folyamat megszakítását eredményezheti.

```
float atanf(float x);
double atan(double x);
long double atanl(long double x);
```

Az arkusz tangens  $\operatorname{tg}^{-1} x$  értékének kiszámítására szolgáló függvények.

A függvények visszatérési értéke megadja, hogy az első paraméterként átadott szám melyik szám tangense. Mivel a tangens függvény periodikus, végtelen sok ilyen szám létezhet, melyek közül ezek a függvények minden a  $-\frac{\pi}{2}$  és  $\frac{\pi}{2}$  közé esőt adják vissza.

## Hatványozás, gyökvonás és logaritmus

A következő néhány matematikai függvény helytelen használat – helytelen paraméterértékkal való hívás – esetén azt eredményezheti, hogy a hívó folyamat jelzést kap, ami a folyamat futását megszakíthatja.

A függvények használatához a `math.h` betöltése és az `m` programkönyvtár programhoz való szerkesztése szükséges (a `gcc -lm` kapcsolója).

```
float expf(float x);
double exp(double x);
```

```
long double expl(long double x);
```

A függvények segítségével kiszámíthatjuk  $e^x$  értékét.

```
float exp2f(float x);
```

```
double exp2(double x);
```

```
long double exp2l(long double x);
```

A függvények segítségével kiszámíthatjuk  $2^x$  értékét.

```
float exp10f(float x);
```

```
double exp10(double x);
```

```
long double exp10l(long double x);
```

A függvények segítségével kiszámíthatjuk  $10^x$  értékét.

```
float logf(float x);
```

```
double log(double x);
```

```
long double logl(long double x);
```

A függvények segítségével kiszámíthatjuk  $\ln x$  (természetes alapú logaritmus) értékét.

```
float log10f(float x);
```

```
double log10(double x);
```

```
long double log10l(long double x);
```

A függvények segítségével kiszámíthatjuk  $\lg x$  (tízes alapú logaritmus) értékét.

```
float log2f(float x);
```

```
double log2(double x);
```

```
long double log2l(long double x);
```

A függvények segítségével kiszámíthatjuk  $\log_2 x$  (kettes alapú logaritmus) értékét.

```
float powf(float alap, float kitevő);
```

```
double pow(double alap, double kitevő);
```

```
long double powl(long double alap, long double kitevő);
```

A függvények segítségével hatványozást végezhetünk( $x^y$ ). A függvények visszaadják az értéket, amelyet akkor kapunk, ha az alapot hatványozzuk a kitevőre.

```
float sqrtf(float x);
```

```
double sqrt(double x);
```

```
long double sqrtl(long double x);
```

A függvények segítségével kiszámíthatjuk  $\sqrt{x}$  értékét.

```
float cbrtf(float x);
```

```
double cbrt(double x);
```

```
long double cbrtl(long double x);
```

A függvények segítségével kiszámíthatjuk  $\sqrt[3]{x}$  értékét.

```
float hypotf(float x, float y);
```

```
double hypot(double x, double y);
```

```
long double hypotl(long double x, long double y);
```

A függvények segítségével kiszámíthatjuk  $\sqrt{x^2 + y^2}$  értékét, ami annak a derékszögű háromszögnek a befogója, amelynek két átfogója  $x$  és  $y$ .

## Véletlenszámok létrehozása

A GNU C könyvtár néhány függvényével véletlenszámokat hozhatunk létre.

Valójában a könyvtár által létrehozott számok nem teljesen véletlenszerűek. Igaz ugyan, hogy meglehetősen nehéz volna megjósolni a következő számot a véletlenszerű számsorozatban, de egy adott folyamat minden indításakor ugyanazt a számsorozatot kapja. Ha kilépünk a programból és újra elindítjuk ismét ugyanazokat a számokat állítja elő számára a C könyvtárban található véletlenszámokat létrehozó függvény. Ez hibakeresés esetén igen hasznos lehet, mivel megjósolhatóvá teszi a program viselkedését.

Ha azt szeretnénk, hogy a véletlenszám-sorozat ne ugyanazokat a számokat tartalmazza, mint a legutóbbi futtatáskor, a véletlenszámokat előállító rendszert újra kell indítanunk. Az újraindításkor át kell adnunk egy számot, amely meghatározza az előállított véletlenszámok sorozatát. Ha ugyanazt a számot adjuk az indításkor, a rendszer ugyanazt a számsorozatot állítja elő. Azt is mondhatnánk, hogy a C könyvtárban igen sok, végtelen számot tartalmazó véletlenszerű számsorozat van, amelyek közül bármikor, bármelyiket kiválaszthatjuk.

A C könyvtár véletlenszámok kezelésére szolgáló függvényei – melyek használatához az `stdlib.h` betöltése szükséges – a következők:

```
int rand(void);
```

A függvény véletlenszámok előállítására szolgál.

A függvény visszatérési értéke egy véletlen szám 0 és `RAND_MAX` közt.

```
void srand(unsigned int sorozat);
```

A függvény segítségével véletlenszámokat létrehozó rendszernek adhatjuk meg a kezdőértéket.

A függvény paramétere határozza meg a rand() függvény által visszaadott véletlenszámok sorozatát. A rand() függvény használható az srand() hívása nélkül is. Ilyen esetben olyan számsorozatot kapunk, mintha az srand(1) hívással állítottuk volna be a kezdőértéket.

Ha a program minden futtatásakor más és más sorozatot akarunk használni, a függvényt a számítógép beépített órájának pillanatnyi értékével hívjuk meg. Az óra lekérdezésére használhatjuk a time() függvényt.

**8. példa.** A következő példaprogram véletlenszerű számsorozatot ír a szabványos kimenetére. A programnak a parancssorban számmal megadhatjuk, hogy hány tagból álljon a sorozat. Ha nem adunk meg paramétert, a program egyetlen véletlenszámot ír a kimenetre. A program minden induláskor más és más sorozatot ír ki, ahogyan ezt a példa mutatja:

```
Bash$ ./rand 2
1277321455
284696243
Bash$ ./rand 2
1739554300
2056568096
Bash$
```

A véletlenszám-sorozatokat előállító program a következő:

	matematika/veletlen/rand.c
1	#include <stdlib.h>
2	#include <stdio.h>
3	#include <time.h>
4	
5	int main(int argc, char *argv[]){
6	int n, N;
7	
8	if( argc==1 )
9	N=1;
10	else
11	N=atoi(argv[1]);
12	
13	srand(time(NULL));
14	
15	for( n=1; n<=N; ++n )

```

16     printf("%d\n", rand());
17 }/*main*/

```

*Figyeljük meg a program 13. sorában, hogyan adjuk át az srand() függvénynek a számítógép beépített órájának pillanatnyi állását annak érdekében, hogy minden futtatáskor új és új számsorozatot kapunk.*

A C könyvtár további függvényei segítségével tágabb határok között lévő véletlenszámokat is előállíthatunk. Erre az itt felsorolt – szintén az stdlib.h fejállományban létrehozott – függvények használhatók.

```
double drand48(void);
```

A függvény segítségével 0 és 1 között állíthatunk elő véletlenszámokat.

```
long int lrand48(void);
```

A függvény segítségével 0 és  $2^{31}$  között állíthatunk elő véletlenszámokat, amelyeket a függvény egész számként ad vissza.

```
long int mrand48(void);
```

A függvény segítségével  $-2^{31}$  és  $2^{31}$  között állíthatunk elő véletlenszerű egész számokat.

```
void srand48(long int kezdeti);
```

A függvény segítségével a véletlenszámot létrehozó drand48(), lrand48(), mrand48(), srand48() függvények kezdőértékét állíthatjuk be az adott folyamatra.

A függvény hívása után létrehozott véletlenszám-sorozatot a függvény paramétereként átadott szám határozza meg.

## A környezeti változók kezelése

Minden Unix rendszeren futó folyamat rendelkezik környezeti változókkal (*environment variables*), amelyek nevükhez híven a futtató környezetet hivatottak leírni. A környezeti változók – mint általában a változók – névvel és értékkel rendelkeznek.

Tudnunk kell, hogy a folyamatok a környezeti változókat a szülőfolyamataiktól öröklök. Amikor egy folyamat egy gyermekfolyamatot indít, a környezeti változói-ról másolat készül, amely másolatot a gyermekfolyamat megkap. A gyermekfolyamatnak csak a számára készült másolathoz van hozzáférési jog, azt olvashatja vagy

módosíthatja, a szülőfolyamat változóihoz azonban nem fér hozzá. A környezeti változók módosításának hatása tehát egyirányú: a szülő meghatározhatja, milyen környezeti beállítások mellett fusson a gyermekfolyamat, de a gyermekfolyamat nem módosíthatja a szülő környezeti változóit.

A C könyvtár néhány egyszerű függvényt biztosít a környezeti változók kezeléséhez, amelyeket ebben a részben mutatunk be. Az itt bemutatott függvények használatához az `stdlib.h` fejállomány betöltése szükséges.

```
char *getenv(const char *név);
```

A függvény egy környezeti változó lekérdezésére szolgál. A függvény egyetlen paramétere a lekérdezni kívánt környezeti változó nevét jelöli ki a memóriában.

A visszaadott mutató a név nevű környezeti változó értéke, ha nincs ilyen változó NULL érték. A visszaadott mutató által kijelölt területet nem szabad megváltoztatni.

```
int putenv(char *értékkedés);
```

A függvény környezeti változók értékének beállítására és környezeti változók törlésére egyaránt használható.

Ha az első paraméterként jelölt karakterlánc tartalma NÉV=ÉRTÉK alakú értékkedés, a függvény elhelyezi az értéket a névvel jelzett környezeti változóban. Ha a környezeti változó nem létezik, a függvény létrehozza. Igaz fontos, hogy az átadott érték memóriaterülete a függvény sikeres vérehajtása esetén a környezet részévé válik, ennek a memóriaterületnek a tartalmát nem szabad megváltoztatnunk.

Ha a függvénynek átadott paraméter nem értékkedés – azaz nincs benne = jel –, a függvény feltételezi, hogy olyan környezeti változónak a neve, amelyet meg akarunk szüntetni és megkísérli törölni a környezeti változót.

A függvény 0 értéket ad, ha a művelet sikeres volt, és -1 értéket, ha nem.

```
int clearenv(void);
```

A függvény az összes környezeti változót törli.

A visszatérési érték 0, ha a művelet hibamentesen hajtódott végre, és -1, ha hiba lépett fel.

**9. példa.** A következő példaprogram megkíséri a kideríteni a folyamatot futtató felhasználó elektronikus levélcímét és azt a szabványos kimenetre írja.

	kornyal/getenv.c
1	#include <stdio.h>
2	#include <stdlib.h>
3	

```

4  main(){
5      printf( "E-mail cím: '%s@%s'\n",
6              getenv("USER"),
7              getenv("HOSTNAME") );
8 }
```

A program tulajdonképpen a *USER* és *HOSTNAME* környezeti változók értékét írja a megfelelő formában a szabványos kimenetre. Természetesen egyáltalán nem biztos, hogy ezek a környezeti változók a felhasználó felhasználói nevét és a számítógép nevét tartalmazzák, hiszen könnyen meg lehet, hogy a folyamatot futtató folyamat azt másképpen állította be. Csak annyit mondhatunk, hogy a hagyomány szerint ezek a változók a felhasználói nevet és a futtató számítógép nevét tartalmazzák.

A következő példaprogram előbb beállítja ezt a két környezeti változót, majd a szokott módon kiíratja a szabványos kimenetet:

<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 4 main(){ 5 6     putenv( "USER=senki" ); 7     putenv( "HOSTNAME=nekeresd.hu" ); 8 9     printf( "E-mail cím: '%s@%s'\n", 10            getenv("USER"), 11            getenv("HOSTNAME") ); 12 }</pre>	kornyval/putenv.c
--	-------------------

A program a 6–7. sorban beállítja a környezeti változók értékét, majd azokat a 9–11. sorban kiírja a szabványos kimenetre. Természetesen a beállított értékeket láthatjuk a kimeneten:

```

Bash$ ./putenv
E-mail cím: 'senki@nekeresd.hu'
Bash$ echo $USER@$HOSTNAME
senki@nekeresd.hu
Bash$
```

Amint a példából is láthatjuk, a program futása után az általa módosított értékek már nem elérhetők, a programot indító folyamat – esetünkben a héj – saját környezeti változókkal rendelkezik.

## Memóriafoglalás

Az alkalmazások, a futó folyamatok a Linux rendszertől memóriát igényelhetnek, amely azt – ha lehetséges – biztosítja a program számára. A Linux pontos nyilvántartást vezet arról, hogy az egyes folyamatok mennyi memóriát foglaltak, illetve hogy a memória egyes területeit melyik folyamatnak osztotta ki.

A rendszer a számítógép processzorában található védőáramkörök segítségével képes megvédeni az egyes folyamatoknak kiosztott memóriaterületet. minden folyamatnak csak azokhoz a memóriaterületekhez van hozzáférési jog, amelyeket a folyamat igényelt és kapott a rendszertől.

A folyamatok a rendszertől igényelt memóriaterületről lemondhatnak, azt felszabadíthatják. Ha a folyamat kilép, de a memóriaterületet nem szabadítja fel, a rendszer automatikusan felszabadítja azt, hogy kioszthassa más folyamatok számára.

A rendszer a memóriaterületeket címük alapján tartja nyilván. Programjainknak maguknak kell gondoskodniuk arról, hogy a foglalt memóriaterületek címét és méretét nyilvántartsák és megfelelőképpen kezeljék. Helytelen memóriakezelés a program megszakítását vonja maga után.

Memóriafoglalásra, a foglalt memória felszabadítására a következő – az `stdlib.h` fejállományban létrehozott – függvények használhatók:

```
void *malloc(size_t méret);
```

A függvény adott méretű memóriaterületet foglal le a folyamat kizártlagos használatára. A memóriafoglalás természetesen csak akkor lesz sikeres, ha a kért memóriamennyiség rendelkezésre áll.

A függvény paramétere a lefoglalni kívánt memóriaterület nagysága.

A lefoglalt memória tartalma ismeretlen, azt az operációs rendszer nem feltétlenül törli.

A Linux nyilvántartást vezet a memóriafoglalásokról; minden memóriaterületre csak az a folyamat írhat, amelyik a memóriaterületet lefoglalta. Az idegen memóriaterületre író vagy onnan olvasó folyamatot az operációs rendszer megszakítja, még mielőtt az írás vagy az olvasás megtörtént volna.

Ha olyan memóriaterületre van szükségünk, amelyet több folyamat is kezel, akkor osztott memóriát (*shared memory*) kell foglalnunk. A megosztott memória kezeléséről a 227. oldalon olvashatunk bővebben.

A folyamat futása után az operációs rendszer automatikusan felszabadítja az általa lefoglalt memóriát, ha azt a folyamat nem tette volna meg, így a számítógép operatív memóriájában nem maradhatnak „elveszett” területek.

A függvény visszatérési értéke a lefoglalt memóriaterület elejét jelöli mutató, ha a művelet sikeres volt, és `NULL` érték, ha a kért méretű memóriaterület nem áll rendelkezésre.

```
void free(void *memória);
```

A függvény a `malloc()` segítségével lefoglalt memóriaterület felszabadítására használható. A memóriaterületet felszabadítása után a folyamat többé már nem használhatja.

A függvény első paramétere a lefoglalt memória elejét kijelölő mutató, az a mutató, amelyet a `malloc()` adott vissza a memóriaterület lefoglalásakor.

Minden folyamat csak az általa lefoglalt memóriát szabadíthatja fel.

Ha a folyamat kilépés előtt nem szabadítja fel az általa lefoglalt memóriát, a Linux felszabadítja azt a kilépése után.

```
void *realloc(void *memória, size_t méret);
```

A függvény az előzőleg lefoglalt memóriaterület méretének megváltoztatására szolgál, általában akkor használjuk, ha növelni szeretnénk a lefoglalt memóriaterület nagyságát.

A függvény első paramétere az előzőleg lefoglalt memória elejére mutat, a második pedig az új memóriaméret.

Előfordulhat, hogy az általunk lefoglalt memóriaterület mellett nincs szabad terület. Ilyenkor a `realloc()` egy másik memóriaterületet foglal az új mérettel és a régi memóriaterület tartalmát átmásolja az új helyre. Mindenképpen a `realloc()` által visszaadott új mutatót kell tehát használnunk a továbbiakban!

A függvény visszatérési értéke a lefoglalt memória elejére mutat, ha a memóriafoglalás sikeres volt, más esetben pedig `NULL`. A visszatérési értékként jelölt memóriaterület a `malloc()` függvényel lefoglalt memóriaterülettel egyenértékű.

```
void *calloc(size_t darab, size_t méret);
```

A függvény memóriaterület lefoglalására szolgál, csakúgy mint a `malloc()`.

A függvény első és második paraméterének értékét összeszorozza, így kapja meg a lefoglalandó memória méretét.

A `calloc()` a lefoglalt memóriát nullázza, teljes egészében 0 bájtokkal tölti fel.

A függvény visszatérési értéke a lefoglalt memóriaterületre mutat, ha a művelet sikeres volt, ellenkező esetben `NULL` érték. A visszatérési értékként jelölt memóriaterület a `malloc()` függvényel lefoglalt memóriaterülettel egyenértékű.

**10. példa.** A következő példaprogram két karakterláncot másol egymás után az általa lefoglalt memóriaterületre.

```

1  #include <stdio.h>                               memoria/malloc1.c
2  #include <stdlib.h>
3  #include <string.h>
4
5  void *foglal( const char *nev, size_t meret ){
6      void     *memoria;
7
8      memoria=malloc( meret );
9      if( memoria==NULL ){
10         fprintf( stderr, "Nincs elegendő memória!\n" );
11         exit(1);
12     }
13     #ifdef DEBUG
14     fprintf( "Memóriafoglalás: %s: %d", nev, meret );
15     #endif
16 }/*foglal*/
17
18 char *osszefuz( const char *a, const char *b ){
19     char     *eredm;
20     int      hossza, hosszb;
21     hossza = strlen(a);
22     hosszb = strlen(b);
23
24     eredm=foglal( __func__, hossza+hosszb+1 );
25     strcpy( eredm, a );
26     strcpy( &erdem[hossza], b );
27     return( eredm );
28 }/*osszefuz*/
29
30 main(){
31     printf( "%s\n", osszefuz("Hello", " virág!") );
32 }/*main*/

```

Figyeljük meg, hogy a memóriafoglaláshoz a függvény átadja a `__func__` értéket, amelynek helyére a C fordító a függvény nevét jelző mutatót helyezi. Az átadott függvénynevet a program kiírja a szabványos kimenetre, ha a `DEBUG` változót az előfeldolgozó számára létrehozzuk. Ehhez a következő sort kell a program elején elhelyeznünk:

```
1  #define DEBUG
```

Ez a módszer általánosan elterjedt a programok tesztelésére. Különösen előnyös, hogy a tesztelőüzeneteket létrehozó utasítások nem kerülnek bele a programba, ha a

*DEBUG előfeldolgozó változó nincs létrehozva, hiszen így a program kisebb lesz.*

## Műveletek a memóriában

A következő néhány függvény segítségével a memóriában – annak tartalmától, a tárolt adatok formátumától függetlenül – különféle műveleteket végezhetünk el. A függvények használatához a `string.h` fejállomány betöltése szükséges.

```
void *memcpy(void *cél, const void *forrás, size_t darab);
```

A függvény segítségével memóriaterület tartalmát másolhatjuk át.

A függvény a második paraméterrel jelölt memóriaterületről másol az első paraméterrel jelölt memóriaterületre a harmadik paraméterként átadott darabszámú bájtot.

A függvény használatakor a forrás és a célterület nem lapolódhat át.

A függvény visszatérési értéke a célterületre mutat, pontosan megegyezik a függvény első paraméterével.

```
void *memmove(void *cél, const void *forrás, size_t darab);
```

A függvény segítségével memóriaterület tartalmát másolhatjuk át.

A függvény a második paraméterrel jelölt memóriaterületet másolja át az első paraméterrel jelölt memóriaterületre. A függvény pontosan annyi bájtot másol át, amennyi a harmadik paraméter értéke.

A függvény használatakor a memóriaterületek átlapolódhatnak.

A függvény visszatérési értéke a célterületet jelöli a memóriában.

```
void *memccpy(void *cél, const void *forrás, int keres, size_t  
darab);
```

A függvény segítségével memóriaterület tartalmát másolhatjuk át adott méret vagy határolójel alapján.

A függvény a második paraméterrel jelölt memóriaterületről másol az első paraméterrel jelölt memóriaterületre. A függvény a másolást addig a pontig végzi, ahol a harmadik paraméterként átadott bájt található vagy ahol a másolt bájtok száma eléri a negyedik paraméterként átadott méretet.

Ha a másolás a harmadik paraméterként átadott bájt megtalálása miatt állt le, a függvény által visszaadott mutató az utolsó másolt bájt utáni helyre mutat, ha a méretkorlát miatt, `NULL` érték.

```
void *memset(void *kezdet, int tölt, size_t méret);
```

A függvény segítségével memóriaterületet tölthatunk fel adott értékkel.

A függvény az első paraméterrel jelölt memóriaterületet tölti fel a második paraméterrel jelölt bájttal. A függvény a feltöltést addig végzi, amíg az írt bajtok száma el nem éri a harmadik paraméterként átadott értéket.

A függvény visszatérési értéke a feltöltött területre mutat, vagyis megegyezik a függvény első paraméterével.

```
int memcmp(const void *a, const void *b, size_t méret);
```

A függvény segítségével memóriaterületeket hasonlíthatunk össze.

A függvény az első paraméterrel jelölt memóriaterületet hasonlítja össze a második paraméterrel jelölt memóriaterülettel. A függvény az összehasonlítást addig végzi, amíg az összehasonlított bajtok száma el nem éri a harmadik paraméterként átadott értéket.

Ha a két memóriaterület tartalma megegyezik, akkor a visszatérési érték 0, ha nem, akkor az első különböző bajtnál számított  $a_n - b_n$  különbségnek megfelelő az előjele, azaz negatív, ha a második memóriaterületen van nagyobb érték és pozitív, ha az elsőn.

```
void *memmem(const void *memória, size_t mhossz, const void
              *tartalom, size_t thossz);
```

A függvény segítségével memóriában kereshetünk meg adott memóriatartalmat.

A függvény az első paraméterrel jelzett memóriaterületen keresi a harmadik paraméterrel jelölt tartalmat. A függvény második paramétere adja meg, hogy mekkora területen kívánunk keresni, a negyedik pedig, hogy a keresett memóriatartalom milyen hosszú.

A függvény visszatérési értéke az átvizsgált memóriaterületen belül jelöli a megtalált memóriatartalom elejét, és NULL, ha a memóriatartalom nem található.

```
void *memchr(const void *memória, char bájt, size_t méret);
```

A függvény segítségével a memóriában kereshetünk meg egy adott értékű bajtot.

A függvény az első paraméterrel jelölt memóriaterületen keresi a második paraméterként átadott értéket. A függvény a keresést a harmadik paraméterként átadott méretig végzi.

A függvény visszatérési értéke a keresett értéket jelöli a memóriában, és NULL, ha a keresés eredménytelen volt.

**11. példa.** A következő példaprogram a `memset()` függvényt használja egy lebegő-pontos számokat tartalmazó tömb elemeinek nullára állítására.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  main(){
6      long double *szam;
7      int          n;
8
9      szam=malloc( sizeof(long double)*10 );
10     memset( szam, 0, sizeof(szam)*10 );
11
12     for( n=0; n<10; ++n )
13         printf( "szam[%d]=%lf\n", n, szam[n] );
14 }/*main*/

```

memoria/memset.c

A program futtatásakor kiírja a szabványos kimenetre a tömbelemek értékét a következő formában:

```

Bash$ ./memset
szam[0]=0.000000
szam[1]=0.000000
...
Bash$

```

## Csatornák megnyitása és lezárása

A csatornákat (*stream*, csatorna) a Unix rendszereken igen sokszor használjuk, ezért mindenkorban meg kell ismerkednünk azzal, hogyan tudjuk őket a programjainkban kezelni.

A csatorna nem más, mint szöveges (ASCII) adatok fogadására vagy küldésére képes adatnyelő vagy adatforrás. A programok számára elérhető szabványos bemenet, szabványos kimenet és szabványos hibacsatorna is ilyen eszköz, de a program saját csatornákat is nyithat az állományrendszerben található állományok felé.

A program, amikor elindítjuk, három csatornát megnyitott állapotban vesz át a rendszertől. Ezekre a C nyelven megírt programban az `stdin` (szabványos bemenet), `stdout` (szabványos kimenet) és `stderr` (szabványos hibacsatorna) szavakkal hivatkozhatunk, ha az `stdio.h` fejállományt betöljtük. Ha állományokat akarunk csatornaként kezelni, a használat előtt meg kell nyitnunk őket.

A következő függvények állományok csatornaként való megnyitására és zárasára szolgálnak. A függvények használatához az stdio.h betöltése szükséges.

```
FILE *fopen(const char *fájlnév, const char mód);
```

Állományokat csatorna (*stream*) módban megnyitni ezzel a függvényel lehet.

Az első paraméter a megnyitandó állomány nevét jelölő mutató, a második a megnyitás módját jelzi. A mód a következők egyike lehet:

"r" Az állomány megnyitása csak olvasható módban.

"w" Az állomány megnyitása csak írható módban. Ha az állomány már létezett, a tartalma elvész.

"a" Az állomány megnyitása hozzáfűzés módban. Ilyenkor a csatornába írt karakterek az állomány végéhez fűződnek.

"r+" Az állomány megnyitása olvasható-írható módban. Ha az állomány már tartalmaz adatokat, a tartalma megmarad, az olvasás-írás helye pedig az első karakter elő kerül.

"w+" Az állomány megnyitása olvasható-írható módban. Ha az állomány már létezik, a tartalma elvész.

"a+" Az állomány megnyitása olvasható-írható módban. Ha az állomány már tartalmaz karaktereket, a tartalma megőrződik.

Kezdetben az olvasási mutató az állomány elejére kerül, a csatornába írt karakterek pedig mindenkor az állomány végéhez fűződnek.

A függvény által visszaadott érték egy mutató, amelynek segítségével a megnyitás után a csatornát használni tudjuk. Ha a megnyitás nem sikerült, a visszaadott érték NULL lesz. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

EINVAL A függvény második paramétereinek átadott megnyitási mód nem megengedett értéket tartalmazott.

ENOMEM A művelet végrehajtásához nem állt rendelkezésre elegendő memória.

EACCES Az állomány megnyitásához vagy a létrehozásához a folyamatnak nincs jog.

EINTR A művelet végrehajtását egy jelzés megszakította.

EISDIR Az első paraméterként jelölt név egy könyvtár neve, amelyet nem lehet írásra megnyitni az fopen() segítségével.

EMFILE A folyamat túl sok állományt tart nyitva, a beállított korlát miatt több állományt már nem nyithat meg.

ENOENT A megnyitandó állomány nem létezik.

**ENOSPC** Az új állomány nem hozható létre, mert a háttértáron már nincs felhasználható üres terület.

**EROFS** A kérő művelet nem végezhető el, mert az állományrendszer, ahol az állomány található – vagy ahol létre akarjuk hozni –, csak olvasható módon van a rendszerbe illesztve.

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

A használat után a csatornákat ezzel a függvényel kell lezárnunk. A függvény visszatérési értéke 0, ha a lezárás sikeres volt, és EOF, ha hiba lépett fel.

```
int fcloseall(void);
```

Ez a függvény lezárja az összes csatornát. Ügyelnünk kell arra, hogy a hívása lezárja a szabványos bemeneti, kimeneti és hibacsatornát is.

A függvény visszatérési értéke minden 0.

**12. példa.** Az itt látható példaprogram megnyitni egy állományt, és ha ez nem sikerül, hibaüzenetben értesíti a felhasználót:

	csatornak/fopen.c
1	#include <stdio.h>
2	#include <stdlib.h>
3	#include <errno.h>
4	
5	void hiba( int hiba ) {
6	char *uzenet;
7	
8	switch( hiba ) {
9	case EINVAL:
10	uzenet="A megnyitási mód érvénytelentl.";
11	break;
12	case ENOMEM:
13	uzenet="Nincs elég memória";
14	break;
15	case EACCES:
16	uzenet="Hozzáférés megtagadva.";
17	break;
18	case EINTR:
19	uzenet="A művelet üzenet miatt megszakadt";
20	break;
21	case EISDIR:
22	uzenet="A könyvtárbejegyzés egy könyvtár";
23	break;
24	case EMFILE:
25	uzenet="Túl sok állomány van nyitva";

```

26     break;
27 case ENOENT:
28     uzenet="Az állomány nem létezik";
29     break;
30 case ENOSPC:
31     uzenet="Nincs hely a meghajtón.";
32     break;
33 case EROFS:
34     uzenet="Csak olvasható állományrendszer.";
35     break;
36 default:
37     uzenet="Ismeretlen hiba.";
38 /*switch*/
39 fprintf( stderr, "Hiba: %s\n", uzenet );
40 exit(hiba);
41 /*hiba*/

42 int main( int argc, char *argv[] ){
43 FILE *teszt;
44
45 teszt=fopen( argv[1], "r" );
46 if( teszt==NULL )
47     hiba( errno );
48 fclose(teszt);
49 return(0);
50 /*main*/
51

```

A példaprogram az üzenetet a szabványos hibacsatornára küldi. Ehhez az `fprintf()` függvényt használja, amelyről bővebben a 119. oldalon olvashatunk.

A következő néhány függvény segítségével a már megnyitott csatornákba írhatunk vagy onnan olvashatunk. A függvények lehetővé teszik karakterek és karakterláncok írását és olvasását is.

Megfigyelhető, hogy ezek a függvények a karakterek – bájtok – átadására az `int` típust használják. Ez a szokás a C programozási nyelvben azért terjedt el, mert így nem csak a karakterek kódolására van mód, hanem egyéb értékeket is elhelyezhetünk a változóban. Ilyen érték az állomány végének kódolására használt EOF (*end of file*, állomány vége) jel is.

A karakterláncokat kezelő függvényeknél megfigyelhető, hogy a kiírás során a karakterláncot lezáró 0 karaktert nem írják a csatornába. Ez azért van így, mert a csatornában és állományokban a karakterláncok lezárására nem használjuk a 0 karaktert. A csatornák olvasásakor általában az újsor karaktert tekintjük a karakterlánc végének, de a beolvasott karakterlánc végén elhelyezzük a 0 karaktert, mert a memóriában a karakterláncok végét 0 karakterrel jelöljük.

Az itt felsorolt függvények használatához az `stdio.h` fejállomány betöltése szükséges.

```
int fputc(int c, FILE *csatorna);
```

A függvény segítségével egy karaktert írhatunk a már megnyitott csatornába.

E függvény karakteres típusúra alakítja az első paraméterként átadott értéket és a második paraméterrel jelölt megnyitott csatornába küldi.

A függvény visszatérési értéke a kiírt karakternek megfelelő egész – azaz az első paraméter –, ha nem lépett fel hiba, különben pedig EOF.

```
int putc(int c, FILE *csatorna);
```

E függvény működése megegyezik az `fputc()` függvénnyel, a különbség csak annyi, hogy ezt a gyorsabb működés érdekében makróként valósították meg.

```
int putchar(int c);
```

A függvény a szabványos kimenetre irányított kimenetű `putc()` függvénnyel egyenértékű.

```
int fputs(const char *szöveg, FILE *csatorna);
```

A függvény segítségével karakterláncot írhatunk a már megnyitott csatornába.

A függvény az első paraméterrel jelölt karakterláncot a második paraméterrel jelölt csatornába írja. A függvény újsor karaktert nem ír a szöveg végére, a karakterlánc végét jelölő 0 karaktert pedig nem írja ki a csatornába.

A függvény visszatérési értéke nem negatív egész szám, ha a művelet sikeres volt, különben pedig EOF.

```
int puts(const char *szöveg);
```

A függvény segítségével karakterláncot írhatunk a szabványos kimenetre.

A függvény az első paraméterrel jelölt karakterláncot a szabványos kimenetre írja. A függvény a karakterlánc kiírásakor a lezáró 0 karaktert nem írja a csatornába, a karakterlánc után azonban egy újsor karaktert kiír.

A függvény visszatérési értéke nem negatív egész, ha a művelet során nem lépett fel hiba, különben pedig EOF.

```
int fgetc(FILE *csatorna);
```

Ez a függvény a következő karaktert olvassa a csatornából és azt egész szám-ként adja vissza.

A függvény visszatérési értéke az olvasott karakternek megfelelő egész, illetve EOF, ha hiba történt az olvasáskor. A függvény szintén EOF értéket ad vissza, ha az olvasás során már előzőleg eljutottunk a csatorna végére, onnan több adat már nem olvasható.

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

Ez a függvény ugyanúgy működik mint az fgetc(), azzal a különbséggel, hogy makróként valósították meg a gyorsabb működés érdekében.

```
int getchar(void);
```

Ez a függvény működésében megegyezik a getc függvénnyel, de a szabványos bemenetet olvassa.

```
ssize_t getline(char **sor, size_t *n, FILE *csatorna);
```

A függvény egy teljes sort olvas be a csatornából és elhelyezi azt a memóriában. Az olvasás során a függvény elhelyezi az újsor karaktert is a memóriaterületen, és ha a sor nem volna 0 karakterrel lezárva, lezárja azt. Mivel a getline() jelenlétéét nem írják elő a szabványok, a GNU C könyvtár dokumentációja szerint szükség lehet a \_GNU\_SOURCE makró létrehozására az stdio.h fejállomány betöltése előtt.

A függvény az olvasott karakterláncot az első paraméterként átadott mutató követésével kapott mutató által jelölt memóriaterületre helyezi.

Ha a második paraméter által jelzett változóban lévő méretnél hosszabb volna a sor, akkor a realloc függvénnyel a szükséges memóriaterületet lefoglalja. Fontos, hogy a függvény az újonnan lefoglalt memóriaterület méretét visszaírja ebbe a változóba, hogy a következő függvényhíváskor (a következő sor olvasásakor) tudjuk, mekkora területet tartunk fenn az olvasott sor számára.

Ha a \*sor változó – az első paraméterként átadott mutató követésével kapott mutató – értéke NULL, a függvény a megfelelő méretű memóriát a malloc() függvénnyel lefoglalja. Az első függvényhívás előtt tehát nem feltétlenül kell helyet foglalnunk az olvasott sor számára.

A függvény visszatérési értéke az olvasott karakterek számát adja meg. Ha az olvasott sor nem volt lezárva 0 karakterrel, akkor az olvasott karakterek számába nem számít bele a függvény által a lezárásra használt 0 karakter. Ha az olvasás nem sikerült – mert például a csővezeték utolsó sorát is elolvastuk már –, a visszaadott érték -1. (A visszatérési érték azért nem size\_t, hogy a -1 értéket is képes legyen hordozni.)

A függvény használatát a 116. oldalon található 15. példa mutatja be.

```
ssize_t getdelim(char **sor, size_t *n, int lezáró, FILE
                  *csatorna);
```

Ez a függvény működésében megegyezik a getline() függvénnyel, azzal a különbséggel, hogy a sor végének jelzésére nem feltétlenül az újsor karaktert használja, hanem a harmadik paraméterben kapott karaktert.

```
char *fgets(char *memória, int méret, FILE *csatorna);
```

A függvény segítségével egy sort olvashatunk be a már nyitott csatornából.

A függvény első paramétere a memóriaterületet jelöli, ahol a függvény az olvasott sort fogja helyezni. A függvény harmadik paramétere a csatornát jelöli, amelyből a függvény a következő sort be fogja olvasni.

A függvény legfeljebb a második paraméterként átadott számnak megfelelő mennyiségi bájtot fog felhasználni. Ennél a számnál eggyel kevesebb bájtot tud beolvasni sorként, hiszen az olvasott sor után el kell helyeznie a memóriában a karakterláncot lezáró 0 karaktert.

A függvény által visszaadott érték a sor elhelyezésére használt memóriaterületet jelöli a memóriában, ha a művelet sikeres volt, illetve NULL, ha hiba lépett fel. A függvény akkor is a NULL értéket adja vissza, ha a csatorna végére értünk és onnan már több karakter nem olvasható.

```
int ungetc(int karakter, FILE *csatorna);
```

A függvény arra szolgál, hogy az éppen olvasott karaktert a csatornába visszahelyezzük. Ha a csatorna egy állományba mutat, a függvény nem változtatja meg az állomány tartalmát, de a legközelebbi olvasás során a visszahelyezett karakter olvasható.

A függvény első paramétere a visszahelyezendő karakter, a második pedig a csatornát jelöli, ahol a karaktert vissza szeretnénk helyezni. Az ungetc() függvénnyel visszahelyezett karakterek nem feltétlenül kell annak lenni, amelyet a legutolsóként olvastunk a csatornából.

Fontos tudnunk, hogy az ungetc() függvényt nem lehet kétszer meghívni egymás után, a két hívás közben legalább egy, olvasást végező függvényhívást kell elvégezni.

A függvény visszatérési értéke a visszahelyezett karakter, ha a visszahelyezés sikeres volt, különben pedig EOF.

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

Ez a függvény a csatorna gyorsítómemoriáját üríti; az összes írásra kijelölt bájtot kiírja a csatornába.

A függvény paramétere a kiírni kívánt csatornát jelöli. Ha a paraméter értéke NULL, a folyamat által birtokolt összes csatornával elvégzi ezt a műveletet.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és EOF hiba esetén.

A függvény használatát mutatja be a 17. példa a 118. oldalon.

```
int feof(FILE *csatorna);
```

A függvény segítségével megállapíthatjuk, hogy a csatorna végéhez értünk-e. A függvény paramétere a vizsgálni kívánt csatornát jelzi. A függvény visszatérési értéke nem 0, ha a csatorna végéhez érkeztünk.

**13. példa.** A következő példaprogram a számok kiszűrésére használható. A betűket, írásjeleket átengedi, de a számjegyek helyére a # karaktert nyomtatja.

```
1  #include <stdio.h>
2
3  main(){
4      int betu;
5      while( (betu=getchar()) != EOF )
6          if( !isdigit(betu) )
7              putchar( betu );
8          else
9              putchar( '#' );
10     }/*main*/
```

A programban megfigyelhetjük, hogyan készíthetünk egyszerűen szűrőt, olyan programot, amely a szabványos bemenetet olvassa és a szabványos kimenetre írja az olvasott adatok módosított változatát.

A program a szabványos bemenetet karakterenként olvassa (5. sor) és karakterenként írja a szabványos kimenetet (7., 9. sor).

**14. példa.** A következő példaprogram a parancssorban kapott összes állományt megnyitja és a bennük található sorokat a szabványos kimenetre írja.

```
1  #include <stdio.h>
2
3  void listaz( FILE *allomany ){
4      char sor[255];
5      while( fgets(sor, 255, allomany)!=NULL )
6          /* A puts() újsor karaktert írna a sorok
7             * végére! */
8          fputs(sor, stdout);
9     }/*listaz*/
10
11  int main( char argc, char *argv[] ){
12      int fileok;
13      FILE *bemenofile;
14      int hibakod=0;
15
16      for( fileok=1; fileok<argc; ++fileok ){
17          bemenofile=fopen( argv[fileok], "r" );
18          if( bemenofile==NULL ){
19              fprintf( stderr,
20                  "A '%s' állományt nem lehet megnyitni. \n",
21              argv[fileok]
22          );
23      }
24  }
```

```

21     argv[fileok] );
22     hibakod++;
23 }else{
24     listaz( bemenofile );
25     fclose( bemenofile );
26 }/*if*/
27 }/*for*/
28 return(hibakod);
29 /*main*/

```

A program állomány megnyitásával létrehozott csővezetéket olvas soronként (5. sor) és a szabványos kimenetre írja az olvasott adatokat (8. sor), szintén soronként.

A 16. sorban kezdődő ciklus hatására a program minden állományt megnyit, amelynek nevét a parancssorban megkapta.

**15. példa.** A következő példaprogram a szabványos bemenetről csak azokat a sorokat másolja át a szabványos kimenetre, amelyekben a parancssorban megadott szövegrészlet megtalálható.

	csatornak/grep.c
--	------------------

```

1 #define _GNU_SOURCE
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int main( int argc, char *argv[] ){
7     char *sor;
8     size_t hossz=255;
9
10    sor=malloc( hossz );
11    if( sor==NULL )
12        exit(1);
13
14    while( getline(&sor, &hossz, stdin) != -1 )
15        if( strstr(sor, argv[1]) != NULL )
16            fputs( sor, stdout );
17
18    free(sor);
19 }/*main*/

```

A példaprogram a szabványos bemenet olvasására a 14. sorban a `getline()` függvényt használja, így a sorok hosszának csak a rendelkezésre álló memória szab határt.

Mivel a `getline()` a GNU C könyvtár bővítése – a szabványok nem írják elő a jelenlétéét –, a program első sorában létrehoztuk a `_GNU_SOURCE` makrót.

**16. példa.** A következő példaprogram megnyitja a parancssorban kapott állományt és az ott található számok átlagát a szabványos kimenetre írja.

```
1      #include <stdio.h>
2      #include <stdlib.h>
3
4      double atlag( FILE *allomany ){
5          char sor[255];
6          double darab=1;
7          double osszeg=0;
8
9          while( fgets(sor, 255, allomany)!=NULL ){
10              osszeg+=strtod(sor, NULL);
11              darab+=1;
12          }/*while*/
13          return(osszeg/darab);
14      }/*atlag*/
15
16      int main( char argc, char *argv[] ){
17          FILE *bemenofile;
18
19          int hibakod=0;
20
21          bemenofile=fopen( argv[1], "r" );
22          if( bemenofile==NULL ){
23              fprintf( stderr,
24                  "Nyitás hiba: '%s'.\n", argv[1] );
25              exit(1);
26          }else{
27              printf( "%f\n", atlag(bemenofile) );
28              fclose( bemenofile );
29          }/*if*/
30          return(hibakod);
31      }/*main*/
```

A program a 10. sorban hívja az strtod() függvényt, amelynek segítségével az állományból olvasott karakterláncot számmá alakítja. A számokat a program összegzi és megszámlálja, hogy kiszámíthassa a matematikai átlagot (13. sor).

Egyeszerűbb programok esetében a programhibák keresésére használhatjuk a printf() vagy az fprintf() függvényeket. A programban ezekkel a függvényekkel üzeneteket írunk a szabványos kimenetre vagy a szabványos hibacsatornára, így futtatás közben folyamatosan figyelhetjük, mi történik.

Vigyáznunk kell azonban az olyan programhibákkal, amelyek a program azonnali megszakítását eredményezik! Ha a Linux megszakítja a program futását, előfordulhat, hogy az átmeneti gyorsítótárban lévő üzenetek nem jelennek meg, így a felhasználó azt hiheti, hogy a program nem ért el azzhoz a részhez, ahol az üzenetet ki kell írnia. Ezt a jelenséget – és természetesen az ellene való védekezés módját – mutatja be a következő példa.

**17. példa.** A következő példaprogram bemutatja, hogyan helyezhetünk el olyan, nyomkövetésre alkalmas üzeneteket, amelyek biztosan megjelennek a kimeneten.

```

1 #include <stdio.h>
2 #define DEBUG
3
4 #ifdef DEBUG
5 void kiir( char *uzenet ){
6     fprintf( stderr, "Nyomkövetés: %s\n", uzenet );
7     fflush( stderr );
8 }/*kiir*/
9 #endif
10
11 int main( int argc, char *argv[] ){
12     int q;
13     int *tomb;
14
15 #ifdef DEBUG
16     kiir( "Eddig még jó..." );
17 #endif
18
19     for( q=0; q<10; q++ )
20         tomb[q]=100;
21
22     return(0);
23 }/*main*/

```

A program 2. sorában létrehozzuk a DEBUG makrót, aminek hatására az 5–8., valamint a 16. sorok bekerülnek a programban. Az 5–8. sorokban egy függvényt hozunk létre, amely a paraméterként jelölt karakterláncot kiírja a szabványos hibacsatornára, valamint a 7. sorban található fflush() hívással gondoskodik róla, hogy az azonnal meg is jelenjen.

A példaprogram egyébként hibás, mivel a 13. sorban létrehozott mutatót memóriaterület írására használja, anélkül, hogy értékét beállítaná.

## Formázott kimenet

A C könyvtár egyik meglehetősen összetett és sokat használt függvénycsaládja a printf() család, amely adatok formázott kiírását teszi lehetővé. A függvénycsalád tagjainak viselkedése majdnem teljesen megegyezik, különbség csak abban van, hogy a kiírás.

A printf() függvénycsalád használatához az stdio.h állomány betöltése szükséges.

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

A függvény formázott szövegkiírást tesz lehetővé a szabványos kimenetre.

A függvény első paramétere a kiírt szöveg formáját meghatározó formázószöveg, a további paraméterei pedig a kiírandó adatokat tartalmazzák. A formázószövegen használható kifejezések leírását a 120. oldalon kezdődő fejezetben találjuk.

A függvény által visszaadott érték a kinyomtatott karakterek száma, illetve negatív, ha valamilyen hiba miatt a művelet nem sikerült.

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

A függvény működésében teljesen megegyezik a printf függvényét, a különbség csak annyi, hogy a művelet tetszőleges csatorna felé irányul.

A függvény első paramétere a csatornát azonosítja, ahová a kiírás történik, további paraméterei és visszatérési értéke pedig megegyeznek a printf() függvény paramétereivel és visszatérési értékével.

```
int sprintf(char *memória, const char *format, ...);
```

A függvény a printf() függvényel megegyező módon működik, az írás azonban nem a szabványos kimenet felé történik, hanem az első paraméterként átadott mutató által meghatározott memóriaterületre.

A függvény az elhelyezett karaktersorozatot lezárja egy 0 karakterrel, amely a visszaadott értékbe – az elhelyezett karakterek számába – nem számít bele.

```
int snprintf(char *memória, size_t méret, const char *format,
             ...);
```

A függvény működésében megegyezik az sprintf() függvényét, azzal a különbséggel, hogy ez a függvény legfeljebb a második paraméterben meghatározott számú karaktert ír a memóriába.

## Kifejezések a formázószövegben

A formázott szövegkiíratásra használható függvények, mint például a `printf()` vagy az `fprintf()`, egy formázószöveget használnak az argumentumok értékének kiírására. A formázószövegben összetett kifejezéseket helyezhetünk el, melyekkel igen rugalmasan határozzuk meg, milyen formában szeretnénk az értékeket kiíratni.

A formázószövegben minden % jellel kezdődő rész egy-egy argumentum értékének kiírására ad utasítást. Ezek a részek a legegyszerűbb esetben csak két betűből állnak, amelyek a következők:

- %d Az argumentum előjeles egész típusú, amelynek kiírása tízes számrendszerben történik. Az ilyen kifejezések nem alkalmazak lebegőpontos számok kezelésére, de minden előjeles egész típust formáthatunk a segítségükkel.
- %u Az argumentum előjel nélküli egész típusú, amelyet tízes számrendszerben íratunk ki. Az ilyen kifejezések minden előjel nélküli egész típus kezelésére alkalmazak.
- %o Az argumentum előjel nélküli egész típus, amelyet nyolcas számrendszerben írunk ki. Az ilyen kifejezések csak előjel nélküli egész számok kezelésére alkalmazak, de azok minden típusa esetén használhatjuk őket.
- %x Az argumentum előjel nélküli egész típus, amelyet tizenhatos számrendszerben íratunk ki. A számjegyek mellett ekkor az abcdef karaktereket is használja a rendszer a szám kiírásakor.
- %X Az argumentum előjel nélküli egész típus, amelyet tizenhatos számrendszerben íratunk ki. A számjegyek mellett ekkor az ABCDEF karaktereket használjuk a szám kiírásakor.
- %f Az argumentum lebegőpontos szám, amelyet tízes számrendszerben íratunk ki. Előbb az egészket, majd a tizedespontot, végül pedig a tizedesjegyeket.
- %e Az argumentum lebegőpontos szám, amelyet tízes számrendszerben, normál alakban íratunk ki. Elöl egy lebegőpontos szám (mantissa) található, amelyet az e betű és a 10 hatvánnyal kiegészítő (exponens) követ.
- %E Ugyanaz mint az %e, de a mantissát és az exponenst az E betű választja el egymástól.
- %g Az argumentum lebegőpontos szám, amelyet tízes számrendszerben íratunk ki %f vagy %e formában, attól függően, hogy az adott érték esetén melyik adja a rövidebb írásmódot.
- %G Az argumentum lebegőpontos szám, amelyet tízes számrendszerben íratunk ki %f vagy %E formában, attól függően, hogy melyik adja a rövidebb írásképet.

%c Az argumentum egyetlen karakterként íródik ki, az ASCII kódtáblának megfelelően. A kiíráskor az argumentumnak csak az alsó 8 bitje lesz figyelembe véve.

%s Az argumentum egy egydimenziós char típusú tömb nulladik elemét jelző mutató. A tömb szövegláncként lesz kiírva az első 0 értékű bájtig.

Óvatosan kell eljárnunk ezzel a kifejezéssel, hiszen használatával a függvények addig olvassák a memóriaterületet, amíg 0 értéket nem találnak. Ha a tömb nincs lezársa ilyen értékkel, könnyen előfordulhat, hogy a függvény túl sok karaktert olvas és így idegen memóriaterületre téved. Ez a program futásának megszakítását eredményezheti.

%p Az argumentum egy mutató, amelynek értéke tizenhatos számrendszerben íródik ki.

%m Az `errno` – a legutóbbi hiba jellegét leíró szám – értékének megfelelő szöveges leírás íródik ki. Ehhez a kifejezéshez nem tartozik paraméter.

%% A % karakter kiírására szolgáló rövidítés.

A két karakterből álló kifejezések a formázószövegben meghatározzák azt, hogy az adott argumentumot milyen formátumban írjuk ki, de az argumentum típusát további betűkkel finomíthatjuk. Bizonyos esetekben – az argumentumok bizonyos típusa esetén – finomítanunk is kell a formázáson, ellenkező esetben a kiírt érték hibás lesz, hiszen az argumentum értékét a kiírást végző függvény hibásan értelmezi.

A két betűből álló formázókifejezésekben, a % jel után a következő kiegészítőket helyezhetjük el:

~~hh~~ Az argumentum `char`, amely lehet előjeles (`signed char`) vagy előjel nélküli (`unsigned char`) értelmezésű is.

h Az argumentum `short int` vagy `unsigned short int` típusú.

j Az argumentum `intmax_t` vagy `uintmax_t` típusú.

l Az argumentum `long int` vagy `unsigned long int` típusú.

ll Az argumentum `long long int` vagy `unsigned long long int` típusú.

z Az argumentum `size_t` típusú.

L Az argumentum `long double` típusú.

A paraméter típusát meghatározó kiegészítőkön kívül elhelyezhetünk olyan kiegészítőket is, amelyek a kiírt értékek formáját határozzák meg. Ezek a kiegészítők a következők lehetnek:

n A % jel után elhelyezett szám az érték számára fenntartott hely méretét, a számára fenntartott karakterek számát jelöli.

n.m A lebegőpontos számok kiírásakor két számot is megadhatunk ponttal elválasztva. Az első szám ilyenkor a kiírt érték számára fenntartott karakterek számát jelöli, a második pedig azt, hogy hány tizedes értéket – milyen pontosággal – szeretnénk a számot kiírni.

- A paraméter kiíratásakor az oszlopon belül balra igazítást használunk.
- + Az előjel helyén a + jel megjelenik, ha a szám pozitív.

**szóköz** Ha kiírt érték nem kezdődik + vagy - karakterrel, egy szóközt helyez el előtte a függvény.

# A kiírt számok számrendszerének jelzését írja elő. A következő jelek jelzik a szám számrendszerét:

- A %o formátumkifejezés (nyolcas számrendszer) esetén a kiírt szám 0 karakterrel kezdődik.
  - A %x és a %X formátumkifejezés (tizenhatos számrendszer) esetén a kiírt szám 0x, illetve 0X jelekkel kezdődik.
  - Lebegőpontos kifejezés kiírása esetén a kiírt kifejezés mindenképpen tartalmazni fog tizedespontot. Ez nem a kiírt kifejezés elején van, de ez is a számformátum felismerését könnyíti meg.
  - Normálalak kiírása esetén a mantissa mindenképpen tartalmazni fogja a tizedesjelet a szám könnyebb felismerése érdekében.
- ’ A számjegyek csoportokba rendezése a könnyebb olvashatóság érdekében.

0 A kiírt szám a rendezéshez nem szóközökkel, hanem 0 karakterrel lesz feltöltve.

**18. példa.** A következő programrészlet egy kétoszlopos táblázatot ír a szabványos kimenetre a printf() segítségével.

```

1 main(){
2     int i;
3     init();
4     for( i=0; i<=5; i++ )
5         printf( "%-8s  %.4f\n", valtozok[i].nev, valtozok[i].ertek );
6 }
```

A program a következő táblázatot készítette:

```
Bash$ ./a.out
E          2.7183
LOG2E      1.4427
LOG10E     0.4343
LN2        0.6931
LN10       2.3026
PI         3.1416
Bash$
```

Amint látjuk, a szöveges értékeket tartalmazó első oszlop bal oldalra rendezett formában tartalmazza az értékeket. A balra rendezett forma és az oszlop állandó szélessége a program 5. sorában található %-8s kifejezésnek köszönhető.

**19. példa.** A következő példaprogram két állományt nyit meg. Az egyikból olvassa az ott található számokat, a másikba pedig írja azokat a szinuszértékkel együtt.

A program a parancssorban kapja az olvasandó és írandó állományok nevét.

	csatornak/ketallomany.c
--	-------------------------

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 void szinusz( FILE *bemenet, FILE *kimenet ){
6     char sor[255];
7     double darab=1;
8     double szam;
9
10    while( fgets(sor, 255, bemenet)!=NULL ){
11        szam=strtod(sor, NULL);
12        fprintf(kimenet, "%8.4f\t%8.4f\n",
13                  szam, sin(szam) );
14    /*while*/
15 }/*szinusz*/
16
17 int main( char argc, char *argv[] ){
18     FILE *bemenofile;
19     FILE *kimenofile;
20
21     bemenofile=fopen( argv[1], "r" );
22     kimenofile=fopen( argv[2], "w" );
23
24     if( bemenofile==NULL ||
25         kimenofile==NULL ){
26         fprintf( stderr, "Nyitás hiba. \n" );
27         exit(1);

```

```

28 }else{
29     szinusz( bemenofile, kimenofile );
30     fclose( kimenofile );
31     fclose( bemenofile );
32 }/*if*/
33 return(0);
34 /*main*/

```

A program a következő formájú kimenetet hozza létre:

1	0.0000	0.0000
2	1.7800	0.9782
3	12.2000	-0.3582
4	6.0000	-0.2794
5	6.3000	0.0168

A program a kimenetén rendezett számoszlopokat hozott létre, amelyeket könnyen olvashatunk. A rendezést a formázószöveg garantálja.

## Formázott bemenet

A formázott kimenet előállítására használt printf() függvénycsaládhoz hasonlóan rendelkezésünkre áll a scanf() függvénycsalád a formázott bemenet kezelésére. A printf() családdal ellentétben azonban ezeket a függvényeket meglehetősen ritkán szokás használni.

A scanf() függvénycsalád használatához az stdio.h fejállományt be kell töltenünk.

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

A függvény formázott bemenetet valósít meg a szabványos bemenetről.

A függvény első paramétere a beolvasott adatok formáját leíró szöveges értéket jelölő mutató. A további paraméterek mutatók, amelyek kijelölik, hogy a beolvasott adatokat hol kell elhelyeznie a függvénynek.

A függvény a szabványos bemenetről olvasott adatokat a formaszövegnek megfelelően részekre vágja és az egyes részeket elhelyezi a megfelelő paraméterek által memóriaterületekre.

A függvény visszatérési értéke a beolvasott adatok részeinek száma, vagyis az, hogy hány memóriaterületre sikerült adatokat helyeznie. Ha ez kevesebb, mint a várt, a beolvasott adatok formátuma nem volt megfelelő. A visszatérési érték EOF, ha a szabványos bemenet nem olvasható.

```
int fscanf(FILE *csatorna, const char *formaszöveg, ...);
```

A függvény formázott bemenetet valósít meg csatornából.

A függvény első paramétere a csatornát azonosítja, ahonnan az adatokat olvassuk, a második a beolvasott adatok formáját leíró szöveges értéket jelölő mutató. A további paraméterek mutatók, amelyek kijelölik, hogy a beolvasott adatokat hol kell elhelyeznie a függvénynek.

A függvény a csatornából olvasott adatokat a formaszövegnek megfelelően részekre vága és az egyes részeket elhelyezi a megfelelő paraméterek által jelölt memóriaterületeken.

A függvény visszatérési értéke a beolvasott adatok részeinek száma, vagyis az, hogy hány memóriaterületre sikerült adatokat helyeznie. Ha ez kevesebb, mint a várt, a beolvasott adatok formátuma nem volt megfelelő. A visszatérési érték EOF, ha a csatorna nem olvasható.

```
int sscanf(const char *memória, const char *formaszöveg, ...);
```

A függvény formázott bemenetet valósít meg memóriaterületről.

A függvény első paramétere a memóriaterület címe, ahonnan az adatokat olvassuk, a második a beolvasott adatok formáját leíró szöveges értéket kijelölő mutató. A további paraméterek mutatók, amelyek kijelölik, hogy a beolvasott adatokat hol kell elhelyeznie a függvénynek.

A függvény a memóriából olvasott adatokat a formaszövegnek megfelelően részekre vága és az egyes részeket elhelyezi a megfelelő paraméterek által kijelölt memóriaterületeken.

A függvény visszatérési értéke a beolvasott adatok részeinek száma, vagyis az, hogy hány memóriaterületre sikerült adatokat helyeznie. Ha ez kevesebb, mint a várt, a beolvasott adatok formátuma nem volt megfelelő. A visszatérési érték EOF, ha a memóriaterületről nem lehet további adatokat olvasni, mert elértük a szöveges érték végét jelző 0 bajtot.

Összehasonlítva a scanf() függvéncsaládot a már ismertetett printf() családdal, meg kell jegyeznünk, hogy a scanf() család esetében az argumentumok minden mutatók. Ez érthető is, hiszen ezek a függvények az argumentumokkal jelzett helyen adják vissza a beolvasott értékeket.

A másik fontos különbség a formátum jelzésére használt karakterláncban található. Ez többé-kevésbé hasonló módon épül fel, mint a printf() függvéncsalád formázószövege, de vannak különbségek. A következő felsorolás azt mutatja be, hogy a scanf() függvéncsaládban használt formaszöveg milyen kifejezéseket tartalmazhat, milyen karaktereket helyezhetünk el benne a % jel után.

% A %% kifejezés a % jelre illeszkedik. Ehhez a kifejezéshez nem tartozik paraméter, amelynek értékét a függvény beállítaná.

%d Előjeles formátumú egész szám beolvasására szolgáló kifejezés. A %d kifejezéshez tartozó argumentum egy int értéket jelölő mutató.

%i Előjeles formátumú egész szám olvasására szolgáló kifejezés. A következő argumentum egy int típus változót jelöl a memóriában.

A %d kifejezéssel ellentétben a %i lehetővé teszi 16-os számrendszerű – 0x vagy 0X kezdetű számok – vagy 8-as számrendszerű – 0 kezdetű számok – használatát.

%o Előjel nélküli szám beolvasása 8-as számrendszerben. A %o kifejezéshez tartozó argumentum unsigned int típusú változót jelöl a memóriában.

%u Előjel nélküli szám beolvasása 10-es számrendszer használatával. A %u kifejezéshez tartozó argumentumnak unsigned int típusú változót kell kijelölnie a memóriában.

%x Előjel nélküli szám beolvasása 16-os számrendszer használatával. A %x kifejezéshez tartozó argumentumnak unsigned int típusúnak kell lennie.

%f Lebegőpontos szám beolvasására szolgáló kifejezés. A %f kifejezéshez tartozó argumentumnak float típusú változót kell kijelölnie a memóriában.

%s Karakterlánc beolvasására szolgáló kifejezés. A %s kifejezéshez tartozó argumentumnak egy karakterlánc tárolására alkalmas tömböt kell kijelölnie a memóriában.

Fontos tudnunk, hogy a függvénycsalád a %s kifejezés feldolgozásakor legfeljebb az első szóközig vagy tabulátor karakterig olvas, vagyis ez a kifejezés nem alkalmas több szó beolvasására.

%c Karakter vagy karakterlánc beolvasására alkalmas kifejezés. A %c kifejezéshez tartozó argumentumnak egy karakterlánc tárolására alkalmas tömböt kell a memóriában kijelölnie.

Ha több karaktert is be akarunk olvasni ezzel a kifejezással, használnunk kell a mező szélességet jelző kiegészítést. A kifejezés hatására a függvénycsalád az elhelyezett karakterláncot nem zárja le 0 karakterrel.

%[ Ez a jel karakterlánc beolvasására szolgáló összetett kifejezések építésére alkalmas. A kifejezéshez tartozó argumentumnak karakterlánc tárolására alkalmas memóriaterületet kell kijelölnie. A %[ jelekkel kezdődő kifejezések egy karakterosztályt írnak le, a függvénycsalád pedig a karaktereket sorra elhelyezi a megfelelő memóriaterületen, amíg nem talál olyan karaktert, amely nem része a karakterosztálynak.

A %[ kifejezés után felsorolhatjuk azokat a karaktereket, amelyek az osztály tagjai. A %[abcABC] kifejezés például egy karakterosztály, melynek tagjai az abc, valamint az ABC betűk. Ha azt szeretnénk, hogy a karakterosztály tagja

legyen a ] betű is, akkor azt közvetlenül a [ jel után kell elhelyeznünk, például %[]abc].

A karakterek között szerepelhet a - jel is, amelynek segítségével tartományt jelölhetünk ki. A %[a-z] a kisbetűket jelenti. Ha azt szeretnénk, hogy a - jel is szerepeljen a karakterosztályban, a - jelet utolsóként kell írnunk, például %[abc-].

A karakterosztály megadható a kivételek felsorolásával is, a ^ jelet használva. A %[^a-z] kifejezés például az összes betűt, számot és írásjelet jelenti a kisbetűk kivételével. Ha a ^ jelet a karakterosztály részeként fel kívánjuk használni, az nem állhat a felsorolás első jeleként közvetlenül a [ jel után.

**%p** Mutató beolvasására szolgáló kifejezés. A %p kifejezéshez tartozó argumentumnak void \* típusú változóra kell mutatnia.

A formaszövegben a % jel után módosítókat helyezhetünk el, melyek a következők lehetnek:

n A legnagyobb szélesség megadása. A függvénycsalád számára a kifejezés által beolvasott karakterek számát korlátozhatjuk a % jel után elhelyezett egész szám segítségével.

A %5c például egy 5 betűből álló karakterlánc beolvasására ad utasítást.

\* Az eredmény elvetése. Ha a kifejezésben a % jel után a \* áll, a kifejezés hatására beolvasott karaktereket a függvénycsalád tagjai egyszerűen eldobják.

a Jelzi, hogy a beolvasott karakterlánc számára a függvénynek kell lefoglalnia a memóriát a malloc() segítségével. A kifejezéshez tartozó argumentumban a lefoglalt memória címét adjva vissza a függvénycsalád.

h Jelzi, hogy a kifejezéshez tartozó argumentum short int típusú változót jelöl a memóriában.

l Egész típusú kifejezésben jelzi, hogy a következő argumentum long int típusú jelöl. Lebegőpontos kifejezésben jelzi, hogy a következő argumentum double típusú jelöl.

L Egész típusú kifejezések esetén jelzi, hogy a következő argumentum long long int típusú jelöl. Lebegőpontos kifejezésekben jelzi, hogy a következő argumentum long double típusú változót jelöl.

**20. példa.** A következő példa sorokat olvas be a scanf() függvény segítségével és kiírja az adatokat a printf() hívásával

```

1 #include <stdio.h>                                csatornak/scanf.c
2
3 main(){
4     int vett;
5     char szoveg[16];
6     int egesz;
7
8     while( scanf( "%[^t]\t%d\n", szoveg, &egesz ) != EOF ){
9         printf("%d %-16s\n", egesz, szoveg );
10    }/*while*/
11 }/*main*/

```

Tegyük fel, hogy egy állományban a következő sorok találhatók:

```

1 első sor      123
2 második sor   2
3 harmadik sor  234

```

A program segítségével az adatok a következő formára alakíthatók:

```

Bash$ scanf teszt.txt | ./scampelda
123 első sor
2 második sor
234 harmadik sor
Bash$

```

**21. példa.** A következő példaprogram egy kérdést tesz fel a felhasználónak és beolvassa a felhasználó válaszát a szabványos bemenetről.

```

1 #include <stdio.h>                                csatornak/begepel.c
2
3 char *beolvas( char *uzenet ){
4     char *olvasott;
5     char c;
6     printf( uzenet );
7     scanf( "%a[^n]%c", &olvasott, &c );
8     return(olvasott);
9 }/*beolvas*/
10
11 main(){
12     char *beolvasott;
13     beolvasott=beolvas( "Név: " );
14     printf( "%s\n", beolvasott );
15 }/*main*/

```

*Figyeljük meg, ahogyan a 7. sorban beolvassuk a felhasználó által beírt szöveget a olvasott memóriaterületre, amelyet a scanf () függvény foglal le. A felhasználó által beírt szövegnek tekintünk minden, ami nem tartalmaz újsor karaktert (%[^n]), utána pedig beolvassuk az újsor karaktert (%c) is.*

## A felhasználói adatbázis

A Unix rendszerek pontos nyilvántartást vezetnek a felhasználóikról. minden felhasználónak van egy felhasználói neve és egy felhasználói azonosítószáma (UID, *user identification number*), de a nyilvántartásban szerepel még néhány más fontos adata is.

A Linux vállalja a felhasználói adatbázis kezelését, a nyilvántartás megosztását más számítógépekkel, a GNU C könyvtár pedig egységes felületet biztosít a felhasználói adatok kezeléséhez. A programozó számára tehát gyakorlatilag mindegy, hogy hol, milyen formában vannak nyilvántartva a felhasználók, a könyvtár függvényei biztosítják az egységes kezelőfelületet programjaink számára.

Nem célunk a felhasználói adatbázis bemutatása, nem tárgyaljuk részletesen, hogy a felhasználóról nyilvántartott adatok pontos jelentése mi – a legtöbb nyilvántartott adat magától értetődő –, de szeretnénk felhívni a figyelmet a felhasználói azonosítószámra (UID), ami az egész nyilvántartás alapját képezi. Tudunk kell, hogy a Unix rendszerekben mindennek – az állományoknak, a folyamatoknak, a szemaforoknak stb. – tulajdonosa van, a tulajdonost pedig minden esetben a felhasználói azonosítószám alapján tárolja a rendszer. Ha tehát ki szeretnénk íratni valaminek a tulajdonosát, mindenkorban ki kell keresnünk a felhasználói adatbázisból a felhasználó nevét, hiszen a programunkkal dolgozó felhasználót nem az azonosítószám, hanem a tulajdonos neve érdekli.

Az itt tárgyalt eszközök használatához szükség lehet a sys/types.h, grp.h és a pwd.h fejállományok betöltésére.

A felhasználókról nyilvántartott legfontosabb információkat a pwd.h fejállományban létrehozott passwd nevű struktúra elemei hordozzák, amely a következő részekből épül fel:

A passwd struktúra	
char *pw_name	A felhasználó felhasználói neve.
char *pw_passwd	A felhasználó kódolt jelszava.
uid_t pw_uid	A felhasználó felhasználói azonosítószáma.
gid_t pw_gid	A felhasználó elsődleges csoportjának csoportazonosítószáma.
char *pw_gecos	Egyéb információk, a /etc/passwd GECOS mezője, amely hagyomány szerint vesszővel elválasztva a felhasználó valódi nevét, irodájának számát, munkahelyi és otthoni telefonszámát tartalmazza.
char *pw_dir	A felhasználó saját könyvtáranak abszolút elérési útja.
char *pw_shell	A felhasználó alapértelmezés szerinti héjprogramja.

A Unix rendszerek lehetőséget adnak arra, hogy a felhasználókat a szervezettebb munkavégzés kedvéért felhasználói csoportokba rendezzük. A felhasználói csoportnak is van neve (csoportnév) és azonosítószáma (GID, *group identification number*). Mivel az erőforrásokhoz a rendszer általában tulajdonoscsoportot is rendel és a tulajdonoscsoportot is azonosítószám alapján tartja nyilván, itt is szükségünk van a felhasználói adatbázis kezelésére.

A felhasználói csoportok adatainak hordozására a grp.h fejállományban létrehozott group struktúra használható, amely a következő részekből áll:

A group struktúra	
char *gr_name	A felhasználói csoport neve.
gid_t gr_gid	A csoport azonosítószáma.
char **gr_mem	A csoporttagok felhasználói nevét jelző mutatókat tartalmazó tömböt jelző mutató.

Az alábbi függvények a felhasználói adatbázis kezelésére szolgálnak. Segítségükkel adatokat kereshetünk ki a felhasználói adatbázisból.

```
struct passwd *getpwuid(uid_t uid);
```

A függvény az első paraméterként neki átadott felhasználói azonosítószám alapján kikeresi a felhasználó adatait a nyilvántartásból és visszaadja az adatokat tartalmazó területet jelző mutatót.

A függvény visszatérési értéke a kikeresett felhasználói adatokat tartalmazó területet jelző mutató, ha a művelet sikeres volt, illetve NULL érték, ha hiba lépett fel. A függvényt újra hívva ez a memóriaterület felülíródik, így ha később is szükségünk van az adatokra, a memóriaterületet mentenünk kell.

```
struct passwd *getpwnam(const char *felhasználónév);
```

A függvény a paraméter által jelzett átadott felhasználói névvel rendelkező felhasználó adatait adja vissza a felhasználói nyilvántartásból.

A visszatérési érték a felhasználó adatait tartalmazó területet jelző mutató, ha a művelet sikeres volt, és NULL érték, ha a művelet végrehajtása közben hiba lépett fel. A függvényt újra hívva ez a memóriaterület felülíródik, így ha később is szükségünk van az adatokra, a memóriaterületet mentenünk kell.

```
struct group *getgrgid(gid_t gid);
```

A függvény a paraméterként átadott csoportazonosítószámmal rendelkező csoport adatait keresi ki a felhasználói nyilvántartásból.

A visszatérési érték a csoport adatait tartalmazó területet jelző mutató, ha a művelet sikeres volt, és NULL érték, ha a művelet során hiba lépett fel. A függvényt újra hívva ez a memóriaterület felülíródik, így ha később is szükségünk van az adatokra, a memóriaterületet mentenünk kell.

```
struct group *getgrnam(const char *csoportnév);
```

A függvény a paraméter által jelzett átadott csoportnévvel rendelkező felhasználói csoport adatait keresi ki a felhasználói adatbázisból.

A visszatérési érték a csoport adatait tartalmazó területet jelző mutató, ha a művelet sikeres volt, és NULL érték, ha a művelet során hiba lépett fel. A függvényt újra hívva ez a memóriaterület felülíródik, így ha később is szükségünk van az adatokra, a memóriaterületet mentenünk kell.

**22. példa.** A következő példaprogram felhasználói nevek kikeresését végzi a felhasználói azonosítószám alapján.

1	#include <stdlib.h>	felhasznalok/getpwuid.c
2	#include <stdio.h>	
3	#include <pwd.h>	
4		
5	int main( int argc, char *argv[] ) {	
6	struct passwd *felhasznalo;	
7		
8	if( argc!=2 ){	
9	fprintf( stderr, "Használat: \n" );	
10	fprintf( stderr, " user <UID>\n" );	
11	exit(1);	
12	}/*if*/	
13		
14	felhasznalo=getpwuid( atoi(argv[1]) );	
15	if(felhasznalo!=NULL){	
16	printf("%s\n", felhasznalo->pw_name );	
17	exit(0);	
18	}else{	
19	exit(1);	

```

20 }/*if*/
21 }/*main*/

```

A program az első paramétereként megkapott azonosítószámot keresi ki a getpwuid() segítségével és kiírja a felhasználó nevét a szabványos kimenetre:

```

Bash$ ./getpwuid 500
pipas
Bash$

```

A program a 8–12. sorban megvizsgálja, hány paramétert kapott. Ha a program neve után nem pontosan egy paramétert írtunk a program indításakor, a program hibaüzenetet ír a szabványos hibacsatornára és kilép.

Ha a paraméterek száma megfelelő, a program a 14. sorban a paramétert átalakítja egész számmá az atoi() függvény hívásával, majd megkíséri kikeresni a felhasználó adatait a getpwiud() függvény hívásával. Ha a művelet sikeres volt – a visszatérési érték nem NULL –, a 16. sorban kiírja a felhasználó felhasználói nevét és 0 hibakóddal kilép. Ha a felhasználói adatai nem érhetők el – a visszatérési érték NULL –, a program üzenet nélkül, 1 hibakóddal kilép.

A felhasználói adatbázist kezelő, eddig bemutatott függvények csak akkor használhatók, ha az adott felhasználó, illetve csoport nevét vagy azonosítóját ismertük. Léteznek olyan függvények is, amelyekkel az összes felhasználót vagy csoportot végegjírhatjuk. Ezeket a függvényeket – melyek használatához a pwd.h, grp.h, valamint a sys/types.h fejállományok betöltése szükséges – ismertetjük itt.

```
void setpwent(void);
```

A függvény segítségével előkészíthetjük a felhasználói adatbázist olvasásra. A folyamat, amelyik ezt a függvényt hívta, későbbiekben a getpwent() függvény segítségével sorra kiolvashatja az adatbázisból a felhasználók adatait.

```
struct passwd *getpwent(void);
```

A függvény segítségével kiolvashatjuk a felhasználói adatbázisból a következő felhasználó adatait. Az adatbázist a függvény hívása előtt meg kell nyitni a setpwent() függvény segítségével.

A függvény visszatérési értéke az olvasott felhasználói adatokat tároló struktúrát jelöli. Fontos tudnunk, hogy ez a memóriaterület a függvény újabb hívásakor felülíródi, ezért, ha az adatokra később is szükségünk van, azokat le kell másolnunk.

A függvény visszatérési értéke NULL, ha már az utolsó felhasználót is beolvastuk vagy ha nem volt elegendő memória a művelet végrehajtásakor.

```
void endpwent(void);
```

A függvény segítségével felszabadíthatjuk a setpwent() hívásakor lefoglalt erőforrásokat. Ha ezután újra használni kívánjuk a getpwent() függvényt, a setpwent() függvényt újra kell hívniunk. Ekkor azonban újra a felhasználói lista elejére kerülünk, újra az első felhasználó adatait olvassa a getpwent().

```
void setgrent(void);
```

A függvény segítségével előkészíthetjük a felhasználói csoportok adatbázisát az olvasásra. A függvény hívása után a getgrent() függvény segítségével olvashatjuk a felhasználói csoportok adatait.

```
struct group *getgrent(void);
```

A függvény segítségével kiolvashatjuk a felhasználói csoportok adatbázisából az adatokat, csoportonként egy-egy függvényhívással. A függvény hívása előtt az adatbázist meg kell nyitnunk a setgrent() függvény hívásával.

A függvény által visszaadott mutató az olvasott csoport adatait jelöli a memóriában. Fontos tudnunk, hogy a függvényt újra hívva ez a memóriaterület felülíródik, így ha később is szükségünk van az adatokra, a memóriaterületet mentenünk kell.

A függvény visszatérési értéke NULL, ha az utolsó csoport adatait is beolvastuk vagy nem állt rendelkezésre elegendő memória a művelet véghajtására.

```
void endgrent(void);
```

A függvény segítségével felszabadíthatjuk a setgrent() függvény által lefoglalt erőforrásokat. Ha ezután újra hívni akarjuk a getgrent() függvényt, a setgrent() függvénytel újra meg kell nyitnunk az adatbázist. Ekkor viszont újra az adatbázis elejéről, az első csoport adatait olvassuk a getgrent() függvénytel.

**23. példa.** A következő példaprogram a getpwent() függvény segítségével a felhasználói adatbázisból sorra kiolvassa az összes felhasználó nevét és kírja a szabványos kimenetre.

	felhasznalok/getpwent.c
1	#include <stdio.h>
2	#include <pwd.h>
3	#include <sys/types.h>
4	
5	main(){
6	struct passwd *felh;
7	setpwent();
8	while( (felh=getpwent())!=NULL )
9	printf("%s\n", felh->pw_name);
10	endpwent();
11	}

## Jelszókezelés

A Unix rendszerek a felhasználók jelszavait csapóajtó kódolással kezelik. Ennek lényege, hogy a jelszó könnyedén kódolható egy titkosított formára, de a visszaállításra nincs mód. A rendszer a jelszó kódolt változatát tárolja, amely alkalmas a felhasználó által begépelt jelszavak ellenőrzésére, de belőle a jelszó nem állítható helyre.

A C könyvtár által a jelszavak kezelésére biztosított eszközök használatához a `crypt.h` fejállomány betöltése szükséges, valamint használnunk kell a `crypt` programkönyvtárat a fordítás során (a `gcc -lcrypt` kapcsolója). A következő függvényeket használhatjuk:

```
char *crypt(const char *jelszó, const char *só);
```

A függvény a Unix rendszereken használatos jelszókódolási módszereket valósítja meg.

A függvény első paramétere a kódolni kívánt jelszót jelöli, a második pedig a változatosságot növelő szöveget. A függvény által visszaadott mutató a jelszó kódolt formája, amely garantáltan csak nyomtatható karaktereket – kis- és nagybetűket, számokat és írásjeleket – tartalmaz. A változatosságot növelő segédváltozót hagyomány szerint sónak (*salt*) nevezzük.

A kódolás lényege, hogy a függvény által visszaadott szöveg egyértelműen jellemzi a jelszót, de abból a jelszó nem állítható helyre. Ha ugyanazt a sót felhasználva, újra kódoljuk a jelszót, akkor annak eredetisége ellenőrizhető, hiszen ugyanazt a kódolt jelszót kapjuk. A Unix rendszerek a jelszó kódolt formáját tárolják és annak felhasználásával ellenőrzik a felhasználók által beírt jelszó valódiságát.

A függvény GNU változata kétféle kódolási rendszert képes használni. A régebbi módszer a DES kódolást használja, az újabb pedig az MD5 módszert. Az MD5 módszer legnagyobb előnye az, hogy a segítségével 8 betűsnél hosszabb jelszót is használhatunk.

Ha a só kétbetűs kis- és nagybetűkből, valamint számjegyekből álló szöveg, a függvény DES kódolást használ. Ekkor a visszaadott szöveg első két betűje a só, utána pedig a jelszó kódolt változata található 11 betűhelyen. A visszaadott kódolt jelszó szintén csak kis- és nagybetűket, valamint számjegyeket tartalmaz.

Ha MD5 kódolással akarjuk a jelszót titkosítani, a sónak \$1\$ karakterekkel kell kezdődnie, ami után egy nyolcbetűs, nagy- és kisbetűkből, valamint számokból álló karaktersorozat szolgál a változatosság növelésére. A nyolc betű után egy \$ jelnek kell következnie, utána pedig a kódolandó jelszónak. Az MD5 kódolás használatakor a jelszó minden betűje hatással van a visszaadott, kódolt szövegre. A visszaadott szöveges érték elején a teljes só található, utána

pedig 22 betűhelyen a nagy- és kisbetűk, valamint számjegyek, amelyek a kódolt jelszót tartalmazzák.

**24. példa.** A következő példaprogram a paraméterként kapott jelszó kódolt formáját kiírja a szabványos kimenetre.

```
titkositas/crypt.c
1 #include <stdio.h>
2 #include <crypt.h>
3
4 int main( int argc, char *argv[] ){
5     printf("%s\n", crypt(argv[1], "ak") );
6 }/*main*/
```

Ha a programot kipróbáljuk, láthatjuk, hogy az ak sót használja – hiszen ez található a kódolt jelszó elején:

```
Bash$ ./crypt hello
akxXauerAFKL.
Bash$
```

Ha átírjuk a programban található sót \$1\$asavabor\$ értékre, MD5 kódolású jelszót kapunk:

```
Bash$ ./crypt pipas
$1$asavabor$daMfzNBIV1c008YU5P4Pb1
Bash$
```

## A futó folyamat tulajdonosa

Mint azt már tudjuk, minden folyamat valamely felhasználó nevében tevékenykedik, minden folyamat valamely felhasználó jogaival rendelkezik.

Valójában minden folyamat esetében beszélhetünk a folyamatot indító személy és csoport azonosítójáról és a folyamat jogait meghatározó személy és csoport azonosítójáról. A folyamat nem minden esetben azokkal a jogokkal rendelkezik, amelyekkel az indító személy. A SUID (*set user ID on execution*) bit bekapcsolása esetén például a folyamat a futtatható bináris állomány tulajdonosának jogaival fog futni.

A következő néhány függvény segítségével azt deríthetjük ki, hogy a program melyik felhasználó jogaival rendelkezik és ki indította a folyamatot. A függvények használatához be kell töltenünk a sys/types.h és unistd.h fejállományokat.

`uid_t getuid(void);` A függvény segítségével lekérdezhetjük, hogy a hívó folyamatot melyik felhasználó indította (*real user ID*, valós felhasználó-azonosító).

A függvény visszatérési értéke a folyamatot futtató felhasználó azonosítója.

```
uid_t geteuid(void);
```

A függvény segítségével lekérdezhetjük, hogy a folyamat melyik felhasználó jogaival rendelkezik (*effective user ID*, tényleges felhasználó-azonosító).

A függvény visszatérési értéke a jogokat meghatározó felhasználó azonosítója.

```
gid_t getgid(void);
```

A függvény segítségével lekérdezhetjük a hívó folyamat indítását kérő felhasználó alapértelmezett csoportjának csoportazonosítóját (*real group ID*, valós csoportazonosító).

A függvény visszatérési értéke a valós felhasználói csoportazonosító.

```
gid_t getegid(void);
```

A függvény segítségével lekérhetjük, hogy a folyamat melyik csoport jogaival rendelkezik (*effective group ID*, tényleges csoportazonosító).

A függvény visszatérési értéke a csoport azonosítója, amely meghatározza a folyamat jogait.

**25. példa.** A következő program lekérdezi a futtatását indító felhasználó azonosítóját és kiírja annak nevét, ha létezik a felhasználói nyilvántartásban.

	felhasznalok/whoami.c
1	#include <stdio.h>
2	#include <unistd.h>
3	#include <pwd.h>
4	#include <sys/types.h>
5	
6	int main(int argc, char *argv[]){
7	struct passwd *pw;
8	uid_t uid;
9	
10	uid = getuid();
11	pw = getpwuid (uid);
12	if( pw != NULL ){
13	puts (pw->pw_name);
14	exit(0);
15	}else{
16	fprintf (stderr, "Ismeretlen felhasználó.\n");
17	exit(1);
18	}/*if*/
19	}/*main*/

# Állománykezelés

Habár a már bemutatott csatornák segítségével az állománykezelés megvalósítható, a Unix rendszerek rendelkeznek egy másik eszközökészlettel is az állománykezelésre. Annak érdekében, hogy a két függvénykészletet meg tudjuk különböztetni egymástól, a szakirodalom ezt az újabb eszköztárat alacsonyszintű állománykezelésnek (*low level file handling*) nevezi. A könyv ezen része az alacsonyszintű állománykezelést mutatja be.

A csatornák használatával összevetve az itt tárgyalt eszközök némi képpen rugalmasabb eszközöket valósítanak meg. Sok olyan állománykezelési művelet van, ami nem végezhető el csatornák használatával, az alacsonyszintű állománykezeléssel azonban igen. Ennek az ára a bonyolultabb felépítés. Ezeket a függvényeket kissé nehezebb kezelni, az eszköztár sokkal több függvényt tartalmaz, de megismerésük mindenkiéppen megéri a fáradtságot, hiszen ezekkel az eszközökkel minden meg tudunk csinálni, amit az állományokkal egyáltalán meg lehet csinálni.

Az alacsonyszintű állománykezelés is hasonló elvek alapján épül fel, mint a már megismert csatornakezelés. Itt is meg kell nyitnunk az állományt, amelyet használni kívánunk, itt is rendelkezésre állnak az állományok írására és olvasására szolgáló függvények és itt is le kell zárnunk az állományt a használat után. Fontos különbség azonban a csatornákkal szemben, hogy itt az állományok nem sor szerkezetűek. Az olvasási és írási műveletek során nem sorok alapján végezzük a munkát, a függvények egyáltalán nem foglalkoznak az olvasott és írt adatok szerkezetével. Azt is mondhatnánk, hogy amíg a csatornakezelés ASCII állományok kezelésére szolgál, az alacsonyszintű állománykezelés bináris állományokkal való munkára készült.

## Alapműveletek

Először az állományokon végzett alapműveleteket mutatjuk be. Ezekkel az eszközökkel az állományokat megnyithatjuk, olvashatjuk vagy írhatjuk, majd lezárhatjuk.

A következő függvényekkel az alacsonyszintű állománykezelés alapvető műveletei végezhetők el. A függvények használatához be kell töltenünk az `fcntl.h` és az `unistd.h` fejállományokat.

```
int open(const char *fájlnév, int kapcsolók);
```

A függvény megnyitja az állományt, amelynek neve az első paraméter által jelzett területen található. A függvény a megnyitást a második paraméterben átadott kapcsolókkal jelzett módon végzi. A függvényt hívó folyamat a függvény hívása során visszakapott szám alapján kezelheti az állományt.

A függvény a megnyitás során egy mutatót rendel az állományhoz, amely a következő írt vagy olvasott adatterületet jelzi. minden olvasási és írási művelet ez alapján a mutató alapján történik, így lehetőségünk van több olvasási vagy

írási művelet segítségével végighaladni a teljes állományon. minden megnyitáshoz tartozik egy mutató, ezért ha egy állományt többször nyitunk meg, az egyes műveletek nem zavarják egymást.

A megnyitás során használt kapcsolók – amelyek meghatározzák, milyen módon használhatjuk a megnyitott állományt – a következő állandók kombinációja lehet (bitenkénti vagy művelet):

`O_RDONLY` Olvasási mód. Az ilyen kapcsolóval megnyitott állományból olvashatunk.

`O_WRONLY` Írási mód. Az ilyen kapcsolóval megnyitott állományba írhatunk.

`O_RDWR` Olvasási és írási mód. Az ilyen kapcsolóval megnyitott állományba írhatunk és onnan olvashatunk is.

`O_EXEC` Végrehajtási mód, megnyitás végrehajtás céljából. Az ilyen módon megnyitott állományt végrehajtás céljából nyitottuk meg. Ez a kapcsoló nem szabványos, a GNU C könyvtár bővíttet szolgáltatása.

`O_CREAT` Megnyitás létrehozással. Az ilyen kapcsolóval megnyitott állományt – ha az nem létezne – a rendszer létrehozza. Mivel üres állományból nem lehet olvasni, általában íráskor használjuk.

`O_EXCL` Megnyitás, ha az állományt létre lehet hozni. Ha az `O_CREAT` be van kapcsolva – vagyis kérjük az állományt létrehozni – és az állomány már létezik, ez a kapcsoló azt eredményezi, hogy a megnyitás sikertelen lesz. Akkor használjuk ezt a kapcsolót, ha biztosak akarunk lenni abban, hogy a megnyitott állomány új.

`O_NONBLOCK` Az állomány megnyitása nem blokkoló módban. A nem blokkoló mód azt jelenti, hogy az állomány használata során nem kívánunk várakozni az állománykezelő műveletek során.

Ha az állomány lassú fizikai eszközön helyezkedik el, előfordulhat, hogy a megnyitási, írási, olvasási vagy zárási művelet „túlságosan hosszú ideig tart”. Amíg a művelet be nem fejeződik, a rendszer a műveletet kérő folyamat futását felfüggeszti, vagyis a folyamat „blokkolódik”. Ha ezt a kapcsolót bekapcsoljuk, a futás soha nem blokkolódik, a rendszer inkább hibaüzenetben jelzi, hogy az erőforrás ideiglenesen nem elérhető.

`O_NOLINK` Ha az állomány közvetett hivatkozás, a közvetett hivatkozás megnyitására kerül sor, nem pedig a hivatkozott könyvtárbejegyzésére. A közvetett hivatkozás ilyen módon megnyitva a hivatkozott könyvtárbejegyzés nevét és elérési útját tartalmazza.

`O_TRUNC` Az állomány csonkolása 0 hosszúságúra a megnyitás során. A csonkoláshoz írási jog szükséges, de nem kell írásra megnyitnunk az állományt, habár olvasásra megnyitott állományt a megnyitáskor 0 hosszúságúra csonkolni nem praktikus.

**0\_APPEND** Megnyitás hozzáfűzésre. Az állomány írása mindenkor az állomány végen történik, függetlenül attól, hogy hová tessük a következő írás helyét jelző mutatót. A rendszer garantálja, hogy ha több folyamat is írja az állományt, mindenkor az éppen érvényes állományvégre kerüljenek a kiírt bájtok.

**0\_FSYNC** Megnyitás szinkron írásra. Az összes írási műveletet kérő függvény hívás addig tart, amíg az adatok a fizikai háttértárra nem kerültek. Ez nagymértékben növeli az adatbiztonságot.

**0\_NOATIME** Megnyitás a hozzáférési idő frissítése nélkül. A megnyitás és az olvasás során az állomány tulajdonságai között szereplő, az utolsó használatot jelző időpont nem frissül. (Ez az időpont nem a közismert „utolsó módosítás időpontja”, olvasásra vonatkozik.)

A függvény visszatérési értéke sikeres végrehajtás esetén egy nem negatív egész szám, ami a megnyitott állomány olvasására és írására használható, hiba esetén pedig -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCES** Az állomány adott módon való megnyitásához vagy létrehozásához a folyamat nem rendelkezik a megfelelő jogokkal.

**EEXIST** Az állományt új állományként kívántuk megnyitni – azaz be volt kapcsolva a `O_CREAT` és a `O_EXCL` is –, az állomány azonban már létezik.

**EINTR** A megnyitást egy jelzés szakította meg.

**EISDIR** A megnyitni kívánt könyvtárbejegyzés könyvtár.

**EMFILE** A folyamat túlságosan sok állományt nyitott meg, több állományt már nem nyithat.

**ENFILE** Az állományrendszerben túl sok a nyitott állomány, több már nem nyitható meg.

**ENOENT** Az állomány nem létezik.

**ENOSPC** Az új állományt nem lehet létrehozni, mert az állományrendszerben már nincs üres hely.

**ENXIO** Az állomány egy csővezeték, amelyet nem blokkoló módon akartunk megnyitni, de a csővezetéket nem olvassa egyetlen folyamat sem.

**EROFS** Az állományt nem lehet létrehozni, mert az csak olvasható állományrendszeren található.

```
int open(const char *fájlnév, int kapcsolók, mode_t jogok);
```

A függvény teljesen megegyezik az előző pontban bemutatott `open()` függvénnel, azzal a különbséggel, hogy itt egy újabb paraméter segítségével beállíthatjuk a létrehozandó állományhoz rendelt jogokat.

A harmadik paramétert a függvény csak akkor veszi figyelembe, ha az állomány megnyitása egy új állomány létrehozását eredményezi. Ekkor a harmadik paraméterből a függvény kivonja az umask értékét és az így kapott értékek határozzák meg a létrehozott állományhoz rendelt jogokat.

A harmadik paraméter típusa `mode_t`, amelyet részletesen a `stat` struktúra ismertetésekor tárgyalunk a 146. oldalon.

```
ssize_t read(int leíró, void *memória, size_t méret);
```

A függvény az `open()` segítségével megnyitott állományból olvas a megnyitott állományhoz rendelt mutatóval jelzett területről. Az olvasás során a mutató az olvasott adatok méretének megfelelő mértékben mozdul el az állomány vége felé.

A függvény első paramétere a leíró, amelyet az állomány azonosítására használunk. Ezt a leírót az `open()` függvény segítségével állíthatjuk elő. A második paraméter a memóriaterület elejének címe, ahová az olvasott állománytartalom kerül. A harmadik paraméter az olvasni kívánt adatmennyiség. A függvény nem garantálhatja, hogy a kívánt mennyiséget be tudja olvasni – ha az olvasás közben az állomány végére ér, az olvasás a kívántnál kevesebb adatot helyez a memóriába –, de a harmadik paraméternél több adatot nem olvas.

A függvény hibamentes futása esetén a visszatérési érték az olvasott bájtok száma. A visszatérési érték 0, ha állomány végéhez ért az olvasás úgy, hogy nem tudott már egyetlen bájtot sem olvasni. A visszaadott érték negatív, ha a művelet végrehajtása során hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EINTR** Az olvasást egy jelzés szakította meg.

**EAGAIN** Pillanatnyilag nincs olvasható adat, de a függvényt később újra hívva valószínűleg képesek leszünk adatokat olvasni. Ezt a hibaüzenetet csak akkor kaphatjuk, ha a `O_NONBLOCK` kapcsolót használtuk az állomány megnyitásakor.

**EIO** Valószínűleg hardverhiba történt.

**EBADFS** A leíró nem megfelelő, a felhasználásával nem lehet állományból olvasni.

```
ssize_t write(int leíró, const void *memória, size_t méret);
```

A függvény adatokat ír a megnyitott állományba.

Az első paraméter az állományleíró, amelyet az állomány azonosítására használunk. A leírót az `open()` függvény segítségével állíthatjuk elő. A második paraméter a memóriaterület elejének a címe, ahonnán az adatokat az állományba kívánjuk írni. A harmadik paraméter a méret, amely meghatározza, hogy mekkora területet kívánunk az állományba írni.

A függvény nem garantálja, hogy a kiírni kívánt adatmennyiséget egy lépéssben az állományba írja, ezért meg kell vizsgálnunk, hogy mekkora adatmennyiség írása történt meg és szükség esetén újra kell hívunk a függvényt.

A függvény az írás során a megnyitott állományhoz rendelt mutatót használja arra, hogy megállapítsa, az állomány melyik részére írja a memória tartalmát. Az írás után ez a mutató az írott adatok mértékének megfelelően elmozdul az állomány vége felé. Ha az írás az állomány végén túlnyúlik, az állomány mérete automatikusan növekszik a megfelelő mértékben.

Az állomány megnyitásának módjától függően az adatok írása esetleg nem történik meg azonnal a fizikai háttértárra, de az adatok a függvény visszatérése után már az állományból visszaolvashatók.

A függvény visszatérési értéke az írt bájtok számát adja meg, hiba esetén pedig -1 lesz. Ekkor a függvény beállítja az `errno` értéket, ami a következők egyike lehet:

**EAGAIN** Az írás nem történt meg, de a függvény újrahívásakor esetleg elvégzhető. Ez a hibajelzés csak akkor fordulhat elő, ha a `O_NONBLOCK` kapcsolóval nyitottuk meg az állományt.

**EBADF** Az első paraméterként adott leíró érvénytelen vagy az állomány nem írásra lett megnyitva.

**EFBIG** Ha az írást elvégezné a rendszer, az állomány nagyobb lenne, mint amekkorát a rendszer kezeli tud.

**EINTR** Az írási műveletet egy jelzés megszakította.

**EIO** Valószínűleg hardverhiba lépett fel.

**ENOSPC** A háttértár betelt.

**EPIPE** A csővezeték nem írható.

**EDQUOT** A felhasználó elérte a számára előírt háttértárkorlátot.

```
int close(int leíró);
```

A függvény zárja a leíróval jelzett állományt, amely művelet után a leíró már nem használható az állomány eléréséhez.

A függvény visszatérési hibamentes futás esetén 0, ha a pedig végrehajtás során hiba lépett fel, -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EBADF** A paraméterként átadott leíró nem érvényes.

**EINTR** A függvény végrehajtását egy jelzés megszakította.

**ENOSPC** Az állomány zárásakor nem volt elegendő hely az eddig nem mentett adatok mentésére.

**EIO** Valószínűleg hardver hiba történt.

**EDQUOT** Az állomány záráskor a nem mentett adatok mentése nem volt elvégezhető, mert a felhasználó túllépte a személyes lemezkorlátot.

```
int access(const char fájlnév, int mód);
```

A függvény segítségével megállapíthatjuk, hogy a megadott módon van-e jog a felhasználónak megnyitni és használni az adott állományt. A függvény olyan programban, ahol a SUID, illetve az SGID bitek be vannak állítva, nem azt állapítja meg, hogy a folyamat számára adott-e a jog, hanem azt, hogy a programot indító felhasználónak van-e jog a művelet elvégzésére. Ha tehát SUID/Sgid bitek beállításával futtatott programból hívjuk a függvényt, ellenőrizhetjük, hogy a programot indító személynek joga volna-e az állományművelethez, attól függetlenül, hogy a programnak van-e.

A program első paramétere a vizsgálni kívánt állomány nevét adja meg. A második paraméter megadja, hogy milyen jogot vagy jogokat szeretnénk megvizsgálni az állományon. Ez a második paraméter a következő állandókból képezhető bitenkénti *vagy* kapcsolattal:

**R\_OK** A függvény megvizsgálja, hogy a felhasználónak van-e joga az állományt olvasni.

**W\_OK** A függvény megvizsgálja, hogy a felhasználónak van-e joga az állományt írni.

**X\_OK** A függvény megvizsgálja, hogy a felhasználónak van-e joga az állományt futtatni.

**F\_OK** A függvény megvizsgálja, hogy az adott állomány létezik-e.

A függvény visszatérési értéke 0, ha az adott joggal a felhasználó rendelkezik (vagy F\_OK esetén az állomány létezik), -1, ha a jog hiányzik (F\_OK esetén az állomány nem létezik), vagy a művelet során hiba lépett fel. Hiba esetén a függvény beállítja az errno értékét, ami a következők egyike lehet:

**EACCES** A vizsgált jog hiányzik.

**ELOOP** Valószínűleg hurok van a könyvtárszerkezetben.

**ENAMETOOLONG** Az első paraméterként átadott állománynév vagy annak valamely része túlságosan hosszú.

**ENOENT** A vizsgálandó állomány nem létezik.

**ENOTDIR** Az első paraméterként átadott állománynévben valamelyik könyvtárnév a valóságban létezik, de nem könyvtár.

**EROFS** A vizsgált állományon nem áll rendelkezésre az írási jog, mert csak olvasható állományrendszerben van.

**EINVAL** A második paraméter értéke érvénytelen.

**EIO** Valószínűleg hardverhiba történt.

ENOMEM A művelet véghajtására nem állt rendelkezésre elegendő memória.

ETXTBSY Írási jogot vizsgáltunk olyan futtatható bináris állományon, amelyet a rendszer pillanatnyilag futtat, ezért az állomány nem írható.

**26. példa.** A következő példaprogram egy állományt másol le egy újonnán létrehozott állományba.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 int main(int argc, char *argv[]){
7     void *memoria;
8     int bemenet, kimenet;
9     int hibakod;
10
11    if( argc != 3 ){
12        fprintf( stderr, "Használat: \n" );
13        fprintf( stderr, " pc <forrás> <cél>\n" );
14        exit(1);
15    }/*if*/
16
17    memoria=malloc( 10240 );
18    if( memoria==NULL ){
19        fprintf( stderr, "Nincs elegendő memória\n" );
20        exit(1);
21    }/*if*/
22
23    bemenet=open( argv[1], O_RDONLY );
24    if( bemenet < 0 ){
25        fprintf( stderr, "%s nem nyitható meg.\n", argv[1] );
26        exit(1);
27    }/*if*/
28
29    kimenet=open( argv[2], O_WRONLY | O_CREAT | O_EXCL );
30    if( kimenet < 0 ){
31        fprintf( stderr, "%s nyitható meg.\n", argv[2] );
32        exit(1);
33    }/*if*/
34
35    while( (hibakod=read(bemenet, memoria, 10240)) > 0 )
36        write( kimenet, memoria, hibakod );
```

```

37     if( hibakod<0 )
38         fprintf( stderr, "Hiba másolás közben. \n" );
39
40     close(bemenet);
41     close(kimenet);
42
43     exit(0);
44 }/*main*/

```

*A bemutatott program nem másolja az állomány jellemzőit, így például a létrehozott új állományhoz tartozó jogok nagy valószínűséggel mások lesznek, mint amit az eredeti állománynál megfigyelhetünk.*

Amint a függvények leírásában láttuk, léteznek olyan állománykezelő műveletek, amelyek végrehajtását a program részére érkezett jelzések megszakíthatják. Ha ezen műveletek végrehajtása közben jelzés érkezik a folyamat számára, a műveletet végrehajtó függvény hibát jelezve tér vissza és beállítja az errno értékét az EINTR állandó alapján. Azt mondjuk ilyenkor, hogy a művelet ideiglenesen nem volt végrehajtható, így meg kell ismételni a függvényhívást.

Az ideiglenesen végre nem hajtható műveletek szükség esetén történő újrahívásáról gondoskodik a Linux rendszereken rendelkezésre álló TEMP\_FAILURE\_RETRY() makró, amely a paraméterében megadott függvényhívást addig ismételgeti, amíg az EINTR hibakódot ad. A makró használatához a következő sorokat kell elhelyeznünk a program elején:

```

1 #define _GNU_SOURCE
2 #include <errno.h>
3 #include <unistd.h>

```

Ezután az ideiglenes hiba esetén újra végrehajtandó függvényhívást a makró paramétereként kell megadnunk, ahogyan az előző példaprogram módosított részletében láthatjuk:

```

1 while( (hibakod=TEMP_FAILURE_RETRY(
2             read(bemenet, memoria, 10240))) > 0 )
3     TEMP_FAILURE_RETRY( write( kimenet, memoria, hibakod ) );

```

## Mozgás az állományban

Az alacsonyszintű állománykezeléssel kezelt állományokban az írást és olvasást egy állománymutató vezérli. Az állománymutató a következő olvasás és írás helyét jelöli ki az állományban. Amikor a megfelelő függvények segítségével az állományba

írunk, az írás az állománymutató által kijelölt helyen kezdődik, ha pedig az állományból olvasunk, az olvasás kezdődik ettől a ponttól.

Igen fontos tudnunk azt, hogy a rendszer minden állomány megnyitásakor, minden állománykezelőhöz egy mutatót rendel, így a folyamatok nem zavarják egymást az állománymutatók mozgatásával. Elképzelhető az is, hogy egy folyamat két hívással kétszer nyitja meg ugyanazt az állományt, így két mutatóhoz jut.

A következő néhány függvény az állománymutatók mozgatásával vagy az állománymutatóktól független működéssel ad újabb eszközöket a számunkra. A függvények némelyikéhez az `_XOPEN_SOURCE` megfelelő beállítása, valamint a `sys/types.h` és az `unistd.h` betöltése szükséges, ahogyan ezt a következő példa bemutatja:

```
1 #define _XOPEN_SOURCE 500
2
3 #include <sys/types.h>
4 #include <unistd.h>
```

Az állománymutatók kezelésére és az azuktól független állománykezelésre a következő függvényeket használhatjuk:

```
ssize_t pread(int leíró, void *memória, size_t méret, off_t cím);
```

A függvény működése és a visszaadott érték teljesen megegyezik a `read()` függvény működésével és visszaadott értékével, azzal a különbséggel, hogy ez a függvény nem az állományhoz rendelt mutató által jelzett helyről olvas, hanem a negyedik paraméter által kijelölt állományterületről. A függvény működése közben nem módosítja az olvasási pontot jelző mutatót.

A függvény a `read()` függvényhez képest az `errno` változóban jelzett hibák sorát két újabbal bővíti:

**EINVAL** A függvénynek átadott cím érvénytelen.

**ESPIPE** Az állomány típusa miatt csak sorosan olvasható, nem lehet benne megváltoztatni az olvasás helyét.

```
ssize_t pwrite(int leíró, const void *memória, size_t méret, off_t cím);
```

A függvény működésében megegyezik a `write()` függvény működésével, de az írás nem az írást jelző mutató által kijelölt helyre történik, hanem a negyedik paraméter által jelölt helyre.

A függvény a `write()` függvény által használt hibakódok sorát a következőkkel bővíti:

**EINVAL** A függvénynek átadott cím érvénytelen.

**ESPIPE** A megnyitott állomány típusa miatt nem lehet előírni az írás helyét.

```
off_t lseek(int leíró, off_t elmozdulás, int kapcs);
```

A függvény segítségével a következő olvasás vagy írás helyét határozhatjuk meg az adott állományon.

A függvény az első paraméterként jelzett megnyitott állományban az olvasás és írás helyét jelző mutatót helyezi át a második paraméterként jelzett értékkel, a harmadik paraméter által jelzett módon.

A harmadik paraméterként átadott kapcsoló a következő értékek egyike lehet:

**SEEK\_SET** A paraméterként megadott méret az állomány elejétől indulva mutatja meg az új olvasási és írási helyet.

**SEEK\_CUR** A méret a jelenlegi olvasási és írási ponttól mutatja meg az új helyet. Ekkor a méret lehet negatív is, ami azt jelzi, hogy az elmozdulás az állomány eleje felé történik.

**SEEK\_END** Az új írási és olvasási pont az állomány végétől számított méret távolságra van. A negatív érték itt is az állomány eleje felé mutat.

Ha az írás során az állomány vége utáni pontra írunk, az állomány nulla értékű bájtokkal egészítődik ki.

A függvény visszatérési értéke az olvasási és írási mutató aktuális értéke az állomány elejétől számítva bájtból. Ha a művelet során hiba lépett fel, a visszaadott érték -1, az **errno** értéke pedig a következők egyike:

**EBADF** A leíró érvénytelen.

**EINVAL** A kapcsoló értéke vagy a kiszámított mutató értéke nem megengedett.

**ESPIPE** Az állomány jellege miatt az olvasási és írási mutató nem módosítható.

## A könyvtárbejegyzések tulajdonságai

A Unix rendszereken minden állománynak szigorú szabványok szerint meghatározott tulajdonságai vannak. minden Unix rendszeren megtalálhatók ezek a tulajdonságok, melyek segítik a rendszer adminisztratív feladatainak elvégzését, lehetővé teszik a megfelelő biztonsági rendszerek kialakítását. Ebben a fejezetben olyan eszközöket mutatunk be, amelyek az állományok tulajdonságait vizsgálják és módosítják. Elsőként néhány struktúrával kell megismernedünk.

A **sys/stat.h** állományban létrehozott **stat** struktúra könyvtárbejegyzések tulajdonságainak tárolására alkalmas:

A stat struktúra	
mode_t st_mode	A könyvtárbejegyzések típusát és a hozzá tartozó jogokat tároló struktúra.
ino_t st_ino	A könyvtárbejegyzés egyedi sorszáma, a könyvtárbejegyzéshez tartozó inode száma.
ev_t st_dev	A könyvtárbejegyzést tartalmazó adathordozó azonosítója.
nlink_t st_nlink	A könyvtárbejegyzés inode-jára való hivatkozások száma.
uid_t st_uid	A könyvtárbejegyzés tulajdonosának felhasználói azonosítója (UID).
gid_t st_gid	A könyvtárbejegyzés tulajdonoscsoportjának csoportazonosító száma (GID).
off_t st_size	A könyvtárbejegyzés mérete.
time_t st_atime	Az állomány utolsó megnyitásának ideje.
unsigned long int st_atime_usec	Az állomány utolsó megnyitásának ezredmásodperceket tartalmazó része.
time_t st_mtime	Az állomány utolsó módosításának időpontja.
unsigned long int st_mtime_usec	Az állomány utolsó módosításának ezredmásodperceket tartalmazó része.
time_t st_ctime	Az állomány tulajdonságai utolsó módosításának időpontja.
unsigned long int st_ctime_usec	Az állománytulajdonságok utolsó módosításának ezredmásodperceket tartalmazó része.
blkcnt_t st_blocks	Az állomány által a háttértáron foglalt 512 bájtos blokkok száma.
unsigned int st_blksize	Az olvasásnál és az írásnál használható leghatékonyabb blokkméret.

A következő függvény segítségével az állományok jellemzőit kérdezhetjük le. A függvény használatához a sys/stat.h állományt be kell töltenünk.

```
int stat(const char *állomány, struct stat *tulajdonságok);
```

A függvény a könyvtárbejegyzés adatait olvassa be. Az első paraméter a könyvtárbejegyzés nevét jelöli mutató, a második pedig azt a területet jelöli ki, ahol a függvény a könyvtárbejegyzés tulajdonságait elhelyezi.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és -1, ha nem. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

EACCES A folyamatnak nincs joga az adott könyvtárból lekérdezni a bejegyzések adatait.

ENAMETOOLONG A könyvtárbejegyzés teljes elérési útja vagy neve túlságosan hosszú.

**ENOENT** A könyvtárbejegyzés nem létezik.

**ENOTDIR** A könyvtárbejegyzést tartalmazó könyvtár létezik, de nem könyvtár.

**ELOOP** Az állomány keresésekor túl sok közvetett hivatkozás végigjárása törtené meg, valószínűleg hurok van a könyvtárszerkezetben.

A stat struktúra mode\_t mezője a sys/stat.h fejállományban található állandók segítségével vizsgálható, amelyek a következők:

**S\_IRUSR** A tulajdonos olvasási joga.

**S\_IWUSR** A tulajdonos írási joga.

**S\_IXUSR** A tulajdonos futtatási, illetve belépéssijoga.

**S\_IRWXU** Ugyanaz, mint az S\_IRUSR | S\_IWUSR | S\_IXUSR.

**S\_IRGRP** A tulajdonoscsoport olvasási joga.

**S\_IWGRP** A tulajdonoscsoport írási joga.

**S\_IXGRP** A tulajdonoscsoport futtatási, illetve belépési joga.

**S\_IRWXG** Ugyanaz, mint az S\_IRGRP | S\_IWGRP | S\_IXGRP.

**S\_IROTH** Mások olvasási joga.

**S\_IWOTH** Mások írási joga.

**S\_IXOTH** Mások futtatási, illetve belépési joga.

**S\_IRWXO** Ugyanaz, mint az S\_IROTH | S\_IWOTH | S\_IXOTH.

**S\_ISUID** A SUID bit.

**S\_ISGID** Az SGID bit.

**S\_ISVTX** A „sticky” bit.

Ezeket az állandókat a bitenkénti és művelet segítségével arra használhatjuk, hogy a mode\_t adott bitjét kiválasszuk. Ezt mutatja be a következő program.

**27. példa.** A példaprogram megvizsgálja, hogy a paraméterként átadott néven található állományt a tulajdonosnak van-e jog a olvasni és futtatni.

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <unistd.h>

5
6 int main(int argc, char *argv[]){
7     struct stat jellemzok;

8
9     if( stat(argv[1], &jellemzok) != 0 ){
10         fprintf(stderr, "stat sikertelen" );
11         exit(1);
12     }/*if*/

13
14     if( (jellemzok.st_mode & S_IRUSR)!=0 &&
15         (jellemzok.st_mode & S_IXUSR)!=0 )
16         printf("A tulajdonos olvashatja és futtathatja.\n");
17 }/*main*/

```

*Figyeljük meg a 14–15. sorban hogyan választjuk ki az & művelettel előbb a tulajdonos olvasás jogát, majd a tulajdonos futtatási jogát.*

A mode\_t nem csak a jogokat, hanem a könyvtárbejegyzés típusát is hordozza. A típus vizsgálatára a következő makrók használhatók:

int S\_ISDIR(mode\_t mód);

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés könyvtár.

int S\_ISCHR(mode\_t mód);

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés karaktereszköz-meghajtó.

int S\_ISBLK(mode\_t mód);

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés blokkész köz-meghajtó.

int S\_ISREG(mode\_t mód);

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés szabályos állomány.

int S\_ISFIFO(mode\_t mód);

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés csővezeték.

int S\_ISLNK(mode\_t mód);

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés közvetett hivatkozás.

```
int S_ISSOCK(mode_t mód);
```

Igaz (nem 0) értéket ad, ha a könyvtárbejegyzés kapu.

Ezen makrók használatát mutatja be a következő példaprogram.

**28. példa.** A következő példaprogram megvizsgálja a paraméterként átadott néven létező állományt és kitírja ha az szabályos állomány illetve könyvtár.

	állomanyok/jellemzok/tipus.c
--	------------------------------

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <unistd.h>
5
6 int main(int argc, char *argv[]){
7     struct stat jellemzok;
8
9     if( stat(argv[1], &jellemzok) != 0 ){
10         fprintf(stderr, "stat sikertelen" );
11         exit(1);
12     }/*if*/
13
14     if( S_ISDIR(jellemzok.st_mode) !=0 )
15         printf("A %s könyvtár.\n", argv[1]);
16     if( S_ISREG(jellemzok.st_mode) !=0 )
17         printf("A %s állomány.\n", argv[1]);
18 }/*main*/

```

## Közlekedés a könyvtárszerkezetben

Minden futó programnak, minden folyamatnak van egy munkakönyvtára, amely az a könyvtár a háttértáron, amelyhez képest a relatív könyvtárleírókat a rendszer értelmezi.

A következő függvényeket a munkakönyvtár lekérdezésére és megváltoztatására használhatjuk. A függvények használatához az unistd.h fejállomány betöltése szükséges.

```
char *getcwd(char *memória, size_t méret);
```

A függvény a munkakönyvtárat adja meg abszolút elérési út segítségével. A függvény első paramétere a memóriaterületet jelöli a memóriában, aholá a munkakönyvtár nevét másolja a függvény, a második paraméter pedig az erre a célra fenntartott memóriaterület nagyságát adja meg.

Ha az átadott memóriacím NULL, a függvény maga foglal memóriát, ha a méret 0, a függvény akkora memóriát foglal, amekkora elegendő a munkakönyvtár abszolút elérési útjának tárolására.

A visszatérési érték a memóriaterületre mutat, ha nem történt hiba, hiba esetén pedig NULL. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

**EINVAL** A méret 0, de a memóriacím nem NULL.

**ERANGE** A méret kisebb, mint a szükséges terület nagysága, a memória nem elegendő a munkakönyvtár abszolút elérési útjának tárolására.

**EACCES** Hozzáférés megtagadva.

```
int chdir(const char *könyvtárnév);
```

A függvény a munkakönyvtár megváltoztatására szolgál, a paraméter a beállítani kívánt könyvtár neve.

A visszatérési érték 0, ha a művelet sikeres volt, illetve -1 hiba esetén. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

**EACCES** A függvényt hívó folyamatnak nincs joga az adott műveletet végre-hajtani.

**ENAMETOOLONG** Az átadott állománynév túlságosan hosszú.

**ENOENT** Az átadott néven könyvtárbejegyzés nem létezik.

**EL0OP** A könyvtár keresése közben túl sok közvetett hivatkozás található, valószínűleg hurok található az állományrendszerben.

**ENOTDIR** Az átadott név nem könyvtárnév.

## Könyvtárak olvasása

A megfelelő eszközökkel megnyitva és olvasva a könyvtárakat, a bennük található könyvtárbejegyzések neveit és alapvető tulajdonságait tudhatjuk meg.

A dirent.h fejállományban létrehozott dirent struktúra könyvtárbejegyzések tulajdonságainak tárolására alkalmas. A következő elemeket tartalmazza:

A dirent struktúra	
char d_name	A könyvtárbejegyzés neve.
ino_t d_fileno	A könyvtárbejegyzés inode száma.
unsigned char d_namlen	A könyvtárbejegyzés nevének hossza a lezáró 0 nélkül.
unsigned char d_type	A könyvtárbejegyzés típusa.

A struktúra `d_type` mezőjének értéke – ami a könyvtárbejegyzés típusát adja meg – a következő állandók egyike lehet.

`DT_UNKNOWN` A könyvtárbejegyzés típusát nem lehet meghatározni.

`DT_REG` A könyvtárbejegyzés szabályos állomány.

`DT_DIR` A könyvtárbejegyzés könyvtár.

`DT_FIFO` A könyvtárbejegyzés névvel ellátott csővezeték.

`DT_SOCK` A könyvtárbejegyzés hálózati kapcsolatot biztosító foglalat.

`DT_CHR` A könyvtárbejegyzés karakteres köz-meghajtó.

`DT_BLK` A könyvtárbejegyzés blokk köz-meghajtó.

`DT_LNK` A könyvtárbejegyzés közvetett hivatkozás.

Könyvtárak tartalmát a következő néhány függvény segítségével olvashatjuk. A függvények használatához a `sys/types.h` és a `dirent.h` betöltése szükséges.

`DIR *opendir(const char *könyvtárnév);`

A függvény segítségével könyvtárakat nyithatunk meg olvasásra, hogy kiolvashassuk a könyvtárban található könyvtárbejegyzések neveit.

A függvény paramétere a megnyitni kívánt könyvtár neve.

A függvény visszatérési értéke a megnyitott könyvtár olvasásához használható mutató, ha a megnyitás sikeres volt, és `NULL`, ha hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

`EACCES` A folyamatnak nincs joga az adott könyvtárból lekérdezni a bejegyzések adatait.

`ENAMETOOLONG` A könyvtárbejegyzés teljes elérési útja vagy neve túlságosan hosszú.

`ENOENT` A könyvtárbejegyzés nem létezik.

`ENOTDIR` A könyvtárbejegyzés létezik, de nem könyvtár.

`ELOOP` A könyvtár keresésekor túl sok közvetett hivatkozás végigjárása törten meg, valószínűleg hurok van a könyvtárszerkezetben.

`EMFILE` A programunk már túl sok állományt nyitott meg, ezért ez a művelet már nem volt végrehajtható.

`struct dirent *readdir(DIR *könyvtár);`

A függvény a megnyitott könyvtárban található könyvtárbejegyzések közül a következőt olvassa be és egy struktúrában adja vissza a legfontosabb adatait.

A függvény paramétere a könyvtárat jelölő mutató, amelyet az opendir() függvényvel való megnyitáskor kaptunk.

A függvény visszatérési értéke a következő könyvtárbejegyzés adatait tartalmazó struktúrát jelöli a memóriában. Ha a művelet során valamilyen hiba lép fel, vagy nincs több könyvtárbejegyzés, a visszatérési érték NULL. Ha a művelet során hiba lépett fel, a függvény beállítja az errno értékét, ami a következő lehet:

EBADF A paraméterként átadott mutató érvénytelen értéket tartalmaz.

```
int closedir(DIR *könyvtár);
```

A megnyitott könyvtár zárása. A függvény hívása után a könyvtárból csak akkor tudunk olvasni, ha az opendir() függvényt újra meghívjuk.

A függvény paramétere a zárnival kívánt könyvtárat jelöli.

A visszaadott érték 0, ha a művelet sikeres volt, illetve -1, ha nem. Ekkor a függvény beállítja az errno értékét, ami a következő lehet:

EBADF A paraméterként átadott mutató érvénytelen értéket tartalmaz.

```
void rewinddir(DIR *könyvtár);
```

A függvény hívásával visszatérhetünk a könyvtár első könyvtárbejegyzésének olvasásához, a readdir() következő hívásával újra a könyvtár elejéről olvashatunk. A függvény hívása után olvashatjuk azokat a könyvtárbejegyzéseket is, amelyek a opendir() hívása után lettek létrehozva, a rewinddir() ugyanis újraolvassa a könyvtárat.

A függvény paramétere a könyvtárat azonosítja.

**29. példa.** A következő példaprogram egy könyvtártartalmat kiírató program, amely a paraméterként megkapott könyvtárnevet megnyitja, majd onnan a könyvtárbejegyzések nevét sorra kiíratja a szabványos kimenetre.

```
1      #include <stdio.h>
2      #include <dirent.h>
3
4      int main(int argc, char *argv[]){
5          DIR           *konyvtar;
6          struct dirent *bejegyzes;
7
8          if(argc==2)
9              konyvtar=opendir( argv[1] );
10         else
11             konyvtar=opendir( "." );
12     }
```

```

13   if( konyvtar==NULL ){
14     fprintf(stderr, "Nem lehet megnyitni a könyvtárat.\n");
15     exit(1);
16   }/*if*/
17
18   while( (bejegyzes=readdir(konyvtar)) != NULL )
19     printf("%s\n", bejegyzes->d_name);
20
21   exit( 0 );
22 }/*main*/

```

A program a következő kimenetet hozza létre:

```
Bash$ dir01
.
..
dir01.c
Makefile
Bash$
```

A következő példaprogram bemutatja a könyvtárak olvasását (`opendir()`, `readdir()` és `closedir()` függvények) és a könyvtárváltást (`chdir()` függvény). A példaprogram különösen hasznos lehet, ha teljes könyvtárágak bejárását kívánjuk megvalósítani programunkban.

**30. példa.** A példaprogram teljes könyvtárágakat jár be önhívó (rekurzív) függvény-hívás segítségével és azok tartalmát a szabványos kimenetre írja.

	allomanyok/konyvtarlista/rekurziv.c
--	-------------------------------------

```

1 #include <stdio.h>
2 #include <dirent.h>
3 #include <unistd.h>
4
5 void rekkdir( int melyseg ){
6   DIR *konyvtar;
7   struct dirent *bejegyzes;
8   int n;
9
10  konyvtar=opendir(".");
11  if( konyvtar==NULL )
12    return;
13
14  while( (bejegyzes=readdir(konyvtar)) != NULL ){
15    if( bejegyzes->d_name[0] != '.' ){

```

```

17     for( n=0; n<melyseg; ++n )
18         printf("   ");
19
20         printf("%s\n", bejegyzes->d_name);
21     }/*if*/
22
23     if( bejegyzes->d_type == DT_DIR &&
24         bejegyzes->d_name[0] != '.' ){
25         chdir(bejegyzes->d_name);
26         rekadir(melyseg+1);
27         chdir("..");
28     }/*if*/
29 }/*while*/
30 closedir(konyvtar);
31 /*rekadir*/
32
33 int main(int argc, char *argv[]){
34     chdir(argv[1]);
35     rekadir(0);
36     exit(0);
37 }
```

A példaprogram 5–31. sorában található a `rekadir()` függvény, amely önhívással bejárja a teljes könyvtárát. A függvény egyetlen paramétere egy egész szám, amely megmutatja, milyen mélységi jutottunk a könyvtárszerkezetben a kiindulási ponttól. A függvény ezt a számot a 17–18. sorban arra használja, hogy a beljebb található könyvtárbejegyzések nevét a képernyőn beljebb írja. A program által kiírt listából leolvasható a könyvtárszerkezet:

```
Bash$ rekurziv
else
masodik
    allomany
    allomany1
    allomany3
    allomany4
Bash$
```

**31. példa.** A következő példaprogram könyvtárak tartalmának kiíratására szolgál. A `stat()` függvény segítségével részletesebb leírást ad a könyvtárbejegyzések röör.

	allomanyok/konyvtarlista/dir02.c
1	#include <stdio.h>
2	#include <dirent.h>

```
3 #include <sys/stat.h>
4 #include <grp.h>
5 #include <pwd.h>
6
7 void egyetkiir( struct dirent *e, struct stat *s ){
8     struct passwd *tulaj;
9     struct group *tulajcsoport;
10
11    switch(e->d_type){
12        case DT_REG:
13            putchar('-');
14            break;
15        case DT_DIR:
16            putchar('d');
17            break;
18        case DT_FIFO:
19            putchar('p');
20            break;
21        case DT_CHR:
22            putchar('c');
23            break;
24        case DT_BLK:
25            putchar('b');
26            break;
27        case DT_LNK:
28            putchar('l');
29            break;
30        case DT_SOCK:
31            putchar('s');
32            break;
33        default:
34            putchar('?');
35    }/*switch*/
36
37    if( (s->st_mode & S_IRUSR) != 0 ) putchar('r');
38    else putchar('-');
39    if( (s->st_mode & S_IWUSR) != 0 ) putchar('w');
40    else putchar('-');
41    if( (s->st_mode & S_IXUSR) != 0 ) putchar('x');
42    else putchar('-');
43
44    if( (s->st_mode & S_IRGRP) != 0 ) putchar('r');
45    else putchar('-');
```

```

46     if( (s->st_mode & S_IWGRP) != 0 ) putchar('w');
47     else putchar('-');
48     if( (s->st_mode & S_IXGRP) != 0 ) putchar('x');
49     else putchar('-');
50
51     if( (s->st_mode & S_IROTH) != 0 ) putchar('r');
52     else putchar('-');
53     if( (s->st_mode & S_IWOTH) != 0 ) putchar('w');
54     else putchar('-');
55     if( (s->st_mode & S_IXOTH) != 0 ) putchar('x');
56     else putchar('-');
57
58     printf(" %4d", s->st_nlink);
59
60     tulaj=getpwuid(s->st_uid);
61     tulajcsoport=getgrgid(s->st_gid);
62     printf("%8s %8s", tulaj->pw_name, tulajcsoport->gr_name );
63     printf("%8d", s->st_size);
64     printf("%s\n", e->d_name);
65 }
66
67 int main(int argc, char *argv[]){
68     DIR *konyvtar;
69     struct dirent *bejegyzes;
70     struct stat tulajdonsagok;
71
72     konyvtar=opendir(".");
73
74     while( (bejegyzes=readdir(konyvtar)) != NULL ){
75         if( stat(bejegyzes->d_name, &tulajdonsagok) == 0 )
76             egyetkiir(bejegyzes, &tulajdonsagok);
77         else
78             printf("%s\n", bejegyzes->d_name);
79     }/*while*/
80
81     exit( 0 );
82 }/*main*/

```

A program a következő táblázatot készít:

Bash\$ dir02

<i>drwxr-xr-x</i>	2	<i>pipas</i>	<i>root</i>	4096 .
<i>drwxr-xr-x</i>	14	<i>pipas</i>	<i>root</i>	4096 ..
<i>-rw-r--r--</i>	1	<i>pipas</i>	<i>root</i>	405 <i>dir01.c</i>

```
-rw-r--r-- 1 pipas root 443 dir02.c
-rw-r--r-- 1 pipas root 1798 dir03.c
-rw-r--r-- 1 root root 272 Makefile
Bash$
```

A `stat()` függvény kapcsán meg kell említenünk, hogy létezik egy `stat` program is, amely a könyvtárbejegyzések tulajdonságait a szabványos kimenetre írja. A parancs a paraméterként átadott néven megtalálható könyvtárbejegyzés minden adatát kiírja, ami a `stat` struktúrában megtalálható.

## Könyvtárbejegyzések létrehozása és törlése

A következő eszközök segítségével könyvtárbejegyzéseket – szabályos állományokat és könyvtárakat – hozhatunk létre és törölhetünk. Ezeknek az eszközöknek a használatához néhány alapismeretet érdemes felelevenítenünk.

Tudnunk kell, hogy Unix rendszereken egy állománynak több neve is lehet. Az állomány a rendszer szempontjából egy azonosítószám, amelynek akár több neve is lehet. Ha egy állománynak több neve is létezik, azt mondjuk, hogy az állományra közvetlen hivatkozást hoztunk létre, habár semmilyen módszerrel nem lehet eldönntení, hogy melyik volt az eredeti név és melyik jött létre a közvetlen hivatkozás létrehozásakor.

Amikor az állományt töröljük, valójában az állomány azonosítószámára hivatkozó nevet töröljük. Az állomány tartalma – az állomány azonosítószáma által kijelölt tárolóterület – csak akkor szabadul fel, ha az állományt jelölő utolsó nevet is töröltük. Ezért van az, hogy az állomány törlését a C könyvtár dokumentációja mint a hivatkozás megszüntetését (*unlink*) emlegeti.

Állományokat kétféleképpen hozhatunk létre. Egyrészt létrehozhatunk állományokat az állományok megnyitására szolgáló függvényekkel – amilyen például az `open()` függvény –, másrészt az ebben a fejezetben tárgyalt `link()` függvénnnyel. Amíg az első esetben az állomány létrehozásakor új tárolóterületet foglalunk, addig a második esetben – a `link()` függvény használatakor – csak egy új név rendelődik a már meglévő adatterülethez. Az itt tárgyalt eszközök tehát csak akkor használhatók állományok létrehozására, amikor már meglévő állomány új nevét kívánjuk létrehozni.

A könyvtárak törlésére vonatkozóan el kell mondanunk, hogy könyvtárak jelzésére nem hozhatunk létre közvetlen hivatkozásokat. Egy másik fontos megkötés, hogy a könyvtárak nem törölhetők, amíg bennük más könyvtárbejegyzés található. Ennek a korlátozó rendelkezésnek nyilvánvalóan az a célja, hogy a könyvtárban található könyvtárbejegyzéseket ne vágassuk el az összefüggő könyvtárfától, hiszen ha ezt megtehetnénk, az elszakadt állományokat többé nem érhetnénk el.

Állományok és könyvtárak létrehozására és törlésére az itt felsorolt függvényeket használhatjuk, amelyek használatához az `unistd.h` és `sys/types.h` fejállományok betöltése szükséges.

```
int link(const char *név, const char *újnév);
```

A függvény segítségével egy meglévő állományhoz újabb nevet készíthetünk.

A függvény első paramétere egy meglévő állomány nevét jelöli a memóriában, a második paramétere pedig egy újabb nevet.

A függvény visszatérési értéke sikeres végrehajtás esetén 0, hiba esetén -1. Ha a végrehajtás közben hiba lép fel, a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EXDEV** Az új név nem a meglévő névvel azonos állományrendszeren van, ezért az új név nem hozható létre.

**EPERM** A könyvtárbejegyzés – amelyre az új nevet létre szeretnénk hozni – egy könyvtár, így az új név nem hozható létre.

**EACCES** Az új név nem hozható létre, mert a befogadó könyvtárra nincs írási jog a folyamatnak.

**ENAMETOOLONG** Az első vagy a második paraméterként átadott név – vagy azok egy része – túlságosan hosszú.

**ENOENT** Az állománynév, amire új nevet akarunk létrehozni, nem létezik.

**ENOTDIR** Az első vagy második paraméterként átadott névben található valamely könyvtár a valóságban nem könyvtár típusú állománybejegyzés.

**ENOMEM** A művelet végrehajtásához nem állt rendelkezésre elegendő memória.

**EROFS** Az állományrendszer csak olvasható, ezért új könyvtárbejegyzés nem hozható létre.

**EEXIST** Az új néven már létezik könyvtárbejegyzés, így nem hozható létre.

**EMLINK** Az állománynak már túl sok neve van, nem hozható létre új.

**EL00P** Valószínűleg hurok van a könyvtárszerkezetben.

**ENOSPC** A háttértár, amelyen az új nevet létre szeretnénk hozni, betelt.

**EIO** Hardverhiba lépett fel az új név létrehozásakor.

```
int unlink(const char *állománynév);
```

A függvény megkísérel törölni egy állománynevet, amelyet paraméterként kap. Ha a törlés sikeres volt és az állománynak ez volt az utolsó neve, az állomány tartalma is törlődik.

A függvény paramétere a törlendő nevet jelöli a memóriában.

A függvény sikeres futása esetén 0 értéket ad vissza, ha pedig a művelet közben hiba lépett fel a visszatérési érték -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCES** Nincs írási jog a könyvtárra, amely az állományt tartalmazza, vagy be van állítva rá a „sticky” bit és a folyamat nem a tulajdonos nevében fut.

**EPERM** Az állománynév egy könyvtár neve, ezért ezzel a függvénytel nem törlhető.

**EISDIR** Az újabb rendszerek ezzel jelzik, ha a függvény paramétere könyvtár típusú könyvtárbejegyzés.

**ENAMETOOLONG** A paraméterként átadott állománynév – vagy annak egy része – túlságosan hosszú.

**ENOENT** Az állomány nem létezik.

**ENOTDIR** A paraméterként átadott állománynév valamely könyvtárat jelöli része a valóságban nem könyvtár.

**ENOMEM** A művelet elvégzésére nem állt rendelkezésre elegendő memória.

**EROFS** Az állomány csak olvasható állományrendszeren található, ezért nem törölhető.

**ELOOP** Valószínűleg hurok van a könyvtárszerkezetben.

**EIO** A művelet végrehajtása közben hardverhiba lépett fel.

```
int truncate(const char *fájlnév, off_t hossz);
```

A függvény segítségével az állomány hosszát változtathatjuk meg.

A függvény első paramétere a módosítani kívánt állomány nevét adja meg, a második pedig a beállítandó méret, bájtban megadva. Ha az állomány eredeti mérete hosszabb volt, mint az új méret, a levágott területen található adatok elvesznek. Ha az új méret hosszabb, mint az eredeti méret volt, az állomány mérete növekszik, az új területről olvasva 0 értékeket kapunk.

A függvény visszatérési értéke sikeres végrehajtás esetén 0, ha pedig hiba lépett fel, -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCESS** Az állomány módosítására a folyamatnak nincs joga.

**EFBIG** A beállítandó méret nagyobb, mint a rendszeren megengedett legnagyobb állományméret.

**EINTR** A műveletet egy jelzés beérkezése szakította meg.

**EINVAL** Az átadott méret érvénytelen értéket hordozott.

**EIO** Valószínűleg hardverhiba történt.

**EISDIR** Az átadott néven könyvtárbejegyzés létezik, de a típusa könyvtár.

**ELOOP** Valószínűleg hurok van a könyvtárszerkezetben.

**ENAMETOOLONG** Az átadott állománynév hosszabb, mint a rendszeren megengetett leghosszabb állománynév.

**ENOENT** A megadott néven könyvtárbejegyzés nem létezik.

**ENOTDIR** A megadott állománynevet tartalmazó könyvtár nem létezik.

**EROFS** A megadott állomány csak olvasható állományrendszerben található, ezért módosítani nem lehet.

**ETXTBSY** A megadott állomány foglalt. Ez csak akkor lehetséges, ha egy futtatható bináris állományról van szó, amelyet a rendszer éppen futtat.

```
int rename(const char *réginév, const char *újnév);
```

A függvény segítségével állományok nevét változtathatjuk meg.

A függvény első paramétere az állomány nevét jelöli a memóriában, a második pedig az állomány új nevét. Az átnevezés után a régi néven létező állományt az új néven érhetjük el. Az átnevezés során az állomány új könyvtárba kerülhet, de ennek az új könyvtárnak az állományrendszeren belül kell lennie.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és -1, ha nem. Ilyenkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCESS** A folyamatnak nincs joga az adott műveletet végrehajtani.

**EBUSY** A könyvtárbejegyzés átnevezése pillanatnyilag nem végezhető el, mert a könyvtárbejegyzést a rendszer használja. Ez a hiba általában csak könyvtárak átnevezésekor fordul elő.

**ENOTEMPTY** Akkor kapjuk ezt a hibát, ha a régi név egy könyvtárat jelöl, az új név pedig egy olyan könyvtárat, amely nem üres.

**EINVAL** A régi név egy könyvtár neve, amely az új név által jelölt állományt is tartalmazza.

**EISDIR** A régi név egy állományt jelöl, az új név pedig egy könyvtárat. A függvény segítségével nem lehet ilyen formában állományt könyvtárba helyezni, meg kell adnunk az állomány új nevét.

**EMLINK** Az állomány nevében túlságosan sok közvetett hivatkozás van, valószerűleg hurok van a könyvtárszerkezetben.

**ENAMETOOLONG** Az állomány új neve túlságosan hosszú, ilyen hosszú nevű állományt a rendszeren nem lehet létrehozni.

**ENOENT** A régi néven állomány nem található.

**ENOSPC** A művelet végrehajtásához – az új könyvtárbejegyzés létrehozásához – nincs elegendő üres hely a háttértárolón.

**ENOTDIR** Az állományt hordozó könyvtár létező könyvtárbejegyzés, de nem könyvtár.

**EROFS** Az állományrendszer csak olvasható módon van beillesztve, ezért a műveletet nem lehet elvégezni.

**EXDEV** A régi név és az új név két külön állományrendszeren található könyvtában található, ezért a művelet ezzel a függvénnel nem végezhető el.

```
int rmdir(const char *könyvtárnév);
```

A függvény segítségével könyvtárat törölhetünk, ha az üres (csak a . és a .. bejegyzés található meg benne).

Sikeressé esetén a függvény 0 értéket ad vissza, hiba esetén pedig -1 a visszatérési érték. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCES** Nincs írási jog a könyvtárra, amely az alkönyvtárat tartalmazza, vagy be van állítva rá a „sticky” bit és a folyamat nem a tulajdonos nevében fut.

**EBUSY** A könyvtár a rendszer által foglalt, ezért nem törölhető.

**ENOENT** A könyvtár nem létezik.

**EPERM** A könyvtárnév nem könyvtár neve, ezért ezzel a függvénnel nem törölhető.

**EROFS** A könyvtár csak olvasható állományrendszeren található, ezért nem törölhető.

**ENOTEMPTY** Az alkönyvtár nem üres, ezért nem törölhető.

```
int remove(const char *állománynév);
```

A függvény segítségével állományokat és könyvtákat törölhetünk.

A függvény paramétere a törölni kívánt könyvtár vagy állomány neve.

A függvény állományok esetében úgy viselkedik, mint az `unlink()` függvény, könyvtárok esetében pedig úgy, mint az `rmdir()` függvény.

```
int mkdir(const char *könyvtárnév, mode_t mód);
```

A függvény megkísérel egy új könyvtárat létrehozni.

A függvény első paramétere a létrehozandó könyvtár neve, a második pedig a létrehozás után beállítandó jogokat adjá meg. A második paraméter használatáról részletesen a `stat` struktúra ismertetésénél olvashattunk a 146. oldalon.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és -1, ha a végrehajtása közben hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCES** A könyvtárra, amelyben az új könyvtárat el akarja helyezni a folyamat, nincs írási joga.

**EEXIST** A létrehozni kívánt név már foglalt.

**EMLINK** A könyvtár, amelyben az új könyvtárbejegyzést el szeretnénk helyezni, már nem képes több könyvtárbejegyzést befogadni.

ENOSPC A háttértáron nem található elegendő üres hely.

EROFS A könyvtár nem hozható létre, mert az állományrendszer csak olvasható.

A függvény használatát a 165. oldalon található 32. példa mutatja be.

```
int symlink(const char *réginév, const char *újnév);
```

A függvény segítségével közvetett hivatkozásokat hozhatunk létre.

A függvény első paramétere egy már létező könyvtárbejegyzést ad meg, míg a második egy újonnan létrehozandó közvetett hivatkozás nevét. A létrehozott közvetett hivatkozás az első paraméterként megadott névre mutat.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és -1, ha hiba lépett fel a végrehajtás közben. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

EPERM Az adott állományrendszer nem támogatja a közvetett hivatkozásokat, így a hivatkozás nem hozható létre.

EFAULT A függvény első vagy második paramétereként jelzett könyvtárbejegyzés nem elérhető.

EACCES A folyamatnak nincs jog a megadott helyre könyvtárbejegyzést létrehozni.

ENAMETOOLONG A megadott állománynevek közül legalább az egyik hosszabb, mint a rendszeren létrehozható leghosszabb állománynév.

ENOENT Az első paraméterként megadott néven nem létezik könyvtárbejegyzés vagy az új név hossza 0.

ENOTDIR A második paraméterként megadott könyvtárbejegyzést hordozó könyvtár nem létezik, vagy létezik, de nem könyvtár.

ENOMEM A művelet végrehajtására nem állt rendelkezésre elegendő memória.

EROFS A létrehozandó közvetett hivatkozást hordozó állományrendszer csak olvasható, a hivatkozás nem hozható létre.

EEXIST A függvény második paramétereként megadott néven már létezik könyvtárbejegyzés, amelyet a függvény nem ír felül.

ELOOP Valószínűleg hurok van a könyvtárszerkezetben.

ENOSPC Az állományrendszerben nincs elegendő szabad hely az új könyvtárbejegyzés létrehozására.

EIO Valószínűleg hardverhiba történt.

```
int readlink(const char *fájlnév, char *memória, size_t méret);
```

A függvény segítségével közvetett hivatkozásokat olvashatunk, azaz megállapíthatjuk, melyik könyvtárbejegyzésre mutatnak.

A függvény első paramétere az olvasni kívánt közvetett hivatkozás nevét adja meg. A második paramétere egy memóriacímet jelöl, ahová a függvény a közvetett hivatkozás tartalmát kiolvassa, a harmadik paramétere pedig azt határozza meg, mekkora helyet foglaltunk a memóriában a tartalom hordozására.

A függvény a beolvasott karakterláncot nem zárja le 0 karakterrel, ha pedig nem áll rendelkezésre elegendő hely – azaz a harmadik paraméter kisebb, mint a közvetett hivatkozás mérete –, a hivatkozott könyvtárbejegyzés nevét csonkolja.

A függvény visszatérési értéke az olvasott bájtok számát adja meg sikeres futás esetén, hiba esetén pedig -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

`ENOTDIR` Az első paraméterként megadott nevet hordozó valamely könyvtár nem létezik.

`EINVAL` Az első paraméterként átadott könyvtárbejegyzés létezik, de nem közvetett hivatkozás, esetleg a fenntartott memóriaterület méretét átadó második paraméter negatív.

`ENAMETOOLONG` Az első paraméterben átadott név – vagy annak valamelyik része – túlságosan hosszú.

`ENOENT` Az első paraméterrel jelzett könyvtárbejegyzés nem létezik.

`EACCES` A folyamat számára a könyvtárbejegyzéshez való hozzáférés nem engedélyezett.

`ELOOP` Valószínűleg hurok van a könyvtárszerkezetben.

`EIO` Valószínűleg hardverhiba történt.

`EFAULT` Az átadott címen található memóriaterület nem elérhető a folyamat számára.

`ENOMEM` A művelet elvégzéséhez nem állt rendelkezésre elegendő memória.

```
char *canonicalize_file_name(const char *fájlnév);
```

A függvény segítségével állománynevekből távolíthatjuk el a bennük található közvetett hivatkozásokat. A függvény az állománynév legegyszerűbb abszolút elérési utat használó formáját adja meg. Ez a függvény nem szerepel a vonatkozó szabványokban, Linux rendszereken azonban elérhető a `stdlib.h` betöltésével, ha a betöltés előtt létrehozzuk a `_GNU_SOURCE` makrót.

A függvény eltávolítja az állománynévből a . és .. részeket, az egymás után többször szereplő / jeleket és a közvetett hivatkozásokat, mégpedig úgy, hogy az eredményül kapott állománynév abszolút elérési úttal az eredetileg jelzett állományra mutasson.

A függvény paramétere az átalakítani kívánt állomány nevét jelöli.

A függvény visszatérési értéke a függvény által a `malloc()` hívásakor lefoglalt memóriaterületre mutat, ami a könyvtárbejegyzés egységesített nevét tartalmazza. A memóriaterületet használat után fel kell szabadítanunk a `free()` függvénytellyel. Hiba esetén a függvény visszatérési értéke NULL érték. Ilyen esetben a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**ENAMETOOLONG** Az eredményül kapott állománynév túlságosan hosszú volna.

**EACCES** A függvény nem tudta végigkövetni az eredeti állománynevet, mert abban legalább egy olyan rész található, amelyhez a folyamatnak nincs hozzáférési joga.

**ENOENT** Az első paraméter által jelölt könyvtárbejegyzés nem létezik, vagy az első paraméterként jelölt karakterlánc hossza 0.

**EL00P** Valószínűleg hurok van a könyvtárszerkezetben.

**32. példa.** A következő példaprogram megkíséri létrehozni a `tmp/ könyvtárat a munkakönyvtárban az mkdir() függvény hívásával.`

```
1 #include <stdio.h>
2 #include <sys/stat.h>
3 #include <sys/types.h>
4
5 main(){
6     if( mkdir("tmp",
7             S_IRUSR | S_IWUSR | S_IXUSR) != 0 )
8         fprintf(stderr, "A létrehozás sikertelen.\n");
9 }/*main*/
```

*Figyeljük meg a 7. sorban, hogy a létrehozott könyvtárhoz a tulajdonos számára adott lesz az írás, olvasás és belépés jog, a többi felhasználó azonban semmiféle jogot nem kap. A létrehozott könyvtár jogai a következők:*

```
Bash$ ls -ld tmp
drwx----- 2 pipas  pipas  4096 máj  2 11:11 tmp
Bash$
```

Előfordulhat, hogy biztosak akarunk lenni abban, hogy egy bizonyos állománynév, amelyet a felhasználó megadott, a könyvtárszerkezet bizonyos ágában van-e. Erre általában biztonsági okokból van szükségünk, amikor meg akarjuk akadályozni, hogy a felhasználó rávegye programunkat arra, hogy a megadott könyvtárából kilépve, máshol végezzen el műveleteket. Ilyen ellenőrzőprogramot mutat be a következő példa.

**33. példa.** A következő függvény megbizonyosodik arról, hogy a megadott állomány a /home/ könyvtáron belül található-e. A függvény 0 értéket ad vissza, ha a paraméterként átadott állománynév a /home/ könyvtáron belül található, egyébként pedig 1-et.

```

1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int homeban( char *fajlnev ){
7      int vissza;
8      char *szabalyos;
9      szabalyos=canonicalize_file_name(fajlnev);
10
11     if( strncmp("/home/", szabalyos, 6)== 0 )
12         vissza=0;
13     else
14         vissza=1;
15     free(szabalyos);
16     return(vissza);
17 }/*homeban*/

```

*Figyeljük meg a programban, hogyan alakítottuk át a tetszőleges formában – akár relatív, vagy .. bejegyzéseket tartalmazó elérési úttal megadott – rendelkezésre álló állománynevet szabályos formájúra a canonicalize\_file\_name() függvény segítségével.*

## Könyvtárbejegyzések tulajdonságainak megváltoztatása

A következő néhány függvény segítségével létező könyvtárbejegyzések tulajdon-ságait állíthatjuk be. A függvények használatához a sys/types.h, sys/stat.h és az unistd.h állományok betöltése lehet szükséges.

```
int chown(const char *fájlnév, uid_t felhasználó, gid_t csoport);
```

A függvény segítségével könyvtárbejegyzések tulajdonosát és tulajdonoscso-portját változtathatjuk meg.

A függvény első paramétere a könyvtárbejegyzés neve, amelynek a tulajdonosát meg akarjuk változtatni, a második a felhasználó felhasználói azonosítószáma, ami a függvényhívás után a könyvtárbejegyzés tulajdonosa lesz. A harmadik paraméter a csoport csoportazonosítószáma, ami a függvény végrehajtása után a könyvtárbejegyzés tulajdonoscsoportja lesz.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, illetve -1 hiba esetén. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCES** A folyamatnak nincs joga a könyvtárbejegyzést tartalmazó könyvtárhoz hozzáérni.

**ENAMETOOLONG** A könyvtárbejegyzés teljes elérési útja vagy neve túlságosan hosszú.

**ENOENT** A könyvtárbejegyzést tartalmazó könyvtár nem létezik.

**ENOTDIR** A könyvtárbejegyzést tartalmazó könyvtár létezik, de nem könyvtár.

**ELoop** Az állomány keresésekor túl sokszor történt közvetett hivatkozás végi járása, valószínűleg hurok van a könyvtárszerkezetben.

**EPERM** A folyamatnak nincs joga a könyvtárbejegyzés tulajdonosának vagy csoport tulajdonosának megváltoztatására. Biztonsági okokból általában csak a rendszergazdának van joga ehhez.

**EROFS** A könyvtárbejegyzés olyan adathordozón van, ami csak olvasható módon van beilleszтve az állományrendszerbe.

```
int chmod(const char *fájl, mode_t mód);
```

A függvény segítségével az állományhoz tartozó jogokat módosíthatjuk.

A függvény első paramétere a módosítani kívánt állomány nevét jelöli a memoriában, a második pedig a jogokat írja le. A második paraméter típusa `mode_t`, amelyet részletesen a `stat` struktúra ismertetésekor tárgyalunk a 146. oldalon.

A függvény visszatérési értéke hibamentes futás esetén 0, hiba esetén -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EPERM** A folyamatnak nincs joga az adott állományhoz tartozó jogokat átállítani.

**EROFS** A jelölt állomány csak olvasható állományrendszerben van, ezért nem lehet módosítani.

**ENAMETOOLONG** Az átadott állománynév vagy annak egy része túlságosan hosszú.

**ENOENT** A módosítani kívánt állomány nem létezik.

**ENOMEM** A művelet végrehajtásához nem állt rendelkezésre elegendő memória.

**ENOTDIR** Az állománynév valamely könyvtárnév-része létezik, de valójában nem könyvtár.

**EACCES** Az állománynév valamely könyvtárnév-részére nem rendelkezik a folyamat olvasási joggal.

ELOOP Valószínűleg hurok van a könyvtárszerkezetben.

EIO Valószínűleg hardverhiba történt.

```
int utime(const char *fájlnév, struct utimbuff *idő);
```

A függvény segítségével az állományhoz rendelt időbélyegeket módosíthatjuk.

A függvény első paramétere az állomány nevét jelöli a memóriában, a második pedig a beállítandó időket. Ez utóbbi struktúra formája a következő:

A utimbuff struktúra	
time_t actime	A legutolsó használat – írás vagy olvasás – időpontja.
time_t modtime	A legutolsó módosítás – írás – időpontja.

Amennyiben a függvény második paramétere NULL értékű, a függvény az állományhoz tartozó időket a számítógép beépített órájának állása szerint állítja be.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, illetve -1, ha hiba lépett fel. Hiba esetén a függvény beállítja az errno értékét, ami a következők egyike lehet:

EACCES A folyamatnak nincs joga az állományhoz tartozó időket beállítani.

ENOENT Az állomány nem létezik.

**34. példa.** A következő példaprogram a saját magát tartalmazható futtatható bináris állományhoz tartozó jogokat állítja át. A futtatás után a jogok -rwxr-x--- formájúak lesznek.

	allomanyok/jellemzok/chmod.c
--	------------------------------

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4
5 int main( int argc, char *argv[] ){
6
7     if( chmod(argv[0], S_IRWXU|S_IRGRP|S_IXGRP)!=0 ){
8         fprintf( stderr, "Nem sikerült.\n" );
9         exit( 1 );
10    }/*if*/
11
12    exit( 0 );
13 }/*main*/

```

## A jogok maszkolása

A jogokat beállító összes függvény figyelembe vesz egy úgynevezett maszkot, amelyet minden esetben kivon a beállítandó jogokból. E maszk segítségével igen praktikus módon, a program egy pontján beállíthatjuk, milyen házirendet akarunk használni a könyvtárbejegyzések jogainak kezelésekor. A következőkben azokat a függvényeket vesszük sorra, amelyek ennek a maszknak a kezelésében segítenek. A függvények használatához be kell töltenünk a `sys/stat.h` állományt.

```
mode_t umask(mode_t újérték);
```

A függvény segítségével beállíthatjuk az állományjogok megváltoztatásakor és beállításakor használatos maszkot, amelyet a függvények kivonnak a mindenkor beállítandó jogokból.

A függvény paramétere a beállítandó maszk, visszatérési értéke pedig az előzőleg használt maszk.

```
mode_t getumask(void);
```

A függvény segítségével anélkül kérdezhetjük le az állományjogok beállításakor használt maszkot, hogy azt megváltoztatnánk. Ez a függvény GNU bővítmény, a használatához létre kell hoznunk a `_GNU_SOURCE` makrót a `sys/stat.h` betöltése előtt.

A függvény visszatérési értéke a programunk által használt állományjogmaszk.

Az állományjogok maszkolásának kapcsán meg kell említenünk, hogy a BASH beépített `umask` parancsa hasonlóképpen használható a héjprogramokban.

## Zárolás

Ha egyszerre több folyamat is használja ugyanazt az állományt, az állomány kezelésében zavarok léphetnek fel. A Linux ugyan képes az állományokat egyszerre több folyamat számára is elérhetővé tenni, de ha több folyamat kezeli egyszerre az állományt, előfordulhat, hogy az olvasott vagy írott adatok összekeverednek.

Ha egy folyamat olvassa az állományt, miközben egy másik folyamat olvassa, előfordulhat, hogy az olvasást végző folyamat félkész adatokat olvas, mivel az író folyamat még nem fejezte be a műveletet. Ha két folyamat írja egyszerre az állományt, előfordulhat, hogy az általuk küldött adatok összekeverednek, mivel nagyjából egyműködően érkeztek.

Az ilyen jellegű problémákat versenyhelyzetnek nevezzük, elkerülésükre pedig a zárolás módszerét használjuk. A Linux és a C könyvtár eszközökét ad számunkra a versenyhelyzetek elkerülésére az állományok kezelése közben.

A Linux kétféle zárolási módszert tesz lehetővé. Használhatunk elrendelő zárolást (*mandatory locking*), amely kötelező minden folyamatra nézve, valamint használhatunk tanácsolt zárolást (*advisory locking*). Az elrendelő zárolást a rendszer kikényszeríti, azaz az állományhozzáférés felügyesztésével lehetetlenné teszi a zárolás megkerülését, míg a tanácsolt zárolást a folyamatok megkerülhetik, az rájuk nézve nem kötelező. Mi az egyszerűség kedvéért csak a tanácsolt zárolást ismertetjük.

A tanácsolt zárolás teljes állományok zárolására használható. Kétféle zárat használhatunk az állomány zárolására. A megosztott zár (*shared lock*) lehetővé teszi, hogy más folyamatokkal együttesen használjuk az állományt, míg a kizárolagos zár (*exclusive lock*) csak egy folyamat számára teszi lehetővé az állomány használatát. Azt is mondhatjuk, hogy egy állományon egyszerre tetszőlegesen sok megosztott zárolás érvényesíthető, míg kizárolagos zárból csak egy, ami a megosztott zárak érvényesítését is lehetetlenné teszi.

A megosztott zár kiválóan alkalmas az olvasás, a kizárolagos zár pedig az írás védelmére. Ha egyszerre több folyamat olvassa az állományt, az nem okozhat gondot, és a megosztott zárolás lehetetlenné teszi az írást, amelyet kizárolagos zárral védünk. A kizárolagos zárral védett írás minden más műveletet kizárt.

A tanácsolt zárolás a következő függvényteljesítménnyel kezelhető. A függvény használatához a `sys/file.h` fejállományt be kell töltenünk.

```
int flock(int leíró, int művelet);
```

A függvény segítségével tanácsolt zárolást kérhetünk és azt feloldhatjuk. Ha a függvény hívásával zárolást kérünk, de az más zárolás miatt nem kapható meg, a függvény a folyamatot blokkolja, amíg a zárolás nem lehetséges. A folyamat blokkolása a zárolás biztosításának módja, tehát ha valamely folyamat úgy kezeli az állományt, hogy az `flock()` függvényt nem hívja, a zárolási rendszert megkerüli. Ezért a függvény csak tanácsolt zárolást tesz lehetővé, amely nem kötelező a folyamatokra nézve.

A függvény első paramétere az állományleíró, amelyet előzőleg az állomány megnyitásakor kaptunk.

A függvény második paramétere adja meg, milyen zárolási műveletet kívánunk elvégezni. Itt a következő állandókat használhatjuk:

`LOCK_SH` Megosztott zár kérése az adott állományra. A művelet hatására az esetlegesen már létező zárat a folyamat elveszíti.

`LOCK_EX` Kizárolagos zár kérése az adott állományra. A művelet hatására az esetlegesen már létező zárat a folyamat elveszíti.

`LOCK_UN` Az előzőleg kapott zár feloldása. Ha az állományt a `close()` függvény hívásával lezárjuk, az esetlegesen kapott zárak automatikusan felszabadulnak.

Ha a második paraméterként átadott állandóhoz a bitenkénti *vagy* művelettel a `LOCK_NB` állandót is hozzákapcsoljuk, a rendszer semmiképpen nem fogja blokkolni a folyamatot, akkor sem, ha az a kért zárat nem kaphatja meg.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és -1, ha nem. A függvény ilyen esetben beállítja az `errno` értékét, ami a következők egyike lehet:

`EWOULDBLOCK` A zárolása blokkolná a folyamatot, de azt úgy kértük, hogy a folyamat ne blokkolódjon. A folyamat a kért zárat nem kapta meg!

`EBADF` Az első paraméterként átadott állományleíró értéke érvénytelen.

`EINTR` A függvény végrehajtását egy beérkező jelzés szakította meg.

`EINVAL` A második paraméterként átadott művelet érvénytelen értéket tartalmazott.

`ENOLCK` Nem áll rendelkezésre a zárolás biztosításához a szükséges erőforrás.

35. példa. A következő példaprogram a tanácsolt zárolás működését és használatát mutatja be. A program megnyit egy állományt a munkakönyvtárban, majd megkíséri azt zárolni kizárolagos zárral. A program a zárat 10 másodpercen keresztül fenntartja, így több példányban futtatva megfigyelhetjük a zárolás hatását.

```
1      #include <stdio.h>
2      #include <sys/file.h>
3      #include <sys/stat.h>
4
5      int main(int argc, char *argv[]){
6          int allomany;
7
8          allomany=open( "temp.text", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR );
9          if( allomany== -1 ) {
10              fprintf( stderr, "A megnyitás sikertelen.\n" );
11              exit( 1 );
12          }/*if*/
13
14          printf( "Kizárolagos zárolás kérése..." );
15          fflush( stdout );
16
17          if( flock(allomany,LOCK_EX)== -1 ) {
18              fprintf( stderr, "A zárolás sikertelen.\n" );
19              exit( 1 );
20          }/*if*/
21
22          printf("Zárolva.\n");
23          fflush( stdout );
24
25          sleep(10);
```

```

26     printf("Zárolás feloldása... ");
27     fflush(stdout);
28
29     if( flock(allomany,LOCK_UN)==-1 ) {
30         fprintf( stderr, "A feloldás sikertelen.\n" );
31         exit( 1 );
32     }/*if*/
33
34     printf("Feloldva.\n");
35
36     if( close(allomany)==-1 ) {
37         fprintf( stderr, "Az állomány zárása sikertelen.\n" );
38         exit( 1 );
39     }/*if*/
40
41     exit( 0 );
42 }/*main*/

```

## Időt kezelő függvények

Most azokat az eszközöket vesszük sorra, amelyekkel az időt mérhetjük és a számítógép beépített óráját állíthatjuk be.

A legegyszerűbb időkezelő adattípus a `time_t`, amely az 1970. január 1. 00:00:00 óta eltelt másodpercekben méri az időt. A GNU C könyvtárban a `time_t` valójában `long int`, más rendszerekben azonban más lehet. Ha kihasználjuk, hogy a `time_t` valójában egyenértékű a `long int` típussal, a programunk veszít a hordozhatóságából.

A következő eszközök használatához a `time.h` betöltése szükséges.

```
time_t time(time_t *eredm);
```

A függvény a számítógép beépített órája által mutatott időt adja vissza.

A függvény paramétere egy memóriaterületet jelöl, ahol a függvény az óra állását fogja tárolni. Ha a paraméter értéke `NULL`, a függvény az óra állását nem tárolja a memóriában, az csak a visszatérési értékként jelenik meg.

```
int stime(time_t új);
```

A függvény a számítógép beépített órájának beállítását végzi.

A függvény paraméterének értéke az időpont, amelyre a beépített órát állítani szeretnénk.

A visszatérési érték nulla, ha a művelet sikeres volt, és -1, ha nem. Az egyetlen ok, ami miatt az óra beállítása meghiúsulhat, az, hogy a beállításhoz rendszer-gazdai jogkör szükséges.

**36. példa.** A következő példaprogram kiírja a szabványos kimenetre, hogy hány másodperc telt el 1970 január 1. 00:00 óta.

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4
5 main(){
6     time_t t;
7     t=time(NULL);
8     printf("time()=%d\n", t);
9 }
```

ido/szuletesnap.c

Ennek a nem túlságosan hasznos programnak a lelkét a 7. sorban található függvényhívás adja, amelynek során a `time()` hívásával lekérdezzük a számítógép belső óráját. A program a következő formájú kimenetet adja:

```

Bash$ ./szuletesnap
time()=1034520775
Bash$
```

A `time_t` típus meglehetősen távol áll a köznapi értelemben használt időtől, bár kétségtelen, hogy könnyű számításokat végezni a felhasználásával. A felhasználóval való kapcsolattartáskor inkább a `tm` struktúrát használjuk, ami a következő mezőkből áll:

A `time_t` struktúra

int <code>tm_sec</code>	A másodpercek.
int <code>tm_min</code>	A percek.
int <code>tm_hour</code>	A órák.
int <code>tm_mday</code>	Nap a hónapban.
int <code>tm_mon</code>	A hónap.
int <code>tm_year</code>	A év.
int <code>tm_wday</code>	Nap a héten.
int <code>tm_yday</code>	Nap az évben.
int <code>tm_isdst</code>	Nyári diószámítás.
long int <code>tm_gmtoff</code>	Időzóna.
const char * <code>tm_zone</code>	Időzóna neve.

`int tm_sec` A másodpercek értéke. Egész, melynek legkisebb értéke 0, legnagyobb értéke pedig 59.

`int tm_min` A percek értéke, szintén 0 és 59 közti egész szám.

`int tm_hour` Az órák értéke, 0 és 23 közti egészszám.

`int tm_mday` A nap sorszáma a hónapban. Az egész értéke 1 és 31 között van, a napok számozása tehát nem 0-tól indul.

`int tm_mon` A hónap sorszáma az évben, 0 és 11 között egész számkként.

`int tm_year` Az 1900. óta eltelt évek száma.

`int tm_wday` A nap sorszáma a hétfő napjai között. Az első nap a vasárnap, 0 értékkel, az utolsó pedig a szombat, 6 értékkel.

`int tm_yday` A nap sorszáma az évben, 0 és 365 között egész számkként.

`int tm_isdst` Ennek a mezőnek az értéke pozitív, ha a számítógépen be van kapcsolva a nyári időszámítás. Ez nem azt jelenti, hogy éppen nyár van, és használjuk a nyári időszámítást, csak azt, hogy az év során valamikor használni kell.

Ha a mező értéke 0, a számítógép beállítása szerint nem kell nyári időszámítást használni az év egyetlen szakaszában sem.

Ha a mező értéke negatív, a nyári időszámításra vonatkozó beállítás nem érhető el.

`long int tm_gmtoff` Ez a mező megmutatja, hogy a pontos idő kiszámításához hány másodperccel kellett eltérni az kezdő időpont óta eltelt másodpercek számától, azaz a `time()` függvény által adott értéktől.

A `time_t` típusban ábrázolt időformátumtól két ok miatt kell eltérni; az időzónák és a nyári időszámítás miatt. A számítógép rendelkezik a megfelelő ismeretekkel – vagyis tudja, hogy melyik időzónában van és milyen nyári időszámítást használnak az adott országban –, így ezeket az információkat felhasználhatja a `tm` kiszámításához.

Ez a mező nem szabványos, GNU és BSD rendszereken elérhető, de máshol nem feltétlenül.

`const char *tm_zone` A használt időzóna szöveges leírása. A mező nem szabványos, de GNU és BSD rendszereken elérhető.

A következő, `time.h` fejállományokban létrehozott függvények az idő és a dátum egyszerű kezelését teszik lehetővé.

```
struct tm *localtime_r(const time_t *alap, struct tm *eredm);
```

A függvény a time\_t adattípust alakítja át tm típusúvá, mégpedig a helyi időzóna és nyári időszámítás beállításainak megfelelően.

A függvény paraméterei közül az első az időpont, amelyet át akarunk alakítani, a második pedig az a memóriaterület, amelyben az eredményt el kell helyeznie.

A visszaadott érték a memóriaterület címe, ahová az eredményt a függvény helyezte, azaz megegyezik a második paraméterrel.

```
struct tm *gmtime_r(const time_t *alap, struct tm *eredm);
```

Ugyanaz, mint a localtime\_r, de a típusátalakítást nem a helyi beállításoknak megfelelően, hanem a greenwich-i középidő (GMT) szerint végzi.

```
time_t mktime(struct tm *alap);
```

A tm összetett típus alakítása time\_t típusúvá. A visszaadott érték az átalakított idő, ha a művelet sikeres volt, illetve -1, ha az összetett típusban beállított értékek nem alakíthatók egyszerű idővé.

A függvény figyelmen kívül hagyja a paraméter tm\_wday és tm\_yday részét a számításkor, de azokat a megfelelő értékkel kitölti a számítás során. Így a függvény öröknaptárként is használható.

A következő függvények az időpontokat olvasható, szöveges formára alakítják át. A függvények használatához a time.h állományt be kell töltenünk.

```
char *asctime_r(const struct tm *idő, char *memória);
```

A függvény segítségével a dátumot és az időt olvasható formára alakíthatjuk.

A függvény első paramétere az átalakítani kívánt időpontot tartalmazza, a második paramétere pedig azt a területet jelöli a memóriában, ahová az eredményt helyeznie kell. Ennek a területnek legalább 26 bájt hordozására kell alkalmasnak lennie.

A függvény visszatérési értéke megegyezik a harmadik paraméterrel.

```
char *ctime_r(const time_t *idő, char *szöveg);
```

Ugyanaz, mint az asctime\_r() és a localtime() egymás után hívva.

```
size_t strftime(char *ide, size_t méret, const char *formátum,
const struct tm idő);
```

A függvény segítségével az időt tetszőleges szöveges formára alakíthatjuk.

A függvény első paramétere azt a helyet jelöli a memóriában, ahol a szöveget el kívánjuk helyezni, a második paraméter pedig azt állítja be, mennyi helyet foglaltunk le a szöveg tárolására. Ha az első paraméter értéke NULL, a

függvény visszaadja, hány karaktert írt volna, illetve mekkora hely elegendő a szöveges idő tárolására.

A harmadik paraméter a szöveges idő formáját meghatározó formázószöveg, a negyedik pedig az idő, amelyet szövegessé kívánunk alakítani.

A függvény harmadik paramétereiként átadott formázószöveg hasonló a printf() függvénycsaládban használt formázószöveghez, benne a következő kifejezések használhatók:

- %a A nap rövidített neve.
- %A A nap teljes neve.
- %b A hónap rövidített neve.
- %B A hónap teljes neve.
- %C Az évszázad.
- %d A nap sorszáma a hónapban.
- %D A dátum a %m/%d/%y formát használva.
- %e A nap sorszáma a hónapban, szükség esetén szóközzel igazítva.
- %F A dátum a %Y-%m-%d formát használva.
- %g Az évszám az évszázad nélkül.
- %H Az óra 24 órás formában.
- %I Az óra 12 órás formában.
- %j A nap sorszáma az évben.
- %k Az óra 24 órás formában, szükség esetén szóközzel igazítva.
- %l Az óra 12 órás formában, szükség esetén szóközzel igazítva.
- %m A hónap sorszáma az évben.
- %M A percek.
- %n Az újsor karakter.
- %R Az óra és a perc kettősponttal elválasztva.
- %S A másodpercek.
- %t A tabulátor karakter.
- %T Az órák, percek és másodpercek kettősponttal elválasztva.
- %u A nap sorszáma a héten. A hétfő az 1-es sorszámú, a kedd a 2. és így tovább.
- %U A hét sorszáma az évben.
- %z Az időzóna számként.
- %Z Az időzóna szövegesen.

%% A % karakter.

A függvény visszatérési értéke az elhelyezett karakterek száma, amelyek után a karakterláncot lezáró 0 karakter következik. Ha a foglalt területen nem volt elegendő hely, a függvény visszatérési értéke 0.

**37. példa.** A következő példaprogram szöveges formában írja ki a számítógép beépített órája által jelzett időt.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4
5 main(){
6     char           szoveg[32];
7     struct tm      ido;
8     time_t         t;
9     time( &t );
10    printf( asctime_r(localtime_r( &t, &ido ), szoveg) );
11 }
```

ido/asctime.c

A program a következő kimenetet adja:

```
Bash$ ./asctime
Sat Jul 19 08:34:38 2003
Bash$
```

## A program sebességének mérése

Az eddig tárgyalt eszközök nem alkalmasak arra, hogy a program egyes részeinek sebességét megmérjük, ahhoz általában nagyobb felbontású eszközre van szükség.

Ebben a részben olyan eszközöket mutatunk be, amelyek alkalmasak gyors folyamatok sebességének vizsgálatára és különösen jól használhatók programrészletek futási sebességének mérésére.

Az itt bemutatott függvény használatához a `time.h` fejállomány betöltése szükséges.

```
clock_t clock(void);
```

A függvény visszaadja a hívó folyamat által felhasznált processzoridőt.

A processzoridő elvont idő, amely a folyamat által felhasznált processzorteljesítményre jellemző. A függvény azt adja vissza, hogy a folyamat mennyi időn

keresztül használta a számítógép processzorát, mégpedig egy olyan mérték-egységben, amely az adott számítógépre jellemző.

Ez a gépre jellemző órajel, a `clock()` által visszaadott érték, általában század vagy ezredmásodperces ütemezésű. A `CLOCKS_PER_SEC` állandó határozza meg, hogy egy másodpercen hány ilyen ütem található.

A `clock()` valójában nem alkalmas az eltelt idő pontos mérésére – hiszen a folyamat más folyamatokkal osztozik a processzoron –, de kitűnően alkalmas arra, hogy a program egyes részeinek futási sebességét összehasonlítsa.

**38. példa.** A következő példaprogram a program által felhasznált processzoridőt méri.

	ido/clock.c
1	#include <stdio.h>
2	#include <time.h>
3	
4	main(){
5	clock_t kezdet, veg;
6	int q;
7	kezdet=clock();
8	
9	for(q=0; q<99999; ++q )
10	printf("%d\r", q);
11	
12	veg=clock();
13	
14	printf("%f\n", ((double)(veg-kezdet))/CLOCKS_PER_SEC );
15	}

A program a 7. sorban tárolja a folyamat által használt processzoridőt, majd egy nagy számításigényű ciklust hajt végre. A ciklus a 9–10. sorokban látható, ennek a processzorfelhasználását mérjük.

A ciklus lefutása után újra lekérdezzük, hogy a mennyi a felhasznált processzoridő. Ha a ciklus utáni és előtti processzoridőt kivonjuk egymásból, megkapjuk, mennyi processzoridőt használt a ciklus. Éppen ezt tesszük a 14. sorban, ahol a különbséget elosztjuk a `CLOCKS_PER_SEC` értékével, hogy az eredményt másodpercben kapjuk.

Ha az elkészült programot a `time` parancssal futtatjuk, láthatjuk, hogy a program által használt processzoridő nagy részét a ciklus használja fel:

```
Bash$ time ./clock
0.050000
```

```
real    0m4.355s
user    0m0.040s
```

```
sys      0m0.010s
Bash$
```

A program által kiírt érték a felhasználói programként felhasznált processzoridő (user) és a folyamat által kezdeményezett rendszerterhelést jellemző processzoridő (sys) összege.

## Folyamatok indítása

A következő néhány oldalon olyan függvényeket vizsgálunk, amelyek programok indítását teszik lehetővé. Igen hasznosak lehetnek ezek a függvények, hiszen segítségükkel kihasználhatjuk a Linux szabványos programjai adta lehetőségeket az általunk írt C programokban is. A függvények használatához az `stdlib.h` és az `stdio.h` fejállományok betöltése szükséges lehet.

```
int system(const char *parancs);
```

A függvény segítségével egyszerűen indíthatunk programokat.

A függvény gyermekfolyamatként futtatja a `/bin/sh` parancsértelmezőt, végrehajtva a függvény paramétereként jelölt parancsot. A futtatás során a parancsértelmező a PATH környezeti változót használja a parancsban esetleg szereplő futtatandó programfájl megkeresésére.

Ha a parancsot nem lehetett futtatni, a visszatérési érték -1, különben a gyermekfolyamat visszatérési értéke.

A függvény használatát a 39. példa mutatja be.

```
FILE *popen(const char *parancs, const char *mód);
```

A függvény segítségével programot indíthatunk oly módon, hogy azzal szabványos csatornát használva kapcsolatban maradunk.

A függvény alfolymatként indítja el az első paraméter által jelölt szöveges változóban található héjparancsot és egy csővezetéket létesít a futó folyamat felé.

Ha a második paraméter értéke "r", a program szabványos kimenetéről olvashatunk a visszaadott leíróval, ha pedig "w", a szabványos bemenetére írhatunk a segítségével.

A függvény hiba esetén NULL értéket ad vissza.

```
int pclose(FILE *cső);
```

A függvény lezárja a `popen()` segítségével megnyitott csővezetéket.

39. példa. A következő példaprogram elektronikus levélben küld el egy állományt a rendszergazdának. A program a `system()` függvényt használja a `sendmail` indítására, amely a levelet kézbesíti.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main(){
5     int hibakod;
6
7     hibakod=system("sendmail root <mail1.c");
8     if( hibakod== -1 )
9         printf( "A programot nem lehet futtatni.\n" );
10    else
11        printf( "Hibakód:%d\n", hibakod );
12 }/*main*/

```

A program a 7. sorban indítja a `sendmail` nevű programot, a `/bin/sh` segítségével, amely a megfelelő módon átadja neki paraméterként a címzett nevét (`root`), és a szabványos bemenetét a `mail1.c` állományból irányítja át.

A program bővíti változata nem állományból veszi a kézbesítendő levelet, hanem maga állítja elő. Itt a `popen()` függvény segítségével nyitjuk meg a csővezetéket, amelybe a levelet küldjük. A csővezeték írására az `fprintf()` függvényt használjuk.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main(){
5     FILE *level;
6
7     level=popen("sendmail root", "w");
8     if( level==NULL ){
9         fprintf( stderr, "A programot nem lehet futtatni.\n" );
10        exit(1);
11    }
12
13    fprintf( level, "Subject: Próbalevél\n" );
14    fprintf( level, "A levél törzse...\n" );
15    pclose( level );
16 }/*main*/

```

A program a 7. sorban csővezetéket nyit a sendmail program itt elindított példánya felé. Ha a csővezeték nyitása sikeres volt, a 13–14. sorban adatokat ír a csővezetékre és a 15. sorban zárja azt. A csővezeték lezárása után a sendmail a levelet továbbítja a címzett felé.

A következő néhány függvény a folyamatok kezeléséhez alacsonyabb szintű eszközöket biztosít.

```
pid_t getpid(void);
```

A függvény a függvényt hívó folyamat folyamatazonosítóját adja vissza.

```
pid_t getppid(void);
```

A függvény a hívó folyamatot létrehozó folyamat – a szülőfolyamat – folyamatazonosítóját adja vissza.

```
pid_t fork(void);
```

A függvény klónozza a futó folyamatot, az a következő utasítástól két példányban fut tovább. A folyamat minden példánya örökli az eredeti folyamat által foglalt erőforrásokat és mindenkor ugyanazokból az utasításokból áll, de a függvény végrehajtása után önálló folyamatként hajtódik végre.

A függvény visszatérési értéke az eredeti folyamat számára a másolatként létrehozott gyermekfolyamat folyamatazonosítója, a gyermekfolyamat számára pedig 0. Ha a művelet sikertelen volt – nem született új folyamat –, a visszatérési érték -1. Ekkor az errno értéke a következők egyike lehet:

EAGAIN Jelenleg nem lehetséges az új folyamat létrehozása, de lehetséges, hogy a későbbiekben sikeres lehet a kérés.

ENOMEM Nincs elegendő memória a művelet végrehajtására.

A fork() függvény használatát a 40. példa mutatja be.

```
int execv(const char *fájlnév, char *const argv[]);
```

A függvény az első argumentumaként kapott állományt programként futtatja. Igen fontos tudnunk, hogy ez a függvény nem hoz létre új folyamatot, hanem az aktuális folyamatot cseréli le az állományban található utasításokra.

A függvény második argumentuma a programnak átadandó argumentumokat kijelölő mutató. Az első mutatónak a futtatandó állomány nevét kell kijelölnie, az utolsónak pedig NULL értéknek kell lennie.

A függvény használatát a 40. példa mutatja be.

```
pid_t waitpid(pid_t PID, int *állapot, int kapcsolók);
```

A függvény a gyermekfolyamatok futásának állapotát adja vissza. A waitpid() függvény alapértelmezés szerint blokkolja a program futását, amíg a gyermekfolyamat be nem fejeződik.

A függvény első argumentuma a folyamat folyamatazonosítója, amelynek állapotát meg kívánjuk vizsgálni, vagy WAIT\_ANY, ha egy tetszőleges gyermekfolyamat befejezódését meg akarjuk várni.

A függvény második paramétere egy mutató, amely azt a területet jelöli ki, ahová az állapot kerül. Ha a függvény második paraméterének értéke NULL, a gyermek állapotát nem mentjük.

A függvény harmadik paramétere a függvény működését befolyásoló kapcsoló, amely a következő két állandót tartalmazhatja bitenkénti vagy művelettel összekapcsolva:

**WNOHANG** A függvény nem blokkolja a program futását; ha van befejezett folyamat, annak elhelyezi az állapotát és visszaadja a folyamatazonosítóját, ha nincs, a visszaadott érték 0.

**WUNTRACED** Nem csak a befejeződött folyamatokról kaphatunk ezzel a kapcsolóval információt, hanem a befagyaszott folyamatokról is.

A függvény visszatérési értéke annak a folyamatnak a folyamatazonosítója, amelynek az állapotát a második paraméterként átadott mutatóval jelzett helyre mentette. Ha nem blokkoló jelleggel indítottuk a függvényt és nincs befejezett folyamat, a függvény 0 értéket ad vissza.

Ha a művelet valamilyen hiba miatt sikertelen volt, a visszatérési érték -1 és a függvény beállítja az errno értékét, ami a következők egyike lehet:

**EINTR** A függvény végrehajtását külső jelzés szakította félbe.

**ECHILD** Nincs gyermekfolyamat, amelynek a befejeződésére várhatnánk, esetleg a függvénynek átadott folyamatazonosító nem a függvényt hívó folyamat gyermekfolyamata.

**EINVAL** A függvénynek átadott kapcsolók érvénytelenek.

A függvény kapsán meg kell jegyeznünk, hogy a rendszer nem törli a folyamattáblából addig a folyamatokat, míg a szülőfolyamatuk nem vette át az állapotukat a waitpid() függvény hívásával. A már befejezett folyamatok, amelyeknek szülőfolyamatuk még fut, de nem vette át a visszatérési értéket, feleslegesen terhelik a rendszert. Ezeknek a folyamatoknak a hagyomány szerint „zombi” folyamat a neve, utalva ezzel előhalott állapotukra.

A waitpid() függvény által visszaadott állapotértéket a következő makrók segítségével értelmezhetjük.

```
int WIFEXITED(int STATUS);
```

E makró visszatérési értéke nem nulla (igaz), ha a gyermekfolyamat szabályos módon, az exit() függvény hívásával lépett ki.

```
int WEXITSTATUS(int STATUS);
```

Ez a függvény a gyermekfolyamat visszatérési értékét adja vissza, ha a WIFEXITED() igaz értéket adott. A visszatérési érték 1 bájtos, a gyermek által a kilépésre használt exit() függvény paramétere.

```
int WIFSIGNALED(int STATUS);
```

Ez a makró nem nulla (igaz) értéket ad vissza, ha a gyermekfolyamat azért fejezte be futását, mert egy jelzést nem fogadott.

```
int WTERMSIG(int STATUS);
```

Ha a WIFSIGNALED() visszatérési értéke igaz, ez a makró annak a jelzésnek az értéke, amelyet a folyamat nem fogadott.

```
int WCOREDUMP(int STATUS);
```

Ez a függvény nem nulla (igaz) visszatérési értéket ad, ha a gyermekfolyamat futása meg lett szakítva és core állomány készült.

```
int WIFSTOPPED(int STATUS);
```

Ez a makró nem nulla (igaz) értéket ad vissza, ha a gyermekfolyamat futása be van fagyasztva.

```
int WSTOPSIG(int STATUS);
```

Ha a WIFSTOPPED() igaz értéket ad vissza, ez a makró megadja annak a jelzésnek a számát, amelyik a befagyasztást okozta.

**40. példa.** A következő példaprogram önmagának egy újabb példányát indítja el gyermekfolyamatként. A gyermek és a szülő egy-egy üzenetet ír a szabványos kiemenetre, amely tartalmazza a folyamatazonosítót is.

folyamatok/fork.c

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 main(){
6     if( fork()==0 )
7         printf("Gyermek vagyok. PID=%d\n", getpid() );
8     else
9         printf("Szülő vagyok. PID=%d\n", getpid() );
10 }/*main*/
```

A program továbbfejlesztett változata a gyermekfolyamatot arra használja, hogy egy másik programot indítson el. A gyermekfolyamat lecseréli önmagát a /bin/ls program futtatásával kapott folyamatra, így a program tulajdonképpen gyermekfolyamatként a /bin/ls -l parancsot adja ki.

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 main(){
6     char *argv[3];
7     argv[0]="/bin/ls";
8     argv[1]="-la";
9     argv[2]=NULL;
10
11    if( fork()==0 ){
12        printf("Gyernekfolyamat");
13        execv(argv[0], argv);
14    }else
15        printf("Szülő vagyok. PID=%d\n", getpid());
16 }/*main*/

```

*Figyeljük meg, hogy ha a program 13. sora után valamelyen végrehajtható C nyelvű parancsot írtunk volna, azt a rendszer már nem hajtotta volna végre, mivel addigra a programunkat lecserélük az ls programra.*

A fork() függvényhez hasonlóképpen működik a daemon() függvény (másolatot hoz létre a folyamatról), amely démonok készítését teszi könnyebbé. A Unix rendszereken démonnak hívjuk azokat a programokat, amelyek a háttérben futva valamely esemény bekövetkezésére várnak. Ezek a programok általában valamelyen szolgáltatást valósítanak meg és azért várakoznak a háttérben, hogy megvárják, amíg valaki igénybe veszi az általuk kínált szolgáltatásokat.

A következő függvény segítségével egyszerűen készíthetünk démont. A függvény használatához az unistd.h fejállományt be kell töltenünk.

```
int daemon(int marad, int konzol);
```

A függvény meghívása után a folyamat démonként fut tovább. Valójában a függvény egy új folyamatot hoz létre, amely ugyanazt a programot hajtja végre, de démonként. A függvény futása során a fork() függvény meghívásával létrehozza önmaga futó másolatát, majd a szülő kilép.

A függvény első paramétere azt határozza meg, hogy a függvény hatására a folyamat munkakönyvtára változzon-e a gyökérkönyvtárra. Ha ez a paraméter 0, a függvény a munkakönyvtárat a gyökérkönyvtárra változtatja. Erre általában azért van szükség, mert a démonok hosszú hónapokig futnak és közben esetleg megszűnik az a könyvtár, amely a munkakönyvtáruk volt az indításukkor.

A függvény második paramétere azt jelzi, hogy a függvény átirányítsa-e a szabványos kimenetét, szabványos bemenetét és szabványos hibacsatornáját. Ha ez a paraméter 0, a függvény a szabványos csatornákat átirányítja a /dev/null karakteresköz-meghajtó állományba, amely minden elnyel. A szabványos csatornák átirányítása azért szokás, mert a démon futásakor az azt indító felhasználó általában már nincs bejelentkezve, ezért nem lehet számára üzeneteket írni a képernyőre.

A függvény visszatérési értéke sikeres végrehajtás esetén 0, hiba esetén -1. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

**EAGAIN** A művelet végrehajtása pillanatnyilag nem lehetséges, mert nem áll rendelkezésre elegendő erőforrás – nem indítható új folyamat –, de lehetséges, hogy a későbbiekben sikeres lehet a kérés.

**ENOMEM** Nincs elegendő memória a művelet végrehajtására.

**41. példa.** A következő példaprogram egy démont mutat be, amelyet a `daemon()` függvény segítségével készítettünk el.

```
folyamatok/demon.c
```

```

1 #include <stdlib.h>
2 #include <unistd.h>
3
4 main(){
5     daemon( 0, 1 );
6     while( 1 ){
7         printf("A démon fut.\n");
8         sleep(15);
9     }/*while*/
10 }/*main*/

```

A program az 5. sorban hívja a `daemon()` függvényt. Mivel a második paraméter értéke nem 0, a függvény nem irányítja át a szabványos csatornákat a /dev/null állományba.

Miután a `daemon()` függvényt meghívottuk, a program a 6–9. sorokban található végtelen ciklust hajtja végre. A ciklusban a program kiírja a szabványos kimenetre az üzenetet, miszerint `fut`, majd várakozik 15 másodpercet és újra kezdi a ciklust. Természetesen nem szerencsés, ha egy démon a szabványos kimenetre írja az üzeneteit, de a példaprogram egyszerűsége érdekében ezt a megoldást választottuk.

Ha a felhasználó elindítja a programot, akkor azonnal visszakapja a parancskérő jelet – hiszen a program, amelyet indított, szinte azonnal ki is lép – és 15 másodpercenként egy üzenetet kap a képernyőre.

Ha az indítást végező felhasználó kilép, a démon tovább fut. Ekkor persze a program kísérletei az üzenet kiíratására kudarcra vannak ítélezve, de tovább fut.

## Naplóbejegyzések elhelyezése

A Unix rendszerek naplót vezetnek a működés során bekövetkezett fontosabb eseményekről. Erre folyamatosan üzemelő, többek által használt rendszer esetében mindenképpen szükség is van.

Ha fel akarjuk készíteni programjainkat naplóbejegyzések elhelyezésére, a C könyvtár néhány függvénye siet a segítségünkre. Ezeknek a függvényeknek és állandóknak a használatához a `syslog.h` fejállomány betöltése szükséges.

```
void openlog(const char *név, int kapcsolók, int szolgáltatás);
```

A függvény kapcsolatot létesít a naplóbejegyzések kezelő alrendszerrel, és beállítja a folyamat által küldött naplóbejegyzések néhány alapvető tulajdon-ságát. A függvényt nem kötelező hívni a naplóbejegyzések elhelyezése előtt, de ha nem hívjuk, a naplóbejegyzések alaptulajdonságait a rendszer határozza meg.

A függvény első paramétere egy nevet jelöl, amely minden naplóbejegyzésbe automatikusan bekerül. Itt általában a programunk nevét adjuk meg.

A függvény második paramétere azt határozza meg, hogy a rendszernaplóval felépített kapcsolat milyen jellegű legyen. Itt a következő állandókat vagy a belőlük *vagy* kapcsolattal készített kifejezéket adhatjuk meg:

`LOG_PERROR` Ha ez a kapcsoló be van kapcsolva, a naplóbejegyzések a pro-gram szabványos hibacsatornáján is megjelennek.

`LOG_CONS` Ha ez a kapcsoló be van kapcsolva, azok az üzenetek, amelyek hiba miatt nem jutnak el a rendszernaplóba, megjelennek a konzolon.

`LOG_PID` Ha ez a kapcsoló be van kapcsolva, a rendszer a naplóbejegyzésben elhelyezi a folyamat azonosítóját (PID) is.

A függvény harmadik paramétere azt jelzi a rendszernaplót kezelő alrendszer számára, hogy az üzenet milyen szolgáltatást végző programtól ered. Itt általában azt adjuk meg, hogy a programunk milyen jellegű feladatot lát el. A rendszernaplót kezelő alrendszer ez alapján az állandó alapján dönti el, hogy hol helyezze el a naplóbejegyzést.

A következő állandók használhatók harmadik paraméterként:

`LOG_AUTHPRIV` Biztonsági, a felhasználók azonosításával kapcsolatos szolgál-tatások.

`LOG_CRON` Időzített, késleltetett feladatvégrehajtással kapcsolatos szolgáltatások.

`LOG_DAEMON` Rendszerdémonok.

`LOG_FTP` FTP szolgáltatást végző programok.

LOG\_KERN A rendszermag üzenetei használják ezt az állandót, felhasználói programokban általában nem használjuk.

LOG\_LOCALn Egyéb témakörökbe be nem sorolható, jellegzetesen helyi feladatok elvégzését végző programok használják ezeket az állandókat. Az n helyén egy 0 és 7 közti számjegy állhat.

LOG\_LPR A nyomtatással kapcsolatos feladatokat ellátó programok használják ezt az állandót.

LOG\_MAIL Az elektronikus levelezést kezelő rendszerprogramok számára fenntartott állandó.

LOG\_NEWS Az internetes hírcsoportok kezelését végző rendszerprogramok használják ezt az állandót.

LOG\_SYSLOG A naplózással kapcsolatos feladatokat ellátó programok – általában a rendszernaplót kezelő alrendszer – használják ezt az állandót.

LOG\_USER A felhasználók által elhelyezett naplóbejegyzések számára fenntartott állandó. (A felhasználók a logger program segítségével helyezhetnek el naplóbejegyzéseket tetszőleges szöveggel.)

LOG\_UUCP Az UUCP kezelését ellátó alrendszer számára fenntartott állandó.

Az openlog() függvénynek nincs visszatérési értéke.

```
void syslog(int fontosság, char *formátum, ...);
```

A függvény segítségével naplóbejegyzést helyezhetünk el a rendszernaplóban. A függvény a bejegyzést a naplót kezelő alrendszer felé továbbítja.

A függvény első paramétere a naplóbejegyzés fontosságát határozza meg. Itt a következő állandókat helyezhetjük el:

LOG\_EMERG A rendszer összeomlott, az adatok elvesztek, a gép lángokban áll.

LOG\_ALERT Vörös riadó, azonnali beavatkozásra van szükség.

LOG\_CRIT Sárga riadó, kritikus helyzet.

LOG\_ERR Hiba lépett fel a rendszer működésében.

LOG\_WARNING Figyelmeztetés rendellenes működésre vagy annak veszélyére.

LOG\_NOTICE A működés megfelelő, de említésre méltó esemény történt.

LOG\_INFO Tájékoztató jellegű információ.

LOG\_DEBUG Lényegtelen apróság, ami csak akkor fontos, ha a rendszer átvizsgálásával vagyunk elfoglalva.

Az első paraméterként megadott állandóval együtt megadható az openlog() harmadik paramétereként átadható témakörök valamelyike is, a vagy művelettel. Ez hasznos lehet, ha programunk többféle témakörben is szeretne naplóbejegyzéseket elhelyezni.

A függvény második és további paraméterei a naplóbejegyzés szövegét határozzák meg. A paraméterek értelmezése a printf() függvénynél megszokott formában történik. A második paraméter a formázószöveg, a további paraméterek pedig olyan értékek, amelyeknek írási formáját a formázószöveg határozza meg.

A függvény nem ad vissza értéket.

```
void closelog(void);
```

A függvény zárja a naplózó alrendszerrel felépített kapcsolatot. A függvény használata nem kötelező.

**42. példa.** A következő példaprogram elhelyez egy naplóbejegyzést a rendszernaplóban, majd kilép.

	naplo/openlogsyslog.c
--	-----------------------

```

1 #include <syslog.h>
2
3 main(){
4     openlog( "Próba", LOG_PERROR|LOG_PID, LOG_AUTHPRIV );
5     syslog( LOG_EMERG, "Kísérleti bejegyzés\n" );
6     closelog();
7 }/*main*/

```

A program futtatáskor a szabványos kimenetre is kiírja az üzenetet, ahogyan a következő példa is mutatja:

```
Bash$ ./openlogsyslog
Próba[2588]: Kísérleti bejegyzés
Bash$
```

## Visszaugrás függvényből

Az itt tárgyalt setjmp(), longjmp() függvények kétségtelenül a C könyvtár legkülönösebb függvényei, amelyek lehetővé teszik, hogy a program végrehajtását egy másik függvényben folytassuk.

E két függvényre akkor lehet szükségünk, ha jelzskezelő (*signal handler*) függvényt szeretnénk írni. Ha nem akarunk jelzskezelőt készíteni a programunkhoz, valószínűleg nem lesz szükségünk ezekre a függvényekre.

A függvénykből valamelyik hívó függvénybe a következő két függvény segítségével ugorhatunk vissza. Ezeknek a függvényeknek a használatához a setjmp.h fejlomány betöltése szükséges.

```
int setjmp(jmp_buf körny);
```

A függvény segítségével menthetjük a program állapotát, hogy ide, a `setjmp()` hívásának helyére visszaugorhassunk. Fontos tudnunk, hogy ha a függvény, amelyből a `setjmp()` függvényt hívtuk, befejezi futását, a mentett programállapot érvényét veszti. A `setjmp()`, `longjmp()` segítségével tehát *nem ugorhatunk vissza olyan függvénybe, amely már befejezte futását.*

A függvény paramétere az a változó, ahová a program állapotát menteni akarjuk.

A függvény visszatérési értéke 0, ha a függvényt azért hajtotta végre a programunk, hogy mentsük az állapotot. Ha a függvény végrehajtására azért kerül sor, mert visszaugrottunk valahonnan az adott helyre, a visszatérési érték nem 0.

```
void longjmp(jmp_buf körny, int viasz);
```

A függvény segítségével egy előzőleg mentett állapot alapján, annak helyére ugorhatunk vissza.

A függvény első paramétere a `setjmp()` függvény segítségével mentett állapotot tartalmazó változó. Ez határozza meg, hol folytatódjon a program végrehajtása. A vezérlés arra a pontra kerül, ahol az első paraméterként átadott változót a `setjmp()` függvénnel előkészítettük.

A függvény második paramétere meghatározza, mi legyen a visszatérési értéke az ugrás után a `setjmp()` függvénynek. Visszaugrás után a `setjmp()` visszatérési értéke nem lehet 0; ha a `longjmp()` második paramétere 0, a `setjmp()` visszatérési értéke 1 lesz.

A függvénynek nincs visszatérési értéke, mert a `longjmp()` soha nem tér vissza. A *program futása nem a függvény hívása utáni utasítással folytatódik.*

**43. példa.** A következő példaprogram bemutatja, hogyan ugorhatunk vissza függvényből a `longjmp()` függvény segítségével.

<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;setjmp.h&gt; 3 #include &lt;unistd.h&gt; 4 5 jmp_buf címke; 6 7 int elso( void ){ 8     return( masodik(1) ); 9 }/*elso*/ 10 11 int masodik( int a ){ </pre>	<span style="border: 1px solid black; padding: 2px;">jelzesek/setjumplongjmp.c</span>
--	---

```

12     longjmp( cimke, 1 );
13 }/*masodik*/
14
15 main(){
16     if( setjmp(cimke)==0 )
17         printf( "Első áthaladás.\n" );
18     else
19         printf( "Kiugrott.\n" );
20
21     sleep(1);
22     printf("Visszatérési érték: %d\n", elso() );
23 }/*main*/

```

A program a 16. sorban elhelyezett `setjmp()` függvény segítségével beállítja a `cimke` nevű változót. A 16. sor lesz az a pont, ahová a program vezérlése kerül a függvényből való kiugrás után. A 16. sorban megvizsgáljuk, hogy mi a `setjmp()` visszatérési értéke és annak megfelelően kiírunk egy üzenetet a szabványos kimenetre.

A program a 22. sorban hívja az `elso()` nevű függvényt, amely a 8. sorban hívja a `masodik()` függvényt. A vezérlés azonban nem a normális úton kerül vissza a hívóhoz, mert a `masodik()` függvényben a 12. sorban egy `longjmp()` hívással visszatérünk a 16. sorba.

Láthatjuk, hogy a program valójában egy végtelen ciklust tartalmaz, amelynek megfelelően minden visszatér a 16. sorba. A program futása ennek megfelelően csak a felhasználó közbeavatkozására szakad meg:

```
Bash$ ./setjumplongjmp
Első áthaladás.
Kiugrott.
Kiugrott.
Kiugrott. [Ctrl]+C
```

Bash\$

## Jelzések

A Unix rendszerek jelzésekét (*signal*) küldenek a folyamatoknak, ha valamilyen különleges esemény következik be, amiről a folyamatnak tudnia kell és lehetővé teszik a folyamatok számára is, hogy jelzéseket küldjenek egymásnak. A felhasználó a `kill` parancs segítségével küldhet jelzést a folyamatai számára.

A folyamatok sorsa szempontjából a jelzéseket három csoportba sorolhatjuk. Az első csoportba tartoznak azok a jelzések, amelyek mindenkorban a folyamat meg-

szakítását okozzák. Ilyen jelzés például a `kill` parancs -9 kapcsolójával küldhető jelzés.

A második csoportba tartoznak azok a jelzések, amelyeket a folyamat fogadhat vagy tilthat annak érdekében, hogy a futását ne szakítsák meg. Az ilyen jelzések nem szakítják meg a folyamat futását, ha az előzőleg úgy rendelkezett. Ilyen jellegű jelzés a futó programnak a `[Ctrl]+[C]` billentyűkombinációval küldhető jelzés.

A harmadik csoportba tartoznak azok a jelzések, amelyeket ha nem fogad a folyamat, akkor is tovább futhat. Az ilyen jelzések miatt a rendszer soha nem szakítja meg a folyamat futását.

Érdekes módon a jelzések nem hordoznak a szó hétköznapi értelmében vett üzenetet, vagyis nincs adat, ami a jelzésbe lenne csomagolva. A jelzésnek csupán típusa van, az információt a pusztta megjelenése hordozza. A folyamat csupán azt tudja meg a jelzés megjelenéséből, hogy a jelzés típusa által jelzett esemény bekövetkezett, annak körülményeiről más forrásból kell tájékozódnia.

A jelzések fogadására Unix rendszereken visszahívható függvényeken (*callback function*) keresztül van mód. A visszahívható függvények olyan C nyelven megírt függvények, amelyeket a felhasználó készít el és helyez el a programban, de a rendszer hív meg.

A felhasználó tehát elkészíti a visszahívható függvényt és elhelyezi a programban, majd értesíti a rendszert, hogy az adott típusú esemény bekövetkezéskor ezt a függvényt kell meghívnia. A felhasználó a bejelentést a visszahívható függvény címének átadásával teheti meg. Erre a cérla a C könyvtár ad eszközt, amelyet ebben a részben mutatunk be.

Mivel a jelzskezelő visszahívható függvényt a rendszer hívja meg, annak típusa kötött. A jelzés fogadására használható függvények típusa – a jelzés típusától függetlenül – a következő:

```
void név( int arg )
```

Minden visszahívható függvénynek, amely jelzéseket fogad, a bemutatott típusúnak kell lennie. A függvénynek átadott paraméter mindenig a jelzés típusa, ami a hívást kiváltotta.

A jelzést fogadni tehát nem jelent többet, mint a bemutatott típusú függvény elkezítése és a rendszeren való bejegyeztetése mint az adott folyamat adott típusú jelzésének kezelője. Jelzskezelő függvényt írni azonban nem egyszerű feladat, ezért a jelzskezelés támogatására használható C könyvtábeli függvények bemutatása után visszatérünk erre a kérdésre.

A GNU C könyvtár a következő – a `signal.h` fejállományban létrehozott – függvényeket biztosítja a jelzések fogadására és küldésére:

```
sighandler_t signal(int jelzés, sighandler_t függvény);
```

A függvény jelzskezelő függvényt vesz nyilvántartásba a hívó folyamat számára.

A függvény első paramétere annak a jelzésnek a száma, amelyet a második paraméterként átadott című függvény kezel.

A `signal()` függvény lefutása után a Linux az adott számú jelzést átadja a függvénynek, azaz a jelzés jelentkezése esetén meghívja a jelzéskezelő függvényt.

Ha a függvény második paramétere a `SIG_DFL`, a függvény törli a nyilvántartásból az adott jelzéstípushoz tartozó jelzéskezelő függvényt.

Ha a függvény második paramétere a `SIG_IGN`, a függvény arra szólítja fel a rendszert, hogy az adott jelzéstípust hagyja figyelmen kívül, az adott típusú jelzéseket semmisítse meg.

```
int raise(int jelzés);
```

A függvény segítségével jelzéseket küldhetünk a saját folyamatnak, annak a folyamatnak, amelyik a függvényt meghívja.

A függvény paramétere a jelzés, amelyet a program önmagának küld.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, illetve nem nulla, ha hiba lépett fel a művelet végrehajtása közben. Mivel a folyamat minden jogosult arra, hogy önmagának jelzést küldjön, csak a jelzés hibás értéke okozhat problémát.

```
int kill(pid_t PID, int jelzés);
```

A függvény jelzést küld egy folyamatnak.

A függvény első paramétere a folyamat folyamatazonosítója, amelynek a jelzést küldeni szeretnénk, a második pedig a jelzés.

A függvény visszatérési értéke 0, ha a jelzés küldése megtörtént, és -1, ha hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EINVAL** A függvény második paramétereként megadott jelzásszám érvénytelen értéket képvisel.

**EPERM** A folyamatnak nincs joga jelzést küldeni az első paraméterként átadott folyamatazonosítóval jelzett folyamatnak.

**ESRCH** Az első paraméterrel meghatározott folyamat nem létezik.

```
void psignal(int jelzés, const char *szoveg);
```

A függvény kiírja a második paramétere által jelzett szöveget a szabványos hi-bacsatornára. A szöveget egy kettőspont követi, amely után az első paraméter által jelzett jelzés szöveges leírása következik.

A jelzéseket nem csak fogadhatjuk, hanem vissza is tarthatjuk. A folyamat beálíthatja, hogy bizonyos jelzések ne jussanak el hozzá, amíg a visszatartást fel nem oldja.

A Linux a jelzés beérkezésekor automatikusan visszatartásra jelöli ki az adott jeléstípust, a jelzéskezelő függvény lefutása után – a függvény végrehajtása vagy a `return` utasítás után – pedig feloldja a visszatartást. Ez azért hasznos, mert így a jelzéskezelő függvény futása alatt a függvény nem hívódhat újra az újabb jelzés miatt.

A jelzések visszatartását és engedélyezését jelzéscsoportok segítségével végezhetjük el. A jelzéscsoportok a `signal.h` fejállományban létrehozott `sigset_t` típusban helyezhetők el és arra szolgálnak, hogy egyszerre több – akár az összes – jeléstípusra hivatkozzunk a segítségükkel.

A GNU C könyvtár a következő – a `signal.h` fejállományban létrehozott – függvényeket biztosítja jelzéscsoportok kezelésére, valamint a jelzések visszatartására és újraengedélyezésére:

```
int sigemptyset(sigset_t *csoporthz);
```

A függvény segítségével egy új jelzéscsoporthz alapbeállítását végezhetjük el.

A függvény első paramétere a jelzéscsoporthz jelöli, amelynek az alapbeállítását el kívánjuk végezni. Az alapbeállítást úgy végzi el a függvény, hogy a jelzéscsoporthz egyetlen jeléstípust se tartalmazzon a függvény lefutása után.

Soha ne használunk olyan jelzéscsoporthzat, amelynek alapbeállítását a `sigemptyset()` vagy a `sigfillset()` függvények valamelyikével nem végeztük el.

A függvény visszatérési értéke mindenkor 0.

```
int sigfillset(sigset_t *csoporthz);
```

A függvény segítségével egy új jelzéscsoporthz alapbeállítást végezhetjük el.

A függvény első paramétere a jelzéscsoporthz jelöli, amelynek az alapbeállítását el kívánjuk végezni. Az alapbeállítást úgy végzi el a függvény, hogy a jelzéscsoporthz minden jeléstípust szerepeljen a függvény lefutása után.

Soha ne használunk olyan jelzéscsoporthzat, amelynek alapbeállítását a `sigemptyset()` vagy a `sigfillset()` függvények valamelyikével nem végeztük el.

A függvény visszatérési értéke mindenkor 0.

```
int sigaddset(sigset_t *csoporthz, int jelzés);
```

A függvény segítségével újabb jeléstípust adhatunk az jelzéscsoporthz.

A függvény első paramétere azt a jelzéscsoporthz jelöli, amelyhez újabb jelzést szeretnénk adni, a második paramétere pedig a csoporthz adandó jeléstípust.

A függvény visszatérési értéke 0, ha a művelet sikeresen lezajlott, és -1, ha hiba lépett fel.

```
int sigdelset(sigset_t *csoporthz, int jelzés);
```

A függvény segítségével jelzéstípust távolíthatunk el a jelzéscsoportból.

A függvény első paramétere azt a jelzéscsoportot jelöli, amelyből a jelzést el szeretnénk távolítani, a második paramétere pedig az eltávolítandó jelzéstípus.

A függvény visszatérési értéke 0, ha a művelet sikeresen lezajlott, és -1, ha hiba lépett fel.

```
int sigprocmask(int parancs, const sigset_t *csoporthz, sigset_t  
*régicsoporthz);
```

A függvény segítségével lekérdezhetjük és módosíthatjuk azoknak a jelzéstípusoknak a halmozát, amelyeket a rendszer a folyamat számára visszatart.

A függvény első paramétere azt határozza meg, hogy milyen műveletet kívánunk elvégezni a jelzéshalmazon, amely meghatározza, mely jelzéstípusokat tart vissza a rendszer a folyamat számára. Az első paraméter a következő állandók egyike lehet:

**SIG\_BLOCK** A függvény második paramétereként jelölt jelzéscsoport elemei hozzá lesznek adva a visszatartandó jelzések listájához, azaz a függvény hívása után a második paraméter által előírt jelzéstípusú jelzések is vissza lesznek tartva a folyamat számára.

**SIG\_UNBLOCK** A függvény második paramétere által jelölt jelzéscsoport elemei el lesznek távolítva a visszatartandó jelzések listájából, azaz a függvény hívása után a második paraméter által előírt típusú jelzéseket a függvény megkapja. A függvény hívása után a felszabadított típussal várakozó jelzéseket is megkapja a folyamat.

**SIG\_SETMASK** A függvény hívása után pontosan azok – és csak azok – a jelzések lesznek visszatartva, amelyek a második paraméterként jelölt jelzéscsoportban megtalálhatók.

A függvény második paramétere a jelzéscsoportot jelöli, amellyel az adott módosítást el szeretnénk érni a visszatartandó jelzések listájával. Ha ez a paraméter NULL, a visszatartandó jelzések listája nem változik. Ezt a módszert akkor használhatjuk, ha a függvény segítségével nem módosítani, csak lekérdezni szeretnénk.

A függvény harmadik paramétere egy területet jelöl a memóriában, ahová a függvény a változtatás előtti állapotot helyezi. Ha nem akarjuk megtudni, hogy mit tartalmazott a visszatartandó jelzések listája a változtatás előtt, a harmadik paraméter legyen NULL.

## Jelzéstípusok

A következő oldalakon sorra veszünk néhány jelzést, amelyeket a rendszer a folyamatoknak küldhet. A jelzéseket a `signal.h` fejállományban létrehozott állandókkal jelöljük, amelyek mindegyike int típusú értéket ad. Ezek az állandók igen hasznosak, hiszen a programot olvashatóvá és hordozhatóvá teszik.

**SIGFPE** A program végzetes matematikai hibát okozott, amilyen például a nullával való osztás vagy a túlcordulás.

**SIGILL** A program nem megengedett utasítást próbált meg végrehajtani. Mivel a C fordító nem készít érvénytelen utasításokból álló programot, ez a jelzés általában a verem sérülését, hibás, függvényt jelző mutatót vagy a memóriában található programkód sérülését jelzi.

**SIGSEGV** A jelzés nem megengedett memóriaelérési kísérletet (szegmentációs hibát) jelez, ami általában hibás mutatókezelés vagy helytelen tömbhasználat miatt alakul ki.

Valójában a szegmentációs hiba olyan általános hiba, amelyet a kezdő programozó általában igen nehezen derít fel.

**SIGBUS** Ez a jelzés olyan memóriaterület elérésére tett kísérletet jelez, amely nincs jelen a rendszerben vagy érvénytelen az adott architektúrán. A hiba oka ugyanaz lehet, mint a SIGSEGV jelzésé, vagyis általában a mutatók helytelen kezelése miatt alakul ki ilyen helyzet.

**SIGTERM** Az operációs rendszer által a programnak küldött jelzés, amely kilépésre szólítja fel a folyamatot. A folyamat ezt a jelzést figyelmen kívül hagyhatja, az üzenetkezelő függvény a `return` utasítással a folyamatot folytathatja.

A `kill` program alapértelmezés szerint (kapcsoló nélkül) ezt az üzenetet küldi.

**SIGINT** Az üzenetet a folyamat megszakítása előtt kapja, akkor, amikor a felhasználó a **[Ctrl]+[C]** billentyűkombinációt használja a program megszakítására. A folyamat ezt az üzenetet figyelmen kívül hagyhatja, az üzenetkezelő függvény a `return` utasítással visszatérve a folyamatot folytathatja.

**SIGKILL** A SIGKILL üzenet küldése a folyamat azonnali megszakítását eredményezi. Az ilyen üzenetek nem kezelhetők, nem hagyhatók figyelmen kívül. Az ilyen üzenetek minden megszakítják a folyamat futását, ezért csak külön kérésre – a felhasználó kérésére – küld ilyen üzenetet a rendszer a folyamatnak.

A `kill` program ezt az üzenetet küldi a -9 kapcsoló hatására.

**SIGHUP** Az üzenet arról értesíti a folyamatot, hogy a futtatást végző felhasználó kilépett, vagy más okból nem használható a konzol a továbbiakban.

A démonoknak általában ezzel az üzenettel jelezhetjük, hogy a beállítóállományok megváltoztak, azokat újra kell olvasni és a megfelelő módon meg kell változtatni a működést.

**SIGALRM** Ez az üzenet azt jelzi a folyamatnak, hogy az általa kért ébresztésnek eljött az ideje. A folyamat az `alarm()` vagy a `setitimer()` függvények valamelyikével kérhet ébresztést.

**SIGCHLD** Ez az üzenet jelzi a folyamat számára, hogy valamelyik gyermekfolyamata befejezte a futását. Ekkor általában a `waitpid()` függvényt kell hívunk.

## Jelzskezelők készítése

A jelzskezelő függvények elkészítése meglehetősen összetett feladat. Az ilyen függvények készítésekor sok olyan dologra kell ügyelnünk, ami a program egyéb részeinek elkészítésekor egyáltalán nem fontos, ezért a jelzskezelő függvények készítéséről kissé részletesebben kell szólnunk.

A jelzskezelő függvény elkészítésekor háromféle stratégia közül választunk, amelyek a következők:

1. Rendelkezhetünk úgy, hogy a jelzskezelő függvény végrehajtása után a folyamat tovább fussen. Ekkor a jelzskezelő függvényből a `return` utasítással kell visszatérnünk. Ha ezt a módszert választjuk, néhány fontos szempontot figyelembe kell vennünk:
  - A legtöbb esetben a jelzskezelő függvényből való visszatérés a folyamat folytatását eredményezi. Ha éppen egy műveletet, egy függvényhívást hajt végre a programunk, azt ott folytatja, ahol a jelzsérkezés a folyamat futását megszakította.

Néhány, a C könyvtárban található függvény esetébe, azonban ez nem így van. Ha ezen függvények végrehajtása közben jelzsérkezik, a függvények végrehajtása megszakad, nem folytatódik. Ilyen függvények például az `open()`, a `close()`, a `write()` vagy a `read()`, amelyek az `errno` változót `EINVAL` értékre állítják és „félbeszakítják” a művelet végrehajtását, ha jelzsérkezik a működésük során.

Ha tehát jelzskezelő függvényt készítünk, ekkor esetleg gondoskodnunk kell arról, hogy minden, jelzserek által megszakítható függvényt a megfelelő módon újraírunk, ha éppen akkor érkezne jelzsérkezés, amikor futnak. (Ha nincs jelzskezelő függvény, a megszakítható függvények futását akkor is megszakítja a beérkezett jelzsérkezés, de akkor általában a folyamat futása is megszakad, ezért nem kell ismételnünk a függvényhívást.)

A 44. példa bemutatja, hogyan szakítja meg a felhasználó által létrehozott jelzés a sleep() függvény végrehajtását a paraméterként átadott idő letelte előtt.

- A kritikus hiba által kiváltott *jelzések* azt jelzik, hogy a programfutás nem folytatható. Az ilyen jelzéseket kezelő függvények nem térhetnek vissza a return utasítással, hiszen az a folyamat folytatását célozná.
- Igen fontos ügyelnünk a jelzskezelő elkészítése során, hogy ne használunk olyan függvényeket, amelyek adott memóriaterületre helyezik el az adataikat.

A readdir() függvény például egy bizonyos memóriaterületen helyezi el a könyvtárból olvasott adatokat. Ez a memóriaterület a *függvény ismételt hívásakor felülíródik*, az ott tárolt adatok elvesznek. Ha éppen az után hívódik meg a jelzskezelő, hogy a readdir() függvényt meghívta a program, a jelzskezelőben hívott readdir() felülírja az adatterületet, amelyben a program számára fontos adatok vannak.

Láthatjuk, hogy bizonyos függvények hívásától mindenkorban őrizkedünk kell a jelzskezelő írása során.

2. Rendelkezhetünk úgy is, hogy a jelzskezelő megszakítsa a program futását. Ehhez a módszerhez általában akkor folyamodunk, ha kritikus hiba lépett fel és a jelzskezelő függvényben – az esetleges adminisztratív teendők elvégzése után – gondoskodni szeretnénk a kilépésről.

Ekkor azonban nem szerencsés, ha a kilépésre a szokásos módot – az exit() függvény hívását – választjuk, mert akkor a szülőfolyamat nem értesül arról, hogy a programfutást jelzs szakította meg. Ha a jelzskezelő függvény kritikus hiba miatt meg akarja szakítani a folyamat futását, először törölne kell a rendszer nyilvántartásából az adott jelzs kezelésére bejegyzett függvényt, majd a saját folyamatnak újra el kell küldenie a jelzést. Ekkor a jelzs – mivel már nincs jelzskezelő – a folyamat megszakítását eredményezi és így a szülőfolyamat értesülhet arról, hogy a programfutást az adott jelzs okozta.

3. Dönthetünk úgy, hogy a jelzs beérkezésekor a program egy bizonyos pontról folytatódjon tovább. Ekkor a setjmp() és longjmp() függvények használatával kell a jelzskezelő függvényből kiugranunk.

Ha ezt a módszert használjuk, igen fontos szem előtt tartani, hogy a longjmp() függvénytel nem léphetünk olyan függvénybe, amely már befejezte futását.

**44. példa.** A következő példaprogram jelzskezelést használ a felhasználó által külödt jelzések fogadására. A jelzskezelő függvény miatt a program a futását a jelzs beérkezése után is folytatja.

```

1 #include <stdlib.h>
2 #include <signal.h>
3
4 void kezelo( int uzenet ){
5     printf("Akkor mi van?\n");
6 }
7
8 main(){
9     signal( SIGINT, kezelo );
10
11    while( 1 ){
12        printf("Futok.\n");
13        sleep(10);
14    }/*while*/
15 }/*main*/

```

A program a 4–6. sorban egy függvényt hoz létre, amely nem tesz semmit, csak kiír egy jelzést a szabványos kimenetre. A program a 9. sorban bejegyzi a függvényt a SIGINT jelzés kezelésére. A program a továbbiakban a 11–14. sorban egy végtelen ciklusban várakozik, néhány másodpercenként üzenetet küldve a felhasználónak.

Ha a programot lefordítjuk és futtatjuk, az érzéketlen lesz a felhasználó megszakítási kísérleteire, bár jelzi, hogy a jelzés kezelésére szolgáló függvény lefut.

```

Bash$ ./a.out
Futok. [Ctrl]+C
Akkor mi van?
Futok. [Ctrl]+C
Akkor mi van?
Futok. [Ctrl]+Z

[1]+  Stopped                  ./a.out
Bash$

```

Amint látjuk, a program felfüggeszthető volt a [Ctrl]+[Z] billentyűkombinációval.

A program futtatása során megfigyelhetjük, hogy a felhasználó által létrehozott jelzés megszakítja a sleep() függvény futását a kért idő letelte előtt. Amikor a felhasználó lenyomja a [Ctrl]+[C] billentyűkombinációt, a program a jelzskezelő végrehajtása után nem folytatja a sleep() végrehajtását.

**45. példa.** A következő példa azt mutatja be, hogyan szakíthatjuk meg a program futását a jelzés hatására a jelzskezelő függvényből. Az itt bemutatott módszer lényeges előnye, hogy a folyamat szülfőfolyamata a folyamat megszakadásának igazi okát – a jelzést – megismerheti.

```

1   jelzesek/jelzeskezelokilep.c
2
3
4 #include <stdio.h>
5 #include <signal.h>
6
7 void kezelo( int uzenet ){
8     printf("Na jó, akkor kilépek... \n");
9     fflush( stdout );
10    signal( uzenet, SIG_DFL );
11    raise( uzenet );
12 }/*kezelo*/
13
14 main(){
15     signal( SIGINT, kezelo );
16
17     while( 1 ){
18         printf("Futok. \n");
19         sleep(10);
20     }/*while*/
21 }/*main*/

```

*Ha a program futása közben megnyomjuk a billentyűkombinációt, a program kilep, de előtte egy üzenetet ír a szabványos kimenetre:*

```

Bash$ ./jelzeskezelokilep
Futok. 
Na jó, akkor kilépek...

```

Bash\$

*A program a 4–9. sorban tartalmazza a jelzskezelő függvényt, amelyet a 12. sorban veszünk nyilvántartásba. A jelzskezelő függvény indulása után az 6. sorban először kiírunk egy üzenetet, majd a 7. sorban töröljük a nyilvántartásból a jelzskezelő függvényt. A 8. sorban ezután újra elküldjük a jelzést a folyamatnak, ami – mivel nincs jelzskezelő nyilvántartva – a folyamat megszakítását vonja maga után.*

**46. példa.** A következő példaprogram azt mutatja be, hogyan lehet egy jelzés beérkezésekor a programot egy adott ponttól újraindítani. A program a setjmp() és longjmp() függvényeket használja.

```

1   jelzesek/jelzeskezeloujra.c
2
3
4 #include <stdio.h>
5 #include <signal.h>
6 #include <setjmp.h>
7
8 jmp_buf eleje;

```

```

6   void kezelo( int uzenet ){
7     sigset_t uzenetek;
8
9
10    sigemptyset( &uzenetek );
11    sigaddset ( &uzenetek, SIGINT );
12    sigprocmask( SIG_UNBLOCK, &uzenetek, NULL );
13
14    printf("Na jó, akkor újra.\n");
15    longjmp( eleje, 1 );
16 }/*kezelo*/
17
18 main(){
19   int n;
20   signal( SIGINT, kezelo );
21
22   if( setjmp(eleje) != 0 )
23     printf("Épp most indulok újra.");
24
25   for( n=1; n<15; ++n ){
26     printf("%d ", n);
27     fflush( stdout );
28     sleep(1);
29   }/*for*/
30 }/*main*/

```

A program 14 másodpercen keresztül számol, 14 számot ír a szabványos kimenetre. Ha a felhasználó közben lenyomja a **[Ctrl]+[C]** billentyűkombinációt, a program makacsul újrakezdi a számolást. Ezt mutatja be a program egy lehetséges futása:

```

Bash$ ./jelzeskezelouja
1 2 3 [Ctrl]+[C] Na jó, akkor újra.
Épp most indulok újra.1 2 3 4 [Ctrl]+[C] Na jó, akkor újra.
Épp most indulok újra.1 2 3 4 5 6 7 8 9 10 11 12 13 14
Bash$

```

## Hálózati kapcsolatok kezelése

Igen izgalmas és hasznos szolgáltatása a C könyvtárnak a hálózati kapcsolatok, a hálózati adatátvitel támogatása. Ez a szolgáltatás talán a legösszetettebb, de szerencsére már viszonylag kevés ismerettel is igénybe vehető.

Mi e fejezetben a foglalatok (*socket*), kezelésének alapjait mutatjuk be. A foglalat kétirányú adatátvitelt tesz lehetővé, amelynek felhasználásával adatokat cserélhetünk számítógéphálózaton keresztül.

A foglalatok igen hasonlítanak az állományleírókhoz, amelyeket az `open()` függvény segítségével hozhatunk létre és állományok olvasására, írására használhatunk. Az állományleírókhoz hasonlóan a foglalatokba is írhatunk a `write()` függvénytel, a foglalatokból is olvashatunk a `read()` függvénytel, sőt a foglalatokat is lezárhatjuk a `close()` függvénytel. A különbség a foglalatok és az állományleírók között csak annyi, hogy másképpen nyitjuk meg őket, ezért alkalmasak két gép közti adatcserére.

Tudunk kell, hogy a Linux többféle hálózati szabványt támogat, így a kapcsolat kialakításakor megadhatjuk, milyen hálózaton keresztül kívánjuk elérni a másik állomást. Mi e fejezetben csak az Internet felépítésében alapvető szerepet játszó IPv4 és IPv6 szabványokat tárgyaljuk, de a C könyvtár dokumentációja alapján más hálózatokat használó programok is készíthetők.

## A számítógép nevének lekérdezése

A következő függvény segítségével lekérdezzük a folyamatot futtató helyi számítógép hálózati nevét. A függvény használatához az `unistd.h` fejállomány betölése szükséges.

```
int gethostname(char *név, int hossz);
```

A számítógép nevét lekérdező függvény.

A függvény első paramétere a területet jelöli, ahol a függvény a helyi számítógép nevét elhelyezi, a második paramétere pedig azt adja meg, hogy ezen a területen mekkora helyet foglaltunk le.

A függvény visszatérési értéke 0, ha a művelet rendben zajlott, és -1, ha hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

`EINVAL` A második paraméter értéke nem megfelelő.

`ENAMETOOLONG` A név túlságosan hosszú, nem fér el a megadott méretű helyen.

A C könyvtár dokumentációja szerint egyes változatok csonkolják a nevet és nem adnak vissza hibajelzést, ha a név nem fér el a lefoglalt területen.

**47. példa.** A következő példaprogram a számítógép nevét kérdezi le és a szabványos kimenetre írja. A gép nevét a `gethostname()` függvénytel kérdezi le egy előre lefoglalt 128 bájt méretű tömbbe.

## halozat/gethostname1.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 main(){
6     char nev[128];
7     int hiba;
8
9     hiba=gethostname(nev, 128);
10    if( hiba != 0 ){
11        fprintf(stderr, "Sikertelen.\n");
12        exit(1);
13    }/*if*/
14
15    printf("Név: %s\n", nev);
16 }/*main*/

```

A példaprogram hiányossága, hogy csak 128 bájt hosszú nevet képes kezelni. Természetesen nagyobb méretet is beállíthatunk, de a példaprogramba bármekkora számot írunk is a tömb foglalásakor, mindenkorban korlátot építünk a programba. A következő példaprogram gyakorlatilag bármekkora hosszságú nevet képes kezelni.

## halozat/gethostname2.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <errno.h>
5
6 main(){
7     char *nev;
8     size_t hossz=16;
9     int hiba;
10
11    nev=(char *)malloc(8);
12
13    while( gethostname(nev, hossz)==-1 &&
14          errno==ENAMETOOLONG ) {
15        hossz*=2;
16        nev=(char *)realloc( nev, hossz );
17    }/*while*/
18
19    printf("%s\n", nev );
20 }/*main*/

```

A program a 13. sorban lekérdezi a gép nevét a `gethostname()` hívásával. Ha a gép nevének lekérdezése nem sikerült és ennek oka a túl kevés hely (a 14. sorban az `errno` értéke `ENAMETOOLONG`) a program növeli a név számára lefoglalt memória méretét (15–16. sor), majd újra megkíséri lekérni a nevet.

## Távoli számítógépek nevei és címei

Mint az az olvasó előtt már bizonyára ismeretes, az Interneten a számítógépekre címükkel és nevükkel is hivatkozhatunk. A címeket (IPv4 szabvány esetén) általában négy számjeggyel, pontokkal elválasztva írjuk – például 172.16.8.72 –, a neveket pedig pontokkal elválasztott szöveges értékként – például `gep.otthon`. A számítógéphálózat működésekor a címek szolgálnak azonosítóul, a nevek pedig az ember számára teszik könnyebbé a gépek azonosítását.

A címeket nevek, a neveket címekre az úgynevezett névfeloldási rendszer fordítja le. A névfeloldási rendszer helyi állományokat – például `/etc/hosts` –, valamint DNS (*domain name service*, körztnév szolgáltatás) kiszolgálókat használ. Az itt tárgyalt néhány függvény a névfeloldást teszi egyszerűvé számunkra, segítségükkel a C könyvtár megkíséri a helyi nyilvántartásból, valamint – ha ez nem járna eredménnyel – a DNS kiszolgáló lekérdezésével kideríteni tetszőleges névből a címet és fordítva.

A névfeloldási rendszer legfontosabb adatszerkezete a `hostent` struktúra, melynek használatához a `netdb.h` fejállomány betöltése szükséges.

A hostent struktúra	
<code>char *h_name</code>	A számítógép elsődleges neve.
<code>char **h_aliases</code>	A számítógép másodlagos nevei.
<code>int h_addrtype</code>	A számítógép IP címének típusa.
<code>int h_length</code>	A cím hossza bájtban.
<code>char **h_addr_list</code>	A számítógép IP címei.
<code>char *h_addr</code>	A számítógép elsődleges IP címe, ami ugyanaz, mint a <code>h_addr_list[0]</code> .

A hostent struktúra első eleme a számítógép elsődleges nevét tartalmazza. Egy számítógépnek, egy hálózati csatlakozási pontnak több neve is lehet, de ezek közül minden létezik egy elsődleges név, amely ide kerül. A név karaktertömbként kerül a memóriába, a `h_name` pedig ezt a területet jelöli ki.

A hostent második eleme egy tömböt jelöl a memóriában, amely az esetleges további neveket jelölő mutatókat tartalmazza. A tömb minden eleme egy karakterláncot jelöl, amely a következő nevet tartalmazza. Kivétel ez alól a tömb utolsó eleme, amely `NULL` értéket tartalmaz. Ez a `NULL` érték jelöli a tömb végét.

A hostent harmadik eleme egy állandó, amely a cím típusát adja meg. Ma a legtöbb helyen az internetes szabvány IPv4 jelölésű 4. változatát használjuk. Az ilyen

címek esetén a harmadik mezőben tárolt állandó értéke AF\_INET. A szabványcsalád legújabb változata, az IPv6 nevű 6. változat, amelynek használata esetén az állandó AF\_INET6.

A hostent struktúra negyedik eleme a cím hosszát adja meg bájtból. Erre azért van szükség, mert a jelenleg használatos 4. és 6. változatú szabványcsaládban használt címek hossza nem egyezik meg.

A hostent ötödik eleme a számítógép címeit adja meg, éppen olyan táblázatos formában, mint ahogyan a második eleme adta a neveket. Az utolsó cím után egy NULL értéket találunk a címeket tartalmazó tömbben.

Az utolsó elem az elsődleges címet tartalmazza, amely megegyezik a tömbben tárolt legelső címmel.

A hostent struktúra használatának megértését hivatott segíteni a 48. példa.

A következő két függvény a nevek címekké, valamint a címek nevekké alakítását végzi. Használatukhoz a netdb.h fejállomány betöltése szükséges.

```
struct hostent *gethostbyname(const char *név);
```

A függvény a számítógép nevének címre történő fordítására használható. A függvény a helyi adatbázisokat és az esetleges DNS kiszolgáló lekérdezését használja a feladat elvégzésére.

A függvény paramétere a nevet tartalmazó karakterláncra mutat.

A függvény visszatérési értéke a számítógép elsődleges és másodlagos neveit, elsődleges és másodlagos címeit tartalmazó hostent struktúrát jelöli a memóriában, ha a művelet sikeres volt. Sikertelen lekérdezés esetén a függvény visszatérési értéke NULL érték. Ilyen esetben a függvény beállítja az errno értékét, ami a következők egyike lehet:

**HOST\_NOT\_FOUND** A számítógép adatai nem érhetők el.

**TRY AGAIN** A DNS kiszolgáló nem volt elérhető, de a művelet ismétlés esetén eredményt adhat.

**NO\_RECOVERY** A fellépő hiba jellege miatt az ismétlés is eredménytelen lesz.

**NO\_ADDRESS** A számítógép neve megtalálható a nyilvántartásban, de nincs cím hozzárendelve.

```
struct hostent *gethostbyaddr(const char *cím, size_t hossz, int tip);
```

A függvény címek feloldását végzi, címeknek nevekre való átalakítására használható.

A függvény első paramétere a címet tartalmazza bináris formában. A második paramétere a cím hosszát adja meg. Ez az elterjedten használt IPv4 szabvány esetében 4, az újabb, IPv6 szabvány szerint pedig 16. A függvény harmadik paramétere a cím típusát adja meg, amely IPv4 esetén AF\_INET, IPv6 esetén pedig AF\_INET6.

A függvény visszatérési értéke a számítógép elsődleges és másodlagos neveit, illetve elsődleges és másodlagos címeit tartalmazó hostent struktúrát jelöli a memóriában, ha a művelet sikeres volt. Sikertelen lekérdezés esetén a függvény visszatérési értéke NULL érték. Ilyen esetben a függvény beállítja az errno értékét, ami a következők egyike lehet:

HOST\_NOT\_FOUND A számítógép adatai nem érhetők el.

TRY AGAIN A DNS kiszolgáló nem volt elérhető, de a művelet ismétlés esetén eredményt adhat.

NO\_RECOVERY A fellépő hiba jellege miatt az ismétlés is eredménytelen lesz.

NO\_ADDRESS A számítógép neve megtalálható a nyilvántartásban, de nincs cím hozzárendelve.

A névfeloldás és a hostent struktúra használatának bemutatására szolgál a következő két példa.

**48. példa.** A következő program számítógépek nevéből állapítja meg azok címeit. A program a gethostbyname() függvényt használja.

```
halozat/gethostbyname.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <netdb.h>
5
6 void hiba( int hibakod ){
7     char *szoveg;
8
9     switch(hibakod){
10         case HOST_NOT_FOUND:
11             szoveg="Ismeretlen számítógép.\n";
12             break;
13         case TRY AGAIN:
14             szoveg="Az információ ideiglenesen nem elérhető.\n";
15             break;
16         case NO_RECOVERY:
17             szoveg="Az információ nem elérhető.\n";
18             break;
19         case NO_ADDRESS:
20             szoveg="A cím nem létezik.\n";
21             break;
22         default:
23             szoveg="Ismeretlen hiba.\n";
24     }/*switch*/
```

```

25     fprintf(stderr, szoveg);
26     exit(1);
27 }/*hiba*/
28
29
30 int main(int argc, char *argv[]){
31 struct hostent *gep;
32 int n;
33
34 if( argc<2 ){
35     fprintf(stderr, "Használat: \n");
36     fprintf(stderr, "  lookup <gépnév>\n");
37     exit(1);
38 }/*if*/
39
40 gep=gethostbyname(argv[1]);
41 if( gep==NULL )
42     hiba(errno);
43
44 printf("%s\n", gep->h_name);
45
46 n=0;
47 while( gep->h_aliases[n] != NULL ){
48     printf("%s\n", gep->h_aliases[n] );
49     n++;
50 }/*while*/
51
52 n=0;
53 while( gep->h_addr_list[n] != NULL ){
54     printf("%hu.%hu.%hu.%hu\n",
55             gep->h_addr_list[n][0],
56             gep->h_addr_list[n][1],
57             gep->h_addr_list[n][2],
58             gep->h_addr_list[n][3]);
59     ++n;
60 }/*while*/
61 }/*main*/

```

*Figyeljük meg a programban a NULL értékekkel lezárt tömbök kezelését, valamint a 14–18. sorban a cím kiíratását! Ez utóbbiból jól látható, milyen óvatosan kell eljárni a címek kezelésénél.*

*A program futtatásakor a lekért számítógép minden címét és nevét megkapjuk:*

Bash\$ ./gethostbyname

*Használat:*

```
lookup <gépnév>
Bash$ ./gethostbyname localhost
localhost.localdomain
localhost
127.0.0.1
Bash$
```

**49. példa.** A következő példaprogram a 127.0.0.1 című számítógép neveit és címeit kérdezi le. Ez a példaprogram az előzőhöz képest egyszerűbb, így kevesebb szolgáltatást is nyújt.

	halozat/gethostbyaddr.c
--	-------------------------

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <netdb.h>
4
5 int main(int argc, char *argv[]){
6     struct hostent *gep;
7     char cim[4];
8     int n;
9
10    cim[0]=127; cim[1]=0; cim[2]=0; cim[3]=1;
11
12    gep=gethostbyaddr( cim, 4, AF_INET );
13    if( gep==NULL ){
14        fprintf(stderr, "A cím nem található.\n");
15        exit(1);
16    }/*if*/
17
18    printf("%s\n", gep->h_name);
19
20    n=0;
21    while( gep->h_aliases[n] != NULL ){
22        printf("%s\n", gep->h_aliases[n] );
23        n++;
24    }/*while*/
25
26    n=0;
27    while( gep->h_addr_list[n] != NULL ){
28        printf("%hu.%hu.%hu.%hu\n",
29               gep->h_addr_list[n][0],
30               gep->h_addr_list[n][1],
31               gep->h_addr_list[n][2],
```

```

32     gep->h_addr_list[n][3]);
33     ++n;
34 }/*while*/
35 }/*main*/

```

A program egy futtatásának eredménye lehet a következő:

```

Bash$ ./gethostbyaddr
localhost.localdomain
localhost
127.0.0.1
Bash$

```

## Bájtsorrend-átalakítás

A ma használt processzorok – és így a ma használt számítógépek – számábrázolása sajnos nem egyezik meg egymással. Léteznek olyan számítógépek, amelyek a magasabb helyi értékű bájtokat magasabb címeken tárolják és léteznek olyanok, amelyek alacsonyabb címeken. Annak érdekében, hogy a különféle számábrázolást használó számítógépek adatokat tudjanak cserélni a hálózaton, a hálózati szabványok rögzítik a bájtsorrendet, amelyhez minden számítógépnek igazodnia kell a hálózatos adatcsere során.

A C könyvtár biztosít néhány függvényt, amelyek a hálózaton használt bájtsorrend és a helyi gépen használt bájtsorrend között végeznek átalakításokat. Ezeknek a függvényeknek a használatához a `netinet/in.h` fejállomány betöltése szükséges.

```
uint16_t htons(uint16_t adat);
```

A függvény a helyi számítógépen használt bájtsorrendről végez átalakítást a hálózaton előírt bájtsorrendre, 16 bites adathosszon.

A függvény paramétere az átalakítandó szám, visszatérési értéke pedig az átalakított szám.

```
uint16_t ntohs(uint16_t adat);
```

A függvény a hálózaton előírt bájtsorrendről végez átalakítást a helyi számítógépen használt bájtsorrendre, 16 bites adathosszon.

A függvény paramétere az átalakítandó szám, visszatérési értéke pedig az átalakított szám.

```
uint32_t htonl(uint32_t adat);
```

A függvény a helyi számítógépen használt bájtsorrendről végez átalakítást a hálózaton előírt bájtsorrendre, 32 bites adathosszon.

A függvény paramétere az átalakítandó szám, visszatérési értéke pedig az átalakított szám.

```
uint32_t ntohs(uint32_t adat);
```

A függvény a hálózaton előírt bájtsorrendről végez átalakítást a helyi számítógépen használt bájtsorrendre, 32 bites adathosszon.

A függvény paramétere az átalakítandó szám, visszatérési értéke pedig az átalakított szám.

**50. példa.** A következő példaprogram megvizsgálja, hogy a futtató számítógép bájtsorrendje megegyezik-e, a hálózati bájtsorrenddel.

```
halozat/htons.c
1 #include <stdio.h>
2 #include <netinet/in.h>
3
4 int main( int argc, char *argv[] ){
5     uint16_t n=1;
6     if( n!=htons(n) )
7         printf( "A bájtsorrend nem egyezik meg.\n" );
8     else
9         printf( "A bájtsorrend megegyezik.\n" );
10
11     exit( 0 );
12 }/*main*/
```

A program futtatásával megtudhatjuk, hogy az Intel processzort használó személyi számítógépek bájtsorrendje nem egyezik meg az Interneten használt bájtsorrenddel.

## Foglalatok

Ha a számítógéphálózat felhasználásával adatokat akarunk küldeni egyik számítógépről a másikra, általában foglalatokat (*socket*) használunk. A foglalatokkal való kapcsolattartás során először a kiszolgáló hoz létre egy foglalatot, megnyitja azt olvasásra, majd ehhez kapcsolódik az ügyfélprogram egy saját foglalattal, amelyet ő hozott létre. Lényeges felfigyelnünk arra, hogy a kapcsolatban részt vevő számítógépek mindegyikén egy-egy foglalatot kell létrehozni az adatátvitelben részt vevő programoknak, a rendszer pedig a két foglalat között valósítja meg az adatátvitelt.

Foglalat létrehozására használható a következő függvény, amelynek használatához a *sys/types.h* és a *sys/socket.h* fejállományokat be kell töltenünk.

```
int socket(int család, int típus, int szabvány);
```

A függvény létrehoz egy foglalatot a megadott adatok alapján. A létrehozott foglalat a függvényhívás után még nem alkalmas a kapcsolattartásra, előbb ki kell építenünk a kapcsolatot a másik állomás által létrehozott foglalattal.

A függvény első paramétere a használni kívánt kapcsolat kiépítésére használt szabványcsaládot adja meg. Habár a GNU C könyvtár többféle szabványt támogat, számunkra a következő két állandó elegendő az első paraméter helyén:

**PF\_INET** A kapcsolattartásra az Internet IPv4 rendszerét kívánjuk használni.

A legtöbb esetben ezt használjuk.

**PF\_INET6** A kapcsolattartásra az Internet újabb, IPv6 rendszerét kívánjuk használni.

A függvény második paramétere a kapcsolat típusát adja meg. A C könyvtár többféle kapcsolattípust támogat, mi azonban csak kettőt tárgyalunk:

**SOCK\_STREAM** Ez a kapcsolattípus kétirányú adatátvitelt tesz lehetővé. Ilyen kapcsolat esetén a rendszer gondoskodik az adatok csomagokra bontásáról, a hibás, elveszett csomagok megismétléséről, a másik állomáson pedig az adatfolyamot újra összeállítja a fogadott csomagokból.

Az ilyen kapcsolattípusok kezelésére az Interneten a TCP (*transmission control protocol*, adatátvitelt kezelő szabvány) szabvány használatos.

**SOCK\_DGRAM** Ez a kapcsolattípus nem teszi lehetővé egy csomagnál nagyobb adatmennyiséget átvitelét, nem tartalmaz hibakezelést, csak egy-egy adatcsomag elküldéséről és fogadásáról gondoskodik.

Az ilyen kapcsolattípus kezelésére az Interneten az UDP (*user datagram protocol*, felhasználói csomag szabvány) szabvány terjedt el.

A függvény harmadik paramétere az adatátvitelhez használt szabványt adja meg. Mivel az általunk tárgyalt két szabványcsomagon belül az általunk tárgyalt két kapcsolattípusra egy-egy szabvány található (TCP és UDP), a harmadik paramétert nem kell megadnunk. A **SOCK\_STREAM** kapcsolattípus választása esetén a TCP, a **SOCK\_DGRAM** kapcsolattípus esetén pedig az UDP szabványt fogja használni a rendszer, harmadik paraméterként átadható a 0 állandó.

A függvény visszatérési értéke a létrehozott foglalatot azonosító egész szám, ha a művelet sikeres volt, és -1, ha nem. Ekkor a függvény beállítja az **errno** értékét, ami a következők egyike lehet:

**EPROTONOSUPPORT** Az adott szabványcsaládban a típus vagy a szabvány nem támogatott.

**EMFILE** A folyamat már túl sok állományt nyitott meg, nem nyithat meg többet.

**ENFILE** A rendszeren már túl sok nyitott állomány van, többet már nem lehet megnyitni.

**EACCES** A folyamatnak nincs joga a megadott adatokkal foglalatot létrehozni.

**ENOBUFS** A rendszer nem rendelkezik elegendő memóriával a feladat végrehajtására.

A `socket()` függvénnyről szóló rész végén még egyszer hangsúlyoznunk kell, hogy a függvény által visszaadott foglalat csak a csatlakoztatás után használható kapcsolattartásra.

Miután létrehoztunk egy foglalatot, azt csatlakoztatnunk kell. A csatlakozáshoz az Interneten használt szabványcsaládok a számítógépek címeit és a használt kapuk (*port*) sorszámát használják. A kapuk szám jelegű azonosítók, amelyek meghatározzák, hogy a két gép közt kiépített adatfolyamok közül melyikhez tartozik az adott adatcsomag, azt az operációs rendszernek melyik alkalmazáshoz kell továbbítania. minden egyes adatkapcsolatot egyértelműen meghatároz a csomagban továbbított küldő és fogadó számítógép címe, illetve a küldő és fogadó kapu száma.

A szakirodalomban gyakran címként (*address*) emlegetik a számítógépcímiből és kapuszámból álló szerkezetet, a foglalat csatlakoztatását pedig címhez való rendelésnek nevezik. Mi az egyértelműség kedvéért *teljes címnek* nevezük azokat az adatszerkezeteket, amelyek egy internetcím és egy kapu tárolására szolgálnak.

Az teljes cím tárolására IPv4 rendszerben a `sockaddr_in` struktúra használható, amelynek használatához a `netinet/in.h` betöltése szükséges:

A <code>sockaddr_in</code> struktúra	
<code>sa_family_t sin_family</code>	A cím típusa ( <code>AF_INET</code> ).
<code>struct in_addr sin_addr</code>	A számítógép címe.
<code>unsigned short int sin_port</code>	A kapu száma.

Amint látható, a struktúrában szerepel a internetcím típusa – ez mindig `AF_INET` –, a internetcím, valamint egy kapu sorszáma. A struktúra tehát alkalmas egy kapcsolat leírására az egyik – fogadó vagy küldő – oldalról.

A struktúra második tagja egy `in_addr` típusú struktúra, amelynek egyetlen, `s_addr` nevű eleme az IPv4 szerinti 32 bites címet tartalmazza, `uint32_t` értéken. Ennek használata során ügyelnünk kell a bájtsorrendre, a C könyvtár függvényeit a hálózati bájtsorrendre alakított címmel kell hívni.

A struktúra második tagjaként megadható az `INADDR_ANY`, amely bármely IPv4 címet jelentheti. Akkor használjuk, ha jelezni kívánjuk, hogy bárhonnan fogadunk kapcsolatkérést. Szintén használható az `INADDR_BROADCAST`, amellyel jelezhetjük, hogy a helyi hálózaton mindenkinél szóló csomagot szeretnénk küldeni.

Hasonlóképpen használható adatkapcsolat adatainak megadására IPv6 rendszerekben a `sockaddr_in6` struktúra, amelynek használatához szintén a `netinet/in.h` fejállomány betöltése szükséges:

A sockaddr_in6 struktúra	
sa_family_t sin6_family	A cím típusa (AF_INET6).
struct in6_addr sin6_addr	A számítógép címe.
uint32_t sin6_flowinfo	Használaton kívüli mező.
uint16_t sin6_port	A kapu száma.

Itt a struktúra második elemeként a in6addr\_any állandó használható annak jelzésére, hogy bármely gépről fogadunk kapcsolódást.

Az egységes használat érdekében a foglalatokat teljes címhez kapcsoló függvények nem a sockaddr\_in és a sockaddr\_in6 struktúrákat használják, hanem egy általános struktúrát, amelynek neve sockaddr. E struktúra tulajdonképpen bármely típusú teljes címet képes hordozni, így többek között a bemutatott sockaddr\_in és sockaddr\_in6 struktúrák általánosítása. A sockaddr használatához a netinet/in.h fejállomány betöltése szükséges.

A sockaddr struktúra	
short int sa_family	A cím típusa.
char sa_data[14]	A cím tárolására használt terület, melynek értelmezése függ a cím típusától.

Figyeljük meg, hogy a sockaddr struktúra mezői megfeleltethetők a sockaddr\_in és a sockaddr\_in6 struktúrának is, de azoknál általánosabb megfogalmazásúak.

A foglalat és teljes cím összekapcsolásához használhatók a következő függvények. Ezek használatához a sys/socket.h fejállomány betöltése szükséges.

```
int bind(int foglalat, struct sockaddr *cím, socklen_t hossz);
```

A függvény segítségével az előzőleg létrehozott foglalatot összekapcsolhatjuk egy, a helyi gépen található címmel. Erre általában kiszolgálóprogramok esetében van szükség, amelyek egy adott helyi kapuhoz csatlakozva várakoznak a beérkező kapcsolatokra.

A függvény első paramétere a foglalat, amelyet előzőleg létrehoztunk a socket() függvény hívásával.

A második paraméter a teljes cím, amelyhez kapcsolni szeretnénk a foglalatot. Ezt a paramétert nem sockaddr struktúraként hozzuk létre, hanem sockaddr vagy sockaddr\_in6 struktúraként.

A harmadik paraméter a második paraméterként jelzett struktúra eredeti típusának – azaz a sockaddr\_in vagy a sockaddr\_in6 típusnak – a mérete.

A függvény sikeres végrehajtás esetén 0 értéket ad vissza, hiba esetén pedig a visszatérési érték -1. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

EBADF Az első paraméterként átadott foglalat érvénytelen értéket tartalmaz.

**ENOTSOCK** Az első paraméterként átadott érték nem foglalat.

**EADDRNOTAVAIL** A második paraméter által kijelölt teljes cím nem elérhető.

**EADDRINUSE** A teljes cím már kapcsolva van valamely más foglalathoz.

**EINVAL** A foglalat már hozzá van rendelve egy címhez.

**EACCES** A hozzáférés megtagadva. Az adott címhez a folyamat nem férhet hozzá. Ennek a hibának gyakran az az oka, hogy az alsó kapukhoz (1024 alattiakhoz) csak a rendszerüzemben van hozzáférési jog.

```
int connect(int foglalat, struct sockaddr *cím, socklen_t hossz);
```

A függvény segítségével az előzőleg létrehozott foglalatot kapcsolhatjuk egy távoli gépre mutató teljes címhez. Erre általában ügyfélprogramok esetében van szükségünk, amelyek a kiszolgálóhoz kapcsolódnak e függvény hívásával.

A függvény első paramétere az előzőleg létrehozott foglalat, amelyet csatlakoztatni akarunk.

A második paraméter a teljes cím, amelyhez kapcsolni szeretnénk a foglalatot. Ezt a paramétert nem sockaddr struktúraként hozzuk létre, hanem sockaddr\_in vagy sockaddr\_in6 struktúraként.

A harmadik paraméter a második paraméterként jelzett struktúra eredeti típusának – azaz a sockaddr\_in vagy a sockaddr\_in6 típusnak – a mérete.

A függvény visszatérési értéke 0, ha a művelet hibamentesen zajlott le, illetve -1 hiba esetén. Ekkor a függvény beállítja az errno értékét, ami a következők egyike lehet:

**EBADF** Az első paraméterként átadott foglalat érvénytelen.

**ENOTSOCK** Az első paraméterként átadott érték nem foglalat.

**EADDRNOTAVAIL** A második paraméterként átadott teljes cím nem elérhető a távoli gépen.

**EAFNOSUPPORT** A második paraméterként átadott teljes cím által használt szabványcsalád ezen a gépen nem támogatott.

**EISCONN** A foglalat már kapcsolva van teljes címhez.

**ETIMEDOUT** A kapcsolat kiépítése közben a folyamat túllépte az időkorlátot, a másik állomás nem válaszol.

**ECONNREFUSED** A másik állomás visszautasította a kapcsolatfelvételt.

**ENETUNREACH** A másik állomás nem elérhető.

**EADDRINUSE** A teljes cím már kapcsolva van foglalathoz.

**EINPROGRESS** A foglalat csatlakoztatása folyamatban van. Ezt a hibákat csak akkor kaphatjuk, ha a foglalatot nem blokkoló típusúra állítottuk.

**EALREADY** A foglalat csatlakoztatása már a függvény előtt folyamatban volt.  
 Ezt a hibakódot csak akkor kaphatjuk, ha a foglalatot nem blokkoló típusúra állítottuk.

Mivel a foglalatok az állományokhoz hasonlóan írhatók és olvashatók a `read()` és `write()` függvényekkel, az eddig bemutatott eszközök is elegendőek hálózaton történő adatátvitelre. Sajnos azonban az eddig tárgyalt eszközökkel még csak `SOCK_DGRAM` típusú adatátvitelt valósíthatunk meg, amely egy-egy csomag elküldését, illetve fogadását teszi lehetővé az UDP szabvány szerint.

**51. példa.** Az itt következő példaprogram egy UDP csomagot fogad az 5433 kapun, kiírja annak tartalmát a szabványos kimenetre, majd kilép.

```
halozat/udpkinszolgalo.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/socket.h>
4 #include <netdb.h>
5
6 #define PORT      5433
7
8 int main (int argc, char *argv[]){
9     int             foglalat;
10    struct sockaddr_in barki;
11    char            memoria[1500];
12    int             olvasott;
13
14    foglalat = socket(PF_INET, SOCK_DGRAM, 0);
15    if( foglalat<0 )
16        fprintf(stderr, "A foglalat létrehozása sikertelen.\n");
17
18    barki.sin_family=AF_INET;
19    barki.sin_port=htons( PORT );
20    barki.sin_addr.s_addr=htonl( INADDR_ANY );
21    if( bind(foglalat, (struct sockaddr *) &barki,
22              sizeof(barki)) < 0 )
23        fprintf(stderr, "Bind sikertelen.\n");
24
25    olvasott=read(foglalat, memoria, 1500);
26    if( olvasott<0 )
27        fprintf(stderr, "Az olvasás sikertelen.\n");
28
29    printf( "Üzenet: %s\n", memoria );
30 }
```

```

31     if( close( foglalat )!=0 )
32         fprintf( stderr, "close sikertelen" );
33
34     exit(0);
35 }/*main*/

```

Kövessük végig, ahogyan a program létrehoz egy foglalatot a 14. sorban, előkészít egy teljes címet a 18–20. sorban, majd hozzákapcsolja azt a foglalathoz a 21–22. sorban. A program a 25. sorban olvassa a bejövő csomagban található adatbájtokat, majd kiírja azokat a szabványos kimenetre és kilép.

A következő program egy adatcsomagban üzenetet küld az 5433 kapura, majd kilép.

halozat/udpugyfel.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/socket.h>
4 #include <netdb.h>
5
6 #define PORT 5433
7
8 int main( int argc, char *argv[] ){
9     int             foglalat;
10    struct sockaddr_in  kiszolgalo;
11    struct hostent      *gep;
12    char              adat[]={ "Linux" };
13
14    kiszolgalo.sin_family=AF_INET;
15    kiszolgalo.sin_port=htons( 5433 );
16    gep=gethostbyname( "localhost" );
17    if(gep == NULL)
18        fprintf ( stderr, "Az IP cím nem található.\n" );
19
20    kiszolgalo.sin_addr = *(struct in_addr *)gep->h_addr;
21
22    foglalat=socket( PF_INET, SOCK_DGRAM, 0 );
23    if( foglalat<0 )
24        fprintf(stderr, "A foglalat létrehozása sikertelen.\n");
25
26    if( connect(foglalat, (struct sockaddr *) &kiszolgalo,
27                 sizeof(kiszolgalo))<0 )
28        fprintf(stderr, "A kapcsolódás sikertelen\n");
29
30    if( write(foglalat, adat, strlen(adat)+1) < 0 )

```

```

31   fprintf( stderr, "Az írás sikertelen\n" );
32
33   close( foglalat );
34   exit(0);
35 }/*main*/

```

*A program a kiszolgálóprogramhoz hasonlóképpen működik, a különbség csak annyi, hogy itt a kapcsolódáshoz a bind() helyett a connect() függvényt használjuk.*

Ha folyamatos átvitelt szeretnénk létesíteni a hálózaton keresztül, az ügyfélprogramban egyszerűen csak a SOCK\_STREAM állandót kell használnunk a foglalat létrehozásakor. Folyamatos kapcsolattal működő kiszolgálóprogram esetén azonban újabb függvényeket kell megismernünk. E függvények használatához be kell töltenünk a socket.h fejállományt:

```
int listen(int foglalat, unsigned int méret);
```

A függvény a foglalatot adatok fogadására engedélyezi. Erre általában kiszolgálóprogramok esetében van szükségünk, amelyek kapcsolatfelvételi kéresekre várakoznak.

Fontos tudnunk, hogy a listen() függvényt nem kell és nem is lehet olyan kapcsolat kezelésére használni, amely egyetlen csomag küldésére szolgál (SOCK\_DGRAM).

A függvény első paramétere a foglalat, amelyen az adatok fogadását engedélyezni kívánjuk.

A második paraméter egy szám, amely megadja, hogy az adott foglalaton hány várakozó kapcsolatkérést engedélyezünk egyidőben. Ha a kiszolgálóprogramhoz olyan gyorsan érkeznek a kérések, hogy egyidőben ennyi kapcsolatkérés várakozik, a rendszer a következő kéréseket elutasítja.

A függvény visszatérési értéke 0, ha a művelet rendben lezajlott, illetve -1 hiba esetén. Ekkor a függvény beállítja az errno értékét, ami a következők egyike:

EADDRINUSE A foglalathoz tartozó teljes cím már használatban van, azaz az adott kapu már meg van nyitva.

EBADF Az első paraméterként átadott foglalat értéke érvénytelen.

ENOTSOCK Az első paraméterként átadott foglalat valójában nem foglalat.

EOPNOTSUPP A foglalaton ezt a műveletet nem lehet végrehajtani, mert a foglalat nem támogatja ezt a műveletet. Ez általában azt jelenti, hogy a foglalat egyetlen csomag továbbítására szolgál (SOCK\_DGRAM).

```
int accept(int foglalat, struct sockaddr *ügyfél, socklen_t  
*méret);
```

A függvény a bejövő kapcsolatfelvétel elfogadására szolgál. Hacsak másként nem rendelkeztünk, a hívó folyamat futása blokkolódik, amíg külső kapcsolatkérés hatására a foglalat adatátvitelre kész nem lesz.

A függvény első paramétere az a foglalat, ahol a bejövő adatokat fogadni kell, a második paramétere pedig az a struktúra, ahol a függvény az ügyfél adatait elhelyezi. A harmadik paraméter a második paraméter mérete. Ennek a függvénynek a használatánál is a `sockaddr_in` vagy `sockaddr_in6` struktúrákat használjuk második paraméterként és ezek méretét adjuk át a harmadik paraméter helyén.

Ha a művelet sikeres volt, a függvény létrehoz egy új foglalatot, amelyen keresztül a létrejött kapcsolat írható és olvasható. Nem az eredeti – a `socket()` segítségével létrehozott – foglalat szolgál tehát az ügyfélprogrammal létrejött kapcsolat kezelésére, hanem egy új, amelyet az `accept()` hoz létre. Az eredeti foglalat – amelyet első paraméterként áadtunk – további kapcsolatkérések fogadására használható.

A függvény által visszaadott érték -1, ha valamilyen hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EAGAIN** A foglalatot nem blokkoló típusúra állítottuk és pillanatnyilag nincs olyan kapcsolatkérés, amelyet el lehet fogadni.

**EWOULDBLOCK** Ugyanaz, mint az **EAGAIN**.

**EBADF** Az első paraméterként átadott foglalat értéke érvénytelen.

**ENOTSOCK** Az első paraméterként átadott foglalat valójában nem foglalat.

**EOPNOTSUPP** A foglalat nem támogatja a műveletet. Ez általában azt jelenti, hogy a foglalat egyetlen csomag továbbítására szolgál (`SOCK_DGRAM`).

**EINTR** A művelet végrehajtását megszakította egy jelzés beérkezése, mielőtt még kapcsolatfelvételi kérés érkezett volna.

**ECONNABORTED** A kapcsolat meg lett szakítva.

**EINVAL** Az első paraméterként átadott foglalat nem kapcsolatok fogadására szolgál. Ez általában azt jelenti, hogy a `listen()` függvényt nem hívtuk meg, vagy nem futott sikeresen az adott foglalattal.

**EMFILE** A folyamatonként megnyitható állományok számát elérte a folyamat, több állományt vagy foglalatot már nem nyithat meg.

**EFAULT** A második paraméterként átadott memóriacím nem írható a folyamat számára.

**ENOMEM** A művelet végrehajtásához nem áll rendelkezésre elegendő memória.

**ENOBUFS** Ugyanaz, mint az **ENOMEM**.

**EPERM** A helyi tűzfalon beállított szabályok nem teszik lehetővé a művelet végrehajtását.

```
int shutdown(int foglalat, int hogyan);
```

A függvény segítségével lezárhatjuk a megnyitott hálózati kapcsolatot úgy, hogy a lezárásról a másik állomás is értesüljön.

A függvény első paramétere a foglalatot adja meg, amely a megnyitott kapcsolatot azonosítja.

A függvény második paramétere azt adja meg, milyen módon kívánjuk lezárnai a kapcsolatot. Itt a következő állandók egyikét adhatjuk meg:

**SHUT\_RD** A foglalatból tovább már nem fogadunk adatokat.

**SHUT\_WR** A foglalatba tovább már nem küldünk adatokat.

**SHUT\_RDWR** A foglalattal jellemzett kapcsolatot teljesen lezárjuk, tovább már sem küldeni, sem fogadni nem kívánunk adatokat.

A függvény visszatérési értéke 0, ha a művelet sikeresen hajtódott végre, és -1, ha hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EBADF** A függvény első paraméterének értéke hibás.

**ENOTSOCK** A függvény első paramétere nem foglalat.

**ENOTCONN** A függvény első paramétereként megadott foglalat nem kapcsolódik teljes címhez vagy állomáshoz.

**52. példa.** A következő példa egy kiszolgálóprogramot mutat be, amely adatokat fogad és azokra válaszokat ír az ügyfélprogrammal való kapcsolattartás során. A kiszolgáló folyamatos adatcsatornát tart nyitva (`SOCK_STREAM`), amíg az ügyféltől utasítást nem kap a kapcsolat bontására.

A példaprogram egyszerre csak egy ügyféllel képes a kapcsolatot tartani, ha azonban a kapcsolat az ügyfél kérésére felbomlik, újabb ügyfélprogramot képes kiszolgálni.

<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;errno.h&gt; 3 #include &lt;stdlib.h&gt; 4 #include &lt;unistd.h&gt; 5 #include &lt;locale.h&gt; 6 #include &lt;sys/socket.h&gt; 7 #include &lt;netinet/in.h&gt; 8 #include &lt;netdb.h&gt; 9 </pre>	<b>halozat/tcpkiszolgalo.c</b>	
---	--------------------------------	--

```
10 #define PORT    2048
11
12 void hiba( char *uzenet ){
13     fprintf( stderr, uzenet, strerror(errno) );
14     exit( 1 );
15 }/*hiba*/
16
17 int _foglalat(uint16_t port){
18     int             foglalat;
19     struct sockaddr_in barki;
20
21     foglalat=socket(PF_INET, SOCK_STREAM, 0);
22     if( foglalat<0 )
23         hiba( "socket(): %s\n" );
24
25     barki.sin_family=AF_INET;
26     barki.sin_port=htons(port);
27     barki.sin_addr.s_addr=htonl(INADDR_ANY);
28     if( bind(foglalat, (struct sockaddr *) &barki,
29               sizeof(barki)) < 0 )
30         hiba( "bind(): %s" );
31
32     return( foglalat );
33 }/*_foglalat*/
34
35 int olvas_ir(int foglalat){
36     char    memoria[2048];
37     int      n;
38
39     while( (n=read( foglalat, memoria, 2048))>= 0 ){
40         if( write( foglalat, memoria, n ) == -1 )
41             hiba( "write(): %s\n" );
42         if( strncmp(memoria, "exit", 4)==0 )
43             return(0);
44     }/*while*/
45
46     return(1);
47 }/*olvas_ir*/
48
49 int main (int argc, char *argv[]){
50     int foglalat, uj;
51     int i;
52     struct sockaddr_in ugyfel;
```

```

53   size_t meret;
54
55   setlocale( LC_ALL, "" );
56
57   foglalat=_foglalat( PORT );
58
59   if( listen(foglalat, 1)<0 )
60     hiba( "listen(): %s\n" );
61
62   fprintf( stderr, "A kiszolgáló várja a kéréseket.\n" );
63   fflush( stderr );
64
65   while(1){
66     meret=sizeof( ugyfel );
67     uj=accept( foglalat,
68               (struct sockaddr *) &ugyfel, &meret );
69     if( uj<0 )
70       hiba( "accept(): %s\n" );
71
72     fprintf (stderr, "Ügyfél: %s, kapu %hd.\n",
73             inet_ntoa(ugyfel.sin_addr),
74             ntohs(ugyfel.sin_port));
75
76     olvas_ir( uj );
77     close( uj );
78   }/*while*/
79 }/*main*/

```

A program a 12–15. sorban hozza létre a hiba() nevű függvényt, amely kiírja a paraméterként kapott üzenetet, a legutóbbi függvényhívás során felmerült hibaüzenetet és kilép. A hibaüzenet előállításához a függvény az strerror() függvényt használja.

A program 17–33. sorban tartalmazza a \_foglalat() nevű függvényt, amely létrehoz egy új foglalatot a 21. sorban, és csatlakoztatja azt a teljes címhez a 28. sorban. A csatlakoztatás során olyan teljes címet használunk, amely bármely külső forrásból fogad kapcsolatkérést (27. sor) a paraméterként kapott kapun (26. sor). Figyeljük meg a 28. sorban a bind() hívásakor használt paramétereket!

A program a 35–47. sorok között található olvas\_ir() nevű függvény segítségével kezeli a hálózatról olvasott adatokat és válaszol azokra. A függvény 39–44. sorában egy ciklus található, amely addig olvassa a foglalatból érkező adatokat, amíg lehetőséges. A ciklusmagban a 40–41. sorban visszaírjuk az olvasott adatot az ügyfélprogram felé. Ha programunk valamilyen hasznos dolgot is csinálna – nem szorítkozna kizárálag az adatok visszaírására –, az adatfeldolgozást az olvasás és az írás között kellene elhelyeznünk.

A függvény 42–43. sorában található egy vizsgálat, amely az ügyfélprogrammal szakítja meg a kapcsolatot. Ha az ügyfélről érkező adatok elején az exit szó található, az olvas\_ir() nevű függvény kilép és ez az ügyfélprogrammal megszakítja a kapcsolatot.

A program működését alapvetően a 49–81. sorokban található main() függvény határozza meg. Az 55. oldalon érvényesítjük a környezeti változókban meghatározott nyelvi beállításokat a setlocale() függvény segítségével.

Ezután az 57. sorban létrehozunk egy foglalatot a \_foglalat() függvény hívásával, amely egy teljes címhez kapcsolja azt. A foglalat létrehozása után az 59. sorban a listen() függvénnnyel jelezzük, hogy készek vagyunk fogadni a bejövő kapcsolatokat.

A tulajdonképpeni működést a 65–78. sorban található végletes ciklus vezérli. Ennek magjában hívjuk az accept() függvényt a 67. sorban. Az accept() egy új foglalatot ad vissza, amelyen keresztül kezelhetjük az ügyfél kérésére kiépült kapcsolatot. Jól látható, hogy ezzel az új foglalattal meghívja a program az olvas\_ir() függvényt a 76. sorban, így a kiépült kapcsolat kezelésének ideje alatt nem képes újabb kapcsolatkéréseket fogadni.

Amint az olvas\_ir() függvény visszatér – az ügyfélről a program az exit szót kapta –, a 77. sorban zárjuk a foglalatot és a ciklus újabb végrehajtása alatt új kapcsolatkéréseket fogadunk.

**53. példa.** A következő példaprogram egy igen egyszerű webkiszolgálót mutat be. A program önmagának több példányban való elindításával képes egyszerre több ügyfélprogram – webböngésző – kiszolgálására.

```
halozat/webkiszolgalo.c
1 #define _GNU_SOURCE
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <syslog.h>
6 #include <unistd.h>
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <netinet/in.h>
10 #include <netdb.h>
11 #include <fcntl.h>
12 #include <unistd.h>
13 #include <sys/stat.h>
14 #include <sys/wait.h>
15 #include <fnmatch.h>
16 #include <libgen.h>
17
18 /*#define DEBUG*/
```

```
19 #define PORT 8080
20 #define DOCUMENTROOT "/home/httpd/html"
21
22 char    allomanynev[1024];
23 void   *memoria;
24 size_t meret;
25 int    fogl;
26
27 void hiba( const char *hiba ){
28     syslog( LOG_WARNING, hiba, strerror(errno) );
29     exit(1);
30 }/*hiba*/
31
32 int betolt( ){
33     struct stat st;
34     int         allomany;
35
36     if( stat(allomanynev, &st )!=0 )
37         hiba( "stat(): %s\n" );
38
39     meret=st.st_size;
40
41     memoria=malloc( meret );
42     if( memoria==NULL )
43         hiba( "malloc(): %s\n" );
44
45     allomany=open( allomanynev, O_RDONLY );
46     if( TEMP_FAILURE_RETRY( read(allomany,
47                               memoria,
48                               meret)
49                               ) != meret )
50         hiba( "file read(): %s\n" );
51
52     close( allomany );
53     return( 0 );
54 }/*sendfile*/
55
56 int kapu_nyitasa( void ){
57     int          foglalat;
58     struct sockaddr_in  barki;
59
60     foglalat=socket(PF_INET, SOCK_STREAM, 0);
61     if( foglalat<0)
```

```
62     hiba( "socket(): %s\n" );
63
64     barki.sin_family=AF_INET;
65     barki.sin_port=htons(PORT);
66     barki.sin_addr.s_addr=htonl(INADDR_ANY);
67     if( bind( foglalat,
68             (struct sockaddr *) &barki,
69             sizeof(barki))<0 )
70         hiba( "bind(): %s\n" );
71
72     return( foglalat );
73 }/*kapu_nyitasa*/
74
75
76 int olvasas( int foglalat ){
77     char memoria[1024];
78     int n;
79     sprintf( allomanynev, DOCUMENTROOT );
80
81     n=TEMP_FAILURE_RETRY(read( foglalat, memoria, 1024));
82     if( n<0 )
83         hiba( "read()\n" );
84
85     sscanf(memoria, "GET %s",
86            &allomanynev[strlen(DOCUMENTROOT)] );
87
88     if( allomanynev[strlen(allomanynev)-1]=='/'
89         sprintf(&allomanynev[strlen(allomanynev)], "index.html" );
90
91 #ifdef DEBUG
92     syslog( 0, "File request: %s\n", allomanynev );
93 #endif
94
95     return( n );
96 }/*olvasas*/
97
98
99 int iras( int foglalat ){
100     char head_template[]="HTTP/1.0 200 OK\r\n"
101     Server: miniserver/0.1\r\n"
102     Connection: close\r\n"
103     Content-Type: %s\r\n\r\n";
104     char head[512];
```

```
105    char *type;
106
107    type="text/html";
108    if( fnmatch("*.gif", basename(allomanynev), 0) == 0 )
109        type="image/gif";
110    if( fnmatch("*.jpg", basename(allomanynev), 0) == 0 )
111        type="image/jpeg";
112    if( fnmatch("*.png", basename(allomanynev), 0) == 0 )
113        type="image/png";
114
115    sprintf( head, head_template, type );
116
117    TEMP_FAILURE_RETRY( write(foglalat, head, strlen(head)) );
118
119    if( memoria!=NULL )
120        TEMP_FAILURE_RETRY( write(foglalat, memoria, meret) );
121    else
122        syslog( 0, "Hiba: nincs küldendő adat.\n" );
123
124    return( 0 );
125 }/*irás*/
126
127
128 int mukodtet( void ){
129     int uj;
130     int i;
131     struct sockaddr_in clientname;
132     size_t size;
133
134     fogl=kapu_nyitasa();
135     if( listen (fogl, 5)<0 )
136         hiba( "listen(): %s\n" );
137
138     while( 1 ){
139         size=sizeof (clientname);
140         uj=accept(fogl, (struct sockaddr *) &clientname, &size);
141
142         if( fork()==0 ){
143             olvasas( uj );
144             betolt();
145             iras( uj );
146             shutdown( uj, SHUT_RDWR );
147             if( memoria != NULL )
```

```

148         free( memoria );
149         return(0);
150     }/*if*/
151     }/*while*/
152 }/*mukodtet*/
153
154 void uzenet_kilepes( int signal ){
155     shutdown( fogl, SHUT_RDWR );
156     hiba("Kilépés... \n");
157 }/*uzenet_kilepes*/
158
159 void uzenet_gyermek( int signal ){
160     while( waitpid(WAIT_ANY, NULL, WNOHANG)>0 )
161         ;
162     return;
163 }/*uzenet_gyermek*/
164
165 int main( int argc, char *argv[] ){
166     daemon( 0, 0 );
167
168     openlog( "httpd", LOG_PID, LOG_DAEMON );
169     syslog( LOG_NOTICE, "Httpd elindult." );
170
171     signal( SIGINT, uzenet_kilepes );
172     signal( SIGKILL, uzenet_kilepes );
173     signal( SIGTERM, uzenet_kilepes );
174     signal( SIGCHLD, uzenet_gyermek );
175
176     mukodtet();
177     return(0);
178 }/*main*/

```

A program az egyszerűség kedvéért négy általános érvényességű változót használ, melyeket a 22–25. sorokban hozunk létre. A program az allomanynev nevű változóban az ügyfélprogram által kért állomány nevét tárolja, a memoria nevű változóban az állomány számára foglalt memóriaterület címét, a meret változóban pedig a méretét. A kapcsolatfelvételre használt foglalat a fogl változóban tárolódik.

A program 27–30. sorában található hiba() nevű függvényt hiba esetén hívhatjuk meg. A függvény naplózza a hibáüzenetet, majd kilep. Azért kellett a hibák jelzésére a syslog() függvénytel megvalósított naplózást választanunk, mert a program démonként fut, így nem írhatja a hibáüzeneteket a szabványos hibacsatornára.

A program a 32–53. sorokban tartalmazza a betolt() nevű függvényt, amely betölt egy állományt a memóriába. A függvény a 36. sorban a stat() függvényt hasz-

nálja az állomány méretének megállapítására, a 41. sorban a `malloc()` hívásával lefoglalja a megfelelő méretű memóriaterületet, majd a 45–52. sorokban megnyitja, beolvassa és lezárja az állományt.

Az 56–73. sorok között található függvény egy foglalatot hoz létre, amelyet előkészít arra, hogy a 8080-as kapun bármely ügyfélről fogadhasson kapcsolatfelvételi kéreseket. E függvényben a már jól ismert `socket()` és `bind()` függvényeket hívjuk.

A 76–96. sorokban található `olvasas()` függvény a hálózatról olvassa az ügyfélprogram által küldött kérést. A függvény feltételezi, hogy a kérés eleje `GET xxx` alakú, ahol az `xxx` helyén a kért állomány neve és elérési útja található. A függvény a kérésből kimásolt állománynév elő bemásolja a `DOCUMENTROOT` makróban található könyvtárnevet – ebben a könyvtárban lesznek a szolgáltatott állományok –, valamint, ha a kért állománynév utolsó betűje a / karakter, utánamásolja az `index.html` állománynevet.

A program 99–126. soraiban található `iras()` függvény előállítja a választ és elküldi az ügyfél felé. A 100–103. sorban található az a fejléc, amelyet az állomány tartalma előtt kell küldeni a webböngésző felé. Figyeljük meg, hogy a fejléc szöveges, a sorait a \r\n kettős jel jelzi, valamint azt, hogy a fejléc végét egy üres sor – azaz a \r\n\r\n karakterek – jelzik.

A fejlécben szerepelnie kell a válaszként küldött állomány típusának. A típusjelzést programunk egyszerűen az állomány végződéséből állítja elő a 108–114. sorokban, a 115–117. sorokban pedig előállítjuk és elküldjük a teljes fejléket. Ezek után a függvény a 119–122. sorokban elküldi az előzőleg betöltött állomány tartalmát.

A program folyamatos működése során a 128–152. sorok között található `mukodtet()` függvényt hajtja végre. Ennek elején, a 134. sorban hozzuk létre a kapcsolatkérések fogadására használt foglalatot, a 135–136. sorokban pedig hívjuk a `listen()` függvényt. A 135–151. sorok között található végtelen ciklus újabb és újabb kapcsolatokat kér és szolgál ki. Elmondhatjuk, hogy ez a ciklus határozza meg a program működését.

A 140. sorban az `accept()` függvénnnyel várakozunk a beérkező kérésekre, valamint előállítjuk a kiépült kapcsolat kezelésére alkalmas foglalatot az új nevű változóban.

Amint kiépült a kapcsolat, a 142. sorban a program végrehajt egy `fork()` hívást, aminek hatására a folyamat megkettőződik. A gyermekfolyamat – a `fork()` visszatérési értéke 0 – felelős a kiépült kapcsolat kezeléséért, a szülő újabb kapcsolatokat fogad. Ennek megfelelően, abban a folyamatban, ahol a `fork()` visszatérési értéke 0, meghívjuk az `olvasas()`, `betolt()` és `iras()` függvényeket, majd bontjuk a kapcsolatot a `shutdown()` függvénnnyel és felszabadítjuk a memóriát (143–148.). Ezek után a gyermek a 149. sorban található `return()` hatására kilép.

A szülő a 142. sorban végrehajtott `fork()` után újrakezdi a ciklust és újabb kapcsolatkéréseket fogad.

A programunk két jelzskezelő függvényt használ. A 154–157. sorokban létrehozott `uzenet_kilepes()` függvényt olyan jelzések kezelésére használjuk, amelyek a program megszakítását kérik. Ennek a függvénynek a meghívásakor először lezárjuk

a kapcsolatok fogadására szolgáló foglalatot a 155. sorban, majd egy naplóbejegyzést helyezünk el a kilépésről és a hiba() függvényen keresztül kilepünk.

Jóval érdekesebb a gyermekfolyamatok kilépését kezelő `uzenet_gyermek()` függvény. Ez a jelzéskezelő a `SIGCHLD` jelzések kezelésére szolgál, ami azt jelzi a folyamat számára, hogy legalább egy gyermekfolyamata befejezte a futását. Az ilyen jelzés beérkezésekor a `waitpid()` függvény hívásával az összes futását befejezett gyermekfolyamatot meg kell semmisítenünk, különben élőhalottként (zombi) maradnak a rendszerben. A 160–161. sorokban egy ciklusban addig ismételgetjük a `waitpid()` hívását, amíg az befejeződött gyermekfolyamatokat érzékel.

Az eddig elmondottak alapján a 165–178. sorokban található `main()` függvény működése könnyen megérthető. A 166. sorban démon üzemmódra kapcsolunk a `daemon()` függvény hívásával, a 168. sorban kapcsolatot nyitunk a rendszernapló felé, a 169. sorban pedig elhelyezünk egy üzenetet a rendszernaplóban arról, hogy a program elindult.

A 171–174. sorokban nyilvántartásba vetjük a jelzéskezelő függvényeinket. A gyermekfolyamatok öröklik a jelzéskezelőket, ezért ezt elegendő egyszer, az indulás-kor megtenniink.

A 176. sorban meghívjuk a `mukodtet()` nevű függvényt és ezzel elkezdjük a tulajdonképpeni munkát.

## Megosztott memória kezelése

A megosztott memória (*shared memory*) a folyamatok közti kapcsolattartás egyik leghatékonyabb eszköze. A módszer lényege, hogy a folyamatok közös memóriaterületen keresztül tartják a kapcsolatot.

Az eddig tárgyalta memóriafoglalási módszerek nem alkalmasak megosztott memóriaterületek foglalására, hiszen az általuk kezelt memóriaterületeket csak egy folyamat használhatja – amelyik azokat lefoglalta. Amíg ez a folyamatok egymástól való védelme szempontjából szükséges, a folyamatközi kapcsolattartás (IPC, *inter process communication*) támogatására hasznos a megosztott memória kezelése, hiszen nincs még olyan gyors eszköz, mint az operatív memória.

A következőkben a legfontosabb függvényeket vesszük sorra, amelyek megosztott memória kezelésére szolgálnak Linux rendszereken. Ezeknek az eszközöknek a használatához szükség lehet a `sys/types.h`, `sys/ipc.h` és a `sys/shm.h` fejállományok betöltésére.

```
key_t ftok(const char *állomány, int változat);
```

A függvény különleges azonosítót hoz létre, amely a folyamatok közti kapcsolattartás folyamán használatos. Valójában a függvény igyekszik egyedi azonosítót készíteni, amely lehetővé teszi, hogy a folyamatok között kiépített kapcsolatok ne keveredjenek össze egymással.

Az azonosító létrehozásához a függvény felhasználja az első paraméterként kapott állománynevet és a második paraméterként kapott változatszámot. Az állománynévnek létező állományra kell mutatnia, mert a függvény az állományt megnyitja és onnan információkat nyer. A függvény az állománynevet és a változatszámot arra használja, hogy az alkalmazásra jellemző egyedi azonosítót hozzon létre, ezért a legszerencsésebb, ha a programot alkotó bináris állomány nevét (`argv[0]`) adjuk át paraméterként.

A függvény a második paraméterként kapott változatszámból csak az alsó nyolc bitet használja fel. Fontos tudnunk, hogy a második paraméterként átadott szám alsó nyolc bitje nem képviselhet 0 értéket.

A függvény visszatérési értéke a létrehozott azonosító, ha a művelet sikeres volt, és -1, ha nem. A függvény által beállított `errno` változó értelmezése ugyanúgy történik, mint a `stat()` függvény esetében, az `ftok()` számára az első paraméterként átadott állománynévre vonatkozik.

```
int shmget(key_t kulcs, int méret, int kapcs);
```

A függvény megosztott memóriát foglal, amely több folyamat által közösen használható. A létrehozott megosztott memóriaterületet a folyamatok – így a létrehozó folyamat is – csak akkor használhatja, ha csatlakozik hozzá.

A függvénynek átadott első paraméter a `kulcs`, amelyet az `ftok()` függvény segítségével hoztunk létre, a második paraméter a lefoglalandó memória mérete, míg a harmadik a következő kapcsolók által beállítható érték:

**IPC\_CREAT** Új memóriaterület foglalására ad utasítást ez a kapcsoló. Ha nincs bekapcsolva, a függvény már létező területhez kísérel meg kapcsolódni.

**IPC\_EXCL** Ha ez a kapcsoló is be van kapcsolva, a függvény hibát ad, ha már le van foglalva memóriaterület a hivatkozott kulccsal.

A függvény által visszaadott érték a memória kezelésére használható szám sikeres művelet esetén, illetve -1, ha hiba lépett fel. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EINVAL** A memóriaterület mérete nem megfelelő vagy létezik már a megosztott memóriaterület az adott kulccsal, de a kért méret nagyobb a méreténél.

**EEXIST** Az `IPC_CREAT` és az `IPC_EXCL` be volt kapcsolva és a megosztott memóriaterület már létezik.

**EIDRM** A terület már létezik, de felszabadításra van kijelölve.

**ENOSPC** A rendszer megosztott memóriaterületre vonatkozó korlátai kimerültek, nem lehetséges újabb megosztott memóriaterületet lefoglalni.

**ENOENT** Az `IPC_CREAT` kapcsoló nem volt bekapcsolva és az adott kulcshoz nincs memóriaterület rendelve.

EACCES Az adott folyamatot futtató felhasználónak nincs joga a megadott megosztott memóriaterülethez férni.

ENOMEM A művelet végrehajtásához nem áll rendelkezésre elegendő memória.

Az 54. példa bemutatja az `shmget()` függvény használatát és azt, hogy hogyan kezelhetjük a megosztott memóriaterületeket a parancssorban indított programok segítségével.

```
void *shmat(int azonosító, const void *cím, int kapcs);
```

A függvény kapcsolódik az első paraméterként megadott azonosítójú megosztott memóriaterülethez, és megadja annak címét.

A függvény második paramétere megadja, hogy milyen memóriacímen szeretnék elérni a megosztott memóriaterületet. A második paraméter értéke nem tetszőleges, a következőképpen alakulhat:

- Ha a függvény második paramétere NULL érték, a rendszer választ egy megfelelő memóriaterületet, amelyen keresztül a folyamat elérheti a megosztott memóriát.
- Ha a második paraméter nem NULL és a harmadik paraméterként átadott kapcsolók között szerepel az SHM\_RND, a rendszer az átadott memóriacímet kerekíti a szükséges értékre és az így kapható területen keresztül teszi elérhetővé a megosztott memóriát.
- Ha a második cím nem NULL és az SHM\_RND nem szerepel a kapcsolók között, az átadott memóriacímnek az SHMLBA állandó többszörösére kell esnie.

A függvénynek átadott kapcsolók a következő értékekkel állíthatók össze:

SHM\_RND A függvény a második paraméterként átadott memóriacímet szükség esetén a megfelelő értékre kerekíti.

SHM\_RDONLY A hozzáférés csak olvasható módon lesz engedélyezve.

A függvény visszatérési értéke a memóriaterület címe, amelyen keresztül a megosztott memória tartalma elérhető, illetve -1 hiba esetén. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

EACCES A folyamatnak nincs joga az adott memóriaterülethez hozzáférni.

EINVAL Érvénytelen azonosító vagy nem megfelelő memóriacím.

ENOMEM A művelet végrehajtásához nem áll rendelkezésre elegendő memória.

```
int shmdt(const void *cím);
```

A függvény a megosztott memóriaterületről való leválásra szolgál. A rendszer akkor tudja felszabadítani a megosztott memóriaterületet, ha az összes csatlapozott folyamat lekapcsolódik róla a shmdt() hívásával. Ha a folyamat kilép, a leválás automatikusan megtörténik.

A függvény egyetlen paramétere az a cím, amelyet a megosztott memóriához kapcsolódáskor adott vissza a shmat() függvény.

A visszatérési érték 0, ha a lekapcsolódás sikeres volt, illetve -1 hiba esetén.

```
int shmctl(int azonosító, int parancs, struct shmid_ds *jell);
```

A függvény segítségével különféle műveleteket végezhetünk a megosztott memóriaterületeken, lekérdezhetjük a tulajdonságait, átállíthatjuk azokat vagy éppen törölhetjük a megosztott memóriaterületet az operációs rendszerből, felszabadítva a területet, amelyet lefoglal.

A parancs első paramétere az azonosító, segítségével a megosztott memóriaterületet jelölhetjük ki, amelyre a második paraméterként átadott parancs vonatkozik. A harmadik paraméter a megosztott memória jellemzőit jelölő mutató, amely egy shmid\_ds struktúrát jelöl a memóriában:

A shmid_ds struktúra	
struct ipc_perm shm_perm	A jogokat szabályozó struktúra.
int shm_segsz	A méret bájtban.
time_t shm_atime	A legutolsó kapcsolódás időpontja.
time_t shm_dtime	A legutolsó lekapcsolódás időpontja.
time_t shm_ctime	A legutolsó változtatás időpontja.
unsigned short shm_cpid	A létrehozó azonosítója.
unsigned short shm_lpid	A legutolsó műveletet kérő azonosítója.
short shm_nattch	A jelenlegi kapcsolatok száma.

A struktúrán belül található az ipc\_perm struktúra, amelynek elemei a következők:

A ipc_perm struktúra	
key_t key	A létrehozáskor használt kulcs.
uid_t uid	A tulajdonos azonosítója.
gid_t gid	A tulajdonoscsoport azonosítója.
uid_t cuid	A létrehozó azonosítója.
gid_t cgid	A létrehozó csoport azonosítója.
unsigned short int mode	Az alsó 9 bit a jogok leírása az állományoknál megszokott módon.

A második paraméterként a következő állandók adhatók át:

**IPC\_STAT** A parancs a megosztott memória adatainak lekérdezésére vonatkozik. A függvény bemásolja a megosztott memória adatait a harmadik paraméterként kapott címre.

A folyamatot futtató személynek a művelet végrehajtásához olvasási joggal kell rendelkeznie a hivatkozott megosztott memóriára.

**IPC\_SET** A művelet a megosztott memória tulajdonságainak megváltoztatására vonatkozik. A függvény harmadik paramétere az új tulajdonságokra mutat.

A folyamatot futtató személynek a megosztott memóriát létrehozó személynek, a megosztott memória tulajdonosának vagy a rendszergazdának kell lennie, hogy a művelet sikeres lehessen.

**IPC\_RMID** A parancs a megosztott memória felszabadítására vonatkozik. A függvény a parancs hatására törlésre jelöli a megosztott memóriát, de az csak akkor szabadul fel, amikor az utolsó csatlakozást is felszabadítjuk.

A folyamatot futtató személynek a megosztott memória létrehozójának, tulajdonosának vagy a rendszergazdának kell lennie.

**SHM\_LOCK** A megosztott memória csereterületre való kimásolásának tiltása. A megosztott memóriaterület ezentúl nem kerülhet a virtuális memóriába. A folyamatot futtató felhasználónak a rendszergazdának kell lennie.

**SHM\_UNLOCK** A memóriaterület virtuális memóriába való másolásának újraengedélyezése. A folyamatot futtató felhasználónak a rendszergazdának kell lennie.

A függvény visszatérési értéke 0, ha a művelet sikeres volt, és -1 hiba esetén. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**EACCES** A memóriaterület adatait nem lehet lekérdezni, mert a folyamatot futtató felhasználónak nincs olvasási joga a memóriaterületre.

**EFAULT** A függvény harmadik paramétere nem elérhető, pedig a második paraméterként átadott parancs szükséges a használatát.

**EINVAL** A függvény első vagy második paramétere érvénytelen értéket hordozott.

**EIDRM** A megosztott memóriaterület már el van távolítva a rendszerből, fel van szabadítva.

**EPERM** A folyamatot futtató személynek nincs joga az adott művelet elvégzésére.

**EOVERFLOW** Az **IPC\_STAT** műveletet nem lehet elvégezni, mert a felhasználói azonosító vagy a csoportazonosító tárolására nem elegendő a hely.

**54. példa.** A következő példaprogram egy kilobájtos megosztott memóriaterületet foglal le.

```

1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <sys/ipc.h>
4
5   int main(int argc, char *argv[]){
6       key_t    kulcs;
7       int      memoria;
8
9       kulcs=ftok( argv[0], 1 );
10      if( kulcs== -1 ){
11          fprintf(stderr, "A kulcs létrehozása sikertelen");
12          exit(1);
13      }/*if*/
14
15      memoria=shmget(kulcs, 1024, IPC_CREAT | IPC_EXCL);
16      if( memoria== -1 ){
17          fprintf( stderr, "A memória létrehozása sikertelen.\n");
18          exit(1);
19      }/*if*/
20      exit(0);
21  }

```

A program a 9–13. sorban egy kulcsot hoz létre, amelyet a 15–19. sorban felhasznál megosztott memória kérésére. A kéréskor átadott kapcsolók új megosztott memóriaterület létrehozására szolgálnak; az `IPC_EXCL` kapcsoló miatt az `shmget()` hibát ad vissza, ha az adott kulccsal már történt területfoglalás.

A program a megosztott memória létrehozása után a 20. sorban kilép, anélkül, hogy bármilyen üzenetet írna a szabványos kimenetre. Ha tehát a memória létrehozása sikeres, a program kilép. Ha most újra futtatjuk a programot, a memória létrehozása nem sikerül, hiszen az már létezik:

```

Bash$ ./shmget
Bash$ ./shmget
A memória létrehozása sikertelen.
Bash$

```

A rendszeren létező megosztott memóriaterületekről információkat az `ipcs` programmal kaphatunk, amelynek `-m` kapcsolója szolgál arra, hogy csak a megosztott memóriáról kapjunk leírást:

```

Bash$ ipcs -m
----- Shared Memory Segments -----

```

```
key          shmid      owner      perms      bytes      nattch      status  
0x0102c011  786432    root        0          1024        0  
Bash$
```

A program kimenetének első oszlopában olvashatjuk a kulcsot, a másodikban a megosztott memóriaterület azonosítóját, az ötödikben a memóriaterület méretét, míg a hatodikban azt, hogy a megosztott memóriaterületet használó folyamatok hányszáma csatlakozási pontot építettek ki a memóriaterülethez.

A megosztott memóriaterületeket törölni az `ipcrm` parancssal lehet. A parancs után az `shm` kulcsszóval kell jelölnünk, hogy megosztott memóriát szeretnénk felszabadítani, utána pedig a memóriaterület azonosítójának kell következnie – amelyet a `ipcs` által kiírt táblázat harmadik oszlopában találunk:

```
Bash$ ipcrm shm 786432  
resource(s) deleted  
Bash$
```

*Ha a megosztott memória törlése után újra futtatjuk a programot, annak újra sikeriül létrehozni a megosztott memóriaterületet, de most már egy új azonosítót kap a rendszertől. A kulcs a régi marad, az azonosító azonban mindenkor új lesz.*

**55. példa.** A következő példaprogram megosztott memóriát használ a gyermekfolyamatával való kapcsolattartásra. A program előbb létrehoz egy megosztott memóriaterületet, majd a `fork()` függvényel létrehozott gyermekfolyamatával közösen használja azt.

```
megosztott/memoriahasznalat.c

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <sys/shm.h>
5 #include <string.h>
6
7 int memoria_letrehoz( const char *allomany ){
8     key_t    kulcs;
9     int      memoria;
10
11     kulcs=ftok( allomany, 1 );
12     if( kulcs== -1 ){
13         fprintf(stderr, "A kulcs létrehozása sikertelen");
14         exit(1);
15     }/*if*/
16
17     memoria=shmget(kulcs, 1024, IPC_CREAT | IPC_EXCL);
18     if( memoria== -1 ){


```

```
19     fprintf( stderr, "A memória létrehozása sikertelen.\n");
20     exit(1);
21 }/*if*/
22
23     return(memoria);
24 }/*memoria_letrehoz*/
25
26 int main(int argc, char *argv[]){
27     int             memoria;
28     char  *cim;
29
30     memoria=memoria_letrehoz( argv[0] );
31
32     if( fork() !=0 ){
33         /*
34          * A szülő feladatai...
35          */
36         cim=shmat( memoria, NULL, 0 );
37         if( (int)cim== -1 ){
38             fprintf(stderr, "Szülő: nem lehetett csatlakozni.\n");
39             exit(1);
40 }/*if*/
41         strcpy( cim, "A küldött üzenet szövege... ");
42         wait(NULL);
43         printf( "Szülő: a fogadott üzenet: %s\n", cim );
44         shmdt(cim);
45         if( shmctl( memoria, IPC_RMID, NULL ) != 0 ){
46             fprintf( stderr, "Szülő: nem lehetett törleni.\n" );
47             exit(1);
48 }/*if*/
49         exit(0);
50     }else{
51         /*
52          * A gyermek feladatai
53          */
54         cim=shmat( memoria, NULL, 0 );
55         if( cim==NULL ){
56             fprintf( stderr, "Gyermek: nem lehetett csatlakozni.\n");
57             exit(1);
58 }/*if*/
59         sleep(5);
60         printf( "Gyermek: a fogadott üzenet: %s\n", cim );
61         strcpy( cim, "A visszaküldött üzenet szövege" );
```

```
62     shmdt(cim);  
63     exit(0);  
64 }/*if*/  
65 }/*main*/
```

A program helyes futás esetén csak két üzenetet ír a képernyőre, az első üzenet jelzi, hogy a gyermek felhasználta a szülő által elhelyezett memóriatartalmat, a második pedig, hogy a szülő megkapta a válaszként adott szöveget a megosztott memóriában.

Bash\$ ./memoriahasznalat

Gyermek: a fogadott üzenet: A küldött üzenet szövege...

Szülő: a fogadott üzenet: A visszaküldött üzenet szövege

Bash\$

A program 7–24. sorában található `memoria_letrehoz()` nevű függvény megosztott memóriát hoz létre és visszaadja az annak kezelésére szolgáló egész számot. A 26. sorban kezdődő `main()` függvény a 30. sorban hívja a függvényt és a memória kezelésére használható egész számot elhelyezi a `memoria` nevű változóban.

A program a 30. sorban található `fork()` hívással új folyamatot hoz létre önmaga lemásolásával. A függvényhívásnak köszönhetően született két folyamat örökli a `memoria` változó értékét, és minden folyamat hozzáérhet az osztott memóriához.

A szülőfolyamat a 33. sortól az 50. sorig tartó utasításokat hajtja végre. Ez a programrészlet először az osztott memóriához csatlakozik a 36. sorban található függvényhívással. A szülő ezután a 41. sorban bemásol egy szöveget a megosztott memóriába az itt kapott cím felhasználásával. A szülő ezzel átadta a szöveget a megosztott memórián keresztül, és a 42. sorban várakozik a gyermekfolyamat befejeződésére. A szülő csak akkor folytatja a 43. sorban a futást, ha a gyermek befejeződött. Ekkor a 43. sorban kiírja a közös memóriaterület tartalmát, amely most már a gyermek válaszát tartalmazza. A szülő ezután a 45. sorban lekapcsolódik a közös memóriaterületről és a 46. sorban kijelöli a memóriaterületet törlésre. A szülő az 50. sorban található függvényhívással lép ki, ha a futás során nem lépett fel hiba.

A gyermek által végrehajtott utasítások az 52. sorban kezdődnak. A gyermek először kapcsolódik az 55. sorban található függvényhívás segítségével a megosztott memóriaterülethez. A memóriaterület kezelését ezután a cím változóban tárolt cím alapján végzi a gyermek, ahogyan a szülő is. A gyermek ezután felfüggeszti futását néhány másodpercre, hogy időt adjon a szülőnek az üzenet elhelyezésére. A várakozás után a gyermek kiírja a szabványos kimenetre a szülő által a megosztott memóriába helyezett szöveget, majd egy másik szöveget helyez oda válaszul. Ezek után a gyermek lekapcsolódik a megosztott memóriáról a 63. sorban és kilép a 64. sorban.

## Szemaforok

Az állományok zárolása kapcsán már említettük, hogy a közös erőforrások használatánál különös gonddal kell eljárnunk. A megosztott memória kezelésére különösen igaz ez, ezért meg kell ismerkednünk a szemaforok (*semaphore*) használatával.

Ha két vagy több folyamat közös erőforrást – például megosztott memóriát – kezel, akkor komoly és nehezen felderíthető hibák eredhetnek abból, ha egyszerre próbálnak meg valamilyen műveletet elvégezni.

Ha például az egyik folyamat módosítja a közös memória tartalmát, míg egy másik olvassa, az olvasó folyamat hibás, félig elkészült adatszerkezeteket olvashat. Azt mondjuk ilyenkor, hogy versenyhelyzet alakult ki a két folyamat között a közös erőforrás kezelése közben. A versenyhelyzet nem azt jelenti tehát, hogy a két folyamat verseng az erőforrás használatáért, hanem azt, hogy egyszerre használják az erőforrást, annak az állapota a folyamatok között fennálló versengéstől függ. Versenyhelyzet esetén a közös erőforrás állapota, a folyamatok futásának végeredménye megjósolhatatlan, kiszámíthatatlan.

A versenyhelyzetet mindenkor kell kerülnünk, gondoskodnunk kell arról, hogy a folyamatok ne használhassák egyszerre az erőforrást.

Azt mondjuk, hogy a folyamatokban, azok futása közben vannak olyan szakaszok, amelyekben közös erőforrásokat használnak. Ezeket a szakaszokat, a programnak ezeket a részeit kritikus szakaszoknak nevezzük. A programozónak gondoskodnia kell arról, hogy a közös erőforrást használó folyamatok között mindenkor csak egy lehessen kritikus szakaszban. Ezt úgy érhetjük el, ha a kritikus szakaszban lévő folyamat – bármelyik is legyen az – megtiltja a másik folyamat számára, hogy a kritikus szakaszba lépjön. Ezt nevezzük a kölcsönös kizárási elvnek. Azért beszélünk kölcsönös kizárásról, mert a kritikus szakaszban – és csak akkor – a folyamatok bármelyike megtiltja a másiknak, hogy belépjen a kritikus szakaszba.

Azt gondolnánk, hogy a kölcsönös kizárás egyszerűen biztosítható egy közös változó segítségével. Amikor a folyamat be kíván lépni a kritikus szakaszba, megvizsgálja a zárolásra használt közös változót, annak értékéből tudva meg, hogy a másik folyamat kritikus szakaszban van-e. Ha nem, a közös változó segítségével megtiltja a másik folyamatnak a kritikus szakaszba való belépést, majd elvégzi a közös erőforrás kezelését.

Sajnos azonban egyszerű változó nem alkalmas a kölcsönös kizárást megvalósítására. Könnyen megérthető, hogy a közös változóval megvalósított kölcsönös kizárás esetében a közös erőforrás a közös változó, a kritikus szakasz a közös változó kezelése és a versenyhelyzet ugyanúgy fennáll, csak most a zárolást végző változó kezelése okoz gondot.

Igazi megoldást csak a szemaforok használata jelent. A szemaforok a kölcsönös kizárást megvalósítására használt, versenyhelyzetmentes kezelésű változók, amelyeket a Linux biztosít számunkra. Ha a közös erőforrásokat használó folyamatokban a kritikus szakaszokat a helyes módon használt szemaforral vagy szemaforokkal védi-

jük, soha nem alakul ki versenyhelyzet. Azt, hogy a Linux milyen programozási trükkökkel biztosítja a szemaforok versenyhelyzetmentes kezelését, nem tárgyaljuk, annál fontosabb azonban az, hogy miképpen használhatjuk azokat programunkban.

A szemaforok egész típusú különleges változók, amelyek értéke 0 és egy maximális érték közt változhat. Ha a szemafor értéke 0, az általa védett erőforrás foglalt, a folyamat nem léphet be a kritikus szakaszba. Ha 0-nál nagyobb a szemafor értéke, a folyamat beléphet a kritikus szakaszba, de előbb a szemafor értékét csökkentenie kell.

A szemaforokon két műveletet értelmezünk:

**Csökkentés** A folyamat, mielőtt belépne a kritikus szakaszba, ezt a műveletet hajtja végre. A művelet kérésekor két eset lehetséges;

1. Ha a szemafor értéke nagyobb volt 0-nál, a szemafor értéke csökken, ezzel a folyamat lefoglalta a közös erőforrást. A folyamat a szemafor csökkentése után beléphet a kritikus szakaszba, végrehajthatja a soron következő utasításait.
2. Ha a szemafor értéke 0, a folyamat blokkolt állapotba kerül. Nem folytatódik a futása, a Linux gondoskodik arról, hogy várakozzék, amíg a közös erőforrás fel nem szabadul, a szemafor értéke nem nő.

**Növelés** A folyamat, miután végrehajtotta a kritikus szakasz utasításait, ezt a műveletet hajtja végre, ezzel a művelettel jelzi, hogy a szemaforral védett erőforrásról lemond.

A növelés során a szemafor értéke nő. Ha lenne olyan folyamat, amely e miatt a szemafor miatt került blokkolt állapotba, a Linux gondoskodik róla, hogy hogy az felszabaduljon a blokkolt állapotból. Miután a folyamat felszabadul a blokkolt állapotból, a szemafor általa kért csökkentése megtörténhet, a folyamat lefoglalhatja az erőforrást és beléphet kritikus szakaszába.

A következő néhány oldalon bemutatjuk, hogy a gyakorlatban miképpen használhatjuk a Linux által biztosított szemaforokat a közösen használt erőforrások kezelésére, a GNU C programkönyvtár mely függvényei szükségesek a szemaforok nyújtotta lehetőségek kihasználására. Ezen eszközök használatához a sys/types.h, a sys/ipc.h és a sys/sem.h fejállományok betöltése szükséges lehet.

```
int semget(key_t kulcs, int darab, int kapcs);
```

A függvény új szemaforkészlet létrehozására használható. minden szemaforkészlet egy vagy több szemafort tartalmazhat. igen fontos tudnunk, hogy a semget() függvény segítségével létrehozott szemaforkészlet *nem használható, amíg az alapérték beállítását el nem végezzük a semctl() függvény segítséggel.*

A függvény első paramétere a kulcs, amely lehetővé teszi, hogy a folyamatok által létrehozott szemaforkészletek ne keveredjenek össze. Az ftok() függvény alkalmas ennek az egyedi kulcsnak a létrehozására. Ha az első paraméter

értéke `IPC_PRIVATE`, a függvény mindenképpen egy új szemaforkészletet hoz létre, de a rendszer nem garantálja, hogy más folyamatok nem használhatják ezt az új szemaforkészletet.

A második paraméter azt határozza meg, hogy hány szemafor legyen a szemaforkészletben. Ez a szám nem lehet nagyobb, mint az `SEMSL`, amely az egy szemaforkészletben elhelyezhető szemaforok számára nézve szab korlátot.

A harmadik paraméterként átadott kapcsoló a következő állandókból állítható össze *vagy* művelet segítségével:

`IPC_CREAT` Új szemaforkészlet létrehozására ad utasítást ez a kapcsoló. Ha nincs bekapcsolva, a függvény már létező készlethez kísérel meg kapcsolódni.

`IPC_EXCL` Ha ez a kapcsoló is ba van kapcsolva, a függvény hibát ad, ha már létezik szemaforkészlet a hivatkozott kulccsal.

A harmadik paraméterként átadott értékhez szintén a bitenkénti *vagy* művelettel kapcsolhatók a jogokat beállító állandók, amelyeket az állománykezelésnél is használhatunk (például `S_IRUSR`, `S_IWUSR` stb.). Ezek az állománykezelésnél megismert formában szabályozzák a szemaforkészlethez való hozzáférést.

A függvény visszatérési értéke a szemaforkészlet azonosítószáma, ha a művelet sikeres volt, és -1, ha nem. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

`EACCES` A kulcsnak megfelelő szemaforkészlet létezik, de a folyamatnak nincs jog a szemaforkészlet használatához.

A szemaforkészlethez kapcsolódó jogokat a `semget()` harmadik paraméterével állíthatjuk be, azok tulajdonosa, tulajdonoscsoportja és a hozzá kapcsolódó jogok az állományoknál megszokott módon értelmezhetők.

`EEXIST` A kulcsnak megfelelő szemaforkészlet már létezik, a `semget()` hívásakor azonban az `IPC_CREAT` és `IPC_EXCL` függvényekkel új szemaforkészlet létrehozását írtuk elő.

`ENOENT` A kulcsnak megfelelő szemaforkészlet nem létezik és nem írtuk elő a létrehozását az `IPC_CREAT` kapcsolóval.

`EINVAL` A második paraméterként megadott szemaforszám értéke érvénytelen vagy nagyobb, mint a kulcsnak megfelelő, már létező szemaforkészletben található szemaforok száma.

`ENOMEM` A művelet végrehajtásához nem állt rendelkezésre elegendő memória.

`ENOSPC` A rendszeren több szemafor már nem hozható létre, mert a szemaforok megengedett legnagyobb számát elértek.

```
int semctl(int semid, int semnum, int cmd, union semun);
```

A függvény segítségével különleges műveleteket végezhetünk a szemaforkészleten vagy annak egy tagján. mindenéppen el kell végeznünk a szemaforkészlet alapbeállítását ezzel a függvényel, miután létrehoztuk azt a `semget()` segítségével.

A függvény negyedik paramétereként átadott paraméter típusa nem minden esetben van létrehozva a fejállományokban, ezért a GNU C könyvtár dokumentációja szerint a következő sorokat kell elhelyeznünk a program elején a fejállományok betöltése után:

```
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
/* Már létre van hozva. */
#else
/* Magunknak kell létrehoznunk. */
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *_buf;
};
#endif
```

A függvény első paramétere a szemaforkészlet azonosítója, amelyen a különleges műveletet el akarjuk végezni.

A függvény második paramétere a kezelní kívánt szemafor sorszáma a szemaforkészletben. A szemaforok számozása 0-tól kezdődik.

A függvény harmadik paramétere egy állandó, amely az elvégezni kívánt műveletet adja meg. Itt – többek között – a következő műveleteket adhatjuk meg:

**IPC\_RMID** A szemaforkészlet azonnali megsemmisítése. Az eltávolítás után a rendszer feloldja azoknak a folyamatoknak a blokkolását, amelyek az adott szemaforkészletre várakoztak, és jelzi számukra, hogy a szemaforkészlet törölve lett.

Ha a függvény hívása ezzel a parancsal történik, a negyedik paraméter lehet `NULL` vagy elhagyható, a függvény három paraméterrel hívható.

**SETVAL** A szemafor értékének beállítása a negyedik paraméterként megadott változó var mezőjének megfelelően. A művelet végrehajtása után a rendszer szükség esetén feloldja a várakozó folyamatok blokkolt állapotát.

Ezt az műveletet – a szemaforok értékének előírt értékre változtatását – általában csak a szemafor kezdeti beállításánál használjuk. A szemaforon használat közben csak a csökkentés és növelés műveletét alkalmazzuk.

A függvény visszatérési értéke sikeres végrehajtás esetén nem negatív, sikertelen végrehajtás esetén pedig -1. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

`EACCES` A folyamatnak nincs joga az adott műveletet végrehajtani.

`EFAULT` A negyedik paraméter értéke hibás.

`EIDRM` A szemaforkészlet törlőve lett.

`EINVAL` Az első vagy a harmadik paraméter – a szemaforkészlet azonosítója vagy a parancs – értéke érvénytelen.

`EPERM` A folyamatnak nincs joga eltávolítani a szemafort.

`ERANGE` A szemafor új értékeként megadott érték érvénytelen.

A szemafor értéke nem lehet 0-nál kisebb vagy a `SEMVMX` állandó értéké-nél nagyobb.

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

A függvény segítségével használhatjuk a szemafort, azaz csökkenthetjük vagy növelhetjük az értékét. A `semctl()` függvénnyel ellentében – amellyel különleges műveleteket végezhetünk a szemaforkészleten – ez a függvény a normális használatot teszi lehetővé. A versenyhelyzet kizárasát az garantálja, hogy a `semop()` hívásakor, ha szükséges, a rendszer blokkolja a folyamatot.

A függvény első paramtere a szemaforkészletet azonosítja, amelynek elemén vagy elemein a műveletet el kívánjuk végezni.

A függvény harmadik paramétere azt adja meg, hogy a szemaforkészlet hány elemén kívánunk műveletet végezni. Ha ennek a paraméternek az értéke 1, csak egy szemaforon végünk műveletet, ha 2, akkor kettőn és így tovább.

Fontos tudnunk, hogy ha több szemaforon végünk műveletet, a rendszer minden szemaforon egyszerre végzi el az érték módosítását, de csak akkor, ha minden módosítás elvégezhető. Ha csak egyetlen művelet is található, amely nem végezhető el – mert a szemafor értéke 0-nál kisebbé válna – a folyamat blokkolódik.

A függvény harmadik paramétere egy mutató, amely egy tömböt jelöl ki a memoriában. Ennek a tömbnek minden eleme egy szemaforműveletet tartalmaz. Értelemszerűen annyi szemaforműveletet kell tartalmazni a második paraméterrel jelölt tömbnek, ahány szemaforon műveletet kívánunk végezni. Így a második paraméterrel jelölt tömb elemeinek számát a harmadik paraméter adja meg.

A második paraméterrel jelölt tömb minden eleme `sembuf` struktúra, amelynek szerkezete a következő:

A sembuf struktúra	
unsigned short sem_num	A szemafor sorszáma a szemaforkészletben.
short sem_op	A szemaforon elvégzendő művelet.
short sem_flg	A művelet módosítókapcsolója.

A struktúrában az első elem, a `sem_num`, annak a szemafornak a sorszámát adja meg, amelyen a műveletet el akarjuk végezni. A szemaforok számozása 0-tól kezdődik.

A struktúra második eleme a szemaforon elvégzendő műveletet adja meg. A műveletet alapvetően meghatározza ennek az értéknek az előjele, a következőképpen:

1. Ha a második elem értéke nullánál nagyobb egész, az adott szemafor ezzel az értékkel növekszik. Ez a művelet soha nem okozza a folyamat blokkolását.

Ezt a műveletet általában a kritikus szakasz végén használjuk, azt jelezük vele, hogy a szemaforral védett közös erőforrás használatát befejeztük.

Ha létezik olyan folyamat, amely a szemafor miatt blokkolt állapotban van, a művelet végrehajtása után a rendszer – ha lehetséges – feloldja a blokkolását.

2. Ha a második elem értéke 0, a szemafor értéke nem változik. Ez a művelet szükség esetén blokkolja a folyamat futását és csak akkor engedélyezi újra, ha a szemafor értéke 0 lesz.

Ez a művelet a 0-ra várakozás, amelyet ritkán használunk, hiszen a szemaforok 0 értéke általában az erőforrás foglalt állapotát jelzi.

3. Ha a második elem értéke negatív, a rendszer megkíséri csökkenteni a szemafor értékét a második elem abszolút értékével. Ha a csökkentés miatt a szemafor értéke negatív lenne, a rendszer blokkolja a folyamat futását, amíg a szemafor nem növekszik akkorára, hogy ez a művelet elvégezhető legyen. (A blokkolást elkerülhetjük, ha a harmadik mezőben nem blokkoló jellegű műveletvégzést kérünk.)

Ezt a műveletet általában a kritikus szakaszba való belépés előtt végezzük el, a szemaforral védett közös erőforrás használatának jelzésére.

A `sembuf` struktúra harmadik eleme a művelet végrehajtását befolyásolja. Itt a következő állandókat és az azokból *vagy* művelettel képzett kifejezéseket adhatjuk meg:

`IPC_NOWAIT` A szemaforművelet nem blokkolja a folyamat futását akkor sem, ha azt nem lehet elvégezni. Ilyen esetben a `semop()` hibával tér vissza és az `errno` értéke `EAGAIN` lesz.

**SEM\_UNDO** A művelet hatását a rendszer visszavonja, amikor a folyamat befejezi futását.

A `semop()` visszatérési értéke 0, ha a művelet sikeresen végrehajtódott, és -1, ha nem. Ekkor a függvény beállítja az `errno` értékét, ami a következők egyike lehet:

**E2BIG** A harmadik paraméterként átadott műveletek száma túl nagy.

**EACCES** A folyamatnak nincs joga a műveletek valamelyikének végrehajtásához.

**EAGAIN** A művelet nem hajtható végre és nem blokkoló jellegű végrehajtást kértünk.

**EFAULT** A második paraméterként átadott cím nem megfelelő.

**EFBIG** A második paraméterként kijelölt tömbben valamelyik művelethez hibás szemaforszám tartozik. Olyan szemaforra hivatkozunk, ami nem létezik a szemaforkészletben.

**EIDRM** A szemafor törölve lett.

**EINTR** A művelet végrehajtása közben a folyamat jelzést kapott.

**EINVAL** Az első paraméterként átadott szemafor azonosító érvénytelen, nem létező szemafort jelöl, vagy a harmadik paraméterként átadott műveletszám érvénytelen.

**ENOMEM** A művelet végrehajtásához nem állt rendelkezésre elegendő memória.

**ERANGE** Valamely szemaforművelet után a szemafor értéke a megengedett értéknél nagyobb volna. A szemafor értéke nem lehet nagyobb, mint a `SEMVMX` állandóban tárolt érték.

A függvények bemutatása után megemlíjtük, hogy az `ipcs` program segítségével a rendszeren található szemaforok kiírathatók, az `ipcrm -s` kapcsolójával pedig törölhetők. A programok használatához természetesen a megfelelő jogkörökkel rendelkeznünk kell.

**56. példa.** A következő példaprogram egy szemafor használatát mutatja be. A program futó példányai egy képzeletbeli erőforrást használnak közösen, szemafor segítségével kizárrva a versenyhelyzetet.

A program az egyszerűség kedvéért a parancssorban kapott kapcsoló alapján működik. Ha a program nem kap kapcsolót, létrehoz egy új szemaforkészletet, elvégzi az alapbeállítását és megkezdi az erőforrás használatát a megfelelő módon használva a szemafort. Ha a program a -c kapcsolót kapja, a már létező szemaforhoz kapcsolódik. Ilyen esetben nem kell elvégeznie a szemafor kezdeti beállítását, azonnal megkezdheti a munkát. Ha a program a -r kapcsolót kapja, megkíséri törölni a szemafort, majd utána azonnal kilép.

A gyakorlatban használt programok természetesen nem kapcsolók szerint kezelik a szemaforokat, a példaprogram csak az egyszerűség kedvéért használja ezt a módszert.

```
megosztott/szemafor.c
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <sys/stat.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/sem.h>
7 #include <string.h>
8 #include <errno.h>
9
10 #if defined(__GNU_LIBRARY__)
11 /* a semun már létre van hozva */
12 #else
13 union semun {
14     int val;
15     struct semid_ds *buf;
16     unsigned short *array;
17
18     struct seminfo *_buf;
19 };
20#endif
21
22 void kiir( char *szoveg ){
23     printf( szoveg );
24     fflush( stdout );
25 }/*kiir*/
26
27 int _szemafor( char *prgnev ){
28     key_t kulcs;
29     int szemafor;
30     kulcs=ftok( prgnev, 1 );
31     if( kulcs== -1 ){
32         fprintf(stderr, "A kulcs létrehozása sikertelen");
33         exit(1);
34 }/*if*/
35
36     szemafor=semget( kulcs, 1, IPC_CREAT|S_IRUSR|S_IWUSR );
37     if( szemafor== -1 ){
38         fprintf( stderr, "semget(): %s\n", strerror(errno));
39         exit( 1 );
40     }else{
```

```
41     printf("Szemafor: %d\n", szemafor);
42     fflush( stdout );
43 }/*if*/
44 return( szemafor );
45}/*_szemafor*/  
  
46  
47 void _szemafor_alapbeall( int szemafor ){
48 union semun alapbeall;
49     alapbeall.val=1;  
50  
51     kiir("A szemafor alapbeállítása...");  
52  
53     if( semctl( szemafor, 0, SETVAL, alapbeall )== -1 ){
54         fprintf( stderr, "semctl: %s\n", strerror(errno) );
55         exit( 1 );
56 }/*if*/
57     kiir("[OK]\n");
58}/*_szemafor_alapbeall*/  
  
59  
60 void _szemafor_torol( int szemafor ){
61     kiir("Szemafor törlése...");  
62     if( semctl( szemafor, 0, IPC_RMID )== -1 ){
63         fprintf( stderr, "semctl: %s\n", strerror(errno) );
64         exit( 1 );
65 }/*if*/
66     kiir("[OK]\n");
67}/*_szemafor_torol*/  
  
68  
69 void zar( int szemafor ){
70     struct sembuf zar;
71     zar.sem_num=0; /* 0-ás szemafor. */
72     zar.sem_op=-1; /* Csökkentés 1-el. */
73     if( semop( szemafor, &zar, 1 )!=0 ){
74         fprintf( stderr, "semop: %s\n", strerror(errno));
75         exit(1);
76 }/*if*/
77}/*zar*/  
  
78  
79 void nyit( int szemafor ){
80     struct sembuf nyit;
81     nyit.sem_num=0; /* 0-ás szemafor. */
82     nyit.sem_op=+1; /* Növelés 1-el. */
```

```
84     if( semop( szemafor, &nyit, 1 )!=0 ){
85         fprintf( stderr, "semop: %s\n", strerror(errno));
86         exit(1);
87     }/*if*/
88 }/*nyit*/  
  
89  
90 int main( int argc, char *argv[]){  
91     int n;  
92     int szemafor;  
93  
94     szemafor=_szemafor( argv[0] );  
95  
96     if( argc>1 && strcmp(argv[1], "-r" )==0 ){
97         _szemafor_torol( szemafor );
98         exit(0);
99     }/*if*/  
100  
101    if( argc>1 && strcmp(argv[1], "-c" )==0 )
102        kiir( "Csatlakozás létező szemaforhoz.\n" );
103    else
104        _szemafor_alapbeall( szemafor );  
105  
106    for( n=0; n<4; ++n){
107        kiir( "Kritikus szakasz..." );
108        zar( szemafor );
109        kiir( "[OK]\n" );
110  
111        /* A kritikus szakasz következik.
112           */
113        sleep( 5 );
114  
115        kiir( "Kritikus szakasz vége..." );
116        nyit( szemafor );
117        kiir( "[OK]\n" );
118  
119        /* A kritikus szakaszon kívüli műveletek.
120           */
121        sleep( 2 );
122    }/*for*/  
123  
124    exit(0);
125 }/*main*/
```

A 23–26. sorban található `kiir()` függvény egy üzenetet ír a szabványos kimenetre, majd gondoskodik az `fflush()` hívásával, hogy az üzenet azonnal meg is jelenjen. Erre azért van szükség, hogy az események egymásutániságát jól megfigyelhessük.

A 28–46. sorban található `_szemafor()` függvény előállítja a program által használt szemaforkészlet azonosítószámát és szükség esetén létre is hozza a szemaforkészletet. A program először egy kulcsot állít elő az `ftok()` függvény segítségével, majd egy szemafort hoz létre a `semget()` hívásával. Ha a szemaforkészlet már létezik, a `semget()` nem hoz létre újat, csak visszaadja a létező szemaforkészlet azonosítószámát.

A 48–59. sorok között található `_szemfor_alapbeall()` függvény a szemaforkészlet alapbeállítását végzi, és 1-re állítja a szemaforkészletben található első szemafor értékét. A függvény az 50. sorban elhelyezi a szemafor értékét az alapbeall változó val mezőjében, majd az 54. sorban hívja a `semctl()` függvényt a `SETVAL` parancccsal utasítva a szemafor értékének beállítására.

A 61–68. sorokban található `_szemafor_torol()` függvény, amely megkíséri a törölni a szemafort. A függvény a 63. sorban hívja a `semctl()` függvényt a `IPC_RMID` parancccsal.

A 70–78. és a 80–89. sorokban található `zar()`, illetve `nyit()` függvények nagyon hasonlítanak egymásra. A `zar()` megkíséri csökkenteni egyelőre a szemafor értékét, ezzel valósítva meg a képzeletbeli erőforrás lefoglalását, a `nyit()` pedig a szemafor növelésével szabadítja fel az erőforrást.

A függvények működésének megértése után a program működése már könnyen átlátható. A 107–123. sorokban található ciklus szerint a program többször foglalja le és szabadítja fel a szemafor segítségével az erőforrást. A 114. és 122. sorban található `sleep()` függvényhívás miatt a program néhány másodpercre megáll, hogy követhessük a működését. Ha grafikus felületen több ablakban, több példányban indítjuk el a példaprogramot – az elsőt paraméter nélkül, a többit pedig a `-c` kapcsolóval – megfigyelhető a szemafor működése.

A szemaforokról szóló rész végén meg kell említenünk, hogy a bemutatott példánál jóval bonyolultabb esetekben is használhatók. Előfordulhat, hogy a feladat bonyolultsága miatt nem elegendő egy szemafor használata, több szemaforral kell védenünk az erőforrást, amelyhez a különféle okokból történő hozzáférést a szemaforok kombinációi szabályozzák. Éppen ez az oka annak, hogy a Linux nem csak egyedi szemaforok kezelésére képes, hanem szemaforokból készített szemafortömbök, szemaforkészletek egyidejű, versenyhelyzetmentes kezelését is lehetővé teszi.

Tudnunk kell azonban, hogy a szemaforok tömbjeivel megoldott feladatok megoldása nehéz feladat elő állítják a programozót, akinek össze kell hangolnia a szemaforok segítségével több folyamat munkáját. Nehéz feladat az egyszerre futó folyamatok történéseit átlátni és komoly nehézséget okoz az is, hogy az esetleges programhibák ritkán előforduló, nehezen tetten érhető rendellenességeket okoznak a közös erőforrás kezelésében. Ilyen, nehezen tetten érhető hiba például a holtpont

jelentkezése, amely az egymást kölcsönösen akadályozó folyamatok révén leállásokat okozhat.

## Ajánlott irodalom

1. DAVID A. CURRY: *UNIX System Programming for SVR4*, O'Reilly, 1996.
2. A GNU C Programkönyvtár dokumentációja: `info libc`
3. BILL O. GALLMEISTER *POSIX.4: Programming for the Real World*, O'Reilly, 1995.
4. PERE LÁSZLÓ: *Linux: Felhasználói ismeretek I.-II.*, Kiskapu, 2002.

# Tárgymutató

$10^x$ , 97  
 $2^x$ , 97  
 $\sqrt[3]{x}$ , 98  
 $\sqrt{x^2 + y^2}$ , 98  
 $\sqrt{x}$ , 98  
 $e^x$ , 97  
 $x^y$ , 97  
,

' ', 48  
\*/ , 26, 27  
+, 122  
-, 122  
->, 52, 53  
-fwriteable-strings, 49  
-lm, 90  
.c, 8  
/\*, 26, 27  
/bin/sh, 179, 180  
/dev/null, 185  
/etc/hosts, 203  
/etc/passwd, 130  
/usr/include/, 19  
#, 122  
#include, 15, 19  
%s, 44  
&, 34  
\_BSD\_SOURCE, 90  
\_GNU\_SOURCE, 70, 113, 116, 164  
\_XOPEN\_SOURCE, 145  
\_\_func\_\_, 105  
{ }, 13  
a.out, 9  
ABDAY\_1, 69  
ABDAY\_7, 69  
ABMON\_1, 69  
ABMON\_12, 69  
accept(), 217, 221, 226  
access(), 142  
acos(), 96  
acosf(), 96  
acosl(), 96  
advisory locking, 170  
AF\_INET, 204, 211  
AF\_INET6, 204  
alarm(), 196  
argc, 57  
argv, 57  
asctime\_r(), 175  
asin(), 95  
asinf(), 95  
asinl(), 95  
atan(), 96  
atanf(), 96  
atanl(), 96  
atof(), 93  
atoi(), 91, 132  
atol(), 91  
atoll(), 91  
awk, 21, 46  
bc, 21, 46  
bin/, 9  
bind(), 212, 216, 220, 226  
bitenkénti és, 48  
bitenkénti vagy, 48  
break, 25  
bsearch(), 84

callback function, 191  
calloc(), 104  
canonicalize\_file\_name(), 164,  
    166  
case, 25, 26  
casting, 35  
cbrt(), 98  
cbrtf(), 98  
cbrtl(), 98  
cc, 9  
char, 27, 29, 30, 43, 46, 73, 121  
char \*, 35, 44, 55  
char \*\*, 55  
char \*\*\*, 56  
char \*\*\*\*, 56  
chdir(), 151, 154  
chmod(), 167  
chown(), 166  
clearenv(), 101  
clock(), 177, 178  
CLOCKS\_PER\_SEC, 178  
close(), 141, 170, 196  
closedir(), 153, 154  
closelog(), 188  
CODESET, 69  
compiling, 17  
connect(), 213, 216  
core, 183  
 $\cos^{-1} x$ , 96  
 $\cos x$ , 95  
 $\cos()$ , 94  
cosf(), 94  
cosl(), 95  
crypt(), 134  
crypt, 134  
crypt.h, 134  
ctime\_r(), 175  
ctype.h, 80, 81  
CURRENCY\_SYMBOL, 70  
D\_FMT, 70  
D\_T\_FMT, 70  
daemon(), 184, 185, 227  
DAY\_1, 69  
DAY\_7, 69  
DECIMAL\_POINT, 70  
default, 26  
dirent, 151  
dirent.h, 151, 152  
domain name service, 203  
double, 30, 89, 93  
drand48(), 100  
DT\_BLK, 152  
DT\_CHR, 152  
DT\_DIR, 152  
DT\_FIFO, 152  
DT\_LNK, 152  
DT\_REG, 152  
DT SOCK, 152  
DT UNKNOWN, 152  
EACCESS, 109, 213  
EADDRINUSE, 213  
EADDRNOTAVAIL, 213  
EBADF, 212  
echo, 57  
effective group ID, 136  
effective user ID, 136  
EINTR, 109, 196  
EINVAL, 109, 201, 213  
EISDIR, 109  
előltesztele ciklus, 22  
ELOOP, 148  
EMFILE, 109  
ENAMETOOLONG, 201  
end of file, 111  
endgrent(), 133  
endpwent(), 132  
ENOENT, 109  
ENOMEM, 109  
ENOSPC, 110  
ENOTDIR, 148  
ENOTSOCK, 213  
environment variables, 100  
EOF, 110, 111, 124, 125  
EROFS, 110

errno, 66  
errno.h, 66  
exclusive lock, 170  
execv(), 181  
exit(), 63, 182, 183  
EXIT\_FAILURE, 67  
EXIT\_SUCCESS, 67  
exp(), 96  
exp10(), 97  
exp10f(), 97  
exp10l(), 97  
exp2(), 97  
exp2f(), 97  
exp2l(), 97  
expf(), 96  
expl(), 97  
  
függvény  
    hívása, 13  
    létrehozása, 13  
    visszatérési értéke, 13  
F\_OK, 142  
fclose(), 110  
fcloseall(), 110  
fcntl.h, 137  
fejállomány, 18  
feltételes utasítésvégrehajtás, 21  
feof(), 114  
fflush(), 114, 118, 246  
fgetc(), 112, 113  
fgets(), 114  
float, 30, 89, 93  
flock(), 170  
FNM\_CASEFOLD, 85  
FNM\_FILE\_NAME, 85  
FNM.LEADING\_DIR, 85  
FNM\_NOESCAPE, 85  
FNM\_NOMATCH, 85  
FNM\_PATHNAME, 85  
FNM\_PERIOD, 85  
fnmatch(), 85  
fnmatch.h, 85  
fopen(), 109  
fork(), 181, 184, 226, 233, 235  
fprintf(), 119, 120, 180  
fputc(), 112  
fputs(), 112  
FRAC\_DIGITS, 70  
free(), 64, 104  
fscanf(), 125  
ftok(), 227, 228, 237, 246  
  
gcc, 9, 49, 134  
getc(), 113  
getchar(), 113  
getcwd(), 150  
getdelim(), 113  
getegid(), 136  
getenv(), 101  
geteuid(), 136  
getgid(), 136  
getgrrent(), 133  
getgrgid(), 131  
getgrnam(), 131  
gethostbyaddr(), 204  
gethostbyname(), 204, 205  
gethostname(), 201, 203  
getline(), 113, 116  
getpid(), 181  
getppid(), 181  
getpwent(), 132, 133  
getpwiud(), 132  
getpwnam(), 130  
getpwuid(), 130, 132  
getuid(), 135  
getumask(), 169  
GID, 130  
global variables, 37  
gmtime\_r(), 175  
group, 130  
group identification number, 130  
grp.h, 129, 130, 132  
  
hátróltesztelő ciklus, 22  
header, 18  
HOST\_NOT\_FOUND, 204, 205  
hostent, 203–205

HOSTNAME, 102  
htonl(), 208  
htons(), 208  
hypot(), 98  
hypotf(), 98  
hypotl(), 98  
  
in6addr\_any, 212  
in\_addr, 211  
INADDR\_ANY, 211  
INADDR\_BROADCAST, 211  
int, 11, 27, 32, 34, 35, 39, 91  
int \*, 34, 35, 39  
INT\_CURR\_SYMBOL, 70  
integer, 11  
inter process communication, 227  
intmax\_t, 121  
IPC\_EXCL, 232  
IPC\_NOWAIT, 241  
IPC\_PRIVATE, 238  
IPC\_RMID, 246  
IPC\_STAT, 231  
ipcrm, 233, 242  
ipcs, 232, 233, 242  
isalnum(), 80  
isalpha(), 80  
isascii(), 81  
isblank(), 80  
isdigit(), 80  
islower(), 80  
ispunct(), 80  
isupper(), 80  
isxdigit(), 80  
  
kill(), 192  
kill, 190, 191, 195  
  
LANG, 69  
langinfo.h, 68, 70, 72  
LC\_ALL, 68, 69  
LC\_COLLATE, 69  
LC\_CTYPE, 69  
LC\_MESSAGES, 69  
LC\_MONETARY, 69  
  
LC\_NUMERIC, 69  
LC\_TIME, 69  
ld, 17  
lfind(), 84  
lgx, 97  
link(), 158, 159  
listen(), 216, 217, 221, 226  
lnx, 97  
local variables, 36  
locale, 69  
locale.h, 68  
localtime(), 175  
localtime\_r(), 175  
localtime\_r, 175  
LOCK\_EX, 170  
LOCK\_NB, 170  
LOCK\_SH, 170  
LOCK\_UN, 170  
log<sub>2</sub>x, 97  
log(), 97  
log10(), 97  
log10f(), 97  
log10l(), 97  
log2(), 97  
log2f(), 97  
log2l(), 97  
LOG\_ALERT, 187  
LOG\_AUTHPRIV, 186  
LOG\_CONS, 186  
LOG\_CRIT, 187  
LOG\_CRON, 186  
LOG\_DAEMON, 186  
LOG\_DEBUG, 187  
LOG\_EMERG, 187  
LOG\_ERR, 187  
LOG\_FTP, 186  
LOG\_INFO, 187  
LOG\_KERN, 187  
LOG\_LOCALn, 187  
LOG\_LPR, 187  
LOG\_MAIL, 187  
LOG\_NEWS, 187  
LOG\_NOTICE, 187

LOG\_PERROR, 186  
LOG\_PID, 186  
LOG\_SYSLOG, 187  
LOG\_USER, 187  
LOG\_UUCP, 187  
LOG\_WARNING, 187  
`logf()`, 97  
`logger`, 187  
logikai és, 48  
logikai vagy, 48  
`logl()`, 97  
`long double`, 30, 89, 93, 121  
`long int`, 27, 91, 92, 121  
`long long int`, 27, 91, 92, 121  
`LONG_MAX`, 92  
`LONG_MIN`, 92  
`longjmp()`, 189, 190, 197, 199  
low level file handling, 137  
`lrand48()`, 100  
`lsearch()`, 84  
`lseek()`, 146  
`m`, 90, 94, 96  
`main()`, 10, 13, 17, 57  
`malloc()`, 63, 64, 74, 75, 103, 104,  
    127, 226  
mandatory locking, 170  
`math.h`, 90, 94, 96  
`memccpy()`, 106  
`memchr()`, 107  
`memcmp()`, 107  
`memcpy()`, 106  
`memmem()`, 107  
`memmove()`, 106  
`memset()`, 107, 108  
`mkdir()`, 162, 165  
`mktimes()`, 175  
`mode_t`, 148, 149  
`MON_1`, 69  
`MON_12`, 69  
`MON_DECIMAL_POINT`, 70  
`MON_THOUSANDS_SEP`, 70  
`mrand48()`, 100  
mutató, 33  
mutató követése, 36  
`NEGATIVE_SIGN`, 70  
`netdb.h`, 203, 204  
`netinet/in.h`, 208, 211, 212  
`nl_langinfo()`, 69, 72  
`NO_ADDRESS`, 204, 205  
`NO_RECOVERY`, 204, 205  
`ntohl()`, 209  
`ntohs()`, 208  
`NULL`, 40, 41, 63  
`O_APPEND`, 139  
`O_CREAT`, 138, 139  
`O_EXCL`, 138, 139  
`O_EXEC`, 138  
`O_FSYNC`, 139  
`O_NOATIME`, 139  
`O_NOLINK`, 138  
`O_NONBLOCK`, 138, 140, 141  
`O_RDONLY`, 138  
`O_RDWR`, 138  
`O_TRUNC`, 138  
`O_WRONLY`, 138  
object code, 17  
`open()`, 137, 139, 140, 158, 196  
`opendir()`, 152–154  
`openlog()`, 186, 187  
`P_CS_PRECEDES`, 70  
`P_SEP_BY_SPACE`, 70  
`passwd`, 129, 130  
`PATH`, 179  
`pclose()`, 179  
`PF_INET`, 210  
`PF_INET6`, 210  
pointer, 33  
pointer dereference, 36  
`popen()`, 179, 180  
port, 211  
`POSITIVE_SIGN`, 70  
`pow()`, 97  
`powf()`, 97

powl(), 97  
pread(), 145  
printf(), 19, 20, 44, 90, 119, 120,  
    122, 124, 125, 127, 188  
program  
    fordítása, 9  
    indítása, 9  
    neve, 9  
psignal(), 192  
putc(), 112  
putchar(), 112  
putenv(), 101  
puts(), 112  
pwd.h, 129, 132  
pwrite(), 145  
  
qsort(), 81–84  
quick sort algorithm, 81  
  
R\_OK, 142  
raise(), 192  
rand(), 98, 99  
RAND\_MAX, 98  
read(), 140, 145, 196, 214  
readdir(), 152–154, 197  
readlink(), 163  
real group ID, 136  
real user ID, 135  
realloc(), 104  
REG\_ESPACE, 87  
REG\_EXTENDED, 86  
REG\_ICASE, 86  
REG\_NEWLINE, 86  
REG\_NOMATCH, 87  
REG\_NOSUB, 86  
REG\_NOTEOL, 86  
REG\_NOTEOL, 86  
regcomp(), 85, 88  
regex.h, 85  
regex\_t, 86  
regexec(), 86, 88  
regmatch\_t, 87, 89  
remove(), 162  
rename(), 161  
  
return(), 63  
return, 13  
return;, 63  
rewinddir(), 153  
rmdir(), 162  
  
s\_addr, 211  
S\_IRGRP, 148  
S\_IROTH, 148  
S\_IRUSR, 148  
S\_IRWXG, 148  
S\_IRWXO, 148  
S\_IRWXU, 148  
S\_ISBLK(), 149  
S\_ISCHR(), 149  
S\_ISDIR(), 149  
S\_ISFIFO(), 149  
S\_ISGID, 148  
S\_ISLNK(), 149  
S\_ISREG(), 149  
S\_ISSOCK(), 150  
S\_ISUID, 148  
S\_ISVTX, 148  
S\_IWGRP, 148  
S\_IWOTH, 148  
S\_IWUSR, 148  
S\_IXGRP, 148  
S\_IXOTH, 148  
S\_IXUSR, 148  
salt, 134  
scanf(), 124, 125, 127, 129  
search.h, 83  
SEEK\_CUR, 146  
SEEK\_END, 146  
SEEK\_SET, 146  
Segmentation fault, 33  
SEM\_UNDO, 242  
semaphore, 236  
sembuf, 240, 241  
semctl(), 239, 240, 246  
semget(), 237–239, 246  
semop(), 240–242  
SEMSL, 238

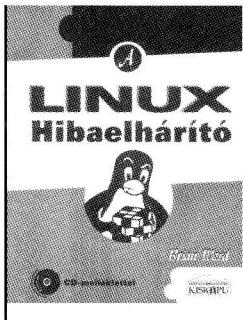
SEMVMX, 240, 242  
sendmail, 180, 181  
setgrent(), 133  
setitimer(), 196  
setjmp(), 189, 190, 197, 199  
setjmp.h, 188  
setlocale(), 68, 72, 76–78, 80, 83,  
    221  
setpwent(), 132, 133  
SETVAL, 246  
shared lock, 170  
shared memory, 103, 227  
SHM\_RND, 229  
shmat(), 229, 230  
shmctl(), 230  
shmdt(), 230  
shmget(), 228, 229, 232  
SHMLBA, 229  
short int, 27, 121  
SHUT\_RD, 218  
SHUT\_RDWR, 218  
SHUT\_WR, 218  
shutdown(), 218, 226  
SIG\_BLOCK, 194  
SIG\_DFL, 192  
SIG\_IGN, 192  
SIG\_SETMASK, 194  
SIG\_UNBLOCK, 194  
sigaddset(), 193  
SIGALRM, 196  
SIGBUS, 195  
SIGCHLD, 196, 227  
sigdelset(), 194  
sigemptyset(), 193  
sigfillset(), 193  
SIGFPE, 195  
SIGHUP, 196  
SIGILL, 195  
SIGINT, 195, 198  
SIGKILL, 195  
signal, 96, 190  
signal(), 191, 192  
signal handler, 188  
signal.h, 191, 193, 195  
signed, 29  
signed char, 121  
signed int, 29  
sigprocmask(), 194  
SIGSEGV, 195  
sigset\_t, 193  
SIGTERM, 195  
 $\sin^{-1} x$ , 95  
 $\sin x$ , 94  
 $\sin()$ , 94  
 $\sincos()$ , 95  
 $\sincosf()$ , 95  
 $\sincosl()$ , 95  
 $\sinf()$ , 94  
 $\sinl()$ , 94  
size\_t, 113, 121  
sizeof(), 27, 29, 63  
sleep(), 198, 246  
snprintf(), 119  
SOCK\_DGRAM, 210, 214  
SOCK\_STREAM, 210, 216  
sockadd\_in, 212  
sockaddr, 212, 213  
sockaddr\_in, 211–213, 217  
sockaddr\_in6, 211–213, 217  
socket, 201, 209  
socket(), 209, 211, 212, 217, 226  
socket.h, 216  
sprintf(), 90, 119  
sqrt(), 98  
sqrtf(), 98  
sqrtl(), 98  
rand(), 99, 100  
rand48(), 100  
sscanf(), 125  
standard input/output, 19  
stat(), 147, 155, 225  
stat, 146, 148, 158  
stderr, 108  
stdin, 108  
stdio.h, 19, 108, 109, 112, 113,  
    119, 124, 179

stdlib.h, 63, 67, 81, 83, 90, 91, 93,  
98, 100, 101, 103, 164, 179  
stdout, 108  
stime(), 172  
stpcpy(), 75  
stpncpy(), 75  
strcasecmp(), 77  
strcasestr(), 78  
strcat(), 75  
strchr(), 78  
strcmp(), 76  
strcoll(), 76, 83  
strcpy(), 46, 74, 75  
strcspn(), 79  
strdup(), 74  
stream, 108, 109  
strerror(), 66, 67, 70, 220  
strftime(), 70, 175  
string.h, 67, 73, 74, 76, 77, 106  
strlen(), 45, 47, 73  
strncasecmp(), 77  
strncat(), 76  
strncmp(), 77  
strncpy(), 74  
strndup(), 75  
strupr(), 79  
strrchr(), 78  
strspn(), 78, 79  
strstr(), 78  
strtod(), 93, 94, 117  
strtodf(), 93, 94  
strtol(), 91–93  
strtold(), 93, 94  
strtoll(), 91, 92  
strtoul(), 92  
strtoull(), 92  
struct, 50  
SUID, 135  
switch, 24, 25  
symlink(), 163  
sys/file.h, 170  
sys/ipc.h, 227, 237  
sys/sem.h, 237  
sys/shm.h, 227  
sys/socket.h, 209, 212  
sys/stat.h, 146–148, 166, 169  
sys/types.h, 129, 132, 135, 145,  
152, 158, 166, 209, 227,  
237  
syslog(), 187, 225  
syslog.h, 186  
system(), 179, 180  
többszörös elágazás, 24  
tömb, 31  
T\_FMT, 70  
típusok  
    egész, 27  
    karakterlánc, 43  
    lebegőpontos, 30  
    mutató, 33  
    mutatótömbök, 56  
    struktúrák, 49  
    struktúratömbök, 58  
    tömb, 31  
tan(), 95  
tanf(), 95  
tanl(), 95  
TCP, 210  
TEMP\_FAILURE\_RETRY(), 144  
tgx, 95  
tg<sup>-1</sup>x, 96  
time(), 99, 172–174  
time, 178  
time.h, 172, 174, 175, 177  
time\_t, 172–175  
tm, 173–175  
tm\_gmtoff, 174  
tm\_hour, 174  
tm\_isdst, 174  
tm\_mday, 174  
tm\_min, 174  
tm\_mon, 174  
tm\_sec, 174  
tm\_wday, 174  
tm\_yday, 174

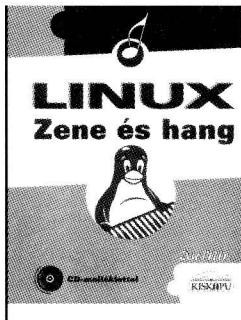
tm\_year, 174  
tm\_zone, 174  
toascii(), 81  
tolower(), 81  
toupper(), 81  
transmission control protocol, 210  
truncate(), 160  
TRY AGAIN, 204, 205  
typedef, 53, 54  
  
UDP, 210  
UID, 129  
uint32\_t, 211  
uintmax\_t, 121  
umask(), 169  
umask, 169  
ungetc(), 114  
unistd.h, 135, 137, 145, 150, 158,  
    166, 184, 201  
unlink, 158  
unlink(), 159, 162  
unsigned, 29, 30  
unsigned char, 121  
unsigned long int, 92, 121  
unsigned long long int, 92, 93,  
    121  
unsigned short int, 121  
USER, 102  
user datagram protocol, 210  
user identification number, 129  
utimbuff, 168  
utime(), 168  
  
változók  
    általános érvényű, 37  
        helyi, 36  
    void, 63  
    void \*, 63  
  
W\_OK, 142  
WAIT\_ANY, 182  
waitpid(), 181, 182, 196, 227  
warning, 35  
WCOREDUMP(), 183  
  
WEXITSTATUS(), 183  
while, 23, 45  
WIFEXITED(), 182, 183  
WIFSIGNALED(), 183  
WIFSTOPPED(), 183  
write(), 140, 145, 196, 214  
WSTOPSIG(), 183  
WTERMSIG(), 183  
  
X\_OK, 142  
  
zombi, 227

# LINUX világába

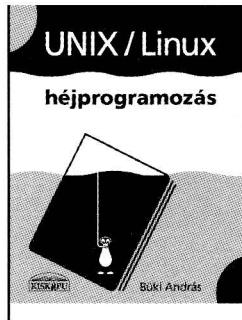
## Kapu a



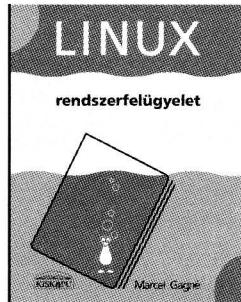
Ár: 3220 Ft  
281 oldal  
felhasználói szint:  
kezdő, haladó  
melléklet: CD



Ár: 4900 Ft  
397 oldal  
felhasználói szint:  
kezdő, haladó  
melléklet: CD



Ár: 2660 Ft  
256 oldal  
felhasználói szint:  
kezdő-haladó



Ár: 6440 Ft  
672 oldal  
felhasználói szint:  
kezdő–profi



Ár: 2660 Ft  
256 oldal  
felhasználói szint:  
kezdő

[www.kiskapu.hu](http://www.kiskapu.hu)

