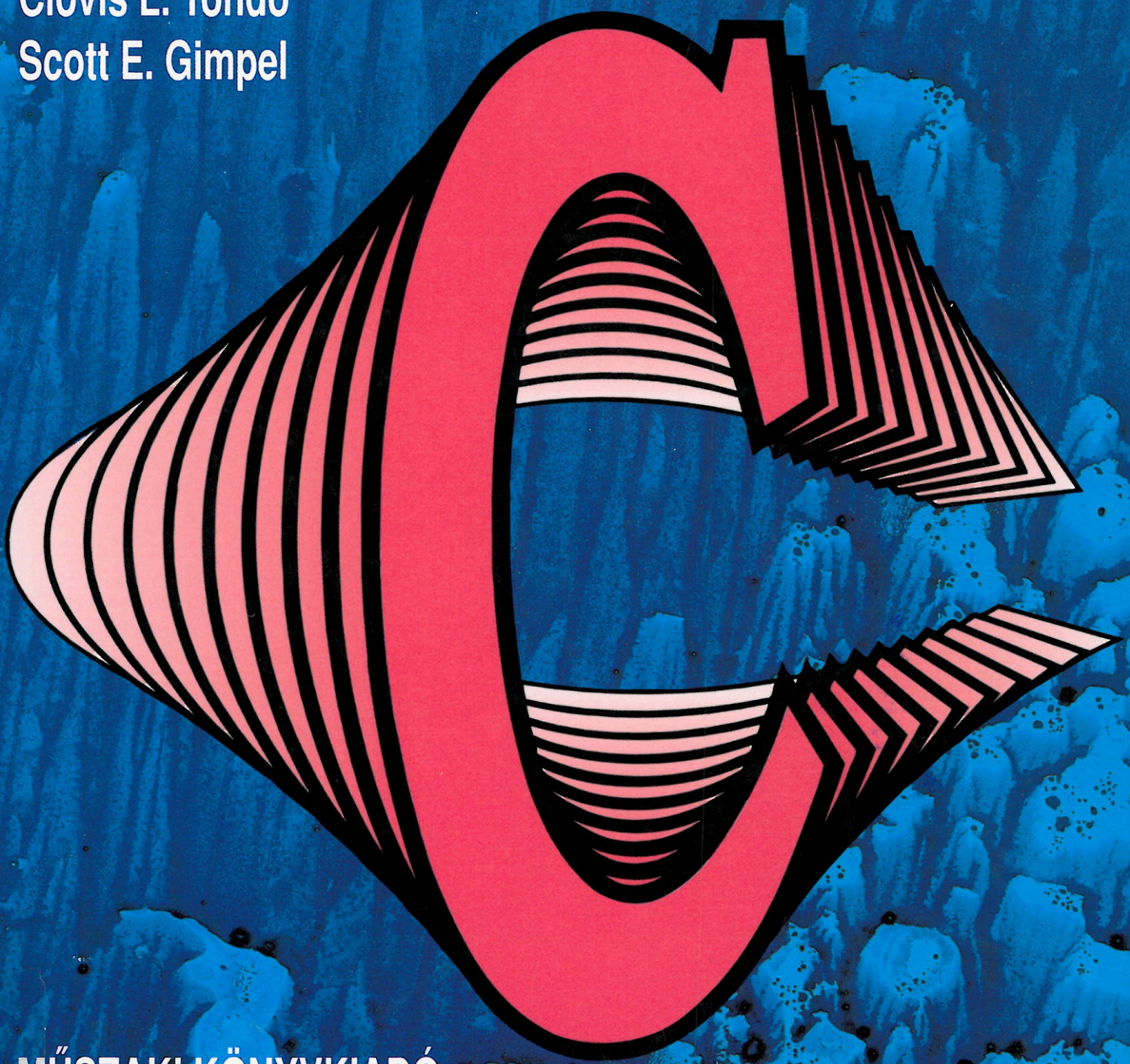
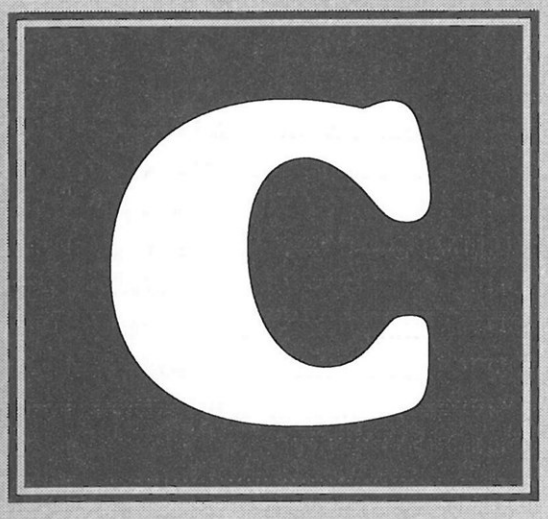
C PROGRAMOZÁSI FELADATOK MEGOLDÁSAI

Clovis L. Tondo
Scott E. Gimpel



MŰSZAKI KÖNYVKIADÓ

Clovis L. Tondo
Scott E. Gimpel



programozási feladatok megoldásai

Műszaki Könyvkiadó, Budapest

Az eredeti mű:
Clovis L. Tondo–Scott E. Gimpel
The C answer book
Prentice Hall, Inc.
Original English language edition published by

ETO: 519.682 C
ISBN 963 16 1067 5

Kiadja a Műszaki Könyvkiadó
Felelős kiadó: Bérczi Sándor ügyvezető igazgató
Felelős szerkesztő: Molnár Ervin okl. villamosmérnök

Franklin Nyomda és Kiadó Kft.
Felelős vezető: a nyomda ügyvezető igazgatója

Műszaki szerkesztő: Bagi Miklós
A borítót tervezte: Kováts Tibor
A könyv formátuma: B/5
Ívterjedelem: 15,5 (A5)
Azonossági szám: 10 287/40
A kézirat lezárva: 1996. szeptember
Készült az MSZ 5601:1983 és 5602:1983 szerint

Tartalom

Előszó

Ez a könyv Brian W. Kernighan és Dennis M. Ritchie: *A C programozási nyelv* című könyv második, átdolgozott kiadásában (amelyet a Prentice Hall 1988-ban, a Műszaki Könyvkiadó 1996-ban jelentetett meg) közölt feladatok megoldásait tartalmazza.

A C programozási nyelv című könyv átdolgozását az tette szükségessé, hogy az ANSI (Amerikai Szabványügyi Hivatal) az első kiadás megjelenését követően szabványosította a C nyelvet. Kernighan és Ritchie ezért teljesen átírta a könyvét, figyelembe véve a szabvány előírásait, és emiatt az első kiadáshoz készített, önálló kötetben megjelentetett feladatmegoldásokat nekünk is át kellett dolgoznunk. Ez az átdolgozott könyv részben *A C programozási nyelv* új feladatainak megoldását tartalmazza, részben pedig már a szabvány előírásait kielégítő megoldásokat ismerteti.

Ezen könyv – K–R könyvével együttesen történő – gondos tanulmányozása várhatóan nagyban segíteni fogja a C programozási nyelv megértését, valamint a megfelelő programozási stílus és gyakorlat elsajátítását. Azt javasoljuk, hogy az Olvasó először a K–R könyvből tanulja meg a C nyelv megfelelő részét, majd oldja meg az ott szereplő feladatokat, és csak ezek után kezdje tanulmányozni az itt közölt megoldásokat. Az egyes megoldások mindig csak a K–R könyvben az adott feladatig tárgyalt ismeretekre támaszkodnak. A későbbiekben, amikor az Olvasó már jobban megismerte a C programozási nyelvet, valószínűleg az itt közöltnél jobb megoldásokat is találhat. Például az

```
if (kifejezés)
    1. utasítás
else
    2. utasítás
```

utasítást a K–R csak a 35. oldalon tárgyalja, így az ezt megelőző gyakorlatok megoldásánál nem használhatjuk azt. Természetesen az 1.8., 1.9. és 1.10. gyakorlatok (K–R 34. oldalán) megoldása az `if - else` utasítás felhasználásával egyszerűsíthető. Az adott feladatok didaktikai szempontok szerint kialakított megoldásai mellett közöljük a javított megoldásokat is.

Az itt közölt megoldásokhoz magyarázatokat is fűzünk. Feltételezzük, hogy az Olvasó már tanulmányozta a K–R adott feladatig terjedő részét, ezért nem akarjuk megismételni a K–R-ben leírtakat, viszont rávilágítunk az egyes megoldások lényeges részeire.

Egy programozási nyelv nem tanulható meg a nyelvi konstrukciók puszta olvasásával! Ehhez programozási gyakorlat kell, azaz saját magunknak kell programokat írni, ill. mások programjait kell tanulmányozni. Könyvünkben mi kihasználtuk a C programozási nyelv kedvező tulajdonságait: modulárissá tettük a programjainkat, bátran használtuk a könyvtári eljárásokat és úgy alakítottuk ki a program formáját, hogy abból világosan látsszon annak logikai menete. Reméljük, könyvünk hozzásegíti az Olvasót ahhoz, hogy gyakorlott C programozóvá váljon.

Itt szeretnénk köszönetet mondani mindazoknak a barátainknak, akik a könyv megírásában segítségünkre voltak. Ők a következők: Brian Kernighan, Don Kostuch, Bruce Leung, Steve Mackey, Joan Magrabi, Julia Mistrello, Rosemary Morrissey, Andrew Nathanson, Sophie Papanikolaou, Dave Perlin, Carlos Tondo, John Wait és Eden Yount.

Clovis L. Tondo

Előszó a magyar kiadáshoz

Már a Kernighan–Ritchie: *A C programozási nyelv* című könyv első magyar kiadásakor (1985-ben) is sokan hiányolták a könyvben levő példák megoldásait. Most, a szerzők átdolgozott, az ANSI szabvány szerint kibővített alapkönyvének megjelenésekor eleget kívántunk tenni a kéréseknek és itt adjuk közre (az első kiadáshoz képest lényegesen bővített) példaanyag megoldásait.

Ahogyan a K–R könyv programjainak többségénél, úgy ebben a példatárban is igyekeztünk a programokat „magyarítani", azaz a függvények és változók nevét, ill. azonosítóit a megfelelő beszédes nevekkel vagy azonosítókkal helyettesíteni. Reméljük, ezzel segítséget nyújtunk a példák tanulmányozásához. Szintén ezt a célt szolgálja, hogy az egyes példaprogramok működésének leírását sokszor szűkszavúnak találtuk és további magyarázatokkal egészítettük ki.

Reméljük, hogy a feladatgyűjtemény példáinak elemzése segítséget fog nyújtani a C nyelvet tanulók számára.

A Kiadó

1.1. gyakorlat

Futassuk a "Halló mindenki!" szöveget kiíró programot a saját rendszerünkön! A programból hagyjunk el részeket és figyeljük meg, milyen hibajelzést ad a rendszer (K–R: 22. oldal)!

```
#include <stdio.h>

main()
{
        printf("Halló mindenki!");
}
```

Ebben a példában az újsor-karakter (\n) hiányzik, és ennek hatására a kurzor a sor végén marad.

A második,

```
#include <stdio.h>

main()
{
        printf("Halló mindenki!\n")
}
```

példában a pontosvessző hiányzik a `printf()` utasítás végén. A C nyelvű program minden utasítását pontosvesszővel kell zárni (K–R: 24. old.). A példa fordítása során a fordítóprogram érzékeli a pontosvessző hiányát, és hibaüzenetet ad.

A harmadik,

```
#include <stdio.h>

main()
{
        printf("Halló mindenki!\n');
}
```

példában a \n utáni idézőjel (") helyett aposztrófot (') írtunk. Az aposztróf az azt követő kerek vég-zárójellel és a pontosvesszővel egy karaktersorozat részét alkotja (mivel a karaktersorozatot az aposztrófok határolják). A fordítóprogram érzékeli a hibát, és úgy magyarázza, hogy vagy egy idézőjel hiányzik, vagy egy kerek vég-zárójel hiányzik a kapcsos vég-zárójel előtt, vagy a karaktersorozat túl hosszú (nincs lezárva egy újabb aposztróffal), vagy egy újsor-karakter fordul elő a karaktersorozatban.

1.2. gyakorlat

Próbáljuk meg kitalálni, hogy mi történik, ha a `printf` eljárás argumentumában a \x jelsorozat szerepel, ahol x valamilyen nem specifikált vezérlőkarakter (K–R: 22. oldal)!

A jelenség vizsgálatához futtassuk le a következő programot:

```
#include <stdio.h>

main()
{
        printf("Halló mindenki!\y");
        printf("Halló mindenki!\7");
        printf("Halló mindenki!\?");
}
```

A K–R függelékében található *Referencia kézikönyv* az ANSI szabványból vett definíció szerint a következőképpen rendelkezik :

> Ha a \ utáni karakter nem egy specifikált vezérlőkarakter, akkor a program viselkedése definiálatlan.

A kísérlet eredménye az alkalmazott fordítóprogramtól függ. Egy lehetséges eredmény a következő:

```
Halló mindenki!yHalló mindenki!<BELL>Halló mindenki!?
```

ahol a <BELL> a 7-es ASCII kód hatására létrejövő rövid hangjelzést jelenti. Mint azt a későbbiekben látni fogjuk, megengedett, hogy a \ karakter után max. három oktális számjegy következzen, amelyek egy karakter kódját adják meg. Ennek alapján a \7 egy rövid hangjelzésnek felel meg az ASCII karakterkészletben.

1.3. gyakorlat

Módosítsuk a hőmérséklet-átalakító programot úgy, hogy a táblázat fölé fejlécet is nyomtasson (K–R: 27. oldal)!

```
#include <stdio.h>

/* Fahrenheit-fok - Celsius-fok táblázat kiírása  F = 0, 20, */
/* ..., 300 Fahrenheit-fokra; a program lebegőpontos változata */
main()
{
      float fahr, celsius;
      int also, felso, lepes;

      also  = 0;     /* a hőmérséklet tábla alsó határa    */
      felso = 300;   /* a felső határ                      */
      lepes = 20;    /* a lépésköz                         */

      printf("Fahr  Celsius\n");
      fahr = also;
      while (fahr <= felso) {
            celsius = (5.0/9.0) * (fahr - 32.0);
            printf("%3.0f  %6.1f\n", fahr, celsius);
            fahr = fahr + lepes;
      }
}
```

A járulékos

```
printf("Fahr  Celsius\n");
```

utasítás beiktatása a ciklus elé létrehozza a megfelelő oszlopok fölött a fejlécet. A `%3.0f` és a `%6.1f` közé szintén elhelyeztünk két szóközt, hogy a számoszlopok a fejléc alá kerüljenek. A program többi része megegyezik a K–R 25. oldalán közölttel.

1.4. gyakorlat

Írjunk programot, amely a Celsius-fokban adott értékeket alakítja Fahrenheit-fokká (K–R: 27. oldal)!

```
#include <stdio.h>

/* Celsius-fok - Fahrenheit-fok táblázat kiírása C = 0, 20,  */
/* ..., 300 Fahrenheit-fokra; a program lebegőpontos változata */
```

```
main()
{
      float fahr, celsius;
      int also, felso, lepes;

      also  = 0;          /* a hőmérséklet tábla alsó határa */
      felso = 300;        /* a felső határ                   */
      lepes = 20;         /* a lépésköz                      */

      printf("Celsius  Fahr\n");
      celsius = also;
      while (celsius <= felso) {
            fahr = (9.0*celsius)/5.0 + 32.0;
            printf("%3.0f  %6.1f\n", celsius, fahr);
            celsius = celsius + lepes;
      }
}
```

A program 0 és 300 °C tartományban kiírja a Celsius-fokokat, ill. az annak megfelelő Fahrenheit-fok értékeket. A Fahrenheit-fokot a

```
fahr = (9.0*celsius)/5.0 + 32.0
```

utasítással számítjuk ki. A program ugyanazon elv alapján működik, mint a Fahrenheit–Celsius táblázatot kiíró program (K–R: 12. oldal). Az `also`, `felso` és `lepes` egész típusú változók a `celsius` változó alsó, ill. felső határát, valamint lépésközét adják meg. A `celsius` változó kezdőértékéül az alsó határt adjuk meg, és a `while` ciklusban kiszámítjuk a megfelelő Fahrenheit-fok értékeket. Az összetartozó °C és °F értékeket a lépésközzel meghatározott lépésekben íratjuk ki. A `while` ciklus addig ismétlődik, amíg a `celsius` változó értéke nagyobb nem lesz a felső határnál.

1.5. gyakorlat

Módosítsuk a hőmérséklet-átalakító programot úgy, hogy a táblázatot fordított (csökkenő) sorrendben, tehát 300 foktól 0 fokig nyomtassa ki (K–R: 28. oldal)!

```
#include <stdio.h>

/* Fahrenheit-fokról Celsius-fokra átszámító táblázat kiírása */
/* fordított sorrendben */
main()
{
      int fahr;

      for (fahr = 300; fahr >= 0; fahr = fahr - 20)
            printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr - 32));
}
```

A program csak egy lényeges módosítást tartalmaz, a

```
for (fahr = 300; fahr >= 0; fahr = fahr - 20)
```

utasítást. A `for` utasítás első része, a

```
fahr = 300
```

beállítja a Fahrenheit-fokot tartalmazó változó (`fahr`) értékét a felső határra. A `for` ciklus második része, a

```
fahr >= 0
```

azt vizsgálja, hogy a Fahrenheit-fokot tartalmazó változó elérte vagy meghaladta-e az alsó határt. A ciklus addig folytatódik, amíg ebben a ciklusrészben a feltétel igaz. A ciklusban szereplő harmadik kifejezés, a

```
fahr = fahr - 20
```

minden lépésben csökkenti a `fahr` változó értékét a lépésközzel.

1.6. gyakorlat

Igazoljuk, hogy a `getchar() != EOF` kifejezés értéke valóban 0 vagy 1 (K–R: 31. oldal)!

```
#include <stdio.h>

main()
{
        int kar;

        while (kar = getchar() != EOF)
                printf("%d\n", kar);
        printf("%d - EOF esetén\n", kar);
}
```

A

```
kar = getchar() != EOF
```

kifejezés egyenértékű a

```
kar = (getchar() != EOF)
```

kifejezéssel (K–R: 31. oldal). A program ezt a kifejezést felhasználva folyamatosan karaktereket

olvas a standard bemeneti egységről. Amíg a `getchar` eljárás karaktereket tud olvasni, addig nem tér vissza a hívó eljárásba az állományvége jelzéssel, így a

```
getchar() != EOF
```

logikai kifejezés értéke igaz és a `kar` értéke 1. Ha a program eléri az állomány végét, a logikai kifejezés értéke hamis lesz, ekkor a `kar` 0 értéket vesz fel és a ciklus befejeződik.

1.7. gyakorlat

Írjunk programot, ami kiírja az `EOF` értékét (K–R: 31. oldal)!

```
#include <stdio.h>

main()
{
      printf("EOF értéke %d\n", EOF);
}
```

Az `EOF` szimbolikus állandó az `<stdio.h>` könyvtárban van definiálva. A fordítás során a `printf()` argumentumában idézőjeleken kívül elhelyezkedő `EOF` a beépített állományban lévő

```
#define EOF
```

utáni szövegre cserélődik. Ez a könyv (és a K–R is) a programokhoz olyan C rendszert használ, amelyben az `EOF` állandó értéke –1. Az `EOF`-hoz hasonló szabványos szimbolikus állandók használata javítja a programjaink hordozhatóságát.

1.8. gyakorlat

Írjunk programot, ami megszámolja a bemenetre adott szövegben lévő szóközöket, tabulátorokat és újsor-karaktereket (K–R: 34. oldal)!

```
#include <stdio.h>

/* Szóközök, tabulátorok és újsor-karakterek megszámolása */
main()
```

```
{
        int kar, nsz, ntab, nujs;

        nsz  = 0;       /* a szóközök száma          */
        ntab = 0;       /* a tabulátorok száma       */
        nujs = 0;       /* az újsor-karakterek száma */
        while ((kar = getchar()) != EOF) {
                if (kar == ' ')
                        ++nsz;
                if (kar == '\t')
                        ++ntab;
                if (kar == '\n')
                        ++nujs;
                }
        printf("%d %d %d\n", nsz, ntab, nujs);
}
```

A programban az `nsz`, `ntab` és `nujs` egész típusú változókat használjuk az előforduló szóközök, tabulátorok és újsor-karakterek számának tárolására. Ezen változók kezdeti értéke természetesen 0.

A `while` ciklus törzsében érzékeljük a beolvasott szövegben előforduló szóköz, tabulátor és újsor-karaktereket. Az összes `if` utasítás végrehajtódik a ciklustörzs minden egyes lefutása során, és ha a beolvasott karakter nem szóköz, tabulátor vagy újsor-karakter, akkor semmi nem történik. Ha a három vizsgált karakter bármelyikét olvastuk be, akkor a neki megfelelő számláló tartalma eggyel nő (inkrementálódik). A `while` ciklus lezárása után (azaz, amikor a `getchar` eljárás `EOF` jelzéssel tér vissza), a program kiírja a kapott eredményeket.

Az `if - else` utasítást a K–R csak az 1.8. gyakorlat után ismerteti (K–R: 35. oldal). Ennek felhasználásával a feladat megoldása:

```
#include <stdio.h>

/* Szóközök, tabulátorok és újsor-karakterek megszámolása */
main()
{
        int kar, nsz, ntab, nujs;

        nsz  = 0;               /* a szóközök száma          */
        ntab = 0;               /* a tabulátorok száma       */
        nujs = 0;               /* az újsor-karakterek száma */
        while ((kar = getchar()) != EOF)
                if (kar == ' ')
                        ++nsz;
                else if (kar == '\t')
                        ++ntab;
                else if (kar == '\n')
                        ++nujs;
        printf("%d %d %d\n", nsz, ntab, nujs);
}
```

1.9. gyakorlat

Írjunk programot, ami a bemenetre adott szöveget úgy másolja át a kimenetre, hogy közben az egy vagy több szóközből álló karaktersorozatokat egyetlen szóközzel helyettesíti (K–R: 34. oldal)!

```
#include <stdio.h>

#define NEMSZOKOZ 'a'

/* Az üres helyeket egyetlen szóközzel helyettesítő program */
main()
{
      int kar, utolsokar;

      utolsokar = NEMSZOKOZ;
      while ((kar = getchar()) != EOF) {
            if (kar != ' ')
                  putchar(kar);
            if (kar == ' ')
                  if (utolsokar != ' ')
                        putchar(kar);
                  utolsokar = kar;
      }
}
```

A `kar` egész típusú változó a bemenetről éppen beolvasott karakter, az `utolsokar` pedig az előzőleg beolvasott karakter ASCII kódját tárolja. A `NEMSZOKOZ` szimbolikus állandóval tetszőleges, szóköztől különböző kezdeti értéket rendelünk az `utolsokar` változóhoz.

A `while` ciklus törzsében lévő első `if` utasítás figyeli a nem szóköz karakterek előfordulását és azonnal kiírja azokat. A második `if` utasítás a szóköz karaktereket kezeli, amíg a harmadik `if` utasítás azt figyeli, hogy a beolvasott szóköz önmagában áll-e vagy egy szóközökből álló karaktersorozat első karaktere-e. A ciklus utolsó utasítása aktualizálja az `utolsokar` változót (a következő karakter beolvasása előtt elmenti a `kar` tartalmát, ha az szóköz volt).

A K–R csak a 35. oldalon ismerteti az `if - else` szerkezetet. Ennek felhasználásával a program a következő módon írható meg:

```
#include <stdio.h>

#define NEMSZOKOZ 'a'

/* Az üres helyeket egyetlen szóközzel helyettesítő program */
main()
{
      int kar, utolsokar;

      utolsokar = NEMSZOKOZ;
```

```
        while ((kar = getchar()) != EOF) {
                if (kar != ' ')
                        putchar(kar);
                else if (utolsokar != ' ')
                        putchar(kar);
                utolsokar = kar;
        }
}
```

A K–R a logikai OR (||) operátort szintén csak a 35. oldalon ismerteti. Ezt felhasználva a program:

```
#include <stdio.h>

#define NEMSZOKOZ 'a'

/* Az üres helyeket egyetlen szóközzel helyettesítő program */
main()
{
        int kar, utolsokar;

        utolsokar = NEMSZOKOZ;
        while ((kar = getchar()) != EOF) {
                if (kar != ' ' || utolsokar != ' ')
                        putchar(kar);
                utolsokar = kar;
        }
}
```

1.10. gyakorlat

Írjunk programot, ami a bemenetre adott szöveget úgy másolja át a kimenetre, hogy közben a tabulátor karaktereket \t, a visszaléptetés (backspace) karaktereket \b és a fordított törtvonal (backslash) karaktereket \\ karakterekkel helyettesíti! Ezzel az átírással a tabulátor és visszaléptetés karakterek a nyomtatásban is láthatóvá válnak (K–R: 34. oldal).

```
#include <stdio.h>

/* A tabulátorokat és visszaléptetéseket nyomtatható karakterekkel */
/* helyettesítő program */
main()
{
        int kar;

        while ((kar = getchar()) != EOF) {
```

```
            if (kar == '\t')
                  printf("\\t");
            if (kar == '\b')
                  printf("\\b");
            if (kar == '\\')
                  printf("\\\\");
            if (kar != '\b')
                  if (kar != '\t')
                        if (kar != '\\')
                              putchar(kar);
      }
}
```

A bemenetről érkező karakter tabulátor, visszaléptetés és fordított törtvonal, valamint bármi más karakter lehet. Ha a bemenetről nem tabulátor, visszaléptetés vagy fordított törtvonal érkezett, akkor a karaktert a program azonnal kiírja (a ciklustörzs utolsó három `if` utasítása és az azokat követő `putchar` utasítás). Ha a megkülönböztetett karakterek valamelyike érkezett, akkor helyettük a `\t`, `\b` vagy `\\` íródik ki (a ciklustörzs első három `if - printf` utasításpárosai).

A C rendszerben a fordított törtvonalat `'\\'` reprezentálja, ennek megfelelően a két fordított törtvonalat a `printf` utasítás argumentumában elhelyezett `"\\\\"` karaktersorozattal írathatjuk ki.

Ugyanez a program a K–R 35. oldalán leírt `if - else` utasítással:

```
#include <stdio.h>

/* A tabulátorokat és visszaléptetéseket nyomtatható karakterekkel*/
/* helyettesítő program */
main()
{
      int kar;

      while ((kar = getchar())!= EOF)
            if (kar == '\t')
                  printf("\\t");
            else if (kar == '\b')
                  printf("\\b");
            else if (kar == '\\')
                  printf("\\\\");
            else
                  putchar(kar);
}
```

1.11. gyakorlat

Hogyan lehet ellenőrizni a szavakat számláló programot? Milyen bemeneti adatsort kell használni, hogy a legnagyobb valószínűséggel érzékeljük a program esetleges hibáit (K–R: 36. oldal)?

A szószámláló programot először ellenőrizzük bemeneti adatsor nélkül! Ekkor a kiírt eredmény 0 0 0 kell legyen (nulla számú újsor-karakter, nulla számú szó és nulla számú karakter).

Ezután egy egykarakteres szóval folytassuk az ellenőrzést! Ekkor a kiírt eredmény 1 1 2 kell legyen (egy újsor-karakter, egy szó és két karakter – azaz egy betű, amit az újsor-karakter követ).

A következő lépésben az ellenőrzéshez egy kétkarakteres szóval folytassuk az ellenőrzést! Ekkor a kiírt eredmény 1 1 3 kell legyen (egy újsor-karakter, egy szó és három karakter – azaz két betű, amit az újsor-karakter követ).

Mindezek után próbálkozzunk két egykarakteres szóval (a kiírt eredmény helyes működés esetén 2 2 4 kell legyen) és két külön sorba írt egykarakteres szóval (a kiírt eredmény helyes működés esetén 2 2 4 kell legyen).

A hibákat a legnagyobb valószínűséggel felfedő bemeneti adatsorok:

- nincs bemeneti adatsor,
- nincs beírt szó, csak újsor-karakterek,
- nincs beírt szó, csak szóközök, tabulátorok és újsor-karakterek,
- soronként egy szóból (szóközök és tabulátorok nélkül) álló adatsor,
- a sor elején kezdődő szóból álló adatsor,
- néhány szóköz után kezdődő szóból álló adatsor.

1.12. gyakorlat

Írjunk programot, ami a bemenetére adott szöveg minden szavát új sorba írja ki (K–R: 36. oldal)!

```
#include <stdio.h>

#define BENT 1    /* a szó belseje */
#define KINT 0    /* a szón kívül  */
/* A szavakat soronként kiíró program */
main()
{
      int kar, allapot;

      allapot = KINT;
      while ((kar = getchar()) != EOF) {
            if (kar == ' ' || kar == '\n' || kar == '\t') {
                  if (allapot == BENT) {
                        putchar('\n');    /* vége a szónak    */
                        allapot = KINT;
                  }
            } else if (allapot == KINT) {
                  allapot = BENT;     /* a szó kezdete        */
                  putchar(kar);
            } else                    /* a szó belsejében van */
```

```
                putchar(kar);
        }
}
```

Az egész típusú `allapot` változó egy logikai értéket tartalmaz, ami azt jelzi, hogy a feldolgozás éppen a szó belsejében folyik-e vagy sem. Mivel a program indításakor még nincs feldolgozandó adat, így `allapot` kezdeti értéke `KINT`.

A ciklus belsejében az első

```
if (kar == ' ' || kar == '\n' || kar == '\t')
```

utasítás azt vizsgálja, hogy a beolvasott `kar` karakter szóhatároló-e (szóhatároló karakter a szóköz, a tabulátor és az újsor). Ha igen, akkor a második

```
if (allapot == BENT)
```

utasítás azt vizsgálja, hogy a `kar` változóban lévő szóhatároló a szó végét jelzi-e (a szó végén `kar` tartalma szóhatároló, és az `allapot` `BENT` értékű). Szó vége esetén a program kiír egy újsor-karaktert és `KINT`-re állítja az `allapot` változót. Ha a `kar` tartalma szóhatároló, de az `allapot` `KINT` volt, akkor nem történik semmi.

Ha `kar` nem szóhatároló, akkor csak a szó első karaktere vagy a szó belsejében lévő egyik karakter lehet, és mindkét esetben a `kar` tartalmát ki kell írni. Szó kezdetén (ha `kar` nem szóhatároló és `allapot` `KINT` értékű) még az `allapot` tartalma is `BENT` értékre vált.

1.13. gyakorlat

Írjunk programot, ami kinyomtatja a bemenetre adott szavak hosszának hisztogramját! A legcélszerűbb, ha a hisztogramot vízszintesen ábrázoljuk, mert a függőleges ábrázolás túl bonyolult lenne (K–R: 38. oldal).

```
#include <stdio.h>

#define MAXHISZT 15  /* a hisztogram max. hossza */
#define MAXSZO   11  /* egy szó max. hossza      */
#define BENT      1  /* a szó belseje            */
#define KINT      0  /* a szón kívül             */

/* Hisztogram nyomtatása vízszintesen */
main()
{
      int kar, i, nsz, allapot;
      int hossz;    /* az egyes oszlopok hossza              */
      int maxertek; /* szóhossz számláló szh[] vektor max.  */
                                                  /* értéke */
```

```
    int tulcs;              /* a túlcsorduló szavak száma  */
    int szh[MAXSZO];        /*a szóhossz számlálók vektora */

    allapot = KINT;
    nsz = 0;          /* a karakterek száma egy szóban      */
    tulcs = 0;        /*a MAXSZO-nál hosszabb szavak száma */
    for (i = 0; i < MAXSZO; ++i)
        szh[i] = 0;         /* a szószámlálók nullázása    */
    while ((kar = getchar()) != EOF) {
        if (kar == ' ' || kar == '\n' || kar == '\t') {
            allapot = KINT;
            if (nsz > 0)
                if (nsz < MAXSZO)
                    ++szh[nsz];
                else
                    ++tulcs;
            nsz = 0;
        } else if (allapot == KINT) {
            allapot = BENT;
            nsz = 1;              /* egy új szó kezdete */
        } else
            ++nsz;                /* a szó belseje      */
    }
    maxertek = 0;
    for (i = 1; i < MAXSZO; ++i)
        if (szh[i] > maxertek)
            maxertek = szh[i];
    for (i = 1; i <MAXSZO; ++i) {
        printf("%5d - %5d : ", i, szh[i]);
        if (szh[i] > 0) {
            if ((hossz = szh[i]*MAXHISZT/maxertek) <= 0)
                hossz = 1;
        } else
            hossz = 0;
        while (hossz > 0) {
            putchar('*');
            --hossz;
        }
        putchar('\n');
    }
    if (tulcs > 0)
        printf("%d darab szó hossza >= %d\n", tulcs, MAXSZO);
}
```

A szó végét szóköz, tabulátor vagy újsor-karakter jelzi. Ha találtunk egy szót (azaz `nsz > 0`) és annak hossza kisebb a max. szóhossznál (`nsz < MAXSZO`), akkor a program eggyel növeli a megfelelő szóhossz számláló tartalmát (`++szh[nsz]`). Ha a szó hossza nagyobb a megengedettnél (`nsz >= MAXSZO`), akkor a program a `tulcs` változó tartalmát növeli eggyel, amiből a program végén megtudjuk a `MAXSZO`-nál hosszabb vagy azzal egyenlő hosszúságú szavak számát.

Amikor a program az összes szót beolvasta, akkor megkeresi az `szh` tömb elemei közül a legnagyobbat és eltárolja a `maxertek` változóba.

A hossz változó a `szh[i]` tömbelem MAXHISZT és `maxertek` arányában skálázott aktuális értékét tartalmazza. Ha `szh[i] > 0`, akkor a program legalább egy csillagot kinyomtat.

A hisztogram függőleges irányú kinyomtatása a következő programmal oldható meg:

```
#include <stdio.h>

#define MAXHISZT 15  /* a hisztogram max. hossza */
#define MAXSZO   11  /* egy szó max. hossza      */
#define BENT      1  /* a szó belseje            */
#define KINT      0  /* a szón kívül             */

/* Hisztogram nyomtatása függőlegesen */
main()
{
      int kar, i, j, nsz, allapot;
      int maxertek;    /* szóhossz számláló szh[] vektor */
                                        /* max. értéke */
      int tulcs;       /* a túlcsorduló szavak száma     */
      int szh[MAXSZO]; /* a szóhossz számlálók vektora   */

      allapot = KINT;
      nsz = 0;   /* a karakterek száma egy szóban        */
      tulcs = 0; /* a MAXSZO-nál hosszabb szavak száma   */
      for (i = 0; i < MAXSZO; ++i)
            szh[i] = 0; /* a szószámlálók nullázása       */
      while ((kar = getchar()) != EOF) {
            if (kar == ' ' || kar == '\n' || kar == '\t') {
                  allapot = KINT;
                  if (nsz > 0)
                        if (nsz < MAXSZO)
                              ++szh[nsz];
                        else
                              ++tulcs;
                  nsz = 0;
            } else if (allapot == KINT) {
                  allapot = BENT;
                  nsz = 1;            /* egy új szó kezdete */
            } else
                  ++nsz;              /* a szó belseje      */
      }
      maxertek = 0;
      for (i = 1; i < MAXSZO; ++i)
            if (szh[i] > maxertek)
                  maxertek = szh[i];

      for (i = MAXHISZT; i > 0; --i) {
            for (j = 1; j < MAXSZO; ++j)
```

```
                if (szh[j]*MAXHISZT/maxertek >= i)
                        printf(" * ");
                else
                        printf("   ");
            putchar('\n');
        }
        for (i = 1; i < MAXSZO; ++i)
            printf("%4d ", i);
        putchar('\n');
        for (i = 1; i < MAXSZO; ++i)
            printf("%4d ", szh[i]);
        putchar('\n');
        if (tulcs > 0)
            printf("%d darab szó hossza >=  %d\n",
                tulcs, MAXSZO);
}
```

Ez a program a `maxertek` meghatározásáig majdnem azonos a vízszintesen nyomtató programmal. Ez után az `szh` tömb minden egyes elemét skálázni és ellenőrizni kell, hogy az egyes elemeknél egy csillagot ki kell-e nyomtatni. Erre az ellenőrzésre azért van szükség, mivel a hisztogram összes oszlopát egyszerre nyomtatja a program (függőleges elrendezés). Az utolsó két `for` ciklus kinyomtatja a `szh` tömb egyes elemeinek indexét és értékét.

1.14. gyakorlat

Írjunk programot, ami kinyomtatja a bemenetre adott különböző karakterek előfordulási gyakoriságának hisztogramját (K–R: 38. oldal)!

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

#define MAXHISZT  15 /* a hisztogram max. hossza                */
#define MAXKAR   128 /* a lehetséges karakterféleségek száma */
/* A különböző karakterek gyakoriság-eloszlásának kinyomtatása */
/* vízszintes hisztogramban */
main()
{
        int kar, i;
        int hossz;        /* az egyes oszlopok max. hossza      */
        int maxertek ;    /* a ksz karakterszámláló max. értéke */
        int ksz[MAXKAR]; /* a karakterszámlálók vektora         */

        for (i = 0; i < MAXKAR; ++i)
            ksz[i] = 0;
        while ((kar = getchar()) != EOF )
            if (kar < MAXKAR)
```

```
                ++ksz[kar];
        maxertek = 0;
        for (i = 1; i < MAXKAR; ++i)
                if  (ksz[i] > maxertek)
                        maxertek = ksz[i];
        for (i = 1; i < MAXKAR; ++i) {
                if (isprint(i))
                        printf("%5d - %c - %5d : ", i, i, ksz[i]);
                else
                        printf("%5d -    - %5d : ",i, ksz[i]);
                if (ksz[i] > 0)   {
                        if ((hossz = ksz[i]*MAXHISZT/maxertek) <= 0)
                                hossz = 1;
                } else
                        hossz = 0;
                while (hossz > 0) {
                        putchar('*');
                        --hossz;
                }
                putchar('\n');
        }
}
```

A program az 1.13. gyakorlat vízszintes hisztogram nyomtató programjához hasonló, csak az egyes karakterek előfordulását számolja. A számláláshoz egy `MAXKAR` elemű tömböt használunk, és egyszerűen elhagyjuk a `MAXKAR`-ral egyenlő vagy nagyobb kódú karaktereket (ha vannak ilyenek a karakterkészletben). Az előző programhoz képest a másik különbség, hogy az `isprint` makrót használjuk a karakter nyomtathatóságának meghatározásához.

Ez a makró a `<ctype.h>` headerben található (K–R: B. függelék), és a header beépítését a K–R az 57. oldalon tárgyalja.

1.15. gyakorlat

Írjuk át a K–R 1.2. pontjában leírt hőmérséklet-átalakító programot úgy, hogy az átalakításhoz függvényt használunk (K–R: 41. oldal)!

```
#include <stdio.h>

float celsius(float fahr);

/* Fahrenheit-fok-Celsius-fok táblázat kiírása F = 0,  */
/* 20, ..., 300 Fahrenheit-fokra; lebegőpontos változat    */
main()
```

```
{
        float fahr;
        int also, felso, lepes;

        also  = 0;                  /* a táblázat alsó határa  */
        felso = 300;                /* a táblázat felső határa */
        lepes = 20;                 /* a táblázat lépésköze    */

        fahr = also;
        while (fahr <= felso) {
                printf("%3.0f %6.1f\n", fahr, celsius(fahr));
                fahr = fahr + lepes;
        }
}

/* celsius függvény: a Fahrenheit-fokot Celsius-fokká alakítja */
float celsius(float fahr)
{
        return(5.0/9.0)*(fahr - 32.0);
}
```

A program a Fahrenheit-fokot egy függvénnyel alakítja Celsius-fok értékké. A függvény neve `celsius`, egy `float` típusú argumentumot igényel és egy `float` típusú eredménnyel tér vissza. A visszatérési érték a `return` utasítás után elhelyezett kifejezés értéke. Néha a függvény visszatérési értékét megadó kifejezés csak egy egyszerű változó (mint pl. a K–R 41. oldalán ismertetett `power` függvény esetén), máskor sokkal összetettebb (mint a jelen példában), de mindkét esetben a visszatérési érték előállítása megoldható a `return` utasítással.

A függvényt

```
float celsius(float fahr);
```

formában deklaráltuk, mivel `float` típusú argumentumot vár, és `float` típusú értékkel tér vissza.

1.16. gyakorlat

Módosítsuk a leghosszabb sort kiíró programot úgy, hogy helyesen adja meg a tetszőlegesen hosszú bemeneti sor méretét és annak szövegéből a lehető legtöbbet írja ki (K–R: 45. oldal)!

```
#include <stdio.h>
#define MAXSOR  1000  /* a bemeneti sor max. mérete */

int getline(char sor[], int maxsor);
void copy(char ba[], char bol[]);
```

```
/* A leghosszabb sor kiírása */
main()
{
        int hossz;                  /* az aktuális sor hossza    */
        int max;                    /* az eddigi leghosszabb sor */
        char sor[MAXSOR];           /* az aktuális sor           */
        char leghosszabb[MAXSOR];/*ide teszi a leghosszabb sort */

        max = 0;
        while ((hossz = getline(sor, MAXSOR)) > 0) {
                printf("%d, %s", hossz, sor);
                if (hossz > max) {
                        max = hossz;
                        copy(leghosszabb, sor);
                }
        }
        if (max > 0)                        /* volt sor, nem EOF */
            printf("%s", leghosszabb);
        return 0;
}

/* getline: egy sort beolvas s-be és visszaadja a hosszát */
int getline(char s[], int lim)
{
        int kar, i, j;

        j = 0;
        for (i = 0; (kar = getchar()) != EOF && kar != '\n'; ++i)
                if (i < lim - 2) {
                        s[j] = kar; /* a tömbhatárokon belül van */
                        ++j;
                }
        if (kar == '\n') {
                s[j] = kar;
                ++j;
                ++i;
        }
        s[j] = '\0';
        return i;
}

/* copy: a "ba" helyre másol a "bol" helyről */
void copy (char ba[], char bol[])
{
        int i;

        i = 0;
        while ((ba[i] = bol[i]) != '\0')
                ++i;
}
```

A `main` eljárásban az egyedüli módosítás a

```
printf("%d, %s", hossz, sor);
```

utasítás, ami kiírja a bevitt sor hosszát (`hossz`) és annyi karaktert, amennyit csak el lehetett menteni a `sor` tömbben.

A `getline` függvényben kevés módosítás történt. A

```
for (i = 0; (kar = getchar()) != EOF && kar != '\n'; ++i)
```

ciklus felső határát a karaktersorozat vége (`EOF` vagy újsor) adja és így a lépésszám tetszőleges lehet. Ez a felső határ nem egy rögzített szám, ezért a `getline` függvény valóban a tetszőleges hosszúságú bemeneti sor hosszát (ami `i` értéke a ciklus végén) adja vissza és amennyit csak lehetséges, elment a szövegből. Az

```
if (i < lim - 2)
```

megvizsgálja, hogy van-e még elegendő hely a tömbben (a tömbhatárokon belül vagyunk-e). A program eredeti változatában (K–R: 43. oldal) a `for` ciklus végfeltétele

```
i < lim - 1
```

volt. Ezt megváltoztattuk, mert az `s` tömb utolsó használható indexe `lim - 1`, mivel `s`-nek `lim` számú eleme van és a bemenetről már beolvastunk egy karaktert. Az

```
i < lim - 2
```

feltétel így helyet hagy egy újsor-karakternek, amit az

```
s[lim - 2] = '\n'
```

utasítással, valamint a karaktersorozat végjelző karakterének, amit az

```
s[lim - 1] = '\0'
```

utasítással helyezünk el a tömbben. A `getline` függvény a karaktersorozat hosszát visszatéréskor az `i` változó tartalmazza és `j` értéke az `s` tömbbe másolt karakterek száma.

1.17. gyakorlat

Írjunk programot, ami kiírja az összes, 80 karakternél hosszabb bemeneti sort (K–R: 45 oldal)!

```
#include <stdio.h>
#define MAXSOR  1000          /* a bemeneti sor max. hossza */
#define HOSSZUSOR 80
```

```
int getline(char sor[], int maxsor);

/* A HOSSZUSOR-nál hosszabb sorok kiírása */
main()

{
      int hossz;                /* az aktuális sor hossza    */
      char sor[MAXSOR];         /* az aktuális bemeneti sor */

      while ((hossz = getline(sor, MAXSOR)) > 0)
            if (hossz > HOSSZUSOR)
                  printf("%s", sor);
      return 0;
}
```

A program a bemeneti sor beolvasásához a `getline` függvényt hívja. A `getline` a beolvasott sor hosszával és a beolvasott sorból a lehetséges leghosszabb eltárolható karaktersorozattal tér vissza a hívó eljárásba. Ha a sor hossza 80 karakternél (`HOSSZUSOR`) nagyobb, akkor a program kiírja a bemeneti sort (vagy legalábbis annak a lehetséges leghosszabb részét). A rövidebb sorokkal nem történik semmi. A `main` eljárás `while` ciklusa addig ismétlődik, amíg a `getline` függvény egyszer nulla sorhosszal nem tér vissza.

A `getline` függvény azonos az 1.16. gyakorlatban használt függvénnyel.

1.18. gyakorlat

Írjunk programot, ami eltávolítja a beolvasott sorok végéről a szóközöket és tabulátorokat, valamint törli a teljesen üres sorokat (K–R: 45. oldal)!

```
#include <stdio.h>
#define MAXSOR  1000 /* a bemeneti sor max. hossza */

int getline(char sor[], int maxsor);
int remove(char s[]);

/* A sorvégi szóközök és tabulátorok eltávolítása, */
/* valamint az üres sorok törlése */
main()
{
      char sor[MAXSOR];         /* az aktuális bemeneti sor */

      while (getline(sor, MAXSOR) > 0)
            if (remove(sor) > 0)
                  printf("%s", sor);
      return 0;
}
```

```
/* A sorvégi szóközök és tabulátorok eltávolítása */
/* az s karaktersorozatból */
int remove(char s[])
{
      int i;

      i = 0;
      while (s[i] != '\n')          /* újsor-karaktert talál */
            ++i;
      --i;                          /* vissza az '\n' elé    */
      while (i >= 0 && (s[i] ==  ' ' || s[i] == '\t'))
            --i;
      if (i >= 0) {                 /* ez nem üres sor?      */
            ++i;
            s[i] = '\n';  /* visszateszi az újsor-karaktert */
            ++i;
            s[i] = '\0'       /* lezárja a karaktersorozatot */
      }
      return i;
}
```

A `remove` függvény megkeresi a `sor` végét, onnan visszafelé haladva eltávolítja az ott lévő szóközöket és tabulátorokat, majd visszatér az új sorhosszal. Ha ez a sorhossz nullánál nagyobb, akkor a `sor` szóköztől és tabulátortól különböző karaktereket is tartalmaz és a program az ilyen sorokat kiírja. Nulla sorhossz esetén a `sor` nem íródik ki, azaz a program törli a kimeneten. Ezzel a módszerrel garantálható, hogy a teljesen üres sorok nem íródnak ki.

A `remove` függvény megkeresi az újsor-karaktert (a sorvégét), majd egy karakterrel visszalép. Erről a helyről indulva lépeget a karaktersorozatban visszafelé, mindaddig, amíg szóköztől vagy tabulátortól különböző karaktert nem talál, ill. el nem fogy a sorozat (`i < 0` lesz). Ha `i >= 0`, akkor a karaktersorozatban legalább egy értékes karakter van. Ekkor a `remove` függvény ezen karakter után elhelyezi az újsor-karaktert, lezárja a karaktersorozatot, majd visszatér az `i` értékével (ami a sorhossz).

A `getline` függvény azonos az 1.16. gyakorlatban használt függvénnyel.

1.19. gyakorlat

Írjunk egy `reverse(s)` függvényt, ami megfordítja az `s` karaktersorozat karaktereit! Használjuk fel ezt a függvényt egy olyan programban, ami soronként megfordítja a beolvasott szöveget (K–R: 45. oldal)!

```
#include <stdio.h>
#define MAXSOR  1000          /* a bemeneti sor. max. hossza */
```

```
int getline(char sor[], int maxsor);
void reverse(char s[]);

/* A beolvasott szöveg megfordítása soronként */
main()
{
      char sor[MAXSOR];       /* a beolvasott aktuális sor */

      while (getline(sor, MAXSOR) > 0) {
            reverse(sor);
            printf("%s", sor);
      }
}

/* reverse: megfordítja az s karaktersorozatot */
void reverse(char s[])
{
      int i, j;
      char atmeneti;

      i = 0;
      while (s[i] != '\0')          /* megkeresi a          */
                                    /* karaktersorozat végét */
            ++i;
      --i;                         /* a '\0' végjel elé áll */
      if (s[i] == '\n')
            --i;       /* az újsor-karaktert a helyén hagyja */
      j = 0;             /* az új s karaktersorozat kezdete  */
      while (j < i) {
            atmeneti = s[j];
            s[j] = s[i];       /* felcseréli a karaktereket */
            s[i] = atmeneti;
            --i;
            ++j;
      }
}
```

A `reverse` függvény megkeresi az `s` karaktersorozat végét, majd a `'\0'` végjel elé lép, így a megfordítás után nem a karaktersorozat végjele lesz az új karaktersorozat első karaktere. Ha a sorban egy `'\n'` újsor-karakter is volt, akkor még ez elé is lép, mert az újsor-karakternek, hasonlóan a `'\0'` végjelhez, a sor végén kell maradni.

A másoláskor `j` a karaktersorozat első karakterének, `i` a karaktersorozat utolsó karakterének indexe lesz. A karaktercsere után `j` értékét eggyel növeljük (haladunk a karaktersorozat vége felé), `i` értékét pedig eggyel csökkentjük (haladunk a karaktersorozat kezdete felé). A csere addig folytatódik, amíg `j` nem lesz kisebb, mint `i`.

A `main` eljárás beolvas egy sort, megfordítja, majd kiírja. Ez az első nulla hosszúságú sor beolvasásáig folytatódik.

A `getline` függvény azonos az 1.16. gyakorlatban használt függvénnyel.

1.20. gyakorlat

Írjunk `detab` néven programot, amely a beolvasott szövegben talált tabulátor karaktereket annyi szóközzel helyettesíti, amennyi a következő tabulátor-pozícióig hátravan! Tételezzük fel, hogy a tabulátor-pozíciók adottak, pl. minden *n*-edik oszlopban. Az *n* értékét változóként vagy szimbolikus állandóként célszerű megadni (K–R: 48. oldal)?

```
#include <stdio.h>

#define TABNOV  8  /* a tabulátorok lépésköze */

/* A tabulátort megfelelő számú szóközzel */
/* helyettesítő program */
main()
{
      int kar, szokoz, hely;

      szokoz = 0;              /* a szükséges szóközök száma */
      hely = 1;                /* a karakter helye a sorban  */
      while ((kar = getchar()) != EOF) {
            if (kar == '\t') {              /* tabulátor volt */
                szokoz = TABNOV - (hely - 1) % TABNOV;
                  while (szokoz > 0) {
                        putchar(' ');
                        ++hely;
                        --szokoz;
                  }
            } else if (kar == '/n') {     /* újsor-karakter */
                  putchar(kar);
                  hely = 1;
            } else {                    /* minden más karakter */
                  putchar(kar);
                  ++hely;
            }
      }
}
```

A tabulátor-pozíciók a `TABNOV` többszöröseinek megfelelő helyen vannak, és ebben a programban `TABNOV` értékét nyolcnak választottuk. A `hely` változó a szöveg soron belüli aktuális helyét adja meg.

Ha a vizsgált karakter tabulátor, akkor a program kiszámítja a következő tabulátor-pozícióig szükséges szóközök számát (`szokoz`). Ezt az értéket a

```
szokoz = TABNOV - (hely - 1) % TABNOV;
```

utasítással határozzuk meg. Ha a vizsgált karakter újsor, akkor azt a program kiírja és a sor kezdetére állítja a `hely` változót (`hely = 1`). Bármilyen más karakter azonnal kiíródik, és a `hely` tartalma eggyel nő (`++hely`).

A TABNOV értékét szimbolikus állandóval adtuk meg. Később, az 5. fejezetben megmutatjuk, hogy hogyan lehet argumentumot átadni a `main()` eljárásnak, és ezzel lehetővé válik, hogy a felhasználó tetszőleges értékre állítsa be a tabulátor-pozíciók helyét. Akkor majd módosítjuk a programot és a TABNOV paramétert változóként fogjuk megadni.

A `detab` program bővített változatát az 5.11. és 5.12. gyakorlatok mutatják be.

1.21. gyakorlat

Írjunk programot `entab` néven, amely a beolvasott szövegben talált, szóközökből álló karaktersorozatot a minimális számú tabulátor-karakterrel és szóközzel helyettesíti úgy, hogy a szövegben a távolság ne változzon! Használjuk ugyanazokat a tabulátor-pozíciókat, mint a `detab` programban! Ha a következő tabulátor-pozíció eléréséhez egyetlen szóköz vagy egyetlen tabulátor-karakter is elegendő, akkor melyiket részesíti előnyben (K–R: 48. oldal)?

```
#include <stdio.h>

#define  TABNOV  8                  /* a tabulátorok lépésköze */

/* a karaktersorozat szóközeit megfelelő számú tabulátorral */
/* és szóközökkel helyettesíti */
main()
{
      int kar, szokoz, tab, hely;

      szokoz = 0;          /* a szükséges szóközök száma     */
      tab    = 0;          /* a szükséges tabulátorok száma */
      for (hely = 1; (kar = getchar()) != EOF; ++hely)
           if (kar == ' ' ) {
                if (hely % TABNOV != 0)
                    ++szokoz;    /* a szóköz-szám növelése     */
                else {
                    szokoz = 0;  /* a szóköz-szám nullázása    */

                    ++tab;       /* eggyel több tabulátor      */
                }
           } else {
                for ( ; tab > 0; --tab)
                     putchar('\t'); /*a tabulátor kiírása*/
                if (kar == '\t')    /* az eddigi szóközök */
                                    /* elhagyása */
                     szokoz = 0;
                else                /* a szóköz(ök) kiírása */
                     for ( ; szokoz > 0; --szokoz)
                          putchar(' ');
                     putchar(kar);
                     if (kar == '\n')
```

```
                    hely = 0;
                else if (kar == '\t')
                    hely = hely + (TABNOV -
                     (hely - 1) % TABNOV) - 1;
            }
}
```

A `szokoz` és `tab` egész típusú változók a szövegben lévő szóköz-sorozat helyettesítéséhez szükséges min. számú szóköz és tabulátor-karakter számát tartalmazzák. A `hely` változó tartalma a szöveg aktuális helye (annak a karakternek a helye, amelynek éppen a feldolgozása folyik).

A program működésének lényege, hogy megtaláljuk az összes szóközt. Amikor a szóköz-sorozaton végighaladunk és a `hely` eléri a `TABNOV` többszörösét, az addigi szóközöket egy tabulátorkarakterrel helyettesítjük.

Ha a program egy szóköztől különböző karaktert talál, akkor azt kiírja, majd számolja a szóközöket a következő nem szóköz karakterig. A `szokoz` és `tab` változók nullázódnak, a `hely` pedig a `sor` kezdetére áll, ha újsor-karaktert olvastunk.

Ha a beolvasott nem szóköz karakter tabulátor, akkor a program csak az aktuális tabulátor-karaktert követően leszámolt tabulátorokat írja ki.

Amikor a következő tabulátor-pozíció eléréséhez egyetlen betűköz is elegendő, akkor azt egyszerűbb tabulátorral helyettesíteni, mint a speciális esetet kezelni.

Az `entab` program bővített változatát az 5.11. és 5.12. gyakorlatok mutatják be.

1.22. gyakorlat

Írjunk programot, amely a hosszú bemeneti sorokat az *n*-edik oszlop előtt előforduló utolsó szóköz után kettő vagy több rövidebb sorba tördeli! Győződjünk meg arról, hogy a program nagyon hosszú sorok és az *n*-edik oszlop előtt sem szóközt, sem tabulátort nem tartalmazó sorok esetén egyaránt helyesen működik (K–R: 48. oldal)!

```
#include <stdio.h>

#define  MAXOSZL  10     /* a bemeneti szöveg máx. oszlopa  */
#define  TABNOV    8      /* a tabulátorok lépésköze         */

char sor[MAXOSZL]         /* a beolvasott sor                */

int tabkifejt (int hely);
int szokozkeres (int hely);
int ujhely (int hely);
void sornyomt (int hely);

/* A hosszú bemeneti sorokat kettő vagy több rövidebb */
/* sorba tördelő program */
main()
{
      int kar, hely;
```

```
        hely = 0;                       /* a soron belüli pozíció */
        while ((kar = getchar()) != EOF) {
                sor[hely] = kar; /* tárolja az aktuális karaktert*/
                if (kar == '\t')    /* kifejti a tabulátort      */
                        hely = tabkifejt(hely);
                else if (kar == '\n') {
                        sornyomt(hely); /*kiírja az aktuális sort */
                        hely = 0;
                } else if (++hely >= MAXOSZL) {
                        hely = szokozkeres(hely);
                        sornyomt(hely);
                        hely = ujhely(hely);
                }
        }
}

/* sornyomt: a hely oszlopig kiírja a sort */
void sornyomt (int hely)
{
        int i;
        for (i = 0; i < hely; ++i)
                putchar(sor[i]);
        if (hely > 0)                   /* minden karaktert kiírt? */
                putchar('\n');
}

/* tabkifejt: a tabulátort szóközökké alakítja */
int tabkifejt (int hely)
{
        sor[hely] = ' '; /* a tabulátor helyére legalább egy */
                                                /* szóköz kerül */
        for(++hely; hely < MAXOSZL && hely % TABNOV != 0; ++hely)
                sor[hely] = ' ';
        if (hely < MAXOSZL)    /* maradt még hely az aktuális */
                                                /* sorban? */
                return hely;
        else {                          /* az aktuális sor betelt */
                sornyomt(hely);
                return 0;           /* sor elejére állítja a helyet*/
        }
}

/* szokozkeres: megkeresi a szóköz helyét     */
int szokozkeres (int hely)
{
        while (hely > 0 && sor[hely] != ' ')
                --hely;
        if (hely == 0)                  /* nincs szóköz a sorban? */
                return MAXOSZL;
```

```
    else                        /* legalább egy szóköz van    */
        return hely + 1;        /* hely a szóköz utáni        */
                                /* pozícióra mutat            */
}

/* ujhely: átrendezi a sort az új helynek megfelelően */
int ujhely (int hely)
{
    int i, j;

    if (hely <= 0 || hely >= MAXOSZL)
        return 0;                       /* nem kell átrendezni */
    else {
        i = 0;
        for (j = hely; j < MAXOSZL; ++j) {
            sor[i] = sor[j];
            ++i;
        }
        return i;                   /* az új pozíció a sorban */
    }
}
```

A programban `MAXOSZL` egy szimbolikus állandó, ami a bemeneti szöveg *n*-edik oszlopát jelzi. Az egész típusú `hely` változó a szövegsoron belül arra a pozícióra mutat, ahol a feldolgozás éppen tart. A program a beolvasott sort az *n*-edik oszlop előtt töri.

Ha a program egy újsor-karaktert talál, akkor szóközökre fejti a tabulátort és kiírja az aktuális bemenet, majd amikor a `hely` eléri a `MAXOSZL` értéket, töri a sort .

A `szokozkeres` függvény a `hely` pozíciótól indulva keresi a szóközt, és ha megtalálta, akkor az utána következő pozícióval, ha nem, akkor a `MAXOSZL` értékkel tér vissza.

A `sornyomt` eljárás kiírja a 0. és a (`hely` - 1)-edik pozíció közötti karaktereket.

Az `ujhely` eljárás úgy rendezi át a sort, hogy a `hely` pozíciótól kezdve a sor elejére másolja a karaktereket, majd a `hely` új értékével tér vissza.

1.23. gyakorlat

Írjunk programot, ami egy C program szövegéből eltávolít minden megjegyzés szövegrészt! Ne feledkezzünk meg az idézőjelek közötti karaktersorozatok és karakterállandók helyes kezeléséről! A C nyelvben a megjegyzés szövegek nem ágyazhatók egymásba (K–R: 48. oldal)!

```
#include <stdio.h>

void commentolv (int kar);
void commentben (void);
void visszair (int kar);
```

```
/* A C nyelvű programból eltávolítja az összes  commentet */
main()
{
      int kar, kar1;

      while ((kar = getchar ()) != EOF)
            commentolv(kar);
      return 0;
}

/* commentolv: egyenként beolvassa a karaktereket */
/* és eltávolítja a commentet */
void commentolv (int kar)
{
      int kar1;

      if (kar == '/')
            if ((kar1 = getchar()) == '*')
                  commentben();          /* a comment kezdete */
            else if (kar1 == '/') {
                  putchar(kar);            /* más törtvonal   */
                  commentolv(kar1);
            } else {
                  putchar(kar);           /* nem comment volt */
                  putchar (kar1);
            }
      else if (kar == '\'' || kar == '"')
            visszair(kar); /* aposztróffal kezdődő rész kezdete*/
      else
            putchar(kar);                 /* nem comment volt */
}

/* commentben: érvényes comment belsejében van */
void commentben (void)
{
      int kar, kar1;

      kar = getchar();                  /* az előző karakter    */
      kar1 = getchar();                 /* az aktuális karakter */
      while (kar != '*' || kar1 != '/') { /*keresi a comment
                                                      végét */
            kar = kar1;
            kar1 = getchar();
      }
}

/* visszair: visszaírja az aposztrófok közötti karaktereket */
void visszair (int kar)
```

```
{
        int kar1;

        putchar(kar);
        while((kar1 = getchar()) != kar) {   /* keresi a végét   */
                putchar(kar1);
                if (kar1 == '\\')
                        putchar(getchar()));      /* törli az escape */
                                                  /* jelsorozatot    */
        }
        putchar(kar1);
}
```

A program feltételezi, hogy a beolvasott szöveg egy helyes C nyelvű program. A `commentolv` eljárás megkeresi a comment kezdetét (a /* karakterpárt) és ha megtalálta, hívja a `commentben` függvényt. Ez a függvény megkeresi a comment végét (a */ karakterpárt) és ezzel garantálja a comment törlését.

A `commentben` függvény ezen kívül még megkeresi az aposztrófokat és idézőjeleket is, és ha talál ilyet, hívja a `visszair` eljárást. A `visszair` eljárás argumentuma jelzi, hogy aposztróf vagy idézőjel volt-e a talált karakter. A `visszair` eljárás kiírja az aposztrófok és idézőjelek közötti részt és nem okoz hibát a commentben. A `visszair` nem foglalkozik a backslasht (\) követő idézőjellel (záró idézőjel, l. a K–R escape jelsorozatokat tárgyaló részét és az 1.2. gyakorlatot), viszont minden más karaktert kiír.

A program akkor fejeződik be, ha a `getchar` eljárás állomány vége jelzéssel tér vissza.

1.24. gyakorlat

Írjon programot, ami egy C program szövegét olyan alapvető szintaktikai hibák szempontjából ellenőrzi, mint a nem azonos számú kerek, szögletes és kapcsos nyitó és záró zárójelek! Ne feledkezzünk meg a programban az aposztrófokról, idézőjelekről, escape jelsorozatokról és megjegyzés szövegekről sem! A programot teljesen általános formában elég nehéz megírni (K–R: 48. oldal)!

```
#include <stdio.h>

int szogl, kapcs, kerek;

void idezoben (int kar);
void commentben (void);
void keres (int kar);

/* Szintaktikai ellenőrző C nyelvű programhoz */
main()
{
        int kar;
        extern int szogl, kapcs, kerek;
```

```
        while ((kar = getchar()) != EOF) {
                if (kar == '/') {
                        if ((kar = getchar()) == '*')
                                commentben();       /* commentben van */
                        else
                                keres(kar);
                } else if (kar == '\'' || kar == '"')
                        idezoben(kar);    /* idézőjelek között van */
                else
                        keres(kar);

                if (kapcs < 0) {                              /* hiba */
                        printf("A nyitó és záró kapcsos zárójelek
                                                    száma nem egyezik\n");
                        kapcs = 0;
                } else if (szogl < 0) {
                        printf("A nyitó és záró szögletes zárójelek
                                                    száma nem egyezik\n");
                        szogl = 0;
                } else if (kerek < 0) {
                        printf("A nyitó és záró kerek zárójelek
                                                    száma nem egyezik\n");
                        kerek = 0;
                }
        }
        if (kapcs > 0)                                        /* hiba */
                        printf("Nem egyenlő számú nyitó és záró
                                                        zárójel\n");
        if (szogl > 0)
                        printf("Nem egyenlő számú nyitó és záró
                                                        zárójel\n");
        if (kerek > 0)
                        printf("Nem egyenlő számú nyitó és záró
                                                        zárójel\n");
}

/* keres: megkeresi a szintaktikai hibát */
void keres (int kar)
{
        extern int szogl, kapcs, kerek;

        if (kar == '{')
                ++kapcs;
        else if (kar == '}')
                --kapcs;
        else if (kar == '[')
                ++szogl;
        else if (kar == ']')
                --szogl;
        else if (kar == '(')
                ++kerek;
```

```
        else if (kar == ')')
                --kerek;
}

/* commentben: érvényes comment belsejében van */
void commentben (void)
{
        int kar, kar1;

        kar = getchar();                /* az előző karakter      */
        kar1 = getchar();              /* az aktuális karakter    */
                while (kar != '*' || kar1 != '/'){
                                        /* keresi a comment végét */
                kar = kar1;
                kar1 = getchar();
        }
}

/* idezoben: az idézőjelek között van */
void idezoben (int kar)
{
        int kar1;

        while((kar1 = getchar()) != kar)    /* keresi a végét  */
                if (kar1 == '\\')
                        getchar(); /* törli az escape jelsorozatot */
}
```

Ez a megoldás nem teljesen általános.A program háromféle szintaktikai hibát ellenőriz: a nem egyenlő számú kerek, kapcsos és szögletes nyitó, ill. záró zárójeleket, és a feldolgozott program többi részéről feltételezi, hogy hibátlan.

A `keres` függvény eggyel növeli a `kapcs` változó tartalmát, ha kezdő kapcsos zárójelet (`'{'`), és eggyel csökkenti `kapcs` tartalmát, ha záró kapcsos zárójelet (`'}'`) talál, ill. hasonlóan kezeli a kerek és szögletes zárójeleket is (a `kerek` és `szogl` változókkal).

A keresés során a `kapcs`, `szogl` és `kerek` változók értékének pozitívnak vagy nullának kell lennie. Ha ezen változók tartalma negatívvá válik, az hibát jelent és a program azonnal kiírja a hibaüzenetet. Átmenetileg pl. a `[[[` (`szogl = 3`) legális, mivel a hiányzó három záró szögletes zárójelet később még megtalálhatja a program. Viszont pl. a `]]]` (`szogl = -3`) illegális, mivel előzőleg nem volt három nyitó szögletes zárójel, ami ezekkel párosítható lenne. Éppen ezért szükséges a

```
if (kapcs < 0) {                                        /* hiba */
        printf("A nyitó és záró kapcsos zárójelek száma
                                                nem egyezik\n");
        kapcs = 0;
} else if (szogl < 0) {
        printf("A nyitó és záró szögletes zárójelek száma
                                                nem egyezik\n");
```

```
        szogl = 0;
} else if (kerek < 0) {
        printf("A nyitó és záró kerek zárójelek száma
                                                    nem egyezik\n");
        kerek = 0;
}
```

programrész, mivel ez azonnal jelzi a hibát és így a később érkező záró zárójelekkel nem kerülnek párba a nyitó zárójelek (ez kivédi a) (,]]] [[[vagy } } } { { { típusú hibákat, amelyek előfordulásakor a `kapcs, szogl` vagy `kerek` változók értéke az ellenőrzés végére nulla lenne).

A `main` eljárás megkeresi a commenteket, az aposztrófokat és az idézőjeleket, és átlépi a közöttük lévő karaktereket. A commentekben, ill. az aposztrófok és idézőjelek között a különböző típusú nyitó és záró zárójeleknek nem feltétlenül kell párosával előfordulniuk.

A program az EOF beolvasása után végzi a végső ellenőrzést, megnézve, hogy van-e lezáratlan nyitó zárójel (`kapcs, szogl` vagy `kerek` változók tartalma >0) és ha igen, akkor kiírja a megfelelő hibaüzenetet.

Típusok, operátorok és kifejezések

2.1. gyakorlat

Írjunk programot, ami meghatározza a `signed` és `unsigned` minősítőjű `char`, `short`, `int` és `long` típusú változók hosszát a szabványos header állományokból vett megfelelő értékek kiírásával és közvetlen számítással! A feladat nehezebb, ha kiszámítjuk a nagyságokat és tovább nehezíthető, ha a lebegőpontos számok nagyságát is meg akarjuk határozni (K–R: 50. oldal).

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

/* Az egyes adattípusok nagyságának meghatározása */
main()
{
            /* signed minősítésű típusok */
      printf("signed char min.  = %d\n", SCHAR_MIN);
      printf("signed char max.  = %d\n", SCHAR_MAX);
      printf("signed short min. = %d\n", SHRT_MIN);
      printf("signed short max. = %d\n", SHRT_MAX);
      printf("signed int min.   = %d\n", INT_MIN);
      printf("signed int max.   = %d\n", INT_MAX);
      printf("signed long min.  = %ld\n",LONG_MIN);
      printf("signed long max.  = %ld\n",LONG_MAX);
            /* unsigned minősítésű típusok */
      printf("unsigned char max.  = %u\n", UCHAR_MAX);
      printf("unsigned short max. = %u\n", USHRT_MAX);
      printf("unsigned int max.   = %u\n", UINT_MAX);
      printf("unsigned long max.  = %lu\n", ULONG_MAX);
}
```

A C nyelv ANSI szabványa az egyes adattípusok nagyságát a `<limits.h>` headerben adja meg. Ezek a nagyságok gépről gépre változnak, mivel a `short`, `int` és `long` alapvető típusok mérete a hardvertől függ. Az egyes adattípusok nagyságát kiszámító program:

```
#include <stdio.h>

/* Az egyes adattípusok nagyságának meghatározása */
```

```
main()
{
            /* signed minősítésű típusok */
      printf("signed char min.    = %d\n", -(char)
                                    ((unsigned char) ~0 >> 1));
      printf("signed char max.    = %d\n", (char)
                                ((unsigned char) ~0 >> 1));
      printf("signed short min.   = %d\n",-(short)
                                ((unsigned short) ~0 >> 1));
      printf("signed short max.   = %d\n", (short)
                                ((unsigned short) ~0 >> 1));
      printf("signed int min.     = %d\n", -(int)
                                    ((unsigned int) ~0 >> 1));
      printf("signed int max.     = %d\n", (int)
                                ((unsigned int) ~0 >> 1));
      printf("signed long min.    = %ld\n", -(long)
                                    ((unsigned long) ~0 >> 1));
      printf("signed long max.    = %ld\n", (long)
                                ((unsigned long) ~0 >> 1));
            /* unsigned minősítésű típusok */
      printf("unsigned char max.  = %u\n",
                                (unsigned char) ~0);
      printf("unsigned short max. = %u\n",
                                (unsigned short) ~0);
      printf("unsigned int max.   = %u\n",
                                (unsigned int) ~0);
      printf("unsigned long max.  = %lu\n",
                                (unsigned long) ~0);
}
```

Az adott típus nagysága bitenkénti logikai operátorokkal (K–R: 62. oldal) számítható ki. Például a

```
(char)((unsigned char) ~0 >> 1)
```

kifejezésben a

```
~0
```

először egy csupa nulla értékű bitből álló adatot hoz létre, majd minden bitjét egy értékűvé alakítja (egyes komplemens). Ezt a

```
(unsigned char) ~0
```

`unsigned char` típusúvá alakítja, majd a

```
(unsigned char) ~0 >> 1
```

művelet hatására az eredmény egy hellyel jobbra lép és az előjel bit törlődik. Ezt a számot a

```
(char)((unsigned char) ~0 >> 1)
```

`char` típusúvá alakítja. A kapott eredmény a `signed char` típus max. értéke.

2.2. gyakorlat

Írjunk az előző `for` ciklussal egyenértékű ciklust, ami nem használja az `&&` vagy `||` operátorokat (K–R: 56. oldal)!

Az eredeti ciklus

```
for (i = 0;i < lim - 1 && (c=getchar()) != '\n' && c != EOF; ++i)
```

volt. Ezzel egyenértékű a következő programrészlet:

```
enum ciklus {NEM, IGEN};
enum ciklus jocikl = IGEN;

i = 0;
while (jocikl == IGEN)
      if (i >= lim - 1)        /* az érvényességi tartományon */
                                                  /* kívül van? */
            jocikl = NEM;
      else if ((c = getchar()) == '\n')
            jocikl = NEM;
      else if (c = EOF)                /* vége az állománynak? */
            jocikl = NEM;
      else {
            s[i] = c;
            ++i;
      }
```

Látható, hogy az `&&` vagy `||` operátorok nélkül az eredeti `for` ciklus `if` utasítások sorozatává alakul. Például az eredeti `for` ciklusban az

```
i < lim - 1
```

azt vizsgálja, hogy `i` még a határok között van-e. Az egyenértékű programban az

```
i >= lim - 1
```

szerepel, ami azt vizsgálja, hogy `i` a határon kívül van-e, mert akkor lezárható a ciklus.

A `jocikl` egy felsorolt típusú változó. Ahogy a feltételek bármelyike teljesül, a `jocikl` NEM értéket kap, aminek hatására a ciklus lezárul.

2.3. gyakorlat

Írjunk `htoi(s)` néven függvényt, amely egy hexadecimális számjegyekből álló karaktersorozatot (beleértve a `0x` vagy `0X` karaktersorozatot is) a megfelelő egész számmá alakít! A megengedett számjegyek `0...9` és `a...f` vagy `A...F` (K–R: 60. oldal).

```
#define    IGEN     1
#define    NEM      0

/* htoi: az s hexadecimális karaktersorozatot egész számmá alakítja */
int htoi (char s[])
{
      int hexdigit, i, hexben, n;

      i = 0;
      if (s[i] == '0' { /* az opcionális 0x vagy 0X átugrása */
            ++i;
            if (s[i] == 'x' || s[i] == 'X')
                  ++i;
      }
      n = 0;                  /* az egész érték, amivel visszatér */
      hexben = IGEN;                /* feltételezi, hogy érvényes */
                                    /* hexadecimális számjegy */
      for (  ; hexben == IGEN; ++i) {
            if (s[i] >= '0' && s[i] <= '9')
                hexdigit = s[i] - '0';
            else if (s[i] >= 'a' && s[i] <= 'f')
                hexdigit = s[i] - 'a' + 10;
            else if (s[i] >= 'A' && s[i] <= 'F')
                hexdigit = s[i] - 'A' + 10;
            else
                hexben = NEM; /* nem érvényes hexadecimális számjegy */
            if (hexben == IGEN)
                  n = 16*n + hexdigit;
      }
      return n;
}
```

A függvény működését a

```
for (  ;  hexben == IGEN;  ++i)
```

ciklus vezérli. Az `i` egész típusú változó az `s` tömb indexe. Amíg `s[i]` egy érvényes hexadecimális számjegy, addig a `hexben` változó értéke `IGEN` és a ciklus folytatódik. A `hexdigit` változó tartalma 0 és 15 közötti egész szám lehet.

Az

```
if (hexben == IGEN)
```

utasítás garantálja, hogy `s[i]` érvényes hexadecimális számjegy, amelynek értéke a `hexdigit` változóba kerül. A ciklus befejeztekor a `htoi` függvény az `n` változó értékével tér vissza.

A `htoi` függvény hasonló az `atoi` függvényhez (K–R: 58. oldal).

2.4. gyakorlat

Írjuk meg a `squeeze(s1, s2)` olyan változatát, amely az `s1` karaktersorozatból töröl minden karaktert, ami az `s2` karaktersorozatban megtalálható (K–R: 62. oldal)!

```
/* squeeze: törli az s1 azon karaktereit, amelyek */
/* megtalálhatók s2-ben */
void squeeze (char s1[], char s2[])
{
      int i, j, k;

      for (i = k = 0; s1[i] != '\0'; i++) {
           for (j = 0; s2[j] != '\0' && s2[j] != s1[i]; j++)
                ;
           if (s2[j] == '\0')  /* vége a karaktersorozatnak, */
                                    /* nincs egyezés */
                s1[k++] = s1[i];
      }
      s1[k] = '\0';
}
```

A függvény

```
for (i = k = 0; s1[i] != '\0'; i++)
```

utasítása egyszerre ad kezdeti értéket az `s1` és az eredmény (ami szintén az `s1`) tömb `i` és `k` indexeinek. Az `s1` tömbből minden karakter, ami megtalálható `s2`-ben törlődni fog. A ciklus akkor ér véget, ha eljutottunk az `s1` karaktersorozat végéig.

A második `for` ciklus ellenőrzi az `s2` karaktereit és azok egyezését az `s1` megfelelő karakterével. Ez a ciklus akkor fejeződik be, ha `s2`-ben már nincs több karakter vagy az `s1` és `s2` megfelelő karaktere megegyezik. Ha nem volt egyező karakter, akkor `s1[i]` az eredmény tömbbe másolódik. Ha volt egyezés, akkor az

```
if   (s2[j] == '\0')
```

utasítás feltétele hamis és `s1[i]` nem másolódik az eredmény tömbbe (kihagyjuk belőle).

2.5. gyakorlat

Írjunk `any(s1, s2)` néven függvényt, amely visszatérési értékként megadja az `s1` karaktersorozat legelső helyét, ahol az `s2` karaktersorozat bármelyik karaktere előfordul! A függvény visszatérési értéke legyen `-1`, ha `s1` egyetlen `s2`-beli karaktert sem tartalmaz. (Az `strpbrk` standard könyvtári függvény ugyanezt teszi, csak visszatérési értékként az adott helyet kijelölő mutatót adja.) (K–R: 62. oldal.)

```
/* any: visszatér az s1 első olyan helyével, ahol a karakter */
/* megegyezik s2 bármelyik karakterével */
int any (char s1[], char s2[])
{
      int i, j;

      for (i = 0; s1[i] != '\0'; i++)
            for (j = 0; s2[j] != '\0'; j++)
                  if (s1[i] == s2[j])/* van egyező karakter? */
                        return i;    /* az első egyezés helye */
      return -1;                 /* egyébként, ha nincs egyezés */
}
```

A

```
for (i = 0; s1[i] != '\0'; i++)
```

utasítás vezérli a függvény működését. Ha a ciklus normális módon fejeződik be (`s1` minden karakterét feldolgoztuk), akkor a függvény `-1` értékkel tér vissza a hívó eljárásba, jelezve, hogy `s2`-ben nincs olyan karakter, ami egyezne az `s1` bármelyik karakterével.

A

```
for (j = 0; s2[j] != '\0'; j++)
```

utasítás minden `i` értékre végbemegy és az utána következő `if` utasítások összehasonlítják `s2` egyes karaktereit `s1[i]`-vel. Ha `s2` valamelyik karaktere megegyezik `s1[i]`-vel, akkor a függvény az `i` értékkel tér vissza (ami az `s1` karaktersorozaton belül az első hely, amelyen a karakter egyezik az `s2`-ben előforduló valamelyik karakterrel).

2.6. gyakorlat

Írjuk meg a `setbits(x, p, n, y)` függvényt, amely egy olyan `x` értékkel tér vissza, amit úgy kap, hogy az `x` `p`-edik pozíciótól jobbra eső `n` bitje helyére bemásolja `y` jobbszélső `n` bitjét, a többi bitet változatlanul hagyva (K–R: 63. oldal)!

```
/* setbits: x p helytől kezdődő n bitje helyébe y */
/* utolsó n bitjét írja */
unsigned setbits (unsigned x, int p, int n, unsigned y)
{
      return x & ~(~0 << n)  << (p + 1 - n)) |
                (y & ~(~0 << n)) << (p + 1 - n);
}
```

A feladat sematikusan ábrázolva:

```
xxx...xnnnx...xxx     x
yyy..........ynnn     y
```

A megoldáshoz az kell, hogy töröljük x kívánt helytől kezdődő n bitjét, és y összes bitjét a jobbszélső n bit kivételével. Ezek után y tartalmát a p-edik pozícióig léptetjük, majd képezzük x és y bitenkénti VAGY kapcsolatát.

```
xxx...x000x...xxx     x
000...0nnn0...000     y
xxx...xnnnx...xxx     x
```

x n bitjét úgy töröljük, hogy ÉS kapcsolatba hozzuk egy olyan számmal, amely a p pozíciótól kezdve n biten nullát tartalmaz, és a többi bitje 1 állapotú. Ezt a számot a következő módon kapjuk: a

```
~0 << n
```

csupa 1 állapotú bitből álló számot hoz létre, amit n hellyel balra léptet. A léptetés során a jobb szélen 0 állapotú bitek lépnek be. A

```
~(~0 << n)
```

egy olyan számot hoz létre, amelyben a jobb szélső n bit 1 állapotú, az összes többi pedig 0 állapotú (az előző adat egyes komplemense). Ezt a számot a

```
~(~0 << n) << (p + 1 - n)
```

művelet balra lépteti, hogy az 1 állapotú bitek a p-edik pozícióhoz kerüljenek, majd a

```
~(~(~0 << n) << (p + 1 - n))
```

művelettel ismét komplementáljuk. Ennek hatására a p-edik pozíciótól kezdve n db 1 állapotú bit lesz és a többi bit 0. Az

```
x & ~(~(~0 << n) << (p + 1 - n))
```

létrehozza az x és a kapott szám ÉS kapcsolatát, ami az x-ből p-től kezdve n bitet kinulláz. y minden bitjét a jobb szélső n bit kivételével úgy nullázzuk, hogy vesszük a

```
~(~0 << n)
```

értéket, ami a jobb szélső n helyen 1 állapotú, minden más helyen 0 állapotú bitet tartalmaz. Ezt ÉS kapcsolatba hozva y-nal:

```
y & ~(~0 << n)
```

előáll a kívánt érték. Ennek a p-edik pozícióba hozása a

```
(y & ~(~0 << n)) << (p + 1 - n)
```

léptetéssel érhető el. A két részkifejezés VAGY kapcsolata

```
x & ~(~(~0 << n) << (p + 1 - n)) |
(y & ~(~0 << n)) << (p + 1 - n)
```

adja a függvény visszatérési értékét.

2.7. gyakorlat

Írjunk egy `invert(x, p, n)` függvényt, amely az x p-edik pozíciótól kezdődő n bitjét invertálja (az 1-eseket 0-ra, a 0-ákat 1-esekre változtatja), a többi bitet pedig változatlanul hagyja (K–R: 63. oldal)!

```
/* invert: x p helytől kezdődő n bitjét invertálja */
unsigned invert (unsigned x, int p, int n)
{
      return x ^ (~(~0 << n) << (p + 1 - n));
}
```

A

```
(~0 << n)
```

kifejezés először csupa 1 állapotú bitből álló számot állít elő, majd ezt *n* lépéssel balra lépteti, amitől a jobb szélen n db 0 állapotú bit lesz. Ennek komplemensét képezzük a

```
~(~0 << n)
```

kifejezéssel, így a jobb szélső n bit 1 állapotú lesz, az összes többi pedig 0. Az n db 1 állapotú bitből álló csoport kezdetét a

```
(~(~0 << n) << (p + 1 - n))
```

kifejezés balra léptetéssel a p-edik pozícióra tolja. Ennek a számnak és az eredeti x adatnak a bitenkénti kizáró-VAGY kapcsolatát képezzük az

```
x ^ (~(~0 << n) << (p + 1 - n))
```

kifejezéssel.

A bitenkénti kizáró-VAGY (^) művelet akkor ad eredményül 1-et, ha a két bit különbözött, azonos állapotú bitek esetén az eredmény nulla. Mivel a cél az volt, hogy invertáljuk x p-edik pozíciótól kezdődő n bitjét, így elegendő, ha képezzük x és a speciálisan kialakított adat (amelyben a p-edik pozíciótól kezdve n db 1 állapotú bit van és az összes többi bit 0 állapotú) kizáró-VAGY kapcsolatát. Ha az n bites szakaszon az eredeti bit 0 állapotú volt, akkor a kizáró-VAGY kapcsolata az 1 állapotú bittel 1 eredményt ad, ami az eredeti bit inverze. Hasonlóan, ha az eredeti bit 1 volt, akkor a kizáró-VAGY az 1 értékkel 0 eredményt ad, ami szintén az eredeti érték inverze.

Az n bites szakaszon kívül mindig 0 állapotú bitekkel képezzük a kizáró-VAGY kapcsolatot, és mivel 0^0 = 0, ill. 1^0 = 1 x eredeti bitjei nem változnak. Így a függvény valóban csak a p pozíciótól kezdődő n bitet invertálja.

2.8. gyakorlat

Írjunk egy rightrot(x, n) függvényt, ami n bittel jobbra forgatja (rotálja) az x egész mennyiséget! Jobbra forgatásnál a jobb szélen kilépő bitek a bal szélen visszakerülnek a szóba (K–R: 64. oldal).

```
/* rightrot: x jobbra rotálása n bittel */
unsigned rightrot (unsigned x, int n)
{
      int szohossz(void);
      int jbit;                          /* a jobb szélső bit */

      while (n-- > 0) {
            jbit = (x & 1) << (szohossz() - 1);
            x = x >> 1; /* x-et egy hellyel jobbra lépteti */
            x = x | jbit;       /* kész egy rotálási ciklus */
      }
      return x;
}
```

A jbit változó először felveszi az x jobb szélső bitjének értékét, majd ezt a bal szélre léptetjük (szohossz() - 1) számú balra léptetéssel.

Ezt követően x-et egy hellyel jobbra léptetjük, majd a bal szélső, léptetés után 0 állapotú bitje helyébe bitenkénti VAGY kapcsolattal beírjuk jbit értékét. Ezzel befejeződött egy teljes rotálás, és a ciklus n ilyen lépést hajt végre.

A szohossz() függvény meghatározza az adott számítógép szóhosszát.

```
/* szohossz: a gép szóhosszának meghatározása */
int szohossz(void)
{
      int i;
      unsigned v = (unsigned) ~0;

      for (i = 1; (v = v >> 1) > 0; i++)
            ;
      return i;
}
```

A feladat megoldására alkalmas más program:

```
/* rightrot: x jobbra rotálása n bittel */
unsigned rightrot (unsigned x, int n)
{
      int szohossz (void);
      unsigned jbitek;

      if ((n = n % szohossz()) > 0) {
           jbitek = ~(~0 << n) & x; /* x n jobb szélső bitje */
            jbitek = jbitek << (szohossz() - n);
                     /* a jobb szélső bitek balra léptetése */
            x = x >> n;      /* x n bittel jobbra léptetése */
            x = x | jbitek;                /* kész a rotálás */
      }
      return x;
}
```

Ha a jobbra rotálások `n` száma megegyezik az előjel nélküli egész adat bitjeinek számával, akkor semmi nem változik, `x` értéke megegyezik a rotálás előtti értékkel. Ha `n` kisebb a szóhossznál, akkor `n` bittel jobbra kell rotálni `x`-et. Ha `n` nagyobb a szóhossznál, akkor az `n/szohossz` egészosztás maradékának megfelelő lépésszámmal kell rotálni. Ez azt eredményezi, hogy nem kell ciklust szervezni. Így

`~0 << n`	a csupa 1 értékű bitből álló adat `n` hellyel balra lép és a jobb szélen `n` db 0 értékű bit lép be;
`~(~0 << n)`	az előbbi adat komplemense, így a jobb szélen `n` db 1 értékű bit lesz.

Ezután ezt az értéket bitenkénti ÉS kapcsolatba hozva `x`-szel, az `n` jobb szélső bit választódik le és kerül a `jbitek` változóba. A `jbitek` tartalmát a bal szélső pozícióba léptetjük, és `x` eredeti értékével bitenkénti VAGY kapcsolatba hozzuk, amivel megkaptuk az `n` bittel rotált értéket.

2.9. gyakorlat

A kettes komplemens kódú aritmetikában az

```
x &= (x - 1)
```

kifejezés törli x jobb szélső bitjét. Ezt kihasználva írjunk egy gyorsabb `bitcount` változatot (K–R: 65. oldal)!

```
/* bitcount: x 1 értékű bitjeinek száma - a gyorsabb változat */
int bitcount (unsigned x)
{
    int b;

    for (b = 0; x != 0; x &= x - 1)
        ++b;
    return b;
}
```

Válasszuk meg pl. `x - 1` értékét bináris `1010`-nak (ez decimális 10-nek felel meg). Így `(x - 1) + 1` adja x értékét:

binárisan		decimálisan
1010	x - 1	+10
+ 1		+ 1
1011	x	Z+11

A választott `(x - 1)`-hez 1-et hozzáadva előállítottuk x-et. `x - 1` jobb szélső, 0 állapotú bitje az eredményben (x) 1 állapotúra változott. Ezért az x 1 állapotú jobb szélső bitje 0 állapotú bitnek felel meg `x-1` esetén. Ebből következik, hogy kettes komplemens kódú aritmetika esetén `x & (x - 1)` törölni fogja x jobb szélső bitjét.

Tekintsük előjel nélküli mennyiségnek a választott négy bites számot. A számban lévő 1 állapotú bitek számát az eredeti `bitcount` függvény a szám négyszeres jobbra léptetésével és a jobb szélső bitek ellenőrzésével határozza meg. Az itt ismertetett változatban kihasználjuk, hogy `x & (x - 1)` törli x jobb szélső bitjét. Például, ha x = 9, akkor

```
1 0 0 1    9 bináris értéke (x)
1 0 0 0    8 bináris értéke (x - 1)
1 0 0 0    x & (x - 1)
```

és x jobb szélső bitje törlődött. A kapott `1000` bináris érték decimális 8-nak felel meg. Az eljárást megismételve:

```
1 0 0 1    8 bináris értéke (x)
0 1 1 1    7 bináris értéke (x - 1)
1 0 0 0    x & (x - 1)
```

és x-ben a jobb szélső 1 állapotú bit ismét törlődött. A kapott eredmény binárisan 0000, decimálisan 0. Most már nincs több 1 állapotú bit x-ben, így a folyamat befejeződött.

Legrosszabb esetben, vagyis ha x minden bitje 1 állapotú, akkor az új függvényben a végrehajtott ÉS műveletek száma megegyezik az eredeti változatban végrehajtott ÉS műveletek számával, minden más esetben az új változat gyorsabb.

2.10. gyakorlat

Írjuk át a nagybetűket kisbetűkké alakító `lower` függvényt úgy, hogy az `if - else` szerkezetet feltételes kifejezéssel helyettesítjük (K–R: 66. oldal)!

```
/* lower: a kar ASCII karakter kisbetűssé alakítása */
int lower (int kar)
{
      return kar >= 'A' && kar <= 'Z' ? kar + 'a' - 'A' : kar;
}
```

Ha a

```
kar >= 'A' && kar <= 'Z'
```

kifejezés igaz, akkor `kar` egy (ASCII kódú) nagybetű. Ekkor a

```
kar + 'a' - 'A'
```

kifejezés értéke a megfelelő kisbetű kódja, és a `lower` függvény ezzel az értékkel fog visszatérni. Minden más esetben (azaz, ha `kar` kisbetű volt) a `lower` függvény a változatlan karakterkóddal tér vissza.

3.1. gyakorlat

A bináris kereső program a ciklus belsejében két vizsgálatot végez, de egy is elegendő lenne (a futási idő csökkentése érdekében érdemes minden lehetséges műveletet a ciklusmagon kívül elvégezni). Írja meg a program olyan változatát, amelyben a cikluson belül csak egy vizsgálat van és hasonlítsa össze a kétféle változat futási idejét (K–R: 72. oldal)!

```
/* binsearch: megkeresi x értékét a növekvő irányba rendezett */
/* v[0]...v[n-1] tömbben */
int binsearch (int x, int v[], int n)
{
      int also, felso, kozep;

      also  = 0;
      felso = n - 1;
      kozep = (also + felso)/2;
      while (also <= felso && x != v[kozep]) {
            if (x < v[kozep])
                  felso = kozep - 1;
            else
                  also = kozep + 1;
            kozep = (also + felso)/2;
      }
      if (x == v[kozep])
            return kozep; /* megtalálta      */
  else
      return -1;     /* nem találta meg */
}
```

Az új programban a `while` ciklus feltételét

```
also <= felso
```

kifejezésről az

```
also <= felso && x != v[kozep]
```

kifejezésre változtattuk, így csak egyetlen `if` utasításra van szükség a ciklusban. Emiatt a

kozep értékét a ciklus indítása előtt is ki kell számítani, valamint a cikluson belül, minden végrehajtási lépésben.

A cikluson kívül kell elhelyezni a ciklus befejeztét ellenőrző utasítást, amellyel meghatározzuk, hogy x megtalálható-e a v tömbben. Ha x megtalálható a tömb elemei között, akkor a binsearch a kozep értékkel tér vissza a hívó eljárásba, különben pedig a –1 értékkel.

A kétféle program futási ideje közötti különbség minimális. Általában a program ilyen kis sebességnövekedéséért nem érdemes feláldozni a programszöveg áttekinthetőségét. Az eredeti binsearch program (K–R: 72. oldal) áttekinthetőbb, mint a javított változat.

3.2. gyakorlat

Írjunk escape(s, t) néven függvényt, amely a t karaktersorozatot az s karaktersorozat végéhez másolja és a másolás során a láthatatlan karaktereket (pl. újsor, tabulátor) látható escape sorozatokká (\n, \t) alakítja! A programot a switch utasítással írjuk meg! Készítsük el a függvény inverzét is, amely az escape sorozatokat a tényleges karakterekké alakítja (K–R: 74. oldal)!

```
/* escape: t s-hez másolása közben az újsor- és tabulátor- */
/* karaktereket látható escape sorozatokká alakítja */
void escape (char s[], char t[])
{
      int i, j;

      for (i = j = 0; t[i] != '\0'; i++)
            switch (t[i]) {
            case '\n' :     /* újsor-karakter */
                  s[j++] = '\\';
                  s[j++] = 'n';
                  break;
            case '\t' :     /* tabulátor      */
                  s[j++] = '\\';
                  s[j++] = 't';
            break;
            default :       /* minden más karakter */
                  s[j++] = t[i];
                  break;
            }
      s[j] = '\0';
}
```

A program

```
for (i = j = 0; t[i] != '\0'; i++)
```

utasítása vezérli a feldolgozó ciklust. Az `i` változó az eredeti `t` karaktersorozat indexe, amíg `j` a módosított `s` karaktersorozat indexe.

A `switch` utasítás három esetet különböztet meg: a `'\n'` az újsor-karaktert, a `'\t'` a tabulátor-karaktert és a `default` az összes egyéb esetet kezeli. Ha a `t[i]` karakter nem újsor vagy tabulátor, akkor az `escape` függvény a `default` esetet hajtja végre, vagyis a `t[i]` karaktert `s` karaktersorozathoz másolja.

Az `escape` függvény inverze, az `unescape` függvény hasonló felépítésű és működésű.

```
/* unescape: t s-hez másolása közben az escape sorozatokat */
/* tényleges karakterekké alakítja */
void unescape (char s[], char t[])
{
      int i, j;

      for (i = j = 0; t[i] != '\0'; i++)
            if (t[i] != '\\')
                  s[j++] = t[i];
            else                    /* backslash karakter volt  */
            switch (t[++i]) {
                  case 'n' :    /* tényleges újsor-karakter */
                        s[j++] = '\n';
                        break;
                  case 't' :  /* tényleges tabulátor */
                        s[j++] = '\t';
                        break;
                  default : /* minden más karakter */
                        s[j++] = '\\';
                        s[j++] = t[i];
                        break;
                  }
      s[j] = '\0';
}
```

Ha a `t[i]` karakter backslash volt, akkor a `switch` utasítást használjuk a `\n` újsor-karakterré és a `\t` tabulátorrá alakítására. A `default` eset a backslash utáni bármilyen más karaktert kezeli, oly módon, hogy a `t[i]` karaktert az `s` karaktersorozathoz másolja.

A `switch` utasítások egymásba ágyazhatók. Ezt felhasználva az `unescape` függvény egy másik változata:

```
/* unescape: t s-hez másolása közben az escape sorozatokat */
/*tényleges karakterekké alakítja */
void unescape (char s[], char t[])
{
      int i, j;

      for (i = j = 0; t[i] != '\0'; i++)
            switch (t[i]) {
```

```
            case '\\' :       /* backslash volt */
                  switch (t[++i]) {
                  case 'n':   /* tényleges újsor-karakter */
                        s[j++] = '\n';
                        break;
                  case 't':   /* tényleges tabulátor      */
                        s[j++] = '\t';
                        break;
                  default : /* minden más karakter      */
                        s[j++] = '\\';
                        s[j++] = t[i];
                        break;
                  }
                  break;
            default :         /* nem backslash volt       */
                  s[j++] = t[i];
                  break;
            }
      s[j] = '\0'
}
```

Ebben a programban a külső `switch` utasítás a backslash karaktert és minden más karaktert (`default` eset) kezeli. A backslash utáni karakterek kezelése a belső `switch` utasítás feladata.

3.3. gyakorlat

Írjunk `expand(s1, s2)` néven függvényt, amely az `s1` karaktersorozatban lévő rövidítéseket az `s2` karaktersorozatban feloldja (pl. az `a-z` helyett kiírja az `abc...xyz` teljes listát)! A program tegye lehetővé a betűk és számjegyek kezelését, és gondoljunk olyan rövidítések feloldására is, mint `a-b-c`, `a-z0-9` vagy `-a-z` is! Célszerű a kezdő vagy záró `-` jelet literálisként kezelni (K–R: 77. oldal).

```
/* expand: az s1-ben lévő rövidítéseket s2-be írva kifejti */
void expand (char s1[], char s2[])
{
      char kar;
      int i, j;

      i = j = 0;
      while ((kar = s1[i++] != '\0')    /* előkészít egy    */
                                        /* karaktert s1-ből */
            if (s1[i] == '-' && s1[i+1] >= kar) {
                  i++;
                  while (kar < s1[i])/* kifejti a rövidítést */
```

```
                    s2[j++] = kar++;
            } else
                    s2[j++] = kar;  /* átmásolja a karaktert */
        s2[j] = '\0';
}
```

A függvény vesz egy karaktert `s1` sorozatból és elmenti a `kar` változóba, majd megvizsgálja a következő karaktert. Ha a következő karakter - jel és az utána következő karakter nagyobb vagy egyenlő mint `kar`, akkor az `expand` elkezdi kifejteni a rövidítést. Minden más esetben a függvény a karaktert átmásolja `s2`-be.

Az `expand` függvény csak ASCII kódú karakterekkel működik. Ennek hatására az `a-z` rövidítést `abc...xyz` listaként fejti ki és a `!-~` rövidítést a `!"#...ABC...XYZ...abc...xyz...|}~` formában fejti ki.

Ezt a megoldást Axel Schreiner készítette az Osnabrucki Egyetemen (Németország).

3.4. gyakorlat

Az `itoa` függvény K–R 78. oldalán ismertetett változata kettes komplemens kódú számábrázolás esetén nem kezeli a legnagyobb negatív számot; azaz az $n = -2^{(\text{szóhossz}-1)}$ értéket. Magyarázzuk meg, hogy miért! Módosítsuk úgy a programot, hogy ezt az értéket is helyesen írja ki, a használt számítógéptől függetlenül (K–R: 78. oldal).

```
#define  abs(x)   ((x) < 0 ? -(x) : (x))

/* itoa: az n számot s karaktersorozattá alakítja */
/* módosított változat */
void itoa (int n, char s[])
{
        int i, sign;
        void reverse(char s[]);

        sign = n; /* elmenti az előjelet */
        i = 0;
        do {       /* generálja a számjegyeket, de */
                                                /* fordított
sorrendben */
                s[i++] = abs(n % 10) + '0';/* a következő számjegy */
        } while ((n /= 10) != 0);      /* törli azt            */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}
```

A problémát az okozza, hogy a

$-2^{(\text{szóhossz}-1)}$

szám nem alakítható pozitív számmá n = -n formában, mivel kettes komplemens kódú számábrázolásban a legnagyobb pozitív szám a

$2^{(\text{szóhossz}-1)} - 1$.

Az előjel megőrzése érdekében a `sign` változóba elmentjük `n` értékét. Az `abs` makroutasítás az `n % 10` abszolút értékét állítja elő, amivel kivédhető a

$-2^{(\text{szóhossz}-1)}$

abszolútértékének meghatározásából adódó probléma, hiszen nem ezt a számot, hanem a modulus operátor eredményét (az egészosztás maradékát) teszi pozitívvá.

Az új program `do - while` utasításában szereplő feltétel

```
(n /= 10) > 0
```

kifejezésről

```
(n /= 10) != 0
```

kifejezésre változott, mivel (ha eredetileg az volt, akkor) `n` a teljes ciklusban negatív marad, így az eredeti feltétellel végtelen ciklust kapnánk.

3.5. gyakorlat

Írjunk `itob(n, s, b)` néven függvényt, amely az `n` egész számot `b` alapú számrendszerben karaktersorozattá alakítja és az `s` karaktersorozatba helyezi! Speciális esetként írjuk meg az `itob(n, s, 16)` függvényt is, amely az `n` értékét hexadecimális formában írja az `s` karaktersorozatba (K–R: 78. oldal).

```
/* itob: az n számot b alapú számrendszerbeli */
/* karaktersorozattá alakítja s-ben */
void itob (int n, char s[], int b)
{
      int i, j, sign;
      void reverse(char s[]);

      if ((sign = n) < 0)              /* elmenti az előjelet */
            n = -n;                    /* n-t pozitívvá teszi */
      i = 0;
      do { /* generálja a számjegyeket, de fordított sorrendben */
```

```
        j = n % b;                  /* a következő számjegy */
        s[i++] = (j <= 9) ? j + '0' : j + 'a' - 10;
    } while ((n /= b) > 0);                  /* törli azt */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

A `b` alapú számrendszerbe alakítást az

```
n % b
```

moduló művelettel végezzük, ami `0` és `b - 1` közötti számot ad eredményül. Az `n`-ből a megfelelő számjegyet az

```
n /= b
```

kifejezéssel töröljük. A ciklus addig folytatódik, amíg `n/b` nullánál nagyobb.

3.6. gyakorlat

Írjuk meg az `itoa` függvénynek azt a változatát, amelynek kettő helyett három argumentuma van! Ez a harmadik argumentum legyen a minimális mezőszélesség, és az átalakított számot szükség esetén balról üres helyekkel töltse fel, hogy elegendően széles legyen (K–R: 78. oldal)!

```
#define  abs(x)   ((x) < 0 ? -(x) : (x))

/* itoa: az n számot w karakter széles karaktersorozattá */
/* alakítja */
void itoa (int n, char s[], int w)
{
    int i, sign;
    void reverse (char s[]);

    sign = n;                       /* elmenti az előjelet */
    i = 0;
    do { /* generálja a számjegyeket, de fordított sorrendben */
        s[i++] = abs(n % 10) + '0';/* a következő számjegy */
    } while ((n /= 10) != 0);                /* törli azt */
    if (sign < 0)
        s[i++] = '-';
    while (i < w)    /* kitölti szóközzel a többi helyet */
```

```
            s[i++] = ' ';
        s[i] = '\0';
        reverse(s);
}
```

A függvény a 3.4. gyakorlat `itoa` függvényéhez hasonló, csak a

```
while (i < w)
        s[i++] = ' ';
```

módosítást tartalmazza, ami egy ciklusban kitölti az `s` további helyeit szóközökkel, ha szükséges.

Függvények és a program szerkezete

4.1. gyakorlat

Írjuk meg az `strrindex(s, t)` függvénynek azt a változatát, amely a `t` minta `s`-beli legutolsó előfordulásának indexével, vagy ha `t` nem található meg `s`-ben, akkor −1-gyel tér vissza (K–R: 85. oldal)!

```
/* strrindex: visszaadja t s-beli legutolsó előfordulásának */
/* indexét, ill. -1-et, ha a keresett minta nincs s-ben */
int strrindex (char, s[], char t[])
{
      int i, j, k, hely;

      hely = -1;
      for (i = 0; s[i] != '\0'; i++) {
            for (j=i, k=0; t[k] != '\0' && s[j] == t[k]; j++, k++)
                  ;
            if (k > 0 && t[k] == '\0')
                  hely = i;
      }
      return hely;
}
```

Az `strrindex` függvény hasonló az `strindex` függvényhez (K–R 84. oldal), amely `t` első elemének `s`-beli előfordulási helyével tér vissza, ha `t` előfordult `s`-ben. Az `strrindex` ezzel szemben eltárolja `t` `s`-beli előfordulásának helyét és folytatja a keresést, mivel a feladat `t` `s`-beli legutolsó előfordulásának meghatározása. Ezt az

```
if (k > 0 && t[k] == '\0')
      hely = i;
```

utasítások végzik.

A feladat egy másik lehetséges megoldása:

```
#include <string.h>

/* strrindex: visszaadja t s-beli legutolsó előfordulásának */
/* indexét, ill. -1-et, ha a keresett minta nincs s-ben */
```

```
int strrindex (char s[], char t[])
{
      int i, j, k;

      for (i = strlen(s) - strlen(t); i >= 0; i--) {
            for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++)
                  ;
            if (k > 0 && t[k] == '\0')
                  return i;
      }
      return -1;
}
```

Ez a feladatnak egy sokkal hatékonyabb megoldása, mivel az `s` feldolgozását az `s` és `t` hosszának különbségénél kezdi (ettől jobbra már biztosan nem lehet egyezés). Ha nincs egyezés, akkor a függvény egy hellyel visszalép (balra lép) `s`-ben és ismét megnézi az egyezést. Ahogy egyezés van, az `strrindex` azonnal visszatér az `i` értékével, mivel ez az egyezések közül a legutolsó (az `s` vége felől kezdett keresés miatt).

4.2. gyakorlat

Bővítsük ki az `atof` függvényt úgy, hogy az pl. az `123.45e-6` alakú tudományos jelölésmódot is kezelni tudja! A bemeneti karaktersorozat a kitevő jelzésére `e` vagy `E` karaktereket használhatja és utána előjeles kitevő következhet (K–R: 88. oldal).

```
#include <ctype.h>

/* atof: az s karaktersorozat duplapontos számmá alakítása */
double atof (char s[])
{
      double val, power;
      int exp, i, sign;

      for (i = 0; isspace(s[i]); i++) /* átlépi az üres helyeket */
            ;
      sign = (s[i] == '-') ? -1 : 1;
      if (s[i] == '+' || s[i] == '-')
            i++;
      for (val = 0.0; isdigit(s[i]); i++)
            val = 10.0*val + (s[i] - '0');
      if (s[i] == '.')
            i++;
      for (power = 1.0; isdigit(s[i]); i++) {
```

```
            val = 10.0*val + (s[i] - '0');
            power *= 10.0;
      }
      val = sign*val/power;

      if (s[i] == 'e' || s[i] == 'E') {
            sign = (s[++i] == '-') ? -1 : 1;
            if (s[i] == '+' || s[i] == '-')
                  i++;
            for (exp = 0; isdigit(s[i]); i++)
                  exp = 10*exp + (s[i] - '0');
            if (sign == 1)
                  while (exp-- > 0)     /* pozitív kitevő */
                        val *= 10;
            else
                  while (exp-- > 0)     /* negatív kitevő */
                        val /= 10;
      }
      return val;
}
```

A program első fele megegyezik az eredeti `atof` függvénnyel (K–R.: 86. oldal). A függvény átlépi az üres helyeket (szóközöket, ill. tabulátorokat), eltárolja az előjelet és kiszámítja a számot. Ezen a ponton az eredeti `atof` visszatér a kapott számmal, a módosított változat viszont a tudományos jelölésmódot kezdi feldolgozni.

A program másik fele kezeli az esetleges kitevőt. Ha nincs kitevő, akkor a függvény a `val` értékkel tér vissza. Ha volt kitevő, akkor a program annak előjelét eltárolja a `sign` változóba, kiszámítja a kitevő értékét és elhelyezi az `exp` változóba.

A módosított program leglényegesebb része a

```
if (sign == 1)
      while (exp-- > 0)     /* pozitív kitevő */
            val *= 10;
else
      while (exp-- > 0)     /* negatív kitevő */
            val /= 10;
```

programrész, amely beállítja a szám nagyságrendjét a kitevőnek megfelelően. Ez úgy történik, hogy ha a kitevő pozitív, akkor a `val`-ban tárolt számot `exp`-szer szorozza tízzel, ill. ha a kitevő negatív, akkor ugyanígy osztja tízzel. Ezek után `val` már a tényleges számot tartalmazza, amivel a függvény visszatér a hívó eljárásba.

A programban a tízzel való osztást használtuk a 0.1-del való szorzás helyett, mivel 0.1 nem ábrázolható véges bináris törtként. A legtöbb számítógép esetén 0.1 bináris értéke egy kicsit kisebb a tényleges értéknél, ezért bináris formában elvégezve a 10*0.1 szorzást nem 1 lesz az eredmény. A tízzel való ismételt osztás a pontosság szempontjából kedvezőbb, mint a 0.1-gyel való ismételt szorzás (bár a pontosság mindenképpen csökken, ha a kitevő nagy).

4.3. gyakorlat

Adott a kalkulátorprogram váza. Bővítsük ezt ki a modulus (%) operátorral és gondoskodjunk a negatív számok (egy operandusú -) kezeléséről (K–R: 93. oldal)!

```
#include <stdio.h>
#include <math.h>    /* az atof miatt */

#define    MAXOP 100 /* az operandus vagy operátor max. hossza */
#define    SZAM  '0' /* jelzi, hogy számot talált */

int getop (char []);
void push (double);
double pop (void);

/* Fordított lengyel jelölésmóddal működő kalkulátorprogram */
main()
{
      int tipus;
      double op2;
      char s[MAXOP];

      while ((tipus = getop(s)) != EOF) {
            switch (tipus) {
            case SZAM :
                  push(atof(s));
                  break;
            case '+' :
                  push(pop() + pop());
                  break;
            case '*' :
                  push(pop()*pop());
                  break;
            case '-' :
                  op2 = pop();
                  push( pop() - op2 );
                  break;
            case '/' :
                  op2 = pop();
                  if (op2 != 0.0)
                        push(pop()/op2);
                  else
                        printf("Hiba: osztás nullával\n");
                  break;
            case '%' :
                  op2 = pop();
```

```
                if (op2 != 0.0)
                        push(fmod(pop(), op2));
                else
                        printf("Hiba: osztás nullával\n");
                break;
        case '\n' :
                printf("\t%.8g\n", pop());
                break;
        default :
                printf("Hiba: ismeretlen parancs %s\n", s);
                break;
        }
    }
    return 0;
}
```

A feladat új megfogalmazása miatt a `main` eljárást és a `getop` függvényt kell módosítani. A `pop` és `push` eljárások változatlanok maradnak (K–R: 91–92. oldal).

A modulus (`%`) operátort az osztás (`/`) operátorhoz hasonlóan kezeljük. Az `fmod` könyvtári függvény kiszámítja a verem tetején lévő két elem egészosztásának maradékát. Az `op2` a verem tetején lévő elem.

A `getop` függvény módosított változata:

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

#define   SZAM   '0' /* jelzi, hogy számot talált */

int getch (void);
void ungetch (int);

/* getop: megadja a következő operátort vagyszámot (operandust) */
int getop (char s[])
{
    int c, i;

    while ((s[0] = c = getch()) ==  ' ' || c == '\t')
        ;
    s[1] = '\0';
    i = 0;
    if (!isdigit(c) && c != '.' && c ! = '-')
        return c;                           /* nem szám */
    if (c == '-')
        if (isdigit(c = getch()) || c == '.')
            s[++i] = c;                     /* negatív szám */
        else {
            if  (c != EOF)
```

```
                ungetch(c);
            return '-';                 /* mínusz előjel */
        }
    if (isdigit(c))         /* összegyűjti az egész részt */
        while ( isdigit(s[++i] = c = getch()) )
            ;
    if (c == '.')               /* összegyűjti a törtrészt */
        while ( isdigit(s[++i] = c = getch()) )
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return  SZAM;
}
```

A `getop` függvény figyeli, hogy a legutolsó karakter a mínusz előjel volt-e, mivel így tudja kezelni a negatív számot. Például:

```
- 1
```

egy mínusz jel, amit egy szám követ (– operátor), viszont a

```
-1.23
```

egy negatív szám (– előjel).

A módosított kalkulátorprogram pl. képes kezelni a következő kifejezéseket is:

```
  1      -1   +, vagy
-10       3   %
```

Az első kifejezés értéke `0`, mivel `1+(-1 = 0`, a második kifejezés értéke `-1`.

4.4. gyakorlat

Bővítsük a kalkulátorprogramot új parancsokkal. Az egyik parancs írja ki a verem tetején lévő elemet anélkül, hogy az a veremből eltűnne, a másik cserélje meg a verem tetején lévő két elemet, a harmadik készítsen másolatot a verem tetején lévő elemről, a negyedik pedig törölje a vermet (K–R: 94. oldal).

```
#include <stdio.h>
#include <math.h>   /* az atof miatt */

#define   MAXOP 100 /* az operandus vagy operátor max. hossza */
```

```
#define   SZAM  '0' /* jelzi, hogy számot talált */

int getop (char []);
void push (double);
double pop (void);
void clear (void);

/* Fordított lengyel jelölésmóddal működő kalkulátorprogram */
main()
{
      int tipus;
      double op1, op2;
      char s[MAXOP];

      while ((tipus = getop(s)) != EOF) {
            switch (tipus) {
            case SZAM :
                  push(atof(s));
                  break;
            case '+' :
                  push(pop() + pop());
                  break;
            case '*' :
                  push(pop()*pop());
                  break;
            case '-' :
                  op2 = pop();
                  push(pop() - op2);
                  break;
            case '/' :
                  op2 = pop();
                  if (op2 != 0.0)
                        push(pop()/op2);
                  else
                        printf("Hiba: osztás nullával\n");
                  break;
            case '?' : /* kiírja a verem tetején lévő elemet */
                  op2 = pop();
                  printf("\t%.8g\n", op2);
                  push(op2);
                  break;
            case 'c' :    /* törli a vermet */
                  clear();
                  break;
            case 'd' : /* duplikálja a verem tetején lévő elemet */
                  op2 = pop();
                  push(op2);
                  push(op2);
                  break;
```

```
            case 's' : /* megcseréli a verem tetején lévő két elemet */
                  op1 = pop();
                  op2 = pop();
                  push(op1);
                  push(op2);
                  break;
            case '\n' :
                  printf("\t%.8g\n", pop());
                  break;
            default :
                  printf("Hiba: ismeretlen parancs %s\n", s);
                  break;
            }
      }
      return 0;
}
```

Az újsor operátor behozza a verem tetején lévő elemet és kiírja az értékét. Az új `'?'` operátor behozza a verem tetején lévő elemet, kiírja az értékét, majd visszateszi a verembe. Az újsor operátortól eltérően nem hagyjuk meg a veremből kivett változót, hanem egy `pop` – nyomtatás – `push` utasítássorozattal tesszük vissza, ami lehetővé teszi, hogy a `main` eljárás mindig pontosan nyomon követhesse a verem és a veremmutató változását.

A verem tetején lévő elemet a kiolvasást követő kétszeri kiírással kettőzzük meg.

A verem tetején lévő két elemet úgy cseréljük fel, hogy kiolvasásuk után fordított sorrendben írjuk vissza azokat.

A verem törlésének legegyszerűbb módja, hogy nulla értéket írunk az `sp` veremmutatóba. Az új parancs pontosan ezt csinálja a `clear` függvény felhasználásával. A `clear` függvényt célszerű a `pop` és `push` függvényekkel együtt elhelyezni, mivel így csak a vermet kezelő függvények férhetnek hozzá a verem változóihoz. A `clear` függvény:

```
/* clear: törli a vermet */
void clear (void)
{
      sp =0;
}
```

4.5. gyakorlat

Tegyük lehetővé, hogy a kalkulátorprogramunk hozzáférjen olyan könyvtári függvényekhez, mint `sin`, `exp` és `pow`. Ezek a függvények a `<math.h>` headerben vannak, amelynek leírása a K–R. B. függelékének 4. pontjában található (K–R: 94. oldal).

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

```
#include <string.h>
#include <math.h>   /* az atof miatt */

#define   MAXOP 100 /* az operandus vagy operátor max. hossza */
#define   SZAM  '0' /* jelzi, hogy számot talált */
#define   NEV   'n' /* jelzi, hogy nevet talált  */

int getop (char []);
void push (double);
double pop (void);
void mathfnc (char []);

/* Fordított lengyel jelölésmóddal működő kalkulátor program */
main()
{
      int tipus;
      double op2;
      char s[MAXOP];

      while ((tipus = getop(s)) != EOF) {
            switch (tipus) {
            case SZAM :
                  push(atof(s));
                  break;
            case NEV :
                  mathfnc(s);
                  break;
            case '+' :
                  push(pop() + pop());
                  break;
            case '*' :
                  push(pop()*pop());
                  break;
            case '-' :
                  op2 = pop();
                  push(pop() - op2);
                  break;
            case '/' :
                  op2 = pop();
                  if (op2 != 0.0)
                        push(pop()/op2);
                  else
                        printf("Hiba: osztás nullával\n");
                  break;
            case '\n' :
                  printf("\t%.8g\n", pop());
                  break;
            default :
```

```
                printf("Hiba: ismeretlen parancs %s\n", s);
                break;
            }
        }
        return 0;
}

/* mathfnc: ellenőrzi az s karaktersorozatot, hogy */
/* tartalmaz-e matematikai függvényt */
void mathfnc (char s[])
{
        double op2;

        if (strcmp(s, "sin") == 0)
                push(sin(pop()));
        else if (strcmp(s, "cos") == 0)
                push(cos(pop()));
        else if (strcmp(s, "exp") == 0)
                push(exp(pop()));
        else if (strcmp(s, "pow") == 0)  {
                op2 = pop();
                push(pow(pop(), op2));
        } else
                printf("Hiba: %s függvény nincs\n", s);
}
```

A `getop` **módosított változata:**

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

#define   SZAM    '0' /* jelzi, hogy számot talált */
#define   NEV     'n' /* jelzi, hogy nevet talált  */

int getch (void);
void ungetch (int);

/* getop: megadja a következő operátort vagy */
/* számot(operandust)vagy matematikai függvényt*/
int getop (char s[])
{
        int c, i;

        while ((s[0] =  c = getch()) ==  ' ' || c == '\t')
                ;
        s[1] = '\0';
        i = 0;
        if (islower(c))  {  /* parancs vagy NEV */
                while (islower(s[++i] = c = getch()))
                        ;
```

```
            s[i] = '\0';
            if (c != EOF)
                  ungetch(c); /* egy karakterrel túlfutott */
            if (strlen(s) > 1)
                  return NEV; /* több mint egy karakter, név
                                                      lehet */
            else
                  return c;   /* ez parancs lehet */
      }
      if (!isdigit(c) && c != '.')
            return c;         /* nem szám */
      if (isdigit(c))         /* összegyűjti az egész részt */
            while (isdigit(s[++i] = c = getch()))
                  ;
      if (c == '.')           /* összegyűjti a törtrészt */
            while (isdigit(s[++i] = c = getch()))
            ;
      s[i] = '\0';
      if (c != EOF)
            ungetch(c);
      return  SZAM;
}
```

Úgy módosítottuk a `getop` eljárást, hogy az képes legyen kisbetűkből álló karaktersorozatok fogadására is, és ilyenkor `NEV` típussal tér vissza. A `main` felismeri, hogy a `NEV` egy legális típus és hívja a `mathfnc` eljárást.

A `mathfnc` eljárás teljesen új. Ez `if` utasítások sorozatával megkeresi az `s` karaktersorozathoz illeszkedő függvénynevet, ill. hibajelzést ad, ha nincs ilyen nevű függvény. Ha `s` tartalma az egyik megengedett függvény neve, akkor a `mathfnc` behozza a veremből a megfelelő számú adatot és kiszámítja a függvény értékét. A `mathfnc` a kapott értéket a verembe téve tér vissza.

Például a `sin` függvény radiánban várja az argumentumot, és `PI/2` szinusza 1. Ezt a programnak

```
3.141592 2 / sin
```

formában írjuk be. Ekkor a program először kiszámítja `PI/2` értékét és az eredményt beírja a verembe. A `sin` függvény a verem legfelső elemét véve kiszámítja annak szinuszát, majd az eredményt ismét visszaírja a verembe. A végeredmény természetesen 1 lesz. A

```
3.141592 2 / sin 0 cos +
```

karaktersorozatot beírva az eredmény 2 lesz, mivel `sin(PI/2)` = 1, `cos(0)` = 1 és 1 + 1 = 2.

Egy további példa:

```
5 2 pow 4 2 pow +
```

az $5^2 + 4^2$ kifejezést jelenti és értéke 41.

A `getop` nem ismeri fel a függvény nevét, csak a megtalált karaktersorozattal tér vissza, ily módon a `mathfnc` tetszőlegesen bővíthető további függvényekkel.

4.6. gyakorlat

Bővítsük úgy a kalkulátorprogramot, hogy képes legyen változók kezelésére is. (Ezt könnyű megvalósítani, ha 26 változót engedünk meg, és minden változó nevéül az angol ábécé egy betűjét választjuk.) Rendeljünk egy változót a legutoljára kiírt értékhez is (K–R: 94. oldal).

```
#include <stdio.h>
#include <math.h>   /* az atof miatt */

#define   MAXOP 100 /* az operandus vagy operátor max. hossza */
#define   SZAM  '0' /* jelzi, hogy számot talált */

int getop (char []);
void push (double);
double pop (void);

/* Fordított lengyel jelölésmóddal működő kalkulátorprogram */
main()
{
      int i, tipus valt = 0;
      double op2 v;
      char s[MAXOP];
      double valtozo[26];

      for (i = 0; i < 26; i++)
            valtozo[i] = 0.0;
      while ((tipus = getop(s)) != EOF) {
            switch (tipus) {
            case SZAM :
                  push(atof(s));
                  break;
            case '+' :
                  push(pop() + pop());
                  break;
            case '*' :
                  push(pop()*pop());
                  break;
            case '-' :
                  op2 = pop();
                  push(pop() - op2);
                  break;
            case '/' :
                  op2 = pop();
                  if (op2 != 0.0)
                        push(pop()/op2);
                  else
```

```
                        printf("Hiba: osztás nullával\n");
                    break;
                case '=' :
                    pop();
                    if (valt >= 'A' && valt <= 'Z')
                        valtozo[valt - 'A'] = pop();
                    else
                        printf("Hiba: nem változónév\n");
                    break;
                case '\n' :
                    v = pop();
                    printf("\t%.8g\n", v);
                    break;
                default :
                    if (tipus >= 'A' && tipus <= 'Z')
                        push(valtozo[tipus - 'A']);
                    else if (tipus == 'v')
                        push(v);
                    else
                        printf("Hiba: ismeretlen parancs %s\n", s);
                    break;
                }
                valt = tipus;
        }
        return 0;
}
```

A változókat az A-tól Z-ig terjedő nagybetűkkel jelöljük, és a változó nevét adó betűt használjuk a változók tömbjének indexelésére. Az egyetlen kisbetűs változó a v, ami az utoljára kiírt értéket tartalmazza.

Amikor a program egy változónevet talál (A–Z vagy v), akkor kirakja annak értékét a verembe. Bevezettünk egy új műveletet (amit az '=' operátor jelöl) is, ami hozzárendeli a veremben lévő értéket az előző változónévhez. Például a

```
3 A =
```

hatására az A változóhoz a 3 érték rendelődik. Ezután a

```
2 A +
```

hatására 2 adódik a 3-hoz (ami az A változó értéke). Az újsor operátor hatására kiírja az eredményt (5) és egyben eltárolja a v változóba is. Ha a következő művelet pl.

```
v 1 +
```

lesz, akkor ez 5+1 = 6 értéket eredményez.

4.7. gyakorlat

Írjunk `ungets(s)` néven függvényt, amely egy teljes karaktersorozatot "visszaír" a bemenetre! Kezelje az `ungets` függvény közvetlenül a `buf` és `bufp` változókat, vagy egyszerűen csak használja az `ungetch` függvényt (K–R: 94. oldal)?

```
#include <string.h>
/*ungets: a karaktersorozatot "visszaírja" a bemenetre*/
void ungets (char s[])
{
      int hossz = strlen(s);
      void ungetch (int);

      while (hossz > 0)
            ungetch(s[-hossz]);
}
```

A `hossz` változó az `s` karaktersorozatban lévő karakterek számát tartalmazza (kivéve a záró `'\0'` karaktert), és ezt az `strlen` (K–R: 53. oldal) határozza meg.

Az `ungets` `hossz`-szor hívja az `ungetch` függvényt (K–R: 93. oldal) és minden alkalommal az `s` karaktersorozat egy karakterét "visszaírja" a bemenetre. Az `ungets` függvény a karaktersorozatot fordított sorrendben írja vissza a bemenetre.

Az `ungets` függvénynek nem szükséges ismernie a `buf` és `bufp` változókat, mivel azokat az `ungetc` függvény kezeli és elvégzi a hibaellenőrzést is.

4.8. gyakorlat

Tegyük fel, hogy soha nem akarunk egynél több karaktert "visszaírni" a bemenetre! Módosítsuk ennek megfelelően a `getch` és `ungetch` függvényeket (K–R: 94. oldal)!

```
#include <stdio.h>

char buf = 0;
/*getch: bevesz egy karaktert, lehetővé téve annak visszaírását*/
int getch (void)
{
      int c;

      if (buf != 0);
            c = buf;
```

```
        else
                c = getchar();
        buf = 0;
        return c;
}

/* ungetch: visszaír egy karaktert a bemenetre */
void ungetch (int c)
{
        if (buf != 0)
                printf("ungetch: puffer-túlcsordulás\n");
        else
                buf = c;
}
```

A `buf` puffer most nem egy karakteres tömb, mivel egyszerre egynél több karakter nem lehet a pufferben.

A program a `buf` változót kellő időben nulla kezdőértékkel inicializálja és a `getch` függvény is minden alkalommal nullázza, amikor beolvas egy karaktert. Az `ungetch` függvény mielőtt "visszaírna" egy karaktert ellenőrzi, hogy üres-e a puffer, és ha nem, akkor hibajelzést ad.

4.9. gyakorlat

A példában használt `getch` és `ungetch` függvények nem kezelik helyesen a "visszaírt" `EOF` karaktert. Határozzuk meg a helyes `EOF` kezelés módját és egészítsük ki ezzel a programtervet (K–R: 94. oldal).

```
#include <stdio.h>

#define  BUFSIZE  100

int buf[BUFSIZE]; /* puffer az ungetch számára          */
int bufp = 0;     /* a következő szabad hely a pufferben */

/* getch: bevesz egy karaktert, lehetővé téve annak visszaírását */
int getch (void)
{
        return (bufp > 0) ? buf[--bufp] : getchar();
}

/* ungetch: visszaír egy karaktert a bemenetre */
void ungetch (int c)
```

```
{
        if (bufp >= BUFSIZE)
                printf("ungets: puffer-túlcsordulás\n");
        else
                buf[bufp++] = c;
}
```

A `getch` és `ungetch` eljárásokban (K–R: 93. oldal) a `buf` puffert karakteres tömbként,

```
char buf[BUFSIZE];
```

módon deklaráltuk.

A C programozási nyelv nem igényli, hogy a `char` típusú változót `signed` vagy `unsigned` módon minősítsük (K–R: 56. oldal), mivel ha egy `char` típusú változót `int` típusúvá alakítunk, soha nem jöhet létre negatív szám. Néhány számítógép esetén viszont a `char` típusú adatok legfelső bitje 1 állapotú, így `int` típusúra alakítva negatív számot kapunk. Más esetekben a `char` – `int` típuskonverzió során a bal szélső bitbe 0 íródik, ami a legfelső bit eredeti állapotától függetlenül garantálja a pozitív eredményt.

Hexadecimális alakban az `EOF`-ot reprezentáló –1 a `0xFFFF` (16 bites) számmal adható meg. A `0xFFFF` értéket `char` típusú változóként tárolva a `0xFF` számot kapjuk, ami `int` típusúvá alakítva vagy `0x00FF` (ami decimális 255-nek felel meg), vagy `0xFFFF` (ami decimális –1 értéknek felel meg) értéket eredményez. Összefoglalva:

negatív szám (–1)	*karakterkód*	*egész szám*	
`0xFFFF`	`0xFF`	`0x00FF`	(255)
`0xFFFF`	`0xFF`	`0xFFFF`	(–1)

Ha az `EOF`-ot (–1) úgy akarjuk kezelni, mint bármelyik más karaktert, akkor a `buf` puffert egész típusú tömbként kell deklarálnunk az

```
int buf[BUFSIZE];
```

formában.

Ebben az esetben semmiféle típuskonverzióra nincs szükség és az `EOF` vagy bármilyen más negatív szám kezelése számítógéptől és fordítóprogramtól függetlenül (hordozható módon) valósul meg.

4.10. gyakorlat

Tegyük fel, hogy egy `getline` függvénnyel a teljes bemeneti sort egyszerre olvassuk be. Ekkor nincs szükség a `getch` és `ungetch` függvényekre. Gondoljuk át, hogy ez hogyan módosítja a kalkulátorprogramot (K–R: 94. oldal)!

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

#define MAXSOR  100
#define SZAM    '0'  /* jelzi, hogy számot talált */

int getline (char sor[], int hatar);

int si = 0;          /* a bemeneti sor indexe      */
char sor[MAXSOR];    /* egy bemeneti sor           */

/* getop: megadja a következő operátort vagy számot (operandust) */
int getop (char s[])
{
      int c, i;

      if (sor[si] == '\0')
            if (getline(sor, MAXSOR) == 0)
                  return EOF;
            else
                  si = 0;
      while ((s[0] = c = sor[si++]) == ' ' || c == '\t')
            ;
      s[1] = '\0';
      if( !isdigit(c)) && c != '.' )
            return c;       /* nem szám */
      i = 0;
      if( isdigit(c) )    /* összegyűjti az egész részt */
            while (isdigit(s[++i] = c = sor[si++]))
                  ;
      if ( c == '.' )     /* összegyűjti a törtrészt */
            while (isdigit(s[++i] = c = sor[si++]))
                  ;
      s[i] = '\0';
      si--;
      return SZAM;
}
```

Ebben a `getop` eljárásban a `getch` és `ungetch` függvények helyett a `getline` eljárást használjuk. A `sor` tömb egyszerre az egész bemeneti sort tartalmazza és `si` a sor következő karakterének indexe. A programban `sor` és `si` változókat `external` (külső) tárolási osztályúnak deklaráltuk, így a lokális változókhoz hasonlóan az egyes eljáráshívások közötti időben is megtartják az értéküket.

Ha a `getop` eljutott a sor végére (vagy még nincs beolvasva egy sor), amit az

```
if (sor[si] == '\0')
```

utasítás vizsgál, akkor hívja a `getline` függvényt és beolvas egy újabb sort.

Az eredeti `getop` (K–R: 92. oldal) minden esetben, ha egy új karakterre van szüksége a `getch` függvényt hívja. Ebben az esetben az új karaktert a `sor` tömb `si` indexű helyén kapjuk

meg és ezután növeljük `si` értékét. A függvény befejeztekor az eredeti változatban hívtuk az `ungetch` függvényt, hogy "visszaírjuk" a feleslegesen beolvasott karaktert a bemenetre, itt viszont elegendő `si` eggyel való csökkentése.

Emlékezzünk rá, hogy bármelyik eljárás használhatja és módosíthatja egy másik eljárás `external` tárolási osztályú változóit, vagyis a `sor` és `si` nem csak a `getop` függvényben használható. Ezt néha el akarjuk kerülni, és ilyenkor ezeket a változókat `static` tárolási osztályúnak deklaráljuk. Itt ezt a módszert nem alkalmazhattuk, mivel a `static` tárolási osztály fogalmát a K–R csak a 97. oldalon tárgyalja.

4.11. gyakorlat

Módosítsuk a `getop` függvényt úgy, hogy ne kelljen használnia az `ungetch` függvényt! Segítséget jelent, ha belső statikus változót használunk (K–R: 98. oldal)!

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

#define  SZAM  '0'  /* jelzi, hogy számot talált */

int getch (void);

/* getop: megadja a következő operátort vagy számot (operandust) */
int getop (char s[])
{
      int c, i;
      static int utkar = 0;

      if (utkar == 0)
         c = getch();
      else {
         c = utkar;
            utkar = 0;
      }
      while ((s[0] = c) == ' ' || c == '\t')
      c = getch();
      s[1] = '\0';
      if (!isdigit(c) && c != '.'
            return c;         /* nem szám */
      i = 0;
      if(isdigit(c))      /* összegyűjti az egész részt */
         while (isdigit(s[++i] = c = getch()))
                  ;
      if (c == '.')           /* összegyűjti a törtrészt */
         while (isdigit(s[++i] = c = getch()))
                  ;
```

```
        s[i] = '\0';
        if (c != EOF)
                utkar = c;
        return SZAM;
}
```

Úgy módosítottuk a `getop` függvényt, hogy az utolsónak beolvasott karaktert egy belső `static` tárolási osztályú változóban tároljuk, ezért szükségtelen annak a bemenetre való "visszaírása". Az utolsó beolvasott karaktert az `utkar` változóban tároljuk.

A `getop` a hívása után először ellenőrzi az `utkar` változót, és ha nincs eltárolt karakter, akkor az új karaktert a `getch` hívásával kéri be. Ha volt előzőleg eltárolt karakter, akkor a `getop` ezt átmásolja a `c` változóba (az aktuális karakter) és nullázza az `utkar` változót. Az új `getop`-ban az első `while` utasítást kissé módosítottuk, mivel csak a `c` változóban tárolt aktuális karakter vizsgálata után kell egy új karaktert bekérni a `getch` függvénnyel.

4.12. gyakorlat

A `printd` függvényben alkalmazott elgondolást felhasználva írjuk meg az `itoa` függvény rekurzív változatát! A függvény rekurzív hívásokkal alakítson egy egész számot karaktersorozattá (K–R: 102. oldal)!

```
#include <math.h>

/* itoa: az n egész számot rekurzív eljárással s */
/* karaktersorozattá alakítja */
void itoa (int n, char s[])
{
        static int i;

        if (n/10)
                itoa(n/10, s);
        else {
                i = 0;
                if (n < 0)
                        s[i++] = '-';
        }
        s[i++] = abs(n) % 10 + '0';
        s[i]   = '\0';
}
```

Az `itoa` függvénynek két argumentuma van: az `n` egész típusú és az `s` karaktertömb típusú változó. Ha az `n/10` egészosztás eredménye nem nulla, akkor a függvény az `n/10` értékkel saját magát hívja. Ezt az

```
if (n/10)
      itoa(n/10, s);
```

utasítások végzik.

Ha az egyik rekurzív hívás során `n/10` nullává válik, akkor éppen `n` legmagasabb helyiértékű számjegyét dolgozzuk fel. A programban `static` tárolási osztályúnak deklarált `i` változó az `s` tömb indexe. Ha `n` negatív, akkor elhelyezzük a mínusz előjelet az `s` tömb első helyén, majd eggyel növeljük `i` értékét. Amikor az `itoa` a rekurzív hívásokból visszatér, akkor balról jobbra sorrendben kiszámította a számjegyeket. Vegyük észre, hogy minden hívási szinten egy `'\0'` végjellel lezárjuk a karaktersorozatot, majd ezt a végjelet – az utolsó lépést kivéve – a következő szinten felülírjuk.

4.13. gyakorlat

Írjuk meg az `s` karaktersorozatot helyben megfordító `reverse(s)` függvény rekurzív változatát (K–R: 102. oldal)!

```
#include <string.h>

/* reverse: az s karaktersorozat megfordítása helyben */
void reverse (char s[])
{
      void reverser (char s[], int i, int hossz)

      reverser(s, 0, strlen(s));
}

/* reverser: az s karaktersorozatot helyben megfordítja */
/* rekurzív változat */
void reverser (char s[], int i, int hossz)
{
      int c, j;

      j = hossz - (i + 1);
      if (i < j) {
            c    = s[i];
            s[i] = s[j];
            s[j] = c;
            reverser(s, ++i, hossz);
      }
}
```

Fontosnak tartottuk, hogy az új függvény megtartsa az eredeti felhasználói interfészt, ezért a `reverse` függvénynek csak egyetlen, karaktersorozat típusú argumentumot adunk át.

A `reverse` meghatározza a karaktersorozat hosszát, majd hívja a `reverser` függvényt, ami helyben, rekurzív módon megfordítja a karaktersorozatot.

A `reverser` három argumentumot kap: a megfordítandó `s` karaktersorozatot, a karaktersorozat `i` bal oldali indexét, valamint a karaktersorozat hossz hosszát (amit az `strlen` függvény határoz meg, K–R: 53. oldal).

A `reverser` első hívásakor `i = 0` és a karaktersorozat `j` jobb oldali indexe a

```
j = hossz - (i + 1);
```

kifejezéssel számítható ki.

A karaktersorozat karakterei kívülről befelé haladva cserélődnek fel, azaz az első két felcserélt karakter az `s[0]` és `s[hossz-1]`, a második kettő az `s[1]` és `s[hossz-2]`. Az `i` bal oldali index a `reverser` minden hívása során eggyel növekszik a

```
reverser(s, ++i, hossz);
```

utasítás hatására.

A cserélés addig folytatódik, amíg a két index ugyanarra a karakterre nem mutat (`i == j`), vagy a bal oldali index a jobb oldali indextől jobbra lévő karakterre nem mutat (`i > j`).

A program nem igazán jó példa a rekurzió alkalmazására. Néhány megoldása természetes módon igényli a rekurziót – lásd, pl. a `treeprint` programot a K–R: 155. oldalon –, más feladatok viszont egyszerűbben oldhatók meg rekurzió nélkül. Az itt tárgyalt feladat az utóbbi csoportba tartozik.

4.14. gyakorlat

Definiáljunk egy `swap(t, x, y)` makrót, amely felcseréli a két `t` típusú argumentumát (`x`-et és `y`-t)! A megoldásban segítségünkre lesz a blokkstruktúra (K–R: 105. oldal).

```
#define  swap(t, x, y)  .{   t _z;      \
                             _z = y;    \
                             y  = x;    \
                             x  = _z; }
```

A kapcsos zárójelekkel egy új blokkot definiáltunk, amelynek kezdetén definiáltuk a cseréhez átmenetileg szükséges `t` típusú `_z` lokális változót.

A `swap` makró csak akkor működik helyesen, ha egyik argumentuma sem `_z`. Ha pl. az egyik argumentum `_z`, akkor a

```
swap(int, _z, x);
```

makrót a fordítóprogram

```
{ int _z; _z = _z; _z = x; x = _z; }
```

formában fejti ki és láthatóan nem cseréli fel a két argumentumot. A makró használatának feltétele, hogy nem használjuk az `_z`-t változónévként.

5.1. gyakorlat

Ahogy ezt a példaprogramban láttuk, a `getint` az olyan + vagy - előjelet, ami után nem következik számjegy, érvényes, nulla értékű adatként kezeli. Szüntessük meg ezt a problémát úgy, hogy egy nullát visszaírunk a bemenetre (K–R: 111. oldal)!

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

int getch (void);
void ungetch (int);

/* getint: a bemenetről beolvas egy egész számot */
/* és a *pn helyre teszi */
int getint (int *pn)
{
      int c, d, sign;

      while (isspace(c = getch())) /* átlépi az üres helyeket */
            ;
      if (!isdigit(c) && c != EOF && c != '+' && c != '-')  {
            ungetch(c);                          /* ez nem szám */
            return 0;
      }
      sign = (c == '-') ? -1 : 1;
      if (c == '+' || c == '-')  {
            d = c;          /* eltárolja az előjel karaktert */
            if (!isdigit(c = getch()))  {
                  if (c != EOF)
                        ungetch(c); /* visszaír egy nem számjegy */
                                                  /* karaktert */
                  ungetch(d) /* visszaírja az előjel karaktert */
                  return d;
            }
      }
      for (*pn = 0; isdigit(c); c = getch())
```

```
            *pn = 10 * *pn + (c - '0');
        *pn *= sign;
        if (c != EOF)
            ungetch(c);
        return c;
}
```

Ha a program előjel karaktert talál, akkor elteszi azt a `d` változóba és veszi a következő karaktert. Ha a következő karakter nem számjegy vagy `EOF`, akkor ezt visszaírja a bemenetre, majd visszaírja az előjel karaktert is, és az előjel karakterrel visszatér a hívó eljárásba, jelezve a helyzetet.

5.2. gyakorlat

Írjuk meg a `getfloat` függvényt, ami a `getint` lebegőpontos megfelelője! Milyen típusú függvényértékkel tér vissza a `getfloat` (K–R: 111. oldal)?

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

int getch (void);
void ungetch (int);

/* getfloat: a bemenetről beolvas egy lebegőpontos */
/* számot és a *pn helyre teszi */
int getfloat (float *pn)
{
        int c, sign;
        float power;

        while (isspace(c=getch())) /* átlépi az üres helyeket */
            ;
        if (!isdigit(c) && c != EOF && c != '+' &&
            c != '-' && c != '.')  {
            ungetch(c);                         /* ez nem szám */
            return 0;
        }
        sign = (c == '-') ? -1 : 1;
        if (c == '+' || c == '-')
            c = getch();
        for (*pn = 0.0; isdigit(c); c = getch())
            *pn = 10.0 * *pn + (c - '-');     /* egész rész */
        if (c == '.')
            c = getch();
        for (power = 1.0; isdigit(c); c = getch())  {
```

```
        *pn = 10.0 * *pn + (c - '0');          /* törtrész */
        power *= 10.0;
    }
    *pn *= sign / power;                    /* a végső szám */
    if (c != EOF)
        ungetch(c);
    return c;
}
```

A `getfloat` függvény teljesen hasonló a `getint` függvényhez (K–R: 111. oldal). A `getfloat` is átlépi az üres helyeket (szóközöket és tabulátorokat), elteszi az előjelet és eltárolja az egész részt a `pn` címre.

A `getfloat` ezek után a szám törtrészét dolgozza fel (de a tudományos jelölésmód kezelésére nem alkalmas). A törtrészt ugyanolyan módon helyezzük a `*pn` helyre, ahogyan az egész részt. Ezt a

```
*pn = 10.0 * *pn + (c - '0');
```

utasítás végzi.

Minden egyes tizedespont utáni számjegy feldolgozása után a `power` változó értéke 10-zel szorzódik. Például, ha a tizedespont után nulla számjegy következik, akkor `power = 1`, ha egy számjegy következik, akkor `power = 10` és ha három számjegy következik, akkor `power = 1000`.

Ezek után a `getfloat` a `*pn` helyen lévő lebegőpontos szám végső értékét úgy kapja meg, hogy az értéket szorozza a `sign/power` számmal.

A `getfloat` függvény `int` típusú és az `EOF`-fal vagy a lebegőpontos szám utáni első ASCII karakterrel tér vissza a hívó eljárásba.

5.3. gyakorlat

Írja meg a K–R 2. fejezetében bemutatott `strcat(s, t)` függvény mutatóval megvalósított változatát! Az `strcat(s, t)` függvény a `t` karaktersorozatot az `s` karaktersorozat végéhez másolja (K–R: 121. oldal).

```
/* strcat: a t karaktersorozatot az s karaktersorozat végéhez */
/* másolja - mutatóval megvalósított változat */
void strcat (char *s, char *t)
{
    while (*s)
        s++;
    while(*s++ = *t++)
        ;
}
```

Az elején s és t a karaktersorozatok kezdetére mutat. Az első while ciklus addig növeli az s mutatót, amíg meg nem találja a karaktersorozat végjelét (a '\0' karaktert). A

```
while (*s)
```

utasítást addig hajtja végre a program, amíg a vizsgált karakter nem végjel.

A második while ciklus s megtalált végjelétől kezdve az s-hez fűzi a t karaktersorozatot. Ezt a

```
while (*s++ = *t++)
       ;
```

utasítások végzik. Ez a ciklus *s-hez rendeli *t-t, majd eggyel növeli mindkét mutatót és addig folytatja a műveletet, amíg csak el nem ér a t karaktersorozat végjeléhez.

5.4. gyakorlat

Írjon strend(s, t) néven függvényt, amely 1 értékkel tér vissza, ha a t karaktersorozat megtalálható az s karaktersorozat végén, és 0 értékkel, ha nem (K–R: 121. oldal)!

```
/* strend: 1 értékkel tér vissza, ha a t karakter-sorozat */
/* megtalálható az s karaktersorozat végén */
int strend (char *s, char *t)
{
      char *bs = s;     /* eltárolja a s kezdetét */
      char *bt = t;     /* eltárolja a t kezdetét */

      for ( ; *s; s++)       /* megkeresi s végét */
            ;
      for ( ; *t, t++)        /*megkeresi t végét */
            ;
      for ( ; *s == *t; s--, t--)
            if (t == bt || s == bs)
                break; /* vége az egyik karaktersorozatnak */
      if (*s == *t && t == bt && *s != '\0')
            return 1;
      else
            return 0;
}
```

A program abs és bt mutatókba eltárolja az s és t karaktersorozatok kezdőcímét és a strcat függvényhez hasonló módon megkeresi az egyes karaktersorozatok végét. Ahhoz, hogy meghatározzuk a t karaktersorozat előfordulását az s karaktersorozat végén, az s utolsó karakterét összehasonlítjuk t utolsó karakterével, majd egyezés esetén visszafele haladva folytatjuk az eljárást.

Az `strend 1` értékkel tér vissza, ha a `t`-ben lévő karakterek megtalálhatók `s` végén, amit az jelez, hogy a `t` mutatója visszaáll a karaktersorozat kezdetére és a karaktersorozatok nem üresek. Ezt az

```
if (*s == *t && t == bt && *s != '\0')
      return 1;
```

utasítások valósítják meg.

5.5. gyakorlat

Írja meg az `strncpy`, `strncat` és `strncmp` könyvtári függvények saját változatát! Ezek a függvények az argumentumként megadott karaktersorozat legfeljebb első `n` karakterével végeznek műveletet, pl. az `strncpy(s,t, n)` a `t` karaktersorozat legfeljebb első `n` karakterét másolja `s`-be. (A könyvtári függvények leírása K–R. B. függelékében található.) (K–R: 121. oldal.)

```
/* strncpy: n karaktert átmásol t-ből s-be */
void strncpy (char *s, char *t, int n)
{
      while (*t && n-- > 0)
            *s++ = *t++;
      while (n-- > 0)
            *s++ = '\0';
}

/* strncat: n karaktert hozzákapcsol t-ből s végéhez */
void strncat (char *s, char *t, int n)
{
      void strncpy (char *s, char *t, int n);
      int strlen (char *);

      strncpy(s + strlen(s), t, n);
}

/*strncmp: t legfeljebb n karakterét összehasonlítja s */
/* karaktereivel*/
int strncmp (char *s, char *t, int n)
{
      for ( ; *s == *t; s++, t++)
            if (*s == '\0' || --n <= 0)
                  return 0;
      return *s - *t;
}
```

Az `strncpy` és `strncat` függvények a K–R 119. oldalán ismertetett `strcpy` függvényhez hasonlóan `void` típusúak. A függvények standard könyvtári változatai a cél-karaktersorozat kezdetét kijelölő mutatóval térnek vissza.

Az `strncpy` `t`-ből legfeljebb `n` karaktert másol `s`-be. Ha `t` rövidebb `n` karakternél, az `s`-ben fennmaradó helyek `'\0'` végjellel töltődnek fel.

Az `strncat` hívja az `strncpy` függvényt és azt felhasználva másolja `t` `n` karakterét az `s` karaktersorozat végéhez.

Az `strncmp` `t` `n` karakterét hasonlítja `s` karaktereihez. A függvény hasonlóan működik, mint a K–R 120. oldalán ismertetett `strcmp`, csak az összehasonlítás akkor fejeződik be, ha elérjük az egyik karaktersorozat végét, vagy ha sikeresen megtörtént az `n` karakter összehasonlítása. Ezt az

```
if (*s == '\0' || --n <= 0)
        return 0;
```

utasításokkal valósítjuk meg.

5.6. gyakorlat

Írjuk át a korábbi fejezetek erre alkalmas példaprogramjait úgy, hogy indexelt tömbök helyett mutatókat használjunk! Erre kiválóan alkalmas az 1. és 4. fejezetben megírt `getline`, a 2., 3. és 4. fejezetben megírt `atoi`, `itoa` minden változata, a 3. fejezetben használt `reverse`, valamint a 4. fejezetben használt `strindex` és `getop` függvény (K–R: 121. oldal).

```
#include <stdio.h>

/* getline: egy sort beolvas s-be és megadja a  hosszát */
int getline (char *s, int hatar)
{
        int c;
        char *t = s;

        while (--hatar > 0 && (c = getchar()) != EOF && c != '\n')
                *s++ = c;
        if (c == '\n')
                *s++ = c;
        *s = '\0';
        return s - t;
}
```

A `getline` egy mutatót rendel a karaktertömbhöz. Az `s[i]` tömb helyett a `*s`-t használjuk, és ezen a mutatón keresztül címezzük a tömb elemeit, ill. ezt a mutatót inkrementálva haladunk végig a karaktersorozaton. Az

```
s[i++] = c;
```

utasítás egyenértékű az

```
*s++ = c;
```

utasítással.
A függvény működésének kezdetén s a karakteres tömb kezdetét jelöli ki, és a tömb kezdetének címét a t mutatóba tároljuk el a

```
char *t = s;
```

utasítással.
Miután a getline beolvasott egy sort, s a végjelre ('\0') mutat, t viszont továbbra is a tömb elejére. Így a sor hossza s - t lesz, ami a visszatérési érték.

```
#include <ctype.h>

/* atoi: s karaktersorozat egész számmá alakítása */
/* - mutatós változat */
int atoi (char *s)
{
      int n, sign;

      for ( ; isspace(*s); s++)    /* átlépi az üres helyeket */
            ;
      sign = (*s == '-' ? -1 : 1;
      if (*s == '+' || *s == '-')     /* átlépi az előjelet */
            s++;
      for (n = 0; isdigit(*s); s++)
            n = 10 * n + *s - '0';
      return sign * n;
}
```

A programban s[i] egyenértékű *s-sel és s[i++] pedig *s++-szal.

Az itoa függvény:

```
void reverse(char *);

/* itoa: az n számot karaktersorozattá  alakítja */
/* -  mutatós változat */
void itoa (int n, char *s)
{
      int sign;
      char *t = s;      /* a mutató mentése s-be */

      if ((sign = n) < n)     /* az előjel eltárolása */
            n = -n;            /* n pozitívvá tétele   */
      do  {       /* generálja a számjegyeket, de     */
                              /* fordított sorrendben */
```

```
        *s++ = n % 10 + '0';        /* a következő számjegy */
    } while ((n /= 10) > 0);                    /* törli azt */
    if (sign < 0)
        *s++ = '-';
    *s = '\0';
    reverse(t);
}
```

A `t` karakteres mutatót az `s` karaktersorozat első elemére állítjuk a

```
char *t = s;
```

utasítással. Az új változat

```
*s++ = n % 10 + '0';
```

utasítása egyenértékű az

```
s[i++] = n % 10 + '0';
```

utasítással.

Az `itoa` függvény az `s` karaktersorozat kezdetét kijelölő mutatóval hívja a `reverse` függvényt.

```
#include <string.h>

/* reverse: az s karaktersorozat megfordítása helyben */
void reverse (char *s)
{
    int c;
    char *t;

    for (t = s + (strlen(s) - 1); s < t; s++, t--)  {
        c  = *s;
        *s = *t;
        *t = c;
    }
}
```

Az s a karaktersorozat első elemére mutat és a `t` mutatót a karaktersorozat utolsó (a `'\0'` végjel kivételével) elemére állítottuk a

```
t = s + (strlen(s) - 1)
```

kifejezéssel. Az új programban `*s` `s[i]`-nek, `*t` `s[j]`-nek felel meg, és a `for` ciklus

```
s < t
```

vizsgálata megfelel a korábbi

```
i < j
```

vizsgálatnak. Az `s++` hatása megegyezik az `i` index inkrementálásával (`i++`), és a `t--` a `j` index dekrementálásával (`j--`) .

Az `strindex` függvény:

```
/* strindex: visszaadja t indexét s-ben, ill. -1-et, */
/* ha a keresett minta nincs a sorban */
int strindex (char *s, char *t)
{
      char *b = s;  /* az s karaktersorozat kezdete */
      char *p, *r;

      for ( ; *s != '\0'; s++)  {
            for (p = s, r = t; *r != '\0' && *p == *r; p++, r++)
                  ;
            if (r > t && *r == '\0')
                  return s - b;
      }
      return -1;
}
```

Itt is az `s[i]`-t `*s`-sel, `s[j]`-t `*p`-vel és `t[k]`-t `*r`-rel helyettesítettük. `b` egy karakteres mutató, ami mindig az `s` karaktersorozat első elemére (`s[0]`-ra) mutat. A `p = s` egyenértékű a `j = i` kifejezéssel és az `r = t` a `k = 0` kifejezésnek felel meg. Amikor az

```
if (r > t && *r == '\0')
```

utasítás igaz értéket ad, akkor volt egyezés és az `strindex` `t` `s`-beli indexével tér vissza, amit a

```
return s - b;
```

kifejezéssel határoz meg.

Az `atof` függvény:

```
#include <ctype.h>

/* atof: az s karaktersorozat duplapontos számmá alakítása    */
double atof (char *s)
{
      double val, power;
      int sign;

      for ( ; isspace(*s); s++)  /* átlépi az üres helyeket */
            ;
      sign = (*s == '-') ? -1 : 1;
```

```
        if (*s == '+' || *s == '-')
                s++;
        for (val = 0.0; isdigit(*s); s++)
                val = 10.0 * val + (*s - '0');
        if (*s == '.')
                s++;
        for (power = 1.0; isdigit(*s); s++)  {
                val = 10.0 * val + (*s - '0');
                power *= 10.0;
        }
        return sign * val / power;
}
```

Az `s[i++]` itt is egyenértékű a `*s++` kifejezéssel.

A `getop` függvény új változata:

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

#define SZAM  '0'  /* jelzi, hogy számot talált */

int getch (void);
void ungetch (int);

/* getop: megadja a következő operátort vagy számot (operandust)*/
/* mutatós változat */
int getop (char *s)
{
        int c;

        while ((*s = c = getch()) == ' ' || c == '\t')
                ;
        *(s + 1) = '\0';
        if (!isdigit(c) && c != '.')
                return c;                          /* nem szám */
        if (isdigit(c))          /* összegyűjti az egész részt */
                while (isdigit(*++s = c = getch()))
                        ;
        if (c == '.')                /* összegyűjti a törtrészt */
                while (isdigit(*++s = c = getch()))
                ;
        *s = '\0';
        if (c != EOF)
                ungetch(c);
        return SZAM;
}
```

Itt is mutatókat használtunk a tömbelemek indexelt hivatkozása helyett. Az ebből adódó változás nyilvánvaló. Az

```
s[1] = '\0';
```

helyett a

```
*(s + 1) = '\0';
```

utasítással helyezzük el a karaktersorozat végjelét a tömb második elemében, anélkül, hogy megváltoztatnánk a mutató értékét.

5.7. gyakorlat

Módosítsuk a `readlines` függvényt úgy, hogy a beolvasott sorokat a `main` eljárás által létrehozott tömbben tárolja, és ne az `alloc` függvényen keresztül kérjen mindig helyet a sor számára! A program mennyivel lesz gyorsabb, ha elmarad az `alloc` hívása (K–R: 124. oldal)?

```
#include <string.h>

#define MAXHOSSZ  1000  /* a sor max. hossza */
#define MAXSTORE  5000  /* a rendelkezésre álló tárterület */

int getline (char *, int);

/* readlines: sorokat olvas be */
int readlines(char *sorptr[], char *sorstore, int maxsor)
{
      int hossz, nsor;
      char sor[MAXHOSSZ];
      char *p = sorstore;
      char *sorveg = sorstore + MAXSTORE;

      nsor = 0;
      while ((hossz = getline(sor, MAXHOSSZ)) > 0)
            if (nsor >= maxsor || p + hossz > sorveg)
                return -1;
            else  {
                sor[hossz - 1] = '\0'; /*törli az újsor-karaktert */
                  strcpy(p, sor);
                  sorptr[nsor++] = p;
                  p += hossz;
            }
      return nsor;
}
```

A `main` eljárásban deklaráljuk a `sorstore` tömböt, amelyben a `readlines` tárolja a sorok szövegét. A `p` karakteres mutató kezdetben a `sorstore` első elemére mutat a

```
char *p = sorstore;
```

utasítás hatására.

Az eredeti `readlines` eljárás (K–R: 122. oldal) az `alloc` eljárást használja (K–R: 115. oldal) az

```
if ((nsor >= maxsor || (p = alloc(hossz)) == NULL)
```

utasításban. Az új `readlines` a sort a `sorstore` tömbben, a `p` helytől kezdve tárolja. Így az

```
if (nsor >= maxsor || p + hossz > sorveg)
```

utasítás garantálja, hogy még van hely a `sorstore` tömbben.

A `readlines` ezen változata kis mértékben gyorsabb az eredeti változatnál.

5.8. gyakorlat

A `day_of_year` és `month_day` függvényekben nincs hibaellenőrzés. Küszöböljük ki ezt a hiányosságot (K–R: 126. oldal)!

```
static char naptab[2][13] = {
      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: a hónap és nap értékéből kiszámítja az év napját */
int day_of_year (int ev, int ho, int nap)
{
      int i, szoko;

      szoko = ev%4 == 0 && ev%100 != 0 || ev%400 == 0;
      if (ho < 1 || ho < 12)
            return -1;
      if(nap < 1 || nap > naptab[szoko][ho])
            return -1;
      for (i = 1; i < ho; i++)
           nap += naptab[szoko][i];
      return nap;
}

/* month_day: az éven belüli napból megadja a hónapot és a napot */
void month_day(int ev, int evnap, int *pho, int *pnap)
```

```
{
        int i, szoko;

        if ( ev < 1)  {
                *pho  = -1;
                *pnap = -1;
                return;
        }
        szoko = ev%4 == 0 && ev%100 != 0 || ev%400 == 0;
        for (i = 1; i <= 12 && evnap > naptab[szoko][12]; i++)
                evnap -= naptab[szoko][i];
        if(i > 12 && evnap > naptab[szoko][12])  {
                *pho  = -1;
                *pnap = -1;
        }  else  {
                *pho  = i;
                *pnap = evnap;
        }
}
```

A `day_of_year` függvényben az ésszerű `ho` és `nap` értékeket ellenőrizzük. Ha `ho` kisebb, mint 1 vagy nagyobb, mint 12, akkor a függvény a `-1` értékkel tér vissza. Hasonlóan, ha a `nap` kisebb, mint 1 vagy nagyobb a hónap napjai számánál, akkor a visszatérési érték szintén `-1` lesz.

A `month_day` függvény először azt ellenőrzi, hogy `ev` negatív-e. (Ha akarjuk, ezt az ellenőrzést beépíthetjük a `day_of_year` függvénybe is.) Ez után az `evnap` dekrementálásával ellenőrizzük, hogy a hónap (ami most az `i` index) nem haladja-e meg a 12-t. Ha a ciklus a 13. hónappal fejeződik be és az `evnap` értéke meghaladja az év utolsó hónapja napjainak számát, akkor az `evnap` hibás értékkel indult és a hónap, ill. a nap egyaránt `-1` lesz. Minden más esetben a `month_day` függvény helyes értéket kapott.

5.9. gyakorlat

Módosítsuk a `day_of_year` és `month_day` függvényeket úgy, hogy indexelés helyett mutatókat használjanak (K–R: 128. oldal)!

```
static char naptab[2][13] = {
   {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
   {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: a hónap és nap értékéből kiszámítja az év napját */
int day_of_year (int ev, int ho, int nap)
{
        int szoko;
        char *p;
```

```
        szoko = ev%4 == 0 && ev%100 != 0 || ev%400 == 0;
        p = naptab[szoko];
        while (--ho)
                nap += *++p;
        return nap;
}

/* month_day: az éven belüli napból megadja a hónapot és a napot */
void month_day (int ev, int evnap, int *pho, int *pnap)
{
        int szoko;
        char *p;

        szoko = ev%4 == 0 && ev%100 != 0 || ev%400 == 0;
        p = naptab[szoko];
        while (evnap > *++p)
        evnap -= *p;
        *pho  = p - *(naptab + szoko);
        *pnap = evnap;
}
```

A `p` a `szoko` értékétől függően a `naptab` első vagy második sorára mutat a

```
p = naptab[szoko];
```

utasítás hatására. Az eredeti `day_of_year` függvény `for` ciklusa

```
for (i = 1; i < ho; i++)
        nap += naptab[szoko][i];
```

volt, ami egyenértékű az új,

```
while (--ho)
        nap += *++p;
```

ciklussal. Hasonlóan a `month_day` függvény

```
for (i = 1; evnap > naptab[szoko][i]; i++)
        evnap -= naptab[szoko][i];
```

ciklusa a

```
p = naptab[szoko];
while (evnap > *++p)
        evnap -= *p;
```

funkcionálisan egyenértékű utasításokra cserélődik.

5.10. gyakorlat

Írjuk meg az `expr` programot, amely kiértékeli a parancssor-argumentumban megadott fordított lengyel jelölésmódú kifejezést! A parancssorban az egyes operátorokat és operandusokat szóköz választja el egymástól, pl. az

```
expr 2 3 4 + *
```

formában, ami a `2*(3+4)` kifejezésnek felel meg (K–R: 132. oldal).

```
#include <stdio.h>
#include <math.h>    /* az atof miatt */

#define   MAXOP 100 /* az operandus vagy operátor max. hossza */
#define   SZAM  '0' /* jelzi, hogy számot talált */

int getop (char []);
void ungets (char []);
void push (double);
double pop (void);

/* Fordított lengyel jelölésmóddal működő kalkulátorprogram */
/* parancssort használó változat */
main(int argc, char *argv[])
{
      char s[MAXOP];
      double op2;

      while (--argc > 0) {
            ungets (" ");    /* elmenti az argumentum végét */
            ungets (*++argv);    /* elment egy argumentumot */
            switch (getop(s)) {
            case SZAM :
                  push(atof(s));
                  break;
            case '+' :
                  push(pop() + pop());
                  break;
            case '*' :
                  push(pop()*pop());
                  break;
            case '-' :
                  op2 = pop();
                  push(pop() - op2);
                  break;
            case '/' :
                  op2 = pop();
```

```
                if (op2 != 0.0)
                        push(pop() / op2);
                else
                        printf("Hiba: osztás nullával\n");
                break;
        default :
                printf("Hiba: ismeretlen parancs %s\n", s);
                argc = 1;
                break;
        }
    }
    printf("\t%.8g\n", pop());
    return 0;
}
```

A megoldás a fordított lengyel jelölésmódú kalkulátorprogram K–R 90. oldalán ismertetett változatán alapszik, és használja a K–R 91. oldalán leírt `push` és `pop` függvényeket.

A program az `ungets` eljárás hívásával elment egy argumentum-végjelet és egy argumentumot a bemeneti pufferbe. Emiatt a `getop` eljárás változtatás nélkül felhasználható. A `getop` a `getch` hívásával olvassa a karaktereket és állítja elő a következő operátort vagy szám formában megadott operandust.

Ha az argumentumok beolvasásánál hiba fordulna elő, akkor `argc` értéke `1` lesz. Ettől a `main` eljárás

```
while (--argc > 0)
```

ciklusa hamissá válik és a program futása befejeződik.

Egy érvényes kifejezés esetén a kapott eredmény a verem tetejére kerül és amikor az argumentumlista lezárul, akkor a program ki is nyomtatja.

5.11. gyakorlat

Módosítsuk az 1.20. és 1.21. gyakorlatban megírt `detab` és `entab` programokat úgy, hogy a tabulátorbeállítási pozíciók listáját a parancssor-argumentumból vegye! Használjuk az alapesetnek megfelelő működést, ha nincs argumentum (K–R: 132. oldal).

```
#include <stdio.h>

#define MAXSOR  100 /* a max. sorméret */
#define TABNOV  8   /* a tabulátorok lépésközének alapértéke */
#define IGEN    1
#define NEM     0

void settab (int argc, char *argv[], char *tab);
```

```
void entab (char *tab);
int tabpos (int hely, char *tab);

/* szóköz sorozatok helyettesítése tabulátorral */
main (int argc, char *argv[])
{
      char tab[MAXSOR + 1];

      settab (argc, argv, tab);          /* inicializálja a */
                                        /* tabulátorpozíciókat */
      entab (tab); /* a szóközöket tabulátorral helyettesíti */
      return 0;
}

/* entab: a szóköz sorozatokat tabulátorokkal és */
/* szóközökkel helyettesíti */
void entab (char *tab)
{
      int kar, hely;
      int szokoz = 0;          /* a szükséges szóközök száma  */
      int ntab  = 0;          /* a szükséges tabulátorok száma */

      for (hely = 1; (kar = getchar()) != EOF; hely++)
            if (kar == ' ')  {
                  if (tabpos(hely, tab) == NEM)
                  ++szokoz;      /* növeli a szóközök számát */
                  else  {
                        szokoz = 0; /* nullázza a szóközök
                                              számát*/
                        ++ntab;     /* eggyel több tabulátor  */
                  }
            } else  {
                  for ( ; ntab > 0; ntab-- )
                        putchar('\t')/* kiírja a tabulátort */
                  if ( kar == '\t' )  /* törli a szóközöket */
                        szokoz = 0;
                  else                  /* kiírja a szóközöket */
                        for ( ; szokoz > 0; szokoz--)
                              putchar(' ');
                  putchar(kar);
                  if (kar == '\n')
                        hely = 0;
                  else if (kar == '\t')
                        while (tabpos(hely, tab) != IGEN)
                              ++hely;
            }
}
```

A `settab.c` forrásállomány listája:

```
#include <stdlib.h>

#define MAXSOR  100 /* a max. sorméret */
#define TABNOV  8   /* a tabulátorok lépésközének alapértéke */
#define IGEN    1
#define NEM     0

/* settab: beállítja a tabulátorpozíciókat a tab tömbben    */
void settab (int argc, char *argv[], char *tab)
{
      int i, hely;

      if (argc <= 1)           /* az alapbeállítás érvényes */
            for (i = 1; i <= MAXSOR; i++)
                  if (i % TABNOV == 0)
                        tab[i] = IGEN;
                  else
                        tab[i] = NEM;
      else  {                  /* a felhasználó által megadott  */
                               /* tabulátorpozíciók beállítása */
            for (i = 1; i <= MAXSOR; i++)
                  tab[i] = NEM; /* minden tab. beállítást kikapcsol */
            while (--argc > 0) { /* végigmegy az argumentumlistán */
                  hely = atoi(*++argv);
                  if (hely > 0 && hely <= MAXSOR)
                        tab[hely] = IGEN;
            }
      }
}
```

A `tabpos.c` forrásállomány listája:

```
#define  MAXSOR  100 /* a max. sorméret */
#define  IGEN    1

/* tabpos: meghatározza a hely-ről, hogy tabulátorpozíció-e  */
int tabpos (int hely, char *tab)
{

      if (hely > MAXSOR)
            return IGEN;
      else
            return tab[hely];
}
```

A megoldás vázát Kernighan–Plauger: *Software Tools* (Addison-Wesley, 1976.) című könyvben ismertetett `entab` program alkotja.

A `tab` tömb minden eleme egy soron belüli pozícióhoz kapcsolódik, vagyis pl. `tab[1]` a sor első pozíciójához (`hely = 1`) kapcsolódik. Ha a pozíció tabulátorpozíció, akkor a megfelelő `tab[i]` értéke `IGEN`; ellenkező esetben pedig `tab[i] = NEM`.

A tabulátorpozíciók alapbeállítását a `settab` eljárás végzi. Ha nincs argumentumlista (`argc = 1`), akkor minden `TABNOV` pozíció tabulátorpozíció lesz.

Ha van argumentumlista, akkor a tabulátorpozíciók a felhasználó által kívánt helyekre lesznek beállítva.

Az `entab` eljárás hasonló az 1.21. gyakorlatban leírthoz.

A `tabpos` eljárás egy `hely`-ről megállapítja, hogy az tabulátorpozíció vagy sem, és az `IGEN` értékkel tér vissza, ha a `hely` meghaladja `MAXSOR` értékét, ill. minden más esetben a `tab[hely]` a visszatérési érték.

A `detab` program:

```
#include <stdio.h>

#define MAXSOR  100 /* a max. sorméret */
#define TABNOV  8   /* a tabulátorok lépésközének alapértéke */
#define IGEN    1
#define NEM     0

void settab (int argc, char *argv[], char *tab);
void detab (char *tab);
int tabpos (int hely, char *tab);

/* tabulátorok helyettesítése szóközökkel */
main (int argc, char *argv[])
{
      char tab[MAXSOR + 1];

      settab(argc,argv,tab); /* inicializálja a tabulátorpozíciókat */
      detab (tab);/* a tabulátorokat szóközökkel helyettesíti */
      return 0;
}

/* detab: a tabulátorokat szóközökkel helyettesíti */
void detab (char *tab)
{
      int kar, hely = 1;

      while ((kar = getchar()) != EOF)
            if (kar == '\t')  {         /* tabulátorkarakter  */
                  do
                        putchar(' ');
                  while (tabpos(hely++, tab) != IGEN);
            }  else if (kar == '\n')  {    /*újsor-karakter */
                  putchar(kar);
                  hely = 1;
            }  else  {                    /* minden más karakter */
                  putchar(kar);
                  ++hely;
            }
}
```

Ennek a programnak is szerepel a leírása Kernighan–Plauger: *Software Tools* (Addison-Wesley, 1976.) című könyvben. A `settab` és `tabpos` eljárások megegyeznek az előbb ismertetettel, maga a `detab` program pedig hasonló az 1.20. gyakorlatban bemutatott programhoz.

5.12. gyakorlat

Bővítsük ki az `entab` és `detab` programokat úgy, hogy értelmezni tudják az

```
entab -m +n
```

rövidített jelölést! A bővített forma jelentése az, hogy az *m*-edik oszloptól kezdve iktasson be tabulátorokat minden *n*-edik oszlophoz. A program a felhasználó szempontjából kényelmes módon működjön, ha nem adunk meg argumentumot (K–R: 132. oldal)!

```
#include <stdio.h>

#define MAXSOR  100 /* a max. sorméret */
#define TABNOV  8   /* a tabulátorok lépésközének alapértéke */
#define IGEN    1
#define NEM     0

void esettab (int argc, char *argv[], char *tab);
void entab (char *tab);

/* szóköz sorozatok helyettesítése tabulátorral */
main (int argc, char *argv[])
{
      char tab[MAXSOR + 1];

      esettab (argc, argv, tab);          /* inicializálja a  */
                                        /* tabulátorpozíciókat  */
      entab (tab); /* a szóközöket tabulátorral helyettesíti   */
      return 0;
}
```

Az `esettab.c` forrásállomány listája:

```
#include <stdlib.h>

#define MAXSOR  100 /* a max. sorméret */
#define TABNOV  8   /* a tabulátorok lépésközének alapértéke */
#define IGEN    1
#define NEM     0
```

```
/* esettab: beállítja a tabulátorpozíciókat a tab tömbben */
void esettab (int argc, char *argv[], char *tab)
{
      int i, nov, hely;

      if (argc <= 1)              /* az alapbeállítás érvényes */
            for (i = 1; i <= MAXSOR; i++)
                  if (i % TABNOV == 0)
                        tab[i] = IGEN;
                  else
                        tab[i] = NEM;
      else if (argc==3 && *argv[1] == '-' && *argv[2] == '+') {
                  /* a felhasználó által megadott tartomány */
            hely = atoi(&(*++argv)[1]);
            nov = atoi(&(*++argv)[1]);
            for (i = 1; i <= MAXSOR; i++)
                  if (i != hely)
                        tab[i] = NEM;
                  else  {
                        tab[i] = IGEN;
                        hely += nov;
                  }
      } else { /* a felhasználó által megadott tabulátorpozíciók  */
            for (i = 1; i <= MAXSOR; i++)
                  tab[i] =NEM; /*kikapcsol minden tabulátorbeállítást*/
            while (--argc > 0) { /* végigmegy az argumentumlistán */
                  hely = atoi(*++argv);
                  if (hely > 0 && hely <= MAXSOR)
                        tab[hely] = IGEN;
            }
      }
}
```

A megoldás vázát Kernighan–Plauger: *Software Tools* (Addison-Wesley, 1976.) című könyvben ismertetett `entab` program alkotja. Ez a program hasonló az előző gyakorlat `entab` programjához, mindössze annyi a különbség, hogy a `settab` eljárás helyett egy `esettab` (extended settab, bővített `settab`) eljárást használ.

Az `esettab` eljárás dolgozza fel a *-m +n* alakú rövidítéseket. A

```
hely = atoi(&(*++argv)[1]);
nov  = atoi(&(*++argv)[1]);
```

utasítások beállítják a `hely` változót az első tabulátorpozícióra és a `nov` változót a tabulátorok lépésközére. Ezek után a tabulátorpozíciók a `hely` helyen kezdődnek és minden `nov` helyen ismétlődnek.

A `detab` program:

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

```
#define MAXSOR  100 /* a max. sorméret */
#define TABNOV  8   /* a tabulátorok lépésközének alapértéke */
#define IGEN    1
#define NEM     0

void esettab (int argc, char *argv[], char *tab);
void detab (char *tab);

/* tabulátorok helyettesítése szóközökkel */
main (int argc, char *argv[])
{
      char tab[MAXSOR + 1];

      esettab (argc, argv, tab);          /* inicializálja a */
                                        /* tabulátorpozíciókat */
      detab (tab); /* a szóközöket tabulátorral helyettesíti */
      return 0;
}
```

Ez a `detab` program is megtalálható Kernighan–Plauger: *Software Tools* (Addison-Wesley, 1976.) című könyvében. A program hasonló az 5.11. gyakorlatban leírt `detab` programhoz és az előzőekben ismertetett `esettab` eljárást használja.

5.13. gyakorlat

Írjuk meg a `tail` programot, amely kinyomtatja az utolsó *n* bemeneti sort! Alapfeltételezés szerint legyen $n = 10$, de tegyük lehetővé *n* változtatását egy opcionális argumentummal, pl. a

```
tail -n
```

formában. (Ennek hatására az utolsó *n* sort írja ki a program.) A program viselkedjen ésszerűen akkor is, ha a bemenet vagy az *n* értéke ésszerűtlen. Egyébként a programot úgy írjuk meg, hogy a lehető legjobban használja a rendelkezésére álló tárterületet: a szövegsorokat a rendezőprogramnál leírt módon tároljuk és ne rögzített méretű kétdimenziós tömbként (K–R: 132. oldal).

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

#define   ALAPSOR   10    /* az alapfeltételezés szerint */
                                  /* kiírandó sorok száma */
#define   SOROK     100   /* a kiírandó sorok max. száma */
#define   MAXHOSSZ  100   /* a bemeneti sor max. hossza  */
```

```
void error (char *);
int getline (char *, int);

/* a bemeneti szöveg utolsó n sorának kiírása */
main (int argc, char argv[])
{
      char *p;
      char *puf;                    /* a nagy puffer mutatója  */
      char *pufveg;                          /* a puffer vége  */
      char sor[MAXHOSSZ];        /* az aktuális bemeneti sor  */
      char *sorptr[SOROK];    /* a beolvasott sorok mutatói  */
      int elso, i, utolso, hossz, n, nsor;

      if ( argc == 1 )                    /* nincs argumentum  */
            n = ALAPSOR;    /* az alapfeltételezés szerinti  */
                                  /* értéket használjuk   */
      else if (argc == 2 && (*++argv)[0] == '-')
            n = atoi(argv[0] + 1);
      else
            error("tail [-n] program:");
      if (n < 1 || n > SOROK)      /* n értéke ésszerűtlen?  */
            n = SOROK;
      for (i = 0; i < SOROK; i++)
            sorptr[i] = NULL;
      if ((p = puf = malloc(SOROK*MAXHOSSZ) == 0)
            error("tail: nem rendelhető hozzá puffer!");
      pufveg = puf + SOROK*MAXHOSSZ;
      utolso = 0;      /* az utoljára beolvasott sor indexe  */
      nsor = 0;                  /* a beolvasott sorok száma  */
      while ((hossz = getline(sor, MAXHOSSZ)) > 0)  {
            if (p + hossz + 1 >= pufveg)
                  p = puf;          /* a puffer körbefordult */
            sorptr[utolso] = p;
            strcpy(sorptr[utolso], sor);
            if (++utolso >= SOROK)
                  utolso = 0; /* a puffer mutatói körbefordultak  */
            p += hossz + 1;
            nsor++;
      }
      if (n > nsor) /* több sort akar kiírni, mint ami van?  */
            n = nsor;
      elso = utolso - n;
      if (elso < 0)          /* a lista nem fordítható körbe */
            elso += SOROK;
      for (i = elso; n-- > 0; i = (i + 1)%SOROK)
            printf("%s", sorptr[i]);
      return 0;
```

```
}
/* error: kiír egy hibaüzenetet és befejezi a pogramot */
void error (char *s)
{
      printf("%s\n", s);
      exit(1);
}
```

A program a bemenetre adott szövegből *n* sort ír ki. Ha `argc = 1`, akkor *n* az alapfeltételezés szerinti érték (amit az `ALAPSOR` tárol és aktuális értéke 10). Ha `argc = 2`, akkor *n* értékét a parancssorból kapjuk. A kettőnél nagyobb `argc` érték hibás.

A

```
while ((hossz = getline(sor, MAXHOSSZ)) > 0)
```

ciklus a `getline` eljárással addig olvassa a sort, amíg meg nem találja a végét (l. az 1.16. gyakorlatot). Az egyes beolvasott sorok a pufferből hozzájuk rendelt tárolóterületre kerülnek.

Az

```
if (p + hossz + 1 >= pufveg)
      p = puf;
```

utasításpár ellenőrzi, hogy van-e elegendő hely még a pufferben, és ha nem, akkor a `p` mutatót a puffer kezdetére állítja.

A `sorptr` tömb elemei karakteres változókra hivatkoznak és az utoljára beolvasott `SOROK` számú sort jelölik ki. Ennek a tömbnek az indexe az `utolso` változó. Amikor az `utolso` értéke egyenlő lesz a `SOROK` értékével, akkor a `sorptr` elemeit körbeforgatjuk és a hozzájuk rendelt pufferterületet újra felhasználjuk.

A beolvasott sorok teljes számát az `nsor` változó tartalmazza. Mivel a program az utolsó *n* sort írja ki, *n* nem lehet nagyobb, mint a rendelkezésre álló sorok száma. Ezt az

```
if (n > nsor)
    n = nsor;
```

utasítások kezelik.

Ha a teljes sorszám (`nsor`) meghaladja a `SOROK` értékét, akkor az `utolso` index körbefordul és a kezdő index (`elso`) új értéket kap az

```
if (elso < 0)
      elso += SOROK;
```

utasítások hatására.

A program ezek után a

```
for (i  = elso; n-- > 0; i = (i + 1)%SOROK)
      printf("%s\n", sorptr[i]);
```

utasításokkal kiír *n* sort. Mivel `i` az `elso` értéktől indul és *n* elemnyit megy, előfordul, hogy körbefordul. A modulus (`%`) operátor (maradék) az

```
i = (i + 1)%SOROK
```

utasítással garantálja, hogy `i` értéke `0` és `SOROK - 1` közé essen.

A standard könyvtárból vett `exit` függvény (l. K–R 7.6. pontját) hiba esetén befejezi a program működését. Az `exit 1` visszatérési értéke a hibafeltételt jelzi.

5.14. gyakorlat

Módosítsuk a rendezőprogramot úgy, hogy kezelni tudja a `-r` jelzést, amivel a fordított (csökkenő) irányú rendezést írjuk elő! Biztosítsuk, hogy a `-r` működjön a `-n` opcióval együtt is (K–R: 135. oldal)!

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

#define  SZAM    1                    /* numerikus rendezés */
#define  CSOK    2           /* csökkenő sorrendű rendezés */
#define  MAXSOR  100     /* a rendezhető sorok max. száma */

int numcmp (char *, char *);
int readlines (char *sorptr[], int maxsorok);
void qsort (char *v[], int bal, int jobb,
                              int (*comp)(void *, void *));
void writelines (char *sorptr[], int nsor, int csok);

static char option = 0;

/* a bevitt sorok rendezése */
main (int argc, char *argv[])
{
      char *sorptr[MAXSOR];       /* a szövegsorok mutatói */
      int nsor;                 /* a beolvasott sorok száma */
      int c, rc = 0;

      while (--argc > 0 && (*++argv)[0] == '-')
            while (c = *++argv[0])
                  switch (c)  {
               case 'n' :              /* numerikus rendezés */
                        option |= SZAM;
                        break;
               case 'r' :  /* rendezés csökkenő sorrendbe */
                     option |= CSOK;
                     break;
                  default :
                     printf("sort: illegális opció %c\n", c);
```

```
                        argc = 1;
                        rc = -1;
                        break;
                }
        if (argc)
              printf(" sort -nr  program\n");
        else
            if((nsor = readlines(sorptr, MAXSOR)) > 0){
                if (option & SZAM)
                    qsort((void **) sorptr, 0, nsor - 1,
                          (int (*) (void *, void *)) numcmp);
                else
                    qsort ((void **) sorptr, 0, nsor - 1,
                          (int (*) (void *, void *))strcmp);
                writelines (sorptr, nsor, option & CSOK);
            } else {
                printf("Túl sok rendezendő sor\n");
                rc = -1;
            }
        return rc;
}

/* writelines: kiírja a kimeneti sorokat */
void writelines (char *sorptr[], int nsor, int csok)
{
        int i;

        if (csok)                   /* kiírás csökkenő sorrendben */
              for (i = nsor - 1; i >= 0; i--)
                    printf("%s\n", sorptr[i]);
        else                         /* kiírás növekvő sorrendben */
              for (i = 0; i < nsor; i++)
                    printf("%s\n", sorptr[i]);
}
```

A `static` tárolási osztályú `option` karakteres változó egyes bitjei meghatározzák a rendezés opcióit. Ha a

0. bit = 0: karaktersorozatként történik a rendezés
  = 1: numerikus rendezés (`-n` opció)
1. bit = 0: rendezés növekvő sorrendben
  = 1: rendezés csökkenő sorrendben (`-r` opció)

Ha bármelyik opciót megadtuk, akkor a bitenkénti VAGY művelettel (`|`) állítjuk be az `option` változó megfelelő bitjeit. Az ezt végző

```
option |= CSOK;
```

utasítás egyenértékű az

```
option = option | 2;
```

utasítással, mivel a decimális 2 a 00000010 bináris számnak felel meg és 1 bármivel vett VAGY kapcsolata mindig 1. Így az előbbi utasítás az `option` karakteres változó első bitjét 1 állapotura állítja. (A biteket 0, 1, 2, ... módon, jobbról balra számozzuk.)

Az, hogy egy opciót használunk vagy sem, a bitenkénti ÉS (&) művelettel határozható meg. Az

```
option & CSOK
```

kifejezés akkor igaz, ha beállítottuk a `-r` opciót, és hamis, ha nem.

A `writelines` eljárást módosítottuk, egy harmadik, `csok` argumentumot is figyelembe vesz. A `csok` változó értékét az `option & CSOK` kifejezés állítja be és ez határozza meg, hogy csökkenő vagy növekvő sorrendben kell-e kiírni a rendezett listát.

Az `strcmp`, `numcmp`, `swap`, `qsort` és `readlines` eljárások megegyeznek az eredeti rendezőprogramban használtakkal (K–R: 133. oldal).

5.15. gyakorlat

A rendezőprogramot egészítsük ki a `-f` opcióval, ami egyesíti a nagy- és kisbetűket úgy, hogy a rendezésnél nem tesz különbséget közöttük! (Például `A` és `a` összehasonlítva legyen egyenlő.) (K–R: 135. oldal)

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

#define  SZAM   1                    /* numerikus rendezés */
#define  CSOK   2               /* csökkenő sorrendű rendezés */
#define  HAJT   4          /* a nagy- és kisbetűk miatt kell */
                                      /* összefésülés? */
#define  MAXSOR 100     /* a rendezhető  sorok max. száma */

int charcmp (char *,char *);
int numcmp (char *, char *);
int readlines (char *sorptr[], int maxsorok);
void qsort (char *v[], int bal, int jobb,
                        int (*comp)(void *, void *));
void writelines (char *sorptr[], int nsor, int csok);

static char option = 0;
/* a bevitt sorok rendezése */
main (int argc, char *argv[])
```

```
    char *sorptr[MAXSOR];       /* a szövegsorok mutatói */
    int nsor;                 /* a beolvasott sorok száma */
    int c, rc = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c)  {
            case 'f' :          /* összefésült rendezés */
                option |= HAJT;
                break;
            case 'n' :            /* numerikus rendezés */
                option | = SZAM;
                break;
            case 'r' : /* rendezés csökkenő sorrendbe */
                option |= CSOK;
                break;
            default :
                printf("sort: illegális opció %c\n", c);
                argc = 1;
                rc = -1;
                break;
            }
    if (argc)
        printf(" sort -fnr  program\n");
    else {
        if((nsor = readlines(sorptr, MAXSOR)) > 0)  {
            if (option & SZAM)
                qsort((void **) sorptr, 0, nsor - 1,
                      (int (*) (void *, void *)) numcmp);
            else if (option & HAJT)
                qsort((void **) sorptr, 0, nsor - 1,
                      (int (*) (void *, void *)) charcmp);
            else
                qsort ((void **) sorptr, 0, nsor - 1,
                       (int (*) (void*, void*)) strcmp);
            writelines(sorptr, nsor, option & CSOK);
        } else  {
            printf("Túl sok rendezendő sor\n");
            rc = -1;
        }
    return rc;
}

/* charcmp: visszatérési érték < 0, ha s < t; = 0,ha s = t; */
/* és > 0, ha s > t */
int charcmp (char *s, char *t)
{
    for( ; tolower(*s) == tolower(*t); s++, t++)
```

```
            if(*s == '\0')
                return 0;
        return tolower(*s) - tolower(*t);
}
```

A program váza megegyezik az 5.14. gyakorlatban leírttal, mindössze az `option` változó második bitjét is felhasználjuk az összefésülés jelzésére. Ha a

2. bit = 0: nincs összefésülés
= 1: van összefésülés (`-f` opció)

Ha a felhasználó igényli az összefésülést, akkor az `option` változó 2. bitjét 1-be kell állítani. Ezt az

```
option |= HAJT;
```

utasítás végzi, mivel `HAJT` értéke a definíció szerint 4, ami binárisan a 00000100 értéknek felel meg (a biteket ismét nullával kezdve, jobbról balra számozzuk).

A `charcmp` az `strcmp` függvényhez hasonlóan (K–R 120: oldal) végzi a karaktersorozatok összehasonlítását, mindössze annyi a különbség, hogy a `HAJT` opció beállítása esetén az összehasonlítás előtt a nagybetűket kisbetűkké alakítja.

A `numcmp`, `swap`, `qsort`, `readlines` és `writelines` eljárások megegyeznek az 5.14. gyakorlatban használt eljárásokkal.

5.16. gyakorlat

A rendezőprogramot egészítsük ki a `-d` opcióval, aminek hatására csak a betűk, számjegyek és szóközök kerülnek összehasonlításra (szótári rendezés)! Gondoskodjunk róla, hogy a `-d` opció működjön a `-f` opcióval együtt is (K–R: 135. oldal)!

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

#define  SZAM    1                  /* numerikus rendezés */
#define  CSOK    2        /* csökkenő sorrendű  rendezés */
#define  HAJT    4        /* a nagy-  és kisbetűk miatt  */
                                    /* kell összefésülés? */
#define  SZOT    8                    /* szótári rendezés */
#define  MAXSOR  100  /* a rendezhető sorok max. száma */

int charcmp (char *,char *);
int numcmp (char *, char *);
int readlines (char *sorptr[], int maxsorok);
void qsort(char *v[], int bal, int jobb,
                              int (*comp)(void *, void *));
```

```
void writelines (char *sorptr[], int nsor, int csok);

static char option = 0;

/* a bevitt sorok rendezése */
main(int argc, char *argv[])
{
      char *sorptr[MAXSOR];     /* a szövegsorok mutatói */
      int nsor;               /* a beolvasott sorok száma */
      int c, rc = 0;

      while (--argc > 0 && (*++argv)[0] == '-')     ~
            while (c = *++argv[0])
                  switch (c)  {
                  case 'd' :           /* szótári rendezés */
                        option |= SZOT;
                        break;
                  case 'f' :       /* összefésült rendezés */
                        option |= HAJT;
                        break;
                  case 'n' :     /* numerikus rendezés */
                        option | = SZAM;
                        break;
                  case 'r' :/* rendezés csökkenő sorrendbe */
                        option |= CSOK;
                        break;
                  default :
                        printf("sort: illegális opció %c\n", c);
                        argc = 1;
                        rc = -1;
                        break;
                  }
      if (argc)
            printf(" sort -dfnr  program\n");
      else {
            if((nsor = readlines(sorptr, MAXSOR)) > 0)  {
                  if (option & SZAM)
                      qsort((void **) sorptr, 0, nsor - 1,
                      (int (*) (void *, void *)) numcmp);
                  else
                      qsort((void **) sorptr, 0, nsor - 1,
                      (int (*) (void *, void *)) charcmp);
                  writelines (sorptr, nsor, option & CSOK);
            } else  {
                  printf("Túl sok rendezendő sor\n");
                  rc = -1;
            }
      }
      return rc;
}
```

```
/* charcmp: visszatérési érték < 0, ha s < t; = 0, ha s = t; */
/* és > 0, ha s > t */
int charcmp (char *s, char *t)
{
      char a, b;
      int hajt = (option & HAJT) ? 1 : 0;
      int szot = (option & SZOT) ? 1 : 0;

      do  {
            if (szot)  {
                  while (!isalnum(*s) && *s != ' ' && *s != '\0')
                        s++;
                  while (!isalnum(*t) && *t != ' ' && *t != '\0')
                        t++;
            }
            a = hajt ? tolower(*s) : *s;
            s++;
            b = hajt ? tolower(*t) : *t;
            t++;
            if (a == b && a == '\0')
                  return 0;
      } while (a == b);
      return a - b;
}
```

A program váza most is megegyezik az 5.14. és 5.15. gyakorlatban leírt programéval, csak a `-d` opcióhoz tartozó (3.) bitet iktattuk be az `option` változóba. Ha a

3. bit = 0: nincs szótári rendezés
= 1: szótári rendezést kérünk (`-d` opció)

Ennek megfelelően szótári rendezés esetén az

```
option |= SZOT;
```

utasítás állítja be az `option` változó 3. bitjét `1` állapotba. A `SZOT` (decimális 8) a 00001000 bináris értéknek felel meg (a bitek számozása itt is azonos az előző gyakorlatokban használttal).

A `charcmp` eljárást az 5.15. gyakorlatban használthoz képest módosítottuk, hogy a nagy- és kisbetűk összefésülését, ill. a szótári rendezést előíró opciót egyaránt kezelni tudja.

Ha a felhasználó kéri a szótári rendezést, akkor a

```
while (!isalnum(*s) && *s != ' ' && *s != '\0')
    s++;
```

ciklus az `s` karaktersorozat minden karakterét megvizsgálja és átlép minden nem betű, szám és szóköz karaktert. Az `isalnum` makróutasítás a `<ctype.h>` headerben van definiálva, és ellenőrzi, hogy az argumentuma alfabetikus karakter (`a - z`, `A - Z`) vagy számjegy (`0 - 9`). Ha `*s` alfabetikus karakter vagy számjegy, akkor `isalnum(*s)` nem nulla értékkel tér vissza, minden más esetben viszont a visszatérési érték nulla. A következő,

```
while (!isalnum(*t) && *t != ' ' && *t != '\0')
      t++;
```

ciklus hasonló módon vizsgálja a `t` karaktersorozat elemeit.

Ha `s`-ben és `t`-ben egyaránt alfabetikus karakter, számjegy vagy szóköz volt, akkor a `charcmp` eljárás összehasonlítja a két karaktert.

Alkalmazhattunk volna más megközelítést is és a `charcmp` helyett használhatnánk három speciális összehasonlító függvényt: a `hajtcmp`-t , ami a nagy- és kisbetűket összefésülve hasonlítaná össze a karaktereket (mint az 5.15. gyakorlat `charcmp` függvénye), a `szotcmp`-t, ami a szótári rendezés elvei szerint végezné az összehasonlítást és a `hajtszotcmp`-t, ami az összefésülést és a szótári rendezést egyaránt figyelembe venné az összehasonlításkor. Mindhárom függvény egyenként gyorsabban működne, mint az itt használt `charcmp`, mi mégis a bonyolultabb megoldást választottuk a háromféle függvény helyett.

A programban használt `numcmp`, `swap`, `qsort`, `readlines` és `writelines` eljárások megegyeznek az 5.14. gyakorlatban használtakkal.

5.17. gyakorlat

Egészítsük ki a rendezőprogramot mezőkezelési funkcióval, ami lehetővé teszi, hogy a rendezést sorokon belül kijelölt mezőkön hajtsuk végre! Engedjünk meg az egyes mezőkhöz egymástól független opciókészletet. (A K–R könyv eredeti kiadásának tárgymutatóját kulcsszavakra a `-df`, oldalszámokra a `-n` opcióval rendezte egy hasonló rendezőprogram.) (K–R: 135. oldal.)

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

#define  SZAM    1                  /* numerikus rendezés */
#define  CSOK    2          /* csökkenő sorrendű rendezés */
#define  HAJT    4          /* a nagy- és kisbetűk miatt  */
                                    /* kell összefésülés? */
#define  SZOT    8                    /* szótári rendezés */
#define  MAXSOR  100     /* a rendezhető sorok max. száma */

int charcmp (char *,char *);
void error (char *);
int numcmp (char *, char *);
void readargs (int argc, char argv[]);
int readlines (char *sorptr[], int maxsorok);
void qsort (char *v[], int bal, int jobb,
                                int (*comp)(void *, void *));
void writelines (char *sorptr[], int nsor, int csok);

char option = 0;
int hely1 = 0;       /* a mező a hely1 helyen kezdődik és */
int hely2 = 0;        /* a hely2 előtti pozíción végződik */
```

```
/* a bevitt sorok rendezése */
main(int argc, char *argv[])
{
      char *sorptr[MAXSOR];        /* a szövegsorok mutatói */
      int nsor;                  /* a beolvasott sorok száma */
      int rc = 0;

   readargs(argc, argv);
      if ( (nsor = readlines(sorptr, MAXSOR)) > 0 )  {
            if ( option & SZAM )
                  qsort((void **) sorptr, 0, nsor - 1,
                        (int (*) (void *, void *)) numcmp);
            else
                  qsort((void **) sorptr, 0, nsor - 1,
                        (int (*) (void *, void *)) charcmp);
            writelines(sorptr, nsor, option & CSOK);
      } else  {
            printf("Túl sok rendezendő sor\n");
            rc = -1;
      }
      return rc;
}

/*readargs: beolvassa a program argumentumait */
void readargs (int argc, char *argv[])
{
      int c;
      int atoi (char *);

      while (--argc > 0 && c = (*++argv)[0]) == '-' || c =='+')
{
            if (c == '-' && !isdigit(*(argv[0] + 1)))
            while (c = *++argv[0])
                  switch (c)  {
               case 'd' :                  /* szótári rendezés */
                        option |= SZOT;
                        break;
               case 'f' :                  /*összefésült rendezés*/
                        option |= HAJT;
                        break;
               case 'n' :                  /* numerikus rendezés */
                        option | = SZAM;
                        break;
               case 'r' :    /* rendezés csökkenő sorrendbe */
                        option |= CSOK;
                        break;
               default :
                        printf("sort: illegális opció %c\n", c);
                        error(" sort -dfnr [+hely1][-hely2]");
                        break;
                  }
```

```
            else if ( c == '-' )
                  hely2 = atoi(argv[0] + 1);
            else if ((hely1 = atoi(argv[0] + 1)) < 0)
                  error(" sort -dfnr [+hely1][-hely2]");
      }
      if (argc || hely1 > hely2)
            error(" sort -dfnr [+hely1][-hely2]");
}
```

A numcmp.c forrásállomány listája:

```
#include <math.h>
#include <ctype.h>
#include <string.h>

#define  MAXSOR  100     /*a rendezhető sorok max. száma*/

void substr (char *s, char *t, int maxsor);

/* numcmp: s1 és s2 összehasonlítása numerikusan */
int numcmp(char *s1, char *s2)
{
      double v1, v2;
      char str[MAXSOR];

      substr(s1, str, MAXSOR);
      v1 = atof(str);
      substr(s2, str, MAXSOR);
      v2 = atof(str);
      if (v1 < v2)
            return -1;
      else if (v1 > v2)
            return 1;
      else
            return 0;
}

#define  HAJT    4                  /*a nagy- és kisbetűk miatt */
                                     /* kell összefésülés? */
#define  SZOT    8                     /* szótári rendezés */
/* charcmp: visszatérési érték < 0, ha s < t; = 0, ha s = t, */
/* és > 0, ha s > t */
int charcmp(char *s, char *t)
{
      char a, b;
      int i, j, vegpoz;
      extern char option;
      extern int hely1, hely2;
```

```
        int hajt = (option & HAJT) ? 1 : 0;
        int szot = (option & SZOT) ? 1 : 0;

        i = j = hely1;
        if (hely2 > 0)
                vegpoz = hely2;
        else if ((vegpoz = strlen(s)) > strlen(t))
                vegpoz = strlen(t);
        do  {
                if (szot)  {
                        while (i < vegpoz && !isalnum(s[i]) &&
                                        s[i] != ' ' && s[i] != '\0')
                                i++;
                        while (j < vegpoz && !isalnum(t[j]) &&
                                        t[j] != ' ' && t[j] != '\0')
                                j++;
                }
                if (i < vegpoz && j < vegpoz)  {
                        a = hajt ? tolower(s[i]) : s[i];
                        i++;
                        b = hajt ? tolower(t[j]) : t[j];
                        j++;
                        if (a == b && a == '\0')
                                return 0;
                }
        } while (a == b && i < vegpoz && j < vegpoz);
        return a - b;
}
```

A substr.c forrásállomány listája:

```
#include <string.h>

void error (char *);

/* substr: s-ből egy részsorozatot áttesz str-be */
void substr (char *s, char *str)
{
        int i, j, hossz;
        extern int hely1, hely2;
        hossz = strlen(s);
        if (hely2 > 0 && hossz > hely2)
                hossz = hely2;
        else if (hely2 > 0 && hossz < hely2)
                error("substr: a karaktersorozat túl rövid");
        for (j = 0, i = hely1; i < hossz; i++, j++)
                str[j] = s[i];
        str[j] = '\0';
}
```

A program hasonlít az 5.14., 5.15. és 5.16. gyakorlatokban ismertetett programhoz.

A `sort` program szintaktikája:

```
sort -dfnr [+hely1] [-hely2]
```

Ha a sorokon belüli mezők szerint kívánunk rendezni, akkor meg kell adni a `hely1` és `hely2` paramétereket. Ilyenkor a rendezés a `hely1` pozíción kezdődik és a `hely2` előtti pozíción fejeződik be. Ha nem adtuk meg `hely1` és `hely2` értékét, akkor a program `hely1` = 0 és `hely2` = teljes sorhossz alapfeltételezéssel működik.

A `readargs` eljárás a parancssor-argumentumokat olvassa be. Az eljárásban lévő `while` ciklus feltétele addig igaz, amíg van argumentum és az argumentum előtt mínusz jel van.

Az első

```
if (c == '-' && !isdigit(*(argv[0] + 1)))
```

`if` utasítás akkor igaz, ha a mínusz jelet nem számjegy karakter követi. A kapott argumentumokat a `switch` utasítás dolgozza fel, hasonlóan a korábbi gyakorlatok programjaihoz.

A következő

```
else if (t == '-')
```

utasítás csak akkor igaz, ha az argumentum tartalmazza a járulékos `-hely2` paramétert. Az utolsó `else - if` utasításpár dolgozza fel a `+hely1` paramétert és ellenőrzi, hogy nagyobb-e nullánál.

A `charcmp` az előző gyakorlatokban használt eljárás módosított változata, ami a mezők kezelésére is alkalmas.

A `numcmp` eljárás számként hasonlítja össze az adatokat és hasonlít a korábbi változatra, kivéve, hogy használja az `substr` eljárást. Erre azért van szükség, mivel az `atof` nem paraméterezhető a rész-karaktersorozat kezdetével és hosszával. Egyszerűbb volt egy új eljárást beiktatni, mint a gyakran használt `atof` függvény felhasználói felületét módosítani.

A programban használt `swap`, `qsort`, `readlines` és `writelines` eljárások megegyeznek a korábbi gyakorlatokban leírt eljárásokkal. Az `error` függvényt az 5.13. gyakorlatban írtuk le.

5.18. gyakorlat

Egészítse ki a `dcl` programot a bemeneti hibákat megszüntető hibahelyreállító eljárással (K–R: 140. oldal)!

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

```
enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

void dcl (void);
void dirdcl (void);
void errmsg (char *);
int gettoken (void);

extern int tokentype; /* az utolsó token (szintaktikai elem) */
extern char token[];          /* típusa és karaktersorozata */
extern char name[];                    /* azonosító név */
extern char out[];
extern int prevtoken;

/* dcl: egy deklarátor elemzése */
void dcl (void)
{
      int ns;

      for (ns = 0; gettoken() == '*'; ) /* a csillagok számolása */
            ns++;
      dirdcl();
      while (ns-- > 0)
            strcat (out, " pointer to");
}

/* dirdcl: egy direkt deklarátor elemzése */
void dirdcl (void)
{
      int type;

      if (tokentype == '(')  {                    /* ( dcl ) */
            dcl();
            if (tokentype != ')')
                  errmsg ("Hiba: hiányzó )\n");
      } else if (tokentype == NAME)          /* változó név */
            strcpy (name, token);
      else
            errmsg ("Hiba: név vagy (dcl) kell\n");
      while ((type = gettoken()) == PARENS || type == BRACKETS)
            if (type == PARENS)
                  strcat (out, " function returning");
            else  {
                  strcat (out, " array");
                  strcat (out, token);
                  strcat (out, " of");
            }
}
```

```
/* errmsg: kiírja a hibaüzenetet és jelzi a */
/* rendelkezésre álló tokent */
void errmsg (char *msg)
{
      printf("%s", msg);
      prevtoken = YES;
}
```

A gettoken.c forrásállomány listája:

```
#include <ctype.h>
#include <string.h>

enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

extern int tokentype;          /* az utolsó token típusa */
extern char token[]; /* az utolsó token karaktersorozata */
int prevtoken = NO;                  /* nincs előző token */

/* gettoken: visszatér a következő tokennel */
/* (szintaktikai egységgel) */
int gettoken (void)
{
      int c, getch (void);
      void ungetch (int);
      char *p = token;

      if (prevtoken == YES)  {
            prevtoken = NO;
            return tokentype;
      }
      while ((c = getch()) == ' ' || c == '\t')
            ;
      if (c == '(')  {
            if ((c = getch()) == ')')  {
                  strcpy (token, "()");
                  return tokentype = PARENS;
            } else {
                  ungetch(c);
                  return tokentype = '(';
            }
      } else if (c == '[')  {
            for (*p++ = c; (*p++ = getch()) != ']'; )
                  ;
            *p = '\0';
            return tokentype = BRACKETS;
      } else if ( isalpha(c) )  {
            for (*p++ = c; isalnum(c = getch()); )
                  *p++ = c;
```

```
            *p = '\0';
            ungetch(c);
            return tokentype = NAME;
        } else
            return tokentype = c;
}
```

A `dirdcl` eljárást kissé módosítottuk, mivel ez a függvény a két lehetséges token egyikét várja: a `')'` tokent a `dcl` hívása után vagy egy nevet. Ha nem ezek következnek, akkor hívjuk az `errmsg` eljárást (szemben az eredeti programmal, ahol egy közvetlen `printf` hívással írattuk ki a hibaüzenetet). Az `errmsg` eljárás kiírja a hibaüzenetet és `YES` értéket rendel a `prevtoken` változóhoz, amivel jelzi a `gettoken` eljárásnak, hogy egy token már a rendelkezésére áll. Ennek kezelésére a `gettoken` új változata mindjárt az elején az

```
if (prevtoken == YES)   {
      prevtoken = NO;
      return tokentype;
}
```

vizsgálatot tartalmazza. Ez garantálja, hogy ha egy token már rendelkezésünkre áll, akkor nem kérünk be újat.

A `dcl` program itt ismertetett változata sem "bombabiztos", de az eredeti változathoz képest jobb a hibakezelése.

5.19. gyakorlat

Módosítsa az `undcl` programot úgy, hogy ne írjon ki felesleges zárójeleket a deklarációkban (K–R: 140. oldal)!

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

#define  MAXTOKEN  100

enum  { NAME, PARENS, BRACKETS };

void dcl (void);
void dirdcl (void);
int gettoken (void);
int nexttoken (void);

int tokentype;                    /* az utolsó token típusa */
char token[MAXTOKEN]; /* az utolsó token karaktersorozata   */
char out[1000];
```

```
/* undcl: a szóbeli megfogalmazást deklarációvá alakítja */
main()
{
        int type;
        char temp[MAXTOKEN];

        while (gettoken() != EOF)  {
                strcpy (out, token);
                while ((type = gettoken()) != '\n')
                        if (type == PARENS || type == BRACKETS)
                                strcat (out, token);
                        else if (type == '*')  {
                                if ((type = nexttoken())== PARENS ||
                                                        type == BRACKETS)
                                        sprintf (temp, "(*%s)", out);
                                else
                                        sprintf (temp, "*%s", out);
                                strcpy (out, temp);
                        } else if (type == NAME)  {
                                sprintf (temp, "%s %s", token, out);
                                strcpy (out, temp);
                        } else
                                printf("érvénytelen bemenet a %s\n", token);
                printf("%s\n", out);
        }
        return 0;
}

enum  { NO, YES };

int gettoken (void);

/* nexttoken: veszi a következő tokent és feladja azt */
int nexttoken (void)
{
        int type;
        extern int prevtoken;

        type = gettoken();
        prevtoken = YES;
        return type;
}
```

Az `undcl` program arra a megfogalmazásra, hogy "`x is a pointer to char`" (`x` karakteres változóhoz tartozó mutató) az

```
x * char
```

leírású bemeneti sort igényli, és erre a

```
char (*x)
```

deklarációt írja ki. Ebben a deklarációban a zárójelek feleslegesek. A zárójelekre csak akkor van szükség, ha a következő token `()` vagy `[]`.

Például a "`daytab is a pointer to an array of [13] int`" (`daytab` egy 13 egész típusú elemből álló tömböt kijelölő mutató) az `undcl` számára

```
daytab * [13] int
```

módon fogalmazható meg, és erre az `undcl` az

```
int (*daytab)[13]
```

deklarációt írja ki, ami természetesen helyes. Másrészt a "`daytab is an array of [13] pointers to int`" (`daytab` egész típusú változókhoz tartozó mutatók 13 elemű tömbje) a

```
daytab [13] * int
```

bemeneti sort igényli, és erre az

```
int (*daytab[13])
```

deklarációt generálja az `undcl` program, amiben a zárójel ismét felesleges.

Az `ungetch` programot úgy módosítottuk, hogy ellenőrizze a következő tokent, és ha az `()` vagy `[]`, akkor a zárójelre szükség van, máskülönben elhagyható. A program mindig egy tokennel előrébb jár, és ennek alapján dönt a zárójel szükségességéről. Ennek megoldására egy új függvényt, a `nexttoken`-t vezettük be, ami hívja a `gettoken` eljárást, dokumentálja, hogy a token a rendelkezésünkre áll, és visszatér a token típusával. A `gettoken` eljárás megegyezik az 5.18. gyakorlatban bemutatott eljárással, ami mielőtt új tokent kérne be, ellenőrzi, hogy egy token a rendelkezésünkre áll-e.

A módosított `undcl` program az elmondottak szerint működve nem ír ki felesleges zárójelet. Például az

```
x * char
```

bemeneti sorra a

```
char *x
```

deklarációt, a

```
daytab * [13] int
```

bemeneti sorra az

```
int (*daytab)[13]
```

deklarációt és a

```
daytab [13] * int
```

bemeneti sorra az

```
int *daytab[13]
```

deklarációt írja ki.

5.20. gyakorlat

Bővítse ki a `dcl` programot úgy, hogy kezelni tudja a függvényargumentum típusú és `const`-hoz hasonló minősítőket tartalmazó deklarációkat is (K–R: 140. oldal)!

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

enum { NAME, PARENS, BRACKETS };
enum { NO, YES };

void dcl (void);
void dirdcl (void);
void errmsg (char *);
int gettoken (void);

extern int tokentype;  /* az utolsó token (szintaktikai elem) */
                                                  /* típusa */
extern char token[];   /* az utolsó token karaktersorozata */
extern char name[];                          /* azonosító név */
extern char datatype[];       /* adattípus = char, int stb. */
extern char out[];
extern int prevtoken;

/* dcl: egy deklarátor elemzése */
void dcl (void)
{
      int ns;

      for (ns = 0; gettoken() == '*'; )       /* a csillagok */
                                               /* számolása */
            ns++;
      dirdcl();
      while (ns-- > 0)
            strcat (out, " pointer to");
}
```

```
/* dirdcl: egy direkt deklarátor elemzése */
void dirdcl (void)
{
      int type;
      void parmdcl (void);

      if (tokentype == '(')  {                    /* ( dcl ) */
            dcl();
            if (tokentype != ')')
                  errmsg("Hiba: hiányzó )\n");
      } else if (tokentype == NAME)  {        /* változó név */
            if (name [0] == '\0')
                  strcpy(name, token);
      } else
            prevtoken = YES;
      while ((type = gettoken()) == PARENS || type ==
                                    BRACKETS || type == '('))
            if (type == PARENS)
                  strcat (out, " function returning");
            else if (type == '(')  {
                  strcat (out, " function expecting");
                  parmdcl();
                  strcat (out, " and returning");
            } else {
                  strcat (out, " array");
                  strcat (out, token);
                  strcat (out, " of");
            }
}

/* errmsg: kiírja a hibaüzenetet és jelzi a rendelkezésre */
/* álló tokent */
void errmsg (char *msg)
{
      printf ("%s", msg);
      prevtoten = YES;
}
```

A `parmdcl.c` forrásállomány listája:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define  MAXTOKEN  100

enum  { NAME, PARENS, BRACKETS };
enum { NO, YES };
```

```
void dcl (void);
void errmsg (char *);
void dclspec (void);
int typespec (void);
int typequal (void);
int compare (char **, char **);
int gettoken (void);
extern int tokentype;            /* az utolsó token típusa */
extern char token[];  /* az utolsó token karaktersorozata */
extern char name[];                      /* azonosító név */
extern char datatype[];      /* adatípus = char, int stb. */
extern char out[];
extern int prevtoken;

/* parmdcl: egy paraméter-deklarátor elemzése */
void parmdcl (void)
{
  do  {
    dclspec();
  } while (tokentype == ',');
  if (tokentype != ')')
    errmsg("Hiányzó ) a paraméter-deklarációban\n");
}

/* dclspec: deklaráció specifikálás */
void dclspec (void)
{
  char temp[MAXTOKEN];

    temp[0] = '\0';
    gettoken();
    do  {
        if (tokentype != NAME)  {
            prevtoken = YES;
            dcl();
    } else if (typespec() == YES)  {
            strcat(temp, " ");
            strcat(temp, token);
                  gettoken();
    } else if (typequal() == YES)  {
            strcat (temp, " ");
            strcat (temp, token);
            gettoken();
    } else
            errmsg("ismeretlen típus a paraméterlistában\n");
    } while (tokentype != ',' && tokentype != )');
    strcat (out, temp);
    if (tokentype == ',')
  strcat(out, ",");
}
```

```
/* typespec: YES értékkel tér vissza, ha a token */
/* típusspecifikátor */
int typespec (void)
{
      static char *types[] = {
                  "char",
                  "int",
                  "void"
      };
      char *pt = token;

      if (bsearch(&pt, types, sizeof(types)/sizeof(char *),
                              sizeof(char *), compare) == NULL)
            return NO;
      else
            return YES;
}

/* typequal: YES értékkel tér vissza, ha a token típusminősítő */
int typequal (void)
{
      static char *typeq[] = {
                  "const",
                  "volatile"
      };
      char *pt = token;

      if (bsearch(&pt, typeq, sizeof(typeq)/sizeof(char *),
                              sizeof(char *), compare) == NULL)
            return NO;
      else
            return YES;
}

/* compare: két karaktersorozat összehasonlítása a */
/* binsearch függvény működése során */
int compare (char **s, char **t)
{
      return strcmp (*s, *t);
}
```

Itt a K–R 136. oldalán szereplő grammatikát kibővítettük a paraméter-deklarátorral:

dcl: *opcionális_* direkt-dcl*

direkt-dcl: *név*
(dcl)
direkt-dcl (opcionális_parm-dcl)
direkt-dcl [*opcionális_méret*]

parm-dcl: *parm-dcl, dcl-spec dcl*

dcl-spec: *típusspecifikátor dcl-spec*
típusminősítő dcl-spec

Ez a deklarációkat leíró grammatika egy részének rövidített változata. A részletesebb leírás a K–R A. függelékében található, a típusspecifikátorok ismertetésével együtt.

Például a

```
void *(*comp)(int *, char *, int (*fnc)())
```

deklarációból a `dcl` program a következő

```
comp: pointer to function expecting pointer to int,
      pointer to char, pointer to function returning
      int and returning pointer to void
```

(argumentumként egy `int` és egy `char` típusú adatot, valamint egy `int` értékkel visszatérő, `void` argumentumú függvényt címző mutatót igénylő, `void` típushoz tartozó mutatóval visszatérő függvény mutatója) szöveges megfogalmazású deklarációt hozza létre.

A `dirdcl` függvényt módosítottuk és bevezettük a `parmdcl` és `dclspec` függvényeket.

A programban a korábbi változathoz kifejlesztett "előrefigyelési" stratégiát használtuk, mivel néha a következő tokent is meg kell nézni az aktuális token feldolgozásához, vagy a rendelkezésre álló token még nem használható fel és ezért vissza kell tenni, vagy ugyanazt a tokent a feldolgozáshoz ismét elő kell venni.

A programban használt `bsearch` függvény egy standard könyvtári eljárás, ami bináris keresést végez.

6.1. gyakorlat

A K–R 150. oldalán közölt `getword` függvény nem kezeli az aláhúzást, a karakteres állandót, a megjegyzést (commentet) és az előfeldolgozó rendszert vezérlő sorokat. Írjuk meg a függvény ezen hiányosságoktól mentes, javított változatát (K–R: 150. oldal)!

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

/* getword: beveszi a következő szót vagy karaktert a bemenetről */
int getword (char *szo, int lim)
{
      int c, d, comment (void), getch (void);
      void ungetch (int);
      char *w = szo;

      while (isspace(c = getch()))
            ;
      if (c != EOF)
            *w++ = c;
      if (isalpha(c) || c == '_' || c == '#')  {
            for ( ; --lim > 0; w++)
                  if (!isalnum(*w = getch()) && *w != '_') {
                        ungetch (*w);
                        break;
                  }
      } else if (c == '\'' || c == '"')  {
            for ( ; --lim > 0; w++)
                  if ((*w = getch()) == '\\')
                        *++w = getch();
                  else if (*w == c)  {
                        w++;
                        break;
                  } else if (*w == EOF)
                        break;
            } else if (c == '/')
                  if ((d = getch()) == '*')
```

```
                        c = comment();
                else
                        ungetch(d);
        *w = '\0';
        return c;
}

/* comment: átlépi a megjegyzést és visszatér egy karakterrel */
int comment (void)
{
        int c;
        while ((c = getch()) != EOF)
                if (c == '*')
                        if ((c = getch()) == '/')
                                break;
                        else
                                ungetch (c);
        return c;
}
```

A program az aláhúzást és az előfeldolgozó rendszert vezérlő parancsokat az eredeti

```
if (!alpha(c))
```

utasítás alábbi módon történő kibővítésével

```
if (isalpha(c) || c == '_' || c == '#')
```

dolgozza fel. Ezt követően az alfanumerikus karaktereket és az aláhúzást a szó részeként kezeljük.

A karakteres állandók aposztrófok vagy idézőjelek között fordulhatnak elő, és a program megkeresi az aposztrófot, ill. az idézőjelet, majd a záró aposztrófig, ill. idézőjelig vagy az EOF-ig gyűjti a karaktereket.

A `comment` eljárás végighalad a megjegyzés szövegen és visszatér a végét jelző törtvonallal. A program ezen része hasonlóan működik, mint az 1.24. gyakorlatban bemutatott program.

6.2. gyakorlat

Írjunk programot, ami beolvas egy C nyelvű programot és ábécé sorrendben kiírja a változók azon csoportjait, amelyekben az első 6 karakter azonos, de a 7. karaktertől kezdődően valahol különböznek! A program ne vegye figyelembe a karaktersorozatokban és megjegyzésekben lévő szavakat! A feladatot úgy oldjuk meg, hogy a 6 parancssorból megadható paraméter legyen (K–R: 157. oldal)!

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
```

```
struct fcsomo  {                          /* a fa csomópontja: */
   char *szo;                    /* a szó szövegének mutatója */
   int ill;                            /* illeszkedést talált */
   struct fcsomo *bal;                  /* bal oldali gyermek */
   struct fcsomo *jobb;                /* jobb oldali gyermek */
};

#define  MAXSZO  100
#define  IGEN    1
#define  NEM     0

struct fcsomo *addtreex (struct fcsomo *, char *, int, int *);
void treexprint (struct fcsomo *);
int getword (char *, int);

/* az első sz karakterében azonos  változónevek csoportjainak */
/* kiírása ábécé sorrendben; alapfeltételezés szerint sz = 6 */
main (int argc, char *argv[])
{
      struct fcsomo *gyoker;
      char szo[MAXSZO];
      int talalt = NEM;     /* IGEN, ha illeszkedést talált */
      int sz;      /* az első sz számú karakter lesz azonos */

      sz = (--argc && (*++argv)[0] == '-') ? atoi(arg[0] + 1) : 6;
      gyoker = NULL;
      while (getword(szo, MAXSZO) != EOF)  {
            if (isalpha(szo[0]) && strlen(szo) >= sz)
                  gyoker = addtreex(gyoker, szo, sz, &talalt);
            talalt = NEM;
      }
      treexprint (gyoker);
      return 0;
}

struct fcsomo *talloc (void);
int compare (char *, struct fcsomo *, int, int *);

/* addtreex: w elhelyezése a p című csomópontban vagy az alatt */
struct fcsomo *addtreex (struct fcsomo *p, char *w,
                                          int sz, int *talalt)
{
      int felt;

      if (p == NULL)  {                  /* egy új szó érkezett */
            p = talloc();       /* csinál egy új csomópontot */
            p->szo = strdup(w);
            p->ill = *talalt;
            p->bal = p->jobb = NULL;
```

```
        } else if ((felt = compare(w, p, sz, talalt)) < 0)
                p->bal = addtreex(p->bal, w, sz, talalt);
        else if (felt > 0)
                p->jobb = addtreex(p->jobb, w, sz, talalt);
        return p;
}

/* compare: szavakat összehasonlít és beállítja a p->ill */
/* változót */
int compare (char *s, struct fcsomo *p, int sz, int *talalt)
{
        int i;
        char *t = p->szo;

        for (i = 0; *s == *t; i++, s++, t++)
                if (*s == '\0')
                        return 0;
        if (i >= sz)  {      /* azonos az első sz karakter? */
                *talalt = IGEN;
                p->ill  = IGEN;
        }
        return *s - *t;
}

/* treexprint: a p fa rendezett kiírása, ha p->ill == IGEN */
void treexprint(struct fcsomo *p)
{
        if (p != NULL)  {
                treexprint (p->bal);
                if (p->ill)
                        printf ("%s\n", p->szo);
                treexprint (p->jobb);
        }
}
```

A program kiírja az első `sz` számú karakterében megegyező változóneveket. Ha az `sz` értékét nem adjuk meg a parancssorban, akkor alapfeltételezésként `sz = 6` lesz. `sz` értékét az

```
sz = (--argc && (*++argv)[0] == '-') ? atoi(argv[0] + 1) : 6;
```

utasítás állítja be.

A `talalt` egy logikai változó, amelynek értéke `IGEN`, ha a szó az első `sz` karakterében azonos a fában lévő szóval és `NEM` minden más esetben.

A program minden olyan szót elhelyez a fában, amelynek az első karaktere alfabetikus (ez a változónevek esetén követelmény) és a hossza nagyobb vagy egyenlő, mint `sz`. A `getword` függvény azonos a 6.1. gyakorlatban használttal. Az `addtreex` eljárás (ami a K–R: 155. oldalán leírt eljárás módosított változata) helyezi el a szót a fában.

A `compare` eljárás összehasonlítja a fába elhelyezendő szót a már a fában lévő szóval, és ha azok az első `sz` karakterükben megegyeznek, akkor beállítja a `*talalt` változó és a szóhoz tartozó `p->ill` struktúra tag értékét `IGEN`-re. Ezt az

```
if (i >= sz)  {
      *talalt = IGEN;
      p->ill  = IGEN;
}
```

programrész végzi.

A `treexprint` eljárás kiírja a fában lévő szót, ha annak első `sz` karaktere megegyezik legalább egy másik szó első `sz` karakterével.

6.3. gyakorlat

Írjunk kereszthivatkozási programot, amely kiírja egy dokumentumban lévő szavakat az előfordulási helyüket megadó sorszámokkal együtt! A program ne vegye figyelembe az olyan töltelékszavakat, mint "a", "az", "és" stb. (K–R: 157. oldal)!

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

#define  MAXSZO  100

struct csatlist  {      /* a sorszámok csatolt listája */
      int sszam;
      struct csatlist *ptr;
};

struct fcsomo  {                    /* a fa csomópontja: */
      char *szo;                      /* a szöveg mutatói */
      struct csatlist *sorok;               /* a sorszámok */
      struct fcsomo *bal;          /* a bal oldali gyermek */
      struct fcsomo *jobb;         /*a jobb oldali gyermek */
};

struct fcsomo *addtreex (struct fcsomo *, char *, int);
int getword (char *, int);
int toltszo (char *);
void treexprint (struct fcsomo *);

/* kereszthivatkozási program */
main()
{
      struct fcsomo *gyoker;
      char szo[MAXSZO];
      int sorszam = 1;
```

```
        gyoker = NULL;
        while (getword(szo, MAXSZO) != EOF)
                if (isalpha(szo[0] && toltszo(szo) == -1)
                        gyoker = addtreex (gyoker, szo, sorszam);
                else if (szo[0] == '\n')
                        sorszam++;
        treexprint (gyoker);
        return 0;
}

struct fcsomo *talloc (void);
struct csatlist *lalloc (void);
void addsor (struct fcsomo *, int);

/* addtreex: w elhelyezése a p című csomópontban vagy az alatt */
struct fcsomo *addtreex (struct fcsomo *p, char *w, int sorszam)
{
        int felt;

        if (p == NULL)  {     /* egy új szó érkezett              */
                p = talloc();       /* csinál egy új csomópontot */
                p->szo = strdup(w);
                p->sorok = lalloc();
                p->sorok->sszam = sorszam;
                p->sorok->ptr = NULL;
                p->bal = p->jobb = NULL;
        } else if ((felt = strcmp(w, p->szo)) == 0)
                addsor(p, sorszam);
        else if (felt < 0)
                p->bal = addtreex(p->bal, w, sorszam);
        else
                p->jobb = addtreex (p->jobb, w, sorszam);
        return p;
}

/* addsor: egy sorszámot iktat a csatolt listába */
void addsor (struct fcsomo *p, int sorszam)
{
        struct csatlist *temp;

        temp = p->sorok;
        while (temp->ptr != NULL && temp->sszam != sorszam)
                temp = temp->ptr;
        if (temp->sszam != sorszam)  {
                temp->ptr = lalloc();
                temp->ptr->sszam = sorszam;
                temp->ptr->ptr = NULL;
        }
}
```

```
/* treexprint: a p fa rendezett kiírása */
void treexprint (struct fcsomo *p)
{
      struct csatlist *temp;

      if (p != NULL)  {
            treexprint (p->bal);
            printf ("%10s: ", p->szo);
            for (temp = p->sorok; temp != NULL; temp = temp->ptr)
                  printf ("%4d ", temp->sszam);
            printf ("\n");
            treexprint(p->jobb);
      }
}

/* lalloc: a csatolt listában egy csomópontot hoz létre */
struct csatlist *lalloc (void)
{
      return (struct csatlist *) malloc (sizeof(struct csatlist));
}

/* toltszo: a szóról megállapítja, hogy töltelékszó-e */
int toltszo (char *w)
{
      static char *nw[] = {
            "a",
            "an",
            "and",
            "are",
            "in",
            "is",
            "of",
            "or",
            "that",
            "the",
            "this",
            "to"
      };
      int felt, kozep;
      int also = 0;
      int felso = sizeof(nw)/sizeof(char *) - 1;

      while (also <= felso)  {
            kozep = (also + felso)/2;
            if ((felt = strcmp(w, nw[kozep])) < 0)
                  felso = kozep - 1;
            else if (felt > 0)
                  also = kozep + 1;
            else
                  return kozep;
      }
      return -1;
}
```

A fában minden önálló szóhoz egy csomópont tartozik. Minden csomópont
- a szó szövegét kijelölő mutatóból (`szo`),
- a sorszámok csatolt listájának mutatójából (`sorok`),
- a bal oldali gyermek csomópont mutatójából (`bal`) és
- a jobb oldali gyermek csomópont mutatójából (`jobb`)

áll.

A sorszámok csatolt listájának minden eleme egy `csatlist` típusú struktúra, és az egyes struktúrák egy sorszámot, valamint a listában következő elemet kijelölő mutatót tartalmazzák. Ha nincs több elem a listában, akkor a mutató értéke `NULL`.

Az `addtreex` az `addtree` eljárás (K–R: 155. oldal) módosított változata, és feladata, hogy elhelyezzen egy szót a fában, ill. egy sorszámot a csatolt listában. Ha ez egy új szó volt, akkor a csatolt lista első eleméhez hozzárendeli a sorszámot a

```
p->sorok->sszam = sorszam;
```

utasítással. Ha a szó már a fában volt, amit a

```
((felt = strcmp(w, p->szo)) == 0)
```

vizsgálat határoz meg, akkor az `addsor` eljárás elhelyezi a sorszámot a csatolt listában.

Az `addsor` eljárás a

```
while (temp->ptr != NULL && temp->sszam != sorszam)
       temp = temp->ptr;
```

ciklussal végigkeresi a csatolt listát, hogy azonos sorszám ne forduljon elő benne. Ha a sorszám nincs a csatolt listában, akkor az

```
if (temp->sszam != sorszam)  {
       temp->ptr = lalloc();
       temp->ptr->sszam = sorszam;
       temp->ptr->ptr = NULL;
}
```

utasításokkal generál egy új csomópontot és ott elhelyezi azt.

A `treexprint` a K–R 155. oldalán leírt `treeprint` módosított változata és feladata a fa ábécé sorrendben történő kiírása. A fa minden szavához a `treexprint` kiírja magát a szót és az összes sorszámot, amelyeken a szó előfordult.

A `toltszo` eljárás a töltelékszavakat tartalmazó `static` tárolási osztályú tömbben keres egy szót, és ha a szó nem töltelékszó, akkor –1 értékkel tér vissza. A töltelékszavak `nw[]` tömbje tetszőleges hosszúságú lehet, kiegészíthető új szavakkal, csak arra kell ügyelni, hogy az ASCII kódok szerint növekvő sorrendbe rendezett legyen (a keresési algoritmus miatt).

A `getword` eljárást (6.1. gyakorlat) módosítani kell, hogy a `'\n'` karakter esetén visszatérve ne vesszen el a sorszám. Ez a

```
while (isspace(c = getch()) && c != '\n')
       ;
```

módosított ciklussal érhető el.

6.4. gyakorlat

Írjunk programot, amely kiírja a beolvasott szöveg egyes szavait azok előfordulási gyakoriságának csökkenő sorrendjében! Minden szó elé írjuk ki az előfordulásának számát is (K–R: 157. oldal)!

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

#define  MAXSZO    100
#define  KULONBOZ  1000

struct fcsomo  {                          /* a fa csomópontja: */
      char *szo;                           /* a szöveg mutatói */
      int szam;                         /* az előfordulások száma */
      struct fcsomo *bal;                  /* bal oldali gyermek */
      struct fcsomo *jobb;                /* jobb oldali gyermek */
};

struct fcsomo *addtree (struct fcsomo *, char *);
int getword (char *, int);
void rendlist (void);
void treestore (struct fcsomo *);

struct fcsomo *list[KULONBOZ]; /* a fa csomópontjainak mutatói */
int ncs = 0;                   /* a fa csomópontjainak száma */

/* a különböző szavak kiírása előfordulási gyakoriságuk */
/* csökkenő sorrendjében */
main()
{
      struct fcsomo *gyoker;
      char szo[MAXSZO];
      int i;

      gyoker = NULL;
      while (getword(szo, MAXSZO) != EOF)
            if (isalpha(szo[0]))
                  gyoker = addtree (gyoker, szo);
      treestore(gyoker);
      rendlist();
      for (i = 0; i < ncs; i++)
            printf ("%2d:%20s\n", list[i]->szam, list[i]->szo);
      return 0;
}
```

```
/* treestore: a fa csomópontjának mutatóját */
/* elhelyezi a list[] helyre */
void treestore (struct fcsomo *p)
{
      if (p != NULL)  {
            treestore (p->bal);
            if (ncs < KULONBOZ)
                  list[ncs++] = p;
         treestore (p->jobb);
      }
}

/* rendlist: a fa csomópontjaihoz tartozó mutatók rendezett */
/* listájának előállítása */
void rendlist ()
{
      int koz, i, j;
      struct fcsomo *temp;

      for (koz = ncs/2; koz > 0; koz /= 2)
            for (i = koz; i < ncs; i++)
                  for (j = i - koz; j >= 0; j -= koz)  {
                        if((list[j]->szo) >=(list[j + koz]>szo))
                              break;
                        temp = list[j];
                        list[j] = list[j + koz];
                        list[j + koz] = temp;
                  }
}
```

A megkülönböztethető szavak max. számát a `KULONBOZ` tartalmazza. A programban használt `fcsomo` struktúra megegyezik a K–R: 154. oldalán leírt struktúrával. A `list` mutatókból álló tömb, amelynek minden eleme egy `fcsomo` típusú struktúrát címez. Az `ncs` változó a csomópontok számát tartalmazza.

A program egyenként beolvassa a szavakat és elhelyezi azokat a fa egy csomópontján. A `treestore` eljárás minden szó csomópontjának mutatóját elhelyezi a `list` tömbben. A `rendlist` eljárás a K–R: 76. oldalán ismertetett `shellsort` eljárás módosított változata, és a `list` tömb elemeit az előfordulási gyakoriságok csökkenő sorrendjébe rendezi.

6.5. gyakorlat

Írjunk `undef` néven függvényt, amely a `lookup` és `install` függvények kezelte táblázatból töröl egy nevet és a hozzá tartozó definíciót (K–R: 159. oldal)!

```
unsigned hash (char *);

/* undef: töröl egy nevet és definíciót a táblázatból */
void undef (char *s)
{
      int h;
      struct nlist *elozo, *np;

      elozo = NULL;
      h = hash(s); /* az s karaktersorozat hash-kódolt értéke   */
      for (np = hashtab[h]; np != NULL; np = np->kovetkezo)
{
            if (strcmp(s, np->nev) == 0)
                  break;
            elozo = np;      /* megőrzi az előző bejegyzést  */
      }
      if (np != NULL)  {                        /* talált nevet */
         if (elozo == NULL)      /* első a hash-listában?     */
             hashtab[h] = np->kovetkezo;
         else                 /* valahol a hash-táblában van    */
         elozo->kovetkezo = np->kovetkezo;
         free((void *) np->nev);
         free((void *) np->hszov);
         free((void *) np); /* felszabadítja a struktúra helyét   */
      }
}
```

Az `undef` eljárás ciklusban keresi az `s` karaktersorozatot a táblázatban és ha megtalálta, kilép a ciklusból az

```
if (strcmp(s, np->nev) == 0)
      break;
```

vizsgálat hatására.

Ha az `s` karaktersorozat nincs a táblázatban, akkor a `for` ciklus normálisan zárul és az `np` mutató `NULL` értéket kap.

Ha `np` nem `NULL` értékű, akkor a nevet (`nev`) és a definíciót (`hszov`) eltávolítja a program a táblázatból. A `hashtab` egy bejegyzése a csatolt lista kezdetét jelöli ki. `np` mutat az eltávolítandó bejegyzésre és `elozo` mutat az `np` előtti bejegyzésre. Amikor `elozo` `NULL` értékű, akkor `np` az első bejegyzés a `hashtab[h]` helyen induló csatolt listában. Ezt az

```
if (elozo == NULL)
      hashtab[h] = np->kovetkezo;
else
      elozo->kovetkezo = np->kovetkezo;
```

utasítássorozat vizsgálja. Az `np` bejegyzés eltávolítása után a név és a definíció, valamint maga a struktúra számára lefoglalt helyet a

```
free ((void *) np->nev);
free ((void *) np->hszov);
free ((void *) np);
```

utasításokkal fel kell szabadítani. A `free` függvény leírása a K–R 183. oldalán található.

6.6. gyakorlat

Ebben a részben ismertetett eljárások, valamint a `getch` és `ungetch` függvények felhasználásával valósítsa meg a `#define` utasítást (direktívát) feldolgozó egyszerű (argumentumokat nem használó) programot! A program legyen alkalmas a C nyelv szintaktikája szerinti, argumentum nélküli `#define` utasítások feldolgozására (K–R: 159. oldal)!

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

#define  MAXSZO   100

struct nlist  {                    /* a táblázat bejegyzései */
      struct nlist *kovetkezo; /* a következő elem a listában */
      char *nev;                          /* a definiált név */
      char *hszov;                /* a helyettesítő szöveg */
};

void error (int, char *);
int getch (void);
void getdef (void);
int getword (char *, int);
struct nlist *install (char *, char *);
struct nlist *lookup (char *);
void skipblanks (void);
void undef (char *);
void ungetch (int);
void ungets (char *);

/* a #define feldolgozó egyszerű változata */
main ()
{
      char w[MAXSZO];
      struct nlist *p;

      while (getword(w, MAXSZO) != EOF)
          if (strcmp(w, "#") == 0)    /* az utasítás kezdete*/
              getdef();
          else if (!isalpha(w[0]))
              printf("%s", w);       /* nem lehet definiált */
          else if ((p = lookup(w)) == NULL)
              printf("%s", w);             /* nem definiált */
          else
              ungets (p->hszov);     /* visszateszi a definíciót */
      return 0;
}
```

```
/* getdef: bekér egy definíciót és elhelyezi */
void getdef (void)
{
      int c, i;
      char def[MAXSZO], dir[MAXSZO], nev[MAXSZO];

      skipblanks();
      if (!isalpha(getword(dir, MAXSZO)))
            error(dir[0], "getdef: egy direktívát vár az # után");
      else if (strcmp(dir, "define") == 0)  {
            skipblanks();
            if (!isalpha(getword(nev, MAXSZO)))
                error(nev[0], "getdef: nem alfabetikus, nevet vár");
            else  {
                  skipblanks();
                  for (i = 0; i < MAXSZO - 1; i++)
                        if ((def[i] = getch()) == EOF || def[i] == '\n')
                            break;   /* vége a definíciónak */
                  def[i] = '\0';
              if (i <= 0)                /* nincs definíció? */
                        error('\n', "getdef: nem teljes
                                     definíció");
              else                      /* elhelyezi a definíciót */
                        install(nev, def);
            }
      } else if (strcmp(dir, "undef") == 0 )  {
            skipblanks();
            if (!isalpha(getword(nev, MAXSZO)))
                error (nev[0], "getdef: nem alfabetikus az undef-ben");
            else
                undef (nev);
      } else
            error (dir[0], "getdef: egy direktívát vár az # után");
}

/* error: kiír egy hibaüzenetet és átlépi a sor maradékát */
void error (int c, char *s)
{
      printf ("Hiba: %s\n", s);
      while (c != EOF && c != '\n')
            c = getch();
}

/* skipblanks: átlépi a szóközöket és tabulátorokat */
void skipblanks (void)
{
      int c;

      while((c = getch()) == ' ' || c == '\t')
            ;
      ungetch(c);
}
```

A `#define` utasítás feldolgozásának gerincét a `main` eljárás tartalmazza. Az utasításoknak (`define`, `undef`) # után kell következni, és ha ezek egyikét találja a program, akkor azokat a `getdef` eljárással dolgozza fel. Ha a `getword` eljárás nem alfabetikus karakterrel tér vissza, akkor a szó nem lehet definíció és a program a szóval együtt kiírja a megfelelő hibaüzenetet, máskülönben pedig a szóhoz keres egy lehetséges definíciót. Ha a definíció létezik, akkor az `ungets` függvény (amelyet a 4.7. gyakorlatban ismertettünk) fordított sorrendben "visszaírja" azt a bemenetre.

A `getdef` függvény a

```
#define    nev        definíció
#undef     nev
```

típusú utasításokat (direktívákat) képes feldolgozni, ahol a `nev` alfanumerikus kell legyen.

A `define` utasítás feldolgozása során a

```
for (i = 0; i < MAXSZO - 1; i++)
      if ((def[i] = getch()) == EOF || def[i] == '\n')
            break;
```

ciklus gyűjti a definíciót a sor vagy az állomány végéig. Ha létezik definíció, akkor a `getdef` az `install` függvényt (K–R: 159. oldal) felhasználva elhelyezi azt a táblázatban.

Az `undef` utasítás a nevet eltávolítja a táblázatból (a 6.5. gyakorlatban leírtak szerint).

A `getword` eljárást úgy módosítottuk, hogy szóközzel tér vissza, így a kimenete hasonlít a bemeneti adatsorhoz.

7.1. gyakorlat

Írjunk programot, amely a hívásakor az `argv[0]`-ban elhelyezett paramétertől függően a nagybetűket kisbetűvé vagy a kisbetűket nagybetűvé alakítja (K–R: 167. oldal)!

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

/* alsó: a  nagybetűket kisbetűvé alakítja */
/* felső: a kisbetűket nagybetűvé alakítja */
main (int argc, char *argv[])
{
      int c;

      if (strcmp(argv[0], "alsó") == 0)
            while ((c = getchar()) != EOF)
                  putchar (tolower(c));
      else
            while ((c = getchar()) != EOF)
                  putchar (toupper(c));
      return 0;
}
```

Amikor a programot az `alsó` névvel hívjuk, akkor az a nagybetűket kisbetűkké alakítja, minden más esetben a kisbetűkből nagybetűk lesznek.

A híváskor alkalmazott nevet az `strcmp` függvénnyel határozzuk meg, mivel az `strcmp` nulla értékkel tér vissza, ha az `argv[0]` az `alsó` karaktersorozatot tartalmazza.

Az

```
if (strcmp(argv[0], "alsó") == 0)
```

utasítás UNIX operációs rendszer alatt is használható, mivel `argv[0]` a program neve, amit a felhasználó ír be. Néhány más operációs rendszer esetén az `argv[0]` a program helyét meghatározó teljes elérési út, ellentétben azzal, amit a felhasználó beírt.

A program használja a `<ctype.h>` headerben található `tolower` és `toupper` könyvtári függvényeket.

7.2. gyakorlat

Írjunk programot, amely a tetszőleges bemeneti szöveget értelmes módon írja ki! A minimális igény, hogy a nem nyomtatható karaktereket a helyi szokásoknak megfelelően oktális vagy hexadecimális számként írja ki, és a túl hosszú sorokat tördeljük rövidebb sorokra (K–R: 169. oldal)!

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

#define  MAXSOR  100  /* karakterek max. száma a sorban */
#define  OKTH      6          /* egy oktális szám hossza */

/* tetszőleges bemeneti szöveg kiírása értelmes módon */
main()
{
      int c, hely;
      int inc (int hely, int n);

      hely = 0;                          /* a hely a soron belül */
      while ((c = getchar()) != EOF)
            if (iscntrl(c) || c == ' ') {  /* nem nyomtatható  */
                                           /* karakter vagy szóköz  */
                  hely = inc(hely, OKTH);
                  printf(" \\%03o ", c); /* újsor-karakter? */
                  if (c == '\n')  {
                        hely = 0;
                        putchar('\n');
                  }
            } else  {                    /* nyomtatható karakter */
                  hely = inc(hely, 1);
                  putchar(c);
            }
      return 0;
}

/* inc: a kimenet helyszámlálójának növelése */
int inc (int hely, int n)
{
      if (hely + n < MAXSOR)
            return hely + n;
      else  {
            putchar ('\n');
            return n;
      }
}
```

A program legfeljebb `MAXSOR` hosszúságú bemeneti sorokat kezel, és a feldolgozáshoz a `<ctype.h>` headerben található `iscntrl` makrót használja, amely megkeresi a nem nyomtatható karaktereket: a `delete` karaktert (oktális `0177`) és a szokásos vezérlőkarak-

tereket (amelyek kódja oktális 040 alatti). A program az üres helyeket is nem nyomtatható karakterként kezeli, és oktális számként a kódjukat írja ki (egy szóközzel kezdve, amit a \ követ, majd OKTH pozíción a szám, utána egy újabb szóközzel). Az újsor-karakter nullára állítja a hely változót és kiíródik az

```
if (c == '\n')  {
      hely = 0;
      putchar('\n');
}
```

utasítások hatására.

Az inc függvény az utolsó felhasználható hellyel tér vissza és töri a sort, ha nincs további n pozíció a sorban.

7.3. gyakorlat

Egészítsük ki a minprintf programot újabb, printf függvényben megengedett lehetőségekkel (K–R: 171. oldal)!

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

#define  LOCALFMT  100

/* minprintf: változó hosszúságú argumentumlistát kezelő, */
/* egyszerűsített printf függvény */
void minprintf (char *fmt, ...)
{
      va_list am;          /* sorjában az egyes név nélküli */
                                  /* argumentumokra mutat */
      char *p, *sert;
      char localfmt[LOCALFMT];
      int i, iert;
      unsigned uert;
      double dert;

      va_start(am, fmt);          /* hatására am az első név */
                        /* nélküli argumentumra fog mutatni */
      for (p = fmt; *p; p++)  {
            if (*p != '%')  {
                  putchar (*p);
                  continue;
      }
```

```
            i = 0;
            localfmt[i++] = '%';     /* indul a helyi formátum */
            while (*(p + 1) && !isalpha(*(p + 1)))
                 localfmt[i++]=*++p; /* gyűjti a karaktereket */
            localfmt[i++] = *(p + 1);     /* a formátum betűje */
            localfmt[i] = '\0';
            switch(*++p) { /* a formátum betűjének feldolgozása */
            case 'd' :
            case 'i' :
                  iert = va_arg(am, int);
                  printf(localfmt, iert);
                  break;
            case 'x' :
            case 'X' :
            case 'u' :
            case 'o' :
                  uert = va_arg(am, unsigned);
                  printf(localfmt, uert);
                  break;
            case 'f' :
                  dert = va_arg(am, double);
                  printf(localfmt, dert);
                  break;
            case 's' :
                  sert = va_arg(am, char *);
                  printf(localfmt, sert);
                  break;
            default :
                  printf(localfmt);
                  break;
            }
      }
      va_end(am);              /* a listafeldolgozás lezárása */
}
```

A `minprintf` program végigmegy az argumentumlistán és feldolgozza azt, de a kényelem kedvéért magát a kiírást a `printf` függvénnyel végeztetjük.

A `printf` függvény lehetőségeinek kezelésére a `%` jelet követő karaktereket a formátumot meghatározó betűig egy `localfmt` karakteres tömbbe gyűjtjük, majd a formátum betűjének feldolgozása során ez a `localfmt` lesz a `printf` függvény formátumát megadó argumentum.

7.4. gyakorlat

Írjuk meg a `scanf` függvény egyszerűsített változatát az előző pontban látott `minprintf` mintájára (K–R: 174. oldal)!

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

#define  LOCALFMT  100

/* minscanf: változó hosszúságú argumentumlistát kezelő, */
/* egyszerűsített scanf függvény */
void minscanf (char *fmt, ...)
{
      va_list am;          /* sorjában az egyes név nélküli */
                           /* argumentumokra mutat */
      char *p, *sert;
      char localfmt[LOCALFMT];
      int c, i, *iert;
      unsigned *uert;
      double *dert;

      i = 0;               /* a helyi formátum indexe */
      va_start(am, fmt);   /* hatására am az első név */
                           /* nélküli argumentumra fog mutatni */
      for (p = fmt; *p; p++)  {
            if (*p != '%')  {
                  localfmt[i++] = *p; /* gyűjti a karaktereket */
                  continue;
            }
            localfmt[i++] = '%';  /* indul a helyi formátum */
            while (*(p + 1) && !isalpha(*(p + 1)))
                 localfmt[i++] = *++p;/* gyűjti a karaktereket */
            localfmt[i++] = *(p + 1);   /* a formátum betűje */
            localfmt[i] = '\0';
            switch(*++p) { /* a formátum betűjének feldolgozása */
            case 'd' :
            case 'i' :
                  iert = va_arg(am, int *);
                  scanf(localfmt, iert);
                  break;
            case 'x' :
            case 'X' :
            case 'u' :
            case 'o' :
                  uert = va_arg(am, unsigned *);
                  scanf(localfmt, uert);
                  break;
            case 'f' :
                  dert = va_arg(am, double *);
                  scanf(localfmt, dert);
                  break;
            case 's' :
                  sert = va_arg(am, char *);
                  scanf(localfmt, sert);
                  break;
```

```
            default :
                  scanf(localfmt);
                  break;
            }
            i = 0;                        /* törli az indexet */
      }
      va_end(am);              /* a listafeldolgozás lezárása */
}
```

A `minscanf` függvény hasonlóan működik, mint a `minprintf` függvény: a formátumot meghatározó, `%` jel utáni betűig begyűjti a formátumot leíró karaktersorozatból a karaktereket. A begyűjtött karaktereket a `localfmt` tömbben helyezi el, és ha eljutott a formátum betűjéig, akkor a `localfmt` tömböt a megfelelő mutatóval együtt átadja a `scanf` függvénynek, amely elvégzi a tényleges beolvasást.

A `scanf` argumentumai mutatók: az első argumentum a formátumot leíró karaktersorozat mutatója, a további argumentumok pedig azon változók mutatói, amelyek az értéküket a `scanf` függvénytől kapják. A változó mutatóját a `va_arg` eljárással határozzuk meg, majd eltároljuk egy helyi változóba, amit argumentumként adunk át a `scanf` függvénynek. Ezek után a `scanf` beolvassa a felhasználó által megadott értéket és elhelyezi a változóban.

7.5. gyakorlat

Írjuk meg a 4. fejezetben ismertetett postfix adatbeírási formátumú kalkulátor program új változatát, amely a bemeneti adatok és számok átalakítását a `scanf` és/vagy `sscanf` függvénnyel valósítja meg (K–R: 174. oldal)!

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

#define  SZAM  '0'              /* jelzi, hogy számot talált */

/* getop: megadja a következő operátort vagy számot(operandust) */
int getop (char s[])
{
      int c, i, rc;
      static char utolsoc[] = " ";

      sscanf(utolsoc, "%c", &c);
      utolsoc[0] = ' ';          /* törli az utolsó karaktert */
      while ((s[0] = c) == ' ' || c == '\t')
            if (scanf("%c", &c) == EOF)
                  c = EOF;
      s[1] = '\0';
      if (!isdigit(c) && c != '.')
            return c;                              /* nem szám */
```

```
    i = 0;
    if (isdigit(c))                    /* gyűjti az egész részt */
        do {
            rc = scanf("%c", &c);
            if (!isdigit(s[++i] = c))
                break;
        } while (rc != EOF);
    if (c == '.')                      /* gyűjti a törtrészt */
        do {
            rc = scanf("%c", &c);
            if (!isdigit(s[++i] = c))
                break;
        } while (rc != EOF);
    s[i] = '\0';
    if (rc != EOF)
        utolsoc[0] = c;
    return SZAM;
}
```

A K–R-ben leírt `getop` eljárást csak kis mértékben módosítottuk.

Az egyik módosítás, hogy az eljárás a `static` tárolási osztályú `utolsoc` változó felhasználásával a hívások között is "emlékszik" a számot követő karakterre. Az `utolsoc` egy kételemű tömb, amelybe eltároljuk az utoljára beolvasott számot (a második elem a végjel, mivel a `sscanf` egy karaktersorozatot vár). A

```
sscanf(utolsoc, "%c", &c)
```

utasítás beolvassa a karaktert és elhelyezi az `utolsoc[0]` helyen. Ezt a

```
c = utolsoc[0]
```

helyett használhatjuk.

A `scanf` a sikeresen illesztett és elhelyezett bemeneti adatok számával tér vissza (K–R: 171. oldal), vagy elérve az állomány végét az `EOF` jelzéssel.

Az eredeti `getop`-hoz képest megváltoztattuk az

```
isdigit(s[++i] = c = getch())
```

kifejezést is az

```
rc = scanf("%c", &c);
if (!isdigit(s[++i] = c))
    break;
```

utasításssorra, mivel a `scanf` függvényt hívjuk, elhelyezzük a karaktereket az `s` karaktersorozatban és csak utána vizsgáljuk, hogy számjegy-e.

Előfordulhat, hogy a `scanf` `EOF` jelzést talál, de a `c` változó nem módosul. Az ebből adódó problémát az

```
rc != EOF
```

vizsgálattal oldjuk meg, de emiatt a `scanf` sem javítja az eredeti `getop` működését, ami egyenként olvassa a karaktereket.

A feladat egy másik megoldása:

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

#define  SZAM  '0' /* jelzi, hogy számot talált */

/* getop: megadja a következő operátort vagy számot(operandust) */
int getop (char s[])
{
      int c, rc;
      float f;

      while ((rc = scanf("%c", &c)) != EOF)
            if ((s[0] = c) != ' ' && c != '\t')
                  break;
      s[1] = '\0';
      if (rc == EOF)
            return EOF;
      else if (!isdigit(c) && c != '.')
            return c;
      ungetc (c, stdin);
      scanf ("%f", &f);
      sprintf (s, "%f", f);
      return SZAM;
}
```

Ennél a megoldásnál addig olvassuk a karaktereket, amíg szóköz vagy tabulátor karaktert nem találunk, ill. a beolvasó ciklust az állomány vége jelzés is leállítja.

Ha a karakter számjegy vagy tizedespont, akkor visszaírjuk a bemenetre az `ungetc` könyvtári függvény felhasználásával, majd a `scanf` függvénnyel számként beolvassuk. Mivel a `getop` a számmal, mint lebegőpontos értékkel tér vissza, az `f` értékét a `sprintf` eljárással alakítjuk `s` karaktersorozattá.

7.6. gyakorlat

Írjunk programot, amely összehasonlít két állományt és kiírja az első olyan sort, ahol különböznek (K–R: 181. oldal)!

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

```
        i = 0;
        if (isdigit(c))                    /* gyűjti az egész részt */
              do  {
                    rc = scanf("%c", &c);
                    if (!isdigit(s[++i] = c))
                          break;
              }  while (rc != EOF);
        if (c == '.')                      /* gyűjti a törtrészt */
              do  {
                    rc = scanf("%c", &c);
                    if (!isdigit(s[++i] = c))
                          break;
              }  while (rc != EOF);
        s[i] = '\0';
        if (rc != EOF)
              utolsoc[0] = c;
        return SZAM;
}
```

A K–R-ben leírt `getop` eljárást csak kis mértékben módosítottuk.

Az egyik módosítás, hogy az eljárás a `static` tárolási osztályú `utolsoc` változó felhasználásával a hívások között is "emlékszik" a számot követő karakterre. Az `utolsoc` egy kételemű tömb, amelybe eltároljuk az utoljára beolvasott számot (a második elem a végjel, mivel a `sscanf` egy karaktersorozatot vár). A

```
sscanf(utolsoc, "%c", &c)
```

utasítás beolvassa a karaktert és elhelyezi az `utolsoc[0]` helyen. Ezt a

```
c = utolsoc[0]
```

helyett használhatjuk.

A `scanf` a sikeresen illesztett és elhelyezett bemeneti adatok számával tér vissza (K–R: 171. oldal), vagy elérve az állomány végét az `EOF` jelzéssel.

Az eredeti `getop`-hoz képest megváltoztattuk az

```
isdigit(s[++i] = c = getch())
```

kifejezést is az

```
rc = scanf("%c", &c);
if (!isdigit(s[++i] = c))
      break;
```

utasítássorra, mivel a `scanf` függvényt hívjuk, elhelyezzük a karaktereket az `s` karaktersorozatban és csak utána vizsgáljuk, hogy számjegy-e.

Előfordulhat, hogy a `scanf` `EOF` jelzést talál, de a `c` változó nem módosul. Az ebből adódó problémát az

```
rc != EOF
```

vizsgálattal oldjuk meg, de emiatt a `scanf` sem javítja az eredeti `getop` működését, ami egyenként olvassa a karaktereket.

A feladat egy másik megoldása:

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

#define  SZAM  '0' /* jelzi, hogy számot talált */

/* getop: megadja a következő operátort vagy számot(operandust) */
int getop (char s[])
{
      int c, rc;
      float f;

      while ((rc = scanf("%c", &c)) != EOF)
            if ((s[0] = c) != ' ' && c != '\t')
                  break;
      s[1] = '\0';
      if (rc == EOF)
            return EOF;
      else if (!isdigit(c) && c != '.')
            return c;
      ungetc (c, stdin);
      scanf ("%f", &f);
      sprintf (s, "%f", f);
      return SZAM;
}
```

Ennél a megoldásnál addig olvassuk a karaktereket, amíg szóköz vagy tabulátor karaktert nem találunk, ill. a beolvasó ciklust az állomány vége jelzés is leállítja.

Ha a karakter számjegy vagy tizedespont, akkor visszaírjuk a bemenetre az `ungetc` könyvtári függvény felhasználásával, majd a `scanf` függvénnyel számként beolvassuk. Mivel a `getop` a számmal, mint lebegőpontos értékkel tér vissza, az `f` értékét a `sprintf` eljárással alakítjuk `s` karaktersorozattá.

7.6. gyakorlat

Írjunk programot, amely összehasonlít két állományt és kiírja az első olyan sort, ahol különböznek (K–R: 181. oldal)!

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

```
#define  MAXSOR  100

/* comp: összehasonlít két állományt és az első különböző */
/* sort kiírja */
main(int argc, char *argv[])
{
      FILE *fp1, *fp2;
      void filecomp (FILE *fp1, FILE *fp2);

      if (argc != 3)  {       /* az argumentumok száma jó? */
         fprintf(stderr, "comp: két filenév kell\n");
         exit(1);
      } else {
         if ((fp1 = fopen(*++argv, "r")) == NULL) {
              fprintf(stderr,"comp: nem lehet megnyitni %s\n",*argv);
              exit(1);
      }  else if ((fp2 = fopen(*++argv, "r")) == NULL)  {
              fprintf(stderr,"comp: nem lehet megnyitni %s\n",*argv);
              exit(1);
      }  else  {      /* megtalálta és megnyitotta az */
                      /* összehasonlítandó állományokat */
              filecomp(fp1, fp2);
              fclose(fp1);
              fclose(fp2);
              exit(0);
        }
      }
}

/* filecomp: soronként összehasonlít két állományt */
void filecomp (FILE *fp1, FILE *fp2)
{
      char sor1[MAXSOR], sor2[MAXSOR];
      char *sp1, *sp2;
      do {
            sp1 = fgets(sor1, MAXSOR, fp1);
            sp2 = fgets(sor2, MAXSOR, fp2);
          if (sp1 == sor1 && sp2 == sor2)  {
             if (strcmp(sor1, sor2) != 0)  {
                 printf("az első különböző sor\n%s\n", sor1);
                 sp1 = sp2 = NULL;
             }
          } else if (sp1 != sor1 && sp2 == sor2)
             printf("vége az első állománynak, az utolsó
                                          sor\n%s\n", sor2);
          else if (sp1 == sor1 && sp2 != sor2)
             printf("vége a második állománynak, az utolsó
                                            sor\n%s\n", sor1);
      } while (sp1 == sor1 && sp2 == sor2);
}
```

Az argumentumok száma három kell legyen: a program neve és a két összehasonlítandó állomány neve. A program megnyitja az állományokat és soronként összehasonlítja azokat a `filecomp` eljárással.

A `filecomp` az állományokból beolvas egy-egy sort az `fgets` függvénnyel, amely visszatér a beolvasott sor mutatójával vagy a `NULL` értékkel, ha elértük az állomány végét. Ha `sp1` és `sp2` a megfelelő sorokra mutatnak és egyik állománynak sincs vége, akkor az `strcmp` összehasonlítja a két sort. Ha a két sor nem egyezik, akkor a `filecomp` kiírja azt a sort, amelyben eltérést talált.

Ha `sp1` és `sp2` nem a megfelelő sorra mutat, akkor elértük az állományok egyikének végét (EOF) és a két állomány különbözik. Ha viszont `sp1` és `sp2` egyike sem mutat a megfelelő sorra, akkor mindkét állomány befejeződött és ilyenkor az állományok nem különböznek.

7.7. gyakorlat

Módosítsuk a K–R 5. fejezetében ismertetett mintakereső programot úgy, hogy bemenetét a parancssor-argumentumként megadott állománynevek listájából, vagy ha nincs argumentum, akkor a standard bemenetről vegye! Ki kell-e íratni az állomány nevét, ha a program egymáshoz illeszkedő sorokat talál (K–R: 181. oldal)?

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

#define  MAXSOR  1000          /* a bemeneti sor max. hossza */

/* find: megkeresi és kiírja az 1. argumentumában megadott */
/* mintát tartalmazó sorokat */
main(int argc, char *argv[])
{
      char minta[MAXSOR];
      int c, kiveve = 0, szam = 0;
      FILE *fp
      void fmint (FILE *fp, char *fnev, char *minta,
                                        int kiveve, int szam);

      while (--argc > 0 && (*++argv)[0] == '-')
            while (c = *++argv[0])
                  switch (c)  {
                  case 'x' :
                        kiveve = 1;
                        break;
                  case 'n' :
                        szam = 1;
                        break;
                  default :
                        printf ("find: illegális opció %c\n", c);
```

```
                        argc = 0;
                        break;
                }
        if (argc >= 1)
                strcpy(minta, *argv);
        else  {
                printf("find [-x] [-n] minta [file...]\n");
                exit (1);
        }
        if (argc == 1)              /* a standard input olvasása */
                fmint(stdin, "", minta, kiveve, szam);
        else
                while (--argc > 0) /* a megnevezett állomány bekérése */
                        if ((fp = fopen(*++argv, "r"))   == NULL)  {
                                fprintf(stderr, "find: nem lehet megnyitni
                                                        %s\n", *argv);
                                exit (1);
                        } else { /* a megnevezett állomány nyitva */
                                fmint(fp, *argv, minta, kiveve, szam);
                                fclose(fp);
                        }
        return 0;
}

/* fmint: megkeresi a mintát */
void fmint (FILE *fp, char *fnev, char *minta,
                                        int kiveve, int szam)
{
        char sor[MAXSOR];
        long sorszam = 0;

        while (fgets(sor, MAXSOR, fp) != NULL)  {
                ++sorszam;
                if ( (strstr(sor, minta) != NULL) != kiveve )  {
                        if (*fnev)              /* van állománynév */
                                printf("%s - ", fnev);
                        if (sorszam)          /* a sorszám kiírása */
                                printf("%ld: ", sorszam);
                        printf("%s", sor);
                }
        }
}
```

A `main` eljárás a K–R: 5. fejezetében leírtak szerint (K–R: 130. oldal) feldolgozza az opciókat megadó argumentumokat, majd ezután – legalább egy argumentumot várva – a keresett mintát olvassa be. Ha nem követi állománynév a mintát, akkor a program a standard bemenetet, ellenkező esetben viszont a megadott nevű állományt használja. Mindkét esetben a kereséshez az `fmint` függvényt használjuk.

Az `fmint` függvény lényegében hasonló az eredeti `find` program `main` eljárásához. A függvény az `fgets`-et híva (K–R: 180. oldal) sorokat olvas, és azokban keresi a mintát. Az olvasás az állomány végéig folytatódik, amikor is `fgets` a `NULL` értékkel tér vissza.

A gyakorlatban előforduló lehetőségek:

```
(strstr(sor, minta) != NULL) != kiveve                    eredmény

0 (nem találta meg a mintát) != 0 (nincs specifikálva)    hamis
1 (megtalálta a mintát)      != 0 (nincs specifikálva)    igaz
0 (nem találta meg a mintát) != 1 (specifikált)           igaz
1 (megtalálta a mintát)      != 1 (specifikált)           hamis
```

Amikor a kifejezés értéke igaz, `fmint` kiírja az állomány nevét (kivéve, ha az a standard input), a sorszámot (ha kértük) és magát a sort.

7.8. gyakorlat

Írjunk programot, amely több állományt ír ki egymás után, minden állományt új oldalon kezdve! A program minden állomány kiírása előtt írja ki a lap tetejére a címet, és az oldalakat állományonként folyamatosan számozza (K–R: 181. oldal)!

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

#define  MAXTET  3      /* a sorok max. száma a lap tetején */
#define  MAXFEJ  5   /* a sorok max. száma a lap fejlécében */
#define  MAXSOR  100                  /* egy sor max. hossza */
#define  MAXLAP  66         /* a sorok max. száma egy lapon */

/* print: egyenként, új oldalon kezdve kiírja az állományokat */
main (int argc, char *argv[])
{
      FILE *fp;
      void fileprint (FILE *fp, char *fnev);

      if (argc == 1) /* nincs argumentum, a standard bemenetre ír */
            fileprint (stdin, " ");
      else
            while (--argc > 0)
                  if((fp = fopen(*++argv, "r")) == NULL)  {
                      fprintf(stderr, "print: nem lehet megnyitni
                                                    %s\n", argv);
                      exit(1);
                  } else  {
                      fileprint(fp, *argv);
                      fclose(fp);
                  }
      return 0;
}
```

```
/* fileprint: kiírja az fnev nevű állományt */
void fileprint (FILE *fp, char *fnev)
{
      int sorsz, lapsz = 1;
      char sor[MAXSOR];
      int fejlec(char *fnev, int lapsz);

      sorsz = fejlec(fnev, lapsz++);
      while (fgets(sor, MAXSOR, fp) != NULL)  {
            if (sorsz == 1)  {
                  fprintf(stdout, "\f");
                  sorsz = fejlec(fnev, lapsz++);
            }
            fputs(sor, stdout);
            if (++sorsz > MAXLAP - MAXTET)
                  sorsz = 1;
      }
      fprintf(stdout, "\f"); /* lapdobás az állománykiírás után */
}

/* fejlec: kiírja a fejlécet és a szükséges számú üres sort */
int fejlec (char *fnev, int lapsz)
{
      int ssz = 3;

      fprintf(stdout, "\n\n");
      fprintf(stdout, "%s  oldal %d\n", fnev, lapsz);
      while (ssz++ < MAXFEJ)
            fprintf(stdout, "\n");
      return ssz;
}
```

A program hasonló a K–R 178. oldalán leírt `cat` programhoz.

A `fileprint` eljárás két argumentumot igényel: egy megnyitott állomány mutatóját és egy állománynév mutatóját (vagy egy üres karaktersorozatot, ha az állomány a standard bemenet). Az eljárás beolvassa az adott állományt, majd kiírja sorait.

A `\f` karakter a lapdobás.

A programban a `sorsz` változó számolja a sorokat a lapon, amelynek hossza `MAXLAP` lehet. Amikor `sorsz = 1`, a `fileprint` kiír egy lapdobást és egy új fejlécet, majd alaphelyzetbe állítja a `sorsz` változót. Az egyes állományok utolsó lapjának kiírása után szintén lapdobás következik.

A `fejlec` függvény kiírja az állomány nevét és a lapszámot, majd megfelelő számú üres sort hagy (hogy a lap tetején `MAXFEJ` számú sor legyen). A `MAXTET` a lap tetején hagyott üres sorok száma.

7.9. gyakorlat

Az `isupper`-hez hasonló függvények helytakarékos vagy időtakarékos változatban írhatók meg. Vizsgálja meg, hogyan lehetséges mindkét változat kialakítása (K–R: 184. oldal)!

```
/* isupper: 1 (igaz) értékkel tér vissza, ha c  nagybetű */
int isupper (char c)
{
      if(c >= 'A' && c <= 'Z')
            return 1;
      else
            return 0;
}
```

Az `isupper`-nek ez a változata egy egyszerű `if - else` szerkezettel vizsgálja a karaktert, és ha annak kódja a nagybetűkhöz tartozó ASCII kódok tartományába esik, akkor az 1 (igaz) értékkel tér vissza, minden más esetben a visszatérési érték 0 (hamis). A függvény ezen változata helytakarékos kialakítású. A

```
#define isupper(c)   ((c) >= 'A' && (c) <= 'Z') ? 1 : 0
```

bár időtakarékos változat, de az előzőhöz képest több helyet igényel.

Az időmegtakarítást az okozza, hogy nincs szükség minden alkalommal a függvény hívására, viszont a makró minden olyan sorban, ahol használjuk kifejtődik, ami a program helyfoglalását növeli. Azt sem szabad elfelejteni, hogy az argumentum kétszeri kiértékeléséből problémák adódhatnak. Például a

```
char *p = "Ez egy szöveg";
if (isupper(*p++))
        ...
```

programrészlet fordítása során a makró a

```
((*p++) >= 'A' && (*p++) <=  'Z') ? 1 : 0
```

formában fejtődik ki, és `*p` értékétől függően a `p` mutató egyszer vagy kétszer inkrementálódik. A második inkrementálás nem fordulhat elő, ha az `isuppert`-t függvényként valósítjuk meg, mert az argumentumot a függvény csak egyszer értékeli ki.

Sok esetben az argumentum második, váratlan kiértékelése gondot okoz, mivel a `p` mutató második inkrementálása hibás eredményre vezet. Makrót alkalmazva a hiba a

```
char *p = "Ez egy szöveg";
if (isupper(*p))
        ...
p++;
```

programszerkezettel küszöbölhető ki.

Legyünk tudatában annak, hogy a makrók argumentuma többször is kiértékelődhet. Az `isupper`-hez hasonló makró a `<ctype.h>` headerben található `toupper` és `tolower` is.

Kapcsolódás a UNIX operációs rendszerhez

8.1. gyakorlat

Írjuk újra a 7. fejezetben megismert `cat` programot úgy, hogy a standard könyvtári függvények helyett a `read`, `write`, `open` és `close` függvényeket használjuk! Végezzünk kísérleteket a két változat futási idejének meghatározására (K–R: 190. oldal)!

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"

void error (char *fmt, ...);

/* cat: állományok konkatenálása - read, write, */
/* open és close függvényekkel */
main (int argc, char *argv[])
{
      int fp;
      void filecopy (int ifp, int ofp);

      if ( argc == 1 )     /* nincs argumentum, a standard */
                                    /* bemenetről másol */
            filecopy(0, 1);
      else
            while (--argc > 0)
                  if ((fp = open(*++argv, O_RDONLY)) == -1)
                     error("cat: nem lehet megnyitni %s", *argv);
                  else  {
                        filecopy (fp, 1);
                        close (fp);
                  }
      return 0;
}

/* filecopy: az ifp-vel címzett állományt az ofp-vel */
/* címzett állományba másolja */
void filecopy (int ifp, int ofp)
```

```
{
        int n;
        char buf[BUFSIZ];

        while ((n = read(ifp, buf, BUFSIZ)) > 0)
                if (write(ofp, buf, n) != n)
                        error ("cat: írás-hiba");
}
```

A program

```
if ((fp = open(*++argv, O_RDONLY)) == -1)
```

utasítása olvasásra megnyitja az állományt és visszatér az (egész típusú) állományleíróval vagy `-1` értékkel, ha hiba volt.

A `filecopy` függvény az `ifp` állományleíró felhasználásával `BUFSIZ` számú karaktert olvas. A `read` függvény a ténylegesen beolvasott karakterek számával (egy bájt-számmal, `n`) tér vissza. Ha ez a bájt-szám nullánál nagyobb, akkor nincs hiba, a nulla bájt-szám az állomány végét jelzi és a `-1` érték hibára utal. A `write` függvény `n` bájtot ír ki, és ha a kiírt bájt-szám nem egyenlő `n`-nel, akkor írás közben hiba történt.

Az `error` függvényt a K–R 190. oldalán írtuk le.

Ez a programváltozat kb. kétszer gyorsabb a K-R: 7. fejezetében leírt eredeti programnál.

8.2. gyakorlat

Írjuk át az `fopen` és `_fillbuf` függvényeket úgy, hogy az explicit bitműveletek helyett bitmezőket használunk! Hasonlítsuk össze a két változat forrásprogramjának méretét és a futási időket (K–R: 194. oldal)!

```
#include <fcntl.h>
#include "syscalls.h"
#define  ENG  0666          /*írás, olvasás a tulajdonosnak, */
                                /* a csoportnak és másoknak */

/* fopen: megnyit egy állományt, visszatér az állománymutatóval */
FILE *fopen (char nev, char *mod)
{
        int fd;
        FILE *fp;

        if (*mod != 'r' && *mod != 'w' && *mod != 'a')
                return NULL;
        for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
                if (fp->flag.is_read == 0 && fp->flag.is_write == 0)
                        break;          /* szabad területet talált */
```

```
    if (fp >= _iob + OPEN_MAX)
        return NULL;                    /* nincs szabad hely */

    if (*mod == 'w')                    /* létrehoz egy állományt */
        fd = creat(nev, ENG);
    else if (*mod == 'a')  {
        if ((fd = open(nev, O_WRONLY, 0) == -1)
            fd = creat(nev, ENG);
        lseek(fd, 0L, 2);
    } else
        fd = open(nev, O_RDONLY, 0);
    if (fd == -1)                       /* a név nem érhető el */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag.is_unbuf = 0;
    fp->flag.is_buf = 1;
    fp->flag.is_eof = 0;
    fp->flag.is_err = 0;
    if (*mod == 'r')  {                          /* olvasás */
        fp->flag.is_read = 1;
        fp->flag.is_write = 0;
    }  else  {                                   /* írás */
        fp->flag.is_read = 0;
        fp->flag.is_write = 1;
    }
    return fp;
}

/* _fillbuf: területet foglal a bemeneti puffernek és feltölti */
int _fillbuf (FILE *fp)
{
    int bufsize;

    if  (fp->flag.is_read == 0 || fp->flag.is_eof == 1
                               || fp->flag.is_err == 1)
        return EOF;
    bufsize = (fp->flag.is_unbuf == 1) ? 1 : BUFSIZ;
    if (fp->base == NULL)               /* még nincs puffer */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF; /* nincs hely a puffer számára */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0)  {
        if (fp->cnt == -1)
            fp->flag.is_eof = 1;
        else
            fp->flag.is_err = 1;
```

```
            fp->cnt = 0;
            return EOF;
        }
        return (unsigned char) *fp->ptr++;
}
```

A `struct _iobuf typedef` utasítása a K–R 192. oldalán található, és az `_iobuf` struktúra egyik tagja a `flag`, amit

```
int flag;
```

módon definiáltunk. Ezt a `flag` változót a megfelelő bitmezőkkel újra kell definiálni:

```
struct flag_field  {
        unsigned is_read    : 1;
        unsigned is_write   : 1;
        unsigned is_unbuf   : 1;
        unsigned is_buf     : 1;
        unsigned is_eof     : 1;
        unsigned is_err     : 1;
};
```

Az eredeti program

```
if ((fp->flag & (_READ | _WRITE)) == 0)
        break;
```

utasításaiban a `_READ` és `_WRITE` VAGY kapcsolata vesz részt, vagyis:

```
        (_READ    |    _WRITE)
          01      |       02          oktálisan
          01      |       10          binárisan
                11                    az eredmény.
```

Ez azt jelenti, hogy az `if` utasítás csak akkor igaz, ha a `flag` két legkisebb helyiértékű bitje nulla állapotú (azaz nem olvasható és nem írható állományról van szó).

Ugyanez a bitmezők felhasználásával:

```
if (fp->flag.is_read == 0 && fp->flag.is_write == 0)
        break;
```

formában írható fel.

A következő módosítás a bitmezők megfelelő bitjeinek beállítása az

```
fp->flag.is_unbuf = 0;
fp->flag.is_buf   = 1;
fp->flag.is_eof   = 0;
fp->flag.is_err   = 0;
```

utasításokkal. Az

```
fp->flag = (*mod == 'r') ? _READ : _WRITE;
```

utasítás `'r'` megadása esetén a `flag`-et `_READ` értékre, különben pedig `_WRITE` értékre állítja.

Bitmezőkkel ugyanez úgy valósítható meg, hogy `'r'` esetén az `is_read` bitet kell `1` állapotba állítani, máskülönben pedig az `is_write` bitet. Ezt az

```
if (*mod == 'r')  {
     fp->flag.is_read  = 1;
     fp->flag.is_write = 0;
}  else  {
     fp->flag.is_read  = 0;
     fp->flag.is_write = 1;
}
```

utasítások hajtják végre. A `_fillbuf` függvény hasonló módon változott. A függvény `EOF` értékkel tér vissza, ha vagy az állomány nincs megnyitva olvasásra, vagy már elértük az állomány végét, vagy hibát észleltünk. Az eredeti `_fillbuf` függvényben ezt az

```
if ((fp->flag & (_READ|_EOF|_ERR)) != _READ)
```

utasítás vizsgálja. Ugyanez bitmezőkkel:

```
if ( fp->flag.is_read == 0 ||
     fp->flag.is_eof  == 1 ||
     fp->flag.is_err  == 1 )
```

alakban írható fel. A

```
bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
```

utasítás a

```
bufsize = (fp->flag.is_unbuf == 1) ? 1 : BUFSIZ;
```

utasításra változott, és az

```
   fp->flag |= _EOF;
else
   fp->flag |= _ERR;
```

utasítások helyett az

```
   fp->flag.is_eof = 1;
else
  fp->flag.is_err = 1;
```

utasításokat kell használni.

A módosított függvény mérete nagyobb volt, mint az eredeti függvényé és a futási idő is növekedett. A bitmezők használata nagymértékben gépfüggő és lassíthatja a programot.

8.3. gyakorlat

Tervezzük meg és írjuk meg a `_flushbuf`, `fflush` és `fclose` függvényeket (K–R: 194. oldal)!

```
#include "syscalls.h"

/* _flushbuf: helyet foglal a kimeneti puffernek és kiüríti */
int _flusbuf (int x, FILE *fp)
{
      unsigned nc;          /* a kitöltött karakterek száma */
      int bufsize;          /* a kijelölt puffer mérete */

      if (fp < _iob || fp >= _iob + OPEN_MAX)
            return EOF;         /* hiba: érvénytelen mutató */
      if ((fp->flag & (_WRITE | _ERR)) != _WRITE)
            return EOF;
      bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
      if (fp->base == NULL)  {          /* még nincs puffer */
            if ((fp->base = (char *) malloc(bufsize)) == NULL)
{
                  fp->flag |= _ERR;
                  return EOF;   /* nem hozható létre puffer */
            }
      }  else  {                      /* a puffer már létezik */
            nc = fp->ptr - fp->base;
            if (write(fp->fd, fp->base, nc) != nc)  {
                  fp->flag |= _ERR;
                  return EOF;    /* hiba: visszatér EOF-fal */
            }
      }
      fp->ptr = fp->base;                /* a puffer kezdete */
      *fp->ptr++ = (char) x; /* elmenti az aktuális karaktert */
      fp->cnt = bufsize - 1;
      return x;
}

/* fclose: lezárja az állományt */
int fclose (FILE *fp)
{
      int rc;                          /* a visszatérési kód */

      if ((rc = fflush(fp)) != EOF)  { /* van mit üríteni? */
            free(fp->base);      /* felszabadítja a puffert */
            fp->ptr = NULL;
            fp->cnt = 0;
            fp->base = NULL;
            fp->flag &= ~(_READ | _WRITE);
      }
  return rc;
}
```

```
/* fflush: az fp-hez tartozó állomány pufferét  kiüríti */
int fflush (FILE *fp)
{
      int rc = 0;

      if (fp < _iob || fp >= _iob + OPEN_MAX)
            return EOF;          /* hiba: érvénytelen mutató */
      if (fp->flag & _WRITE)
            rc = _flusbuf(0, fp);
      fp->ptr = fp->base;
      fp->cnt = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
      return rc;
}
```

A `_flusbuf` függvény `EOF` értékkel tér vissza, ha az állomány nem nyitható meg írásra vagy bármi hiba fordult elő. Ezt az

```
if ((fp->flag & (_WRITE | _ERR)) != _WRITE)
   return EOF;
```

utasítások kezelik.

Ha még nincs puffer, akkor a `flushbuff` a `fillbuf`-hoz hasonlóan (K–R: 194. oldal) létrehoz egyet, ha viszont létezik, akkor kiüríti a benne lévő karaktereket.

A program következő tevékenysége az argumentum elmentése a pufferbe, amit a

```
*fp->ptr++ = (char) x;
```

utasítás végez.

A lehetséges karakterek száma (`fp->cnt`) a pufferben eggyel kisebb a puffer méreténél, mivel egy karakter már el van mentve.

Az `fclose` hívja az `fflush` függvényt, mert ha az állományt írásra megnyitottuk, szükségszerűen írtunk bele néhány karaktert. A függvény alaphelyzetbe állítja az `_iobuf` struktúra megfelelő tagjait, így az `fopen` nem talál majd értelmetlen értékeket. Ha nem történt hiba, az `fclose` `0` értékkel tér vissza.

Az `fflush` ellenőrzi, hogy a mutató érvényes-e és hívja az `_flushbuf` függvényt, ha az állomány írásra volt megnyitva. Végül az `fflush` alaphelyzetbe állítja `cnt` és `ptr` értékét, majd `rc` értékkel tér vissza.

8.4. gyakorlat

A standard könyvtár

```
int fseek (FILE *fp, long offset, int bazis)
```

függvénye megegyezik az `lseek` függvénnyel, kivéve, hogy az `fp` állománymutatót használja az állományleíró helyett és hogy a visszatérési értéke az `int` típusú állapotjelzés, nem pedig

egy pozíció. Írjuk meg az `fseek` függvényt! Gondoskodjunk arról, hogy az általunk írt `fseek` pufferkezelése összhangban legyen a könyvtár többi függvényével (K–R: 194. oldal)!

```
#include "syscalls.h"

/* fseek: keresés állománymutatóval */
int fseek (FILE *fp, long offset, int bazis)
{
      unsigned nc;       /* a kiürítendő karakterek száma */
      long rc = 0;       /* a visszatérési kód */

      if (fp->flag & _READ)  {
            if (bazis == 1) /* az aktuális helytől keressen? */
                offset -= fp->cnt; /*megjegyzi a karaktereket */
                                              /* a pufferben */
            rc = lseek(fp->fd, offset, bazis);
            fp->cnt = 0;        /* nincs pufferelt karakter */
      }  else if (fp->flag & _WRITE)  {
            if ((nc = fp->ptr - fp->base) > 0)
                 (write(fp->fd, fp->base, nc) != nc)
                       rc = -1;
            if (rc != -1)                    /* nem volt hiba? */
                  rc = lseek(fp->fd, offset, bazis);
      }
      return (rc == -1) ? -1 : 0;
}
```

Az `rc` változó tartalmazza a visszatérési kódot, és ha ez `-1` értékű, akkor hiba történt.

Az `fseek` használatakor két lehetőség van: az állomány olvasásra van megnyitva vagy az állomány írásra van megnyitva.

Ha az állomány olvasásra van megnyitva és a `bazis = 1`, akkor az `offset` az aktuális pozíciótól számlálódik (további lehetőség, ha a `bazis = 0`, akkor az `offset` az állomány kezdetétől, ha pedig `bazis = 2`, akkor az állomány végétől számlálódik). Az `offset` aktuális pozíciótól való számlálásához az `fseeek` figyelembe veszi a már pufferben lévő karaktereket is az

```
if (bazis == 1)
      offset -= fp->cnt;
```

utasításokkal.

Az `fseek` ezután hívja az `lseek` függvényt és eldobja a pufferelt karaktereket az

```
rc = lseek(fp->fd, offset, bazis);
fp->cnt = 0;
```

utasításokkal.

Ha az állományt írásra nyitottuk meg, akkor az `fseek` először kiüríti a puffert (ha volt benne valami) az

```
if ((nc = fp->ptr - fp->base) > 0)
      if (write(fp->fd, fp->base, nc) != nc)
            rc = -1;
```

utasításokkal.

Ha nem fordult elő hiba, akkor az `fseek` hívja az `lseek` függvényt:

```
if ( rc != -1 )
      rc = lseek(fp->fd, offset, bazis);
```

Az `fseek` a sikeres keresés után `0` értékkel tér vissza.

8.5. gyakorlat

Módosítsuk az `fsize` programot úgy, hogy más, az inode táblázatban szereplő információt is kiírjon (K–R: 200. oldal)!

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>          /* jelzők írásra és olvasásra */
#include <sys/types.h>              /* typedef utasítások */
#include <sys/stat.h>  /* stat-ból visszaadott struktúra */
#include "dirent.h"

int stat (char *, struct stat *);
void dirwalk (char *, void (*fcn)(char *));

/* fsize: kiírja az adott nevű állományhoz tartozó inode számát */
/* és az inod mode, nlink paraméterét, ill. az állomány méretét */
void fsize (char *nev)
{
      struct stat stbuf;

      if (stat(nev, &stbuf) == -1)  {
          fprintf(stderr, "fsize: nem hozzáférhető %s\n", nev);
            return;
      }
      if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
            dirwalk(nev, fsize);
      printf("%S5u %6o %3u %8ld %s\n", stbuf.st_ino, stbuf.st_mode,
                              stbuf.st_nlink, stbuf.st_size, nev);
}
```

A módosított `fsize` kiírja az adott nevű állományhoz tartozó inod-számot (`st_ino`), oktális számként a mód bitek értékét (`st_mode`), a linket számát (`st_nlink`), az állomány méretét (`st_size`) és az állomány nevét. Még további információkat is kiírathatunk, ha azok lényegesek a számunkra.

A programban használt `dirwalk` függvény leírása a K–R: 198. oldalán található.

8.6. gyakorlat

A standard könyvtárban található `calloc(n, size)` függvény *n* darab `size` méretű objektum számára lefoglalt és nulla kezdeti értékkel feltöltött tárolóterület mutatójával tér vissza. Írjuk meg a `calloc` függvényt úgy, hogy az hívja a `malloc`-ot, vagy megfelelően módosítsuk a `malloc` függvényt (K–R: 205. oldal)!

```
#include "syscalls.h"

/* calloc: n db size méretű objektum számára helyet foglal */
void *calloc (unsigned n, unsigned size)
{
      unsigned i, nb;
      char *p, *q;

      nb = n * size;
      if ((p = q = malloc(nb) != NULL)
            for (i = 0; i < nb; i++)
                  *p++ = 0;
            return q;
}
```

A `calloc` függvény *n* db `size` méretű objektum számára foglal helyet. A lefoglalt bájtok teljes száma `nb`, ami

```
nb = n * size;
```

utasítással határozható meg. A `malloc` ezek után egy `nb` bájtos tárolóterület mutatójával tér vissza és a `p`, ill. `q` mutatók megőrzik a lefoglalt tárolóterület kezdőcímét. Ha a helyfoglalás sikeres volt, akkor az `nb` bájtot a

```
for (i = 0; i < nb; i++)
   *p++ = 0;
```

ciklus tölti fel a `0` kezdeti értékkel.

A `calloc` a lefoglalt és nullával feltöltött tárolóterület kezdőcímét kijelölő mutatóval tér vissza.

8.7. gyakorlat

A `malloc` a kért méretet ellenőrzés nélkül elfogadja és a `free` feltételezi, hogy a felszabadítandó blokk mérete érvényes. Javítsuk ki úgy ezeket a programokat, hogy nagyobb gondot fordítsanak a hibaellenőrzésre (K–R: 205. oldal)!

```
#include "syscalls.h"

#define   MAXBYTES   (unsigned)   10240

static unsigned maxalloc; /* a lefoglalt egységek max. száma */
static Header   base;           /* üres lista az induláshoz */
static Header  *freep = NULL;     /* az üres lista kezdete */

/* malloc: általános célú tárterület-foglaló program */
void *malloc (unsigned nbytes)
{
      Header *p, *prevp;
      static Header *morecore(unsigned);
      unsigned nunits;

      if (nbytes > MAXBYTES)  {  /* nem több, mint MAXBYTES */
            fprintf(stderr, "alloc: nem kérhető %u bájtnál
                                              több\n", MAXBYTES);
            return NULL;
      }
      nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
      if ((prevp = freep) == NULL) { /* nincs még szabad lista */
            base.s.ptr = freep = prevp = &base;
            base.s.size = 0;
      }
      for (p = prevp->s.ptr;  ;prevp = p, p = p->s.ptr)  {
            if (p->s.size >= nunits) { /* elég nagy a hely? */
                  if (p->s.size == nunits) /* a méretek egyeznek? */
                        prevp->s.ptr = p->s.ptr;
                  else  {           /* kiadja a blokk végét */
                        p->s.size -= nunits;
                        p += p->s.size;
                        p->s.size = nunits;
                  }
                  freep = prevp;
                  return (void *) (p + 1);
            }
            if (p == freep)           /* körbement a listán */
                  if ((p = morecore(nunits)) == NULL)
                        return NULL;    /* nincs több hely */
      }
}
```

```
#define   NALLOC  1024     /* a min. terület a malloc által */
                                    /* igényelt egységekben */

/* morecore: az operációs rendszertől tárterületet kér */
static Header *morecore (unsigned nu)
{
      char *cp, *sbrk(int);
      Header *up;

      if (nu < NALLOC)
          nu = NALLOC;
      cp = sbrk(nu * sizeof(Header));
      if (cp == (char *) - 1)          /* nincs több terület */
          return NULL;
      up = (Header *) cp;
      up->s.size = nu;
      maxalloc = (up->s.size > maxalloc) ? up->s.size : maxalloc;
      free((void *)(up + 1));
      return freep;
}

/* free: visszarak egy blokkot a szabad blokkok listájába */
void free (void *ap)
{
      Header *bp, *p;

      bp = (Header *) ap - 1;        /* a blokk fejére mutat */
      if (bp->s.size == 0 || bp->s.size > maxalloc)  {
            fprintf(stderr, "free: nem szabadítható fel %u
                                          egység\n", bp->s.size);
            return;
      }
      for (p = freep; !(bp > && bp < p->s.ptr); p = p->s.ptr)
            if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
                  break;   /* a felszabadult blokk a lista  */
                           /* elejére vagy végére kerül  */

      if (bp + bp->s.size == p->s.ptr) {          /* a felső  */
                                  /* szomszédhoz kapcsoljuk  */
            bp->s.size += p->s.ptr->s.size;
            bp->s.ptr = p->s.ptr->s.ptr;
      }  else
            bp->s.ptr = p->s.ptr;
      if (p + p->s.size == bp)  {                  /* az alsó  */
                                 /* szomszédhoz kapcsoljuk  */
            p->s.size += bp->s.size;
            p->s.ptr = bp->s.ptr;
      }  else
            p->s.ptr = bp;
      freep = p;
}
```

Az új `malloc` függvény a tetszőlegesen választható `MAXBYTES` állandóval való összehasonlítással ellenőrzi az igényelt bájtok számát. `MAXBYTES` értékét az alkalmazott operációs rendszerhez illesztve kell megválasztani.

Amikor a `morecore` egy új blokkot ad, a `static` tárolási osztályúnak definiált `maxalloc` változó megjegyzi az eddigi legnagyobb felhasznált blokk méretét. Így a `free` függvény ellenőrizheti, hogy a visszaadott blokk mérete nem nulla vagy nem nagyobb a kiosztott legnagyobb blokknál.

8.8. gyakorlat

A `malloc` és a `free` függvények felhasználásával írjuk meg a `bfree(p, n)` függvényt úgy, hogy az felszabadítsa az *n* karakterből álló tetszőleges `p` blokkot. Ezt a `bfree` függvényt alkalmazva a felhasználó bármikor beiktathat a szabad blokkok listájába egy statikus vagy külső tömböt (K–R: 205. oldal).

```
#include "syscalls.h"

/* bfree: p db n karakteres blokkot szabadit fel */
unsigned bfree (char *p, unsigned n)
{
      Header *hp;

      if (n < sizeof(Header))
            return 0;          /* túl kicsi, nem használható */
      hp = (Header *) p;
      hp->s.size = n / sizeof(Header);
      free((void *)(hp + 1));
      return hp->s.size;
}
```

A `bfree` függvény két argumentumot igényel: egy `p` mutatót és a karakterek `n` számát. A függvény csak akkor szabadítja fel a blokkot, ha annak mérete legalább `sizeof(Header)` és ha a méret ezt a határt nem éri el, akkor `0` értékkel tér vissza. A `p` mutatóhoz kényszerített típusmódosítással `Header` típust rendelünk és értékét átadjuk a `hp` változónak a

```
hp = (Header *) p;
```

utasítással.

A blokk mérete `sizeof(Header)` egységekben

```
hp->s.size = n / sizeof(Header);
```

A program utolsó lépésben hívja a `free` függvényt. Mivel `free` egy mutatót vár, ami éppen a blokkfej utáni címre mutat, a programban a `(hp + 1)` értéket használjuk, amit kényszerített típusmódosítással `(void *)` típusúvá alakítunk – mint ahogy korábban a `morecore` függvénynél is tettük.

A `bfree` eljárás `0` értékkel tér vissza, ha a felszabadítandó blokk mérete túl kicsi és minden más esetben a visszatérési érték a blokk `sizeof(Header)` egységekben mért mérete.

C PROGRAMOZÁSI FELADATOK MEGOLDÁSAI

Kernighan és Ritchie szabványosított C nyelvet ismertető könyve számos megoldandó programozási feladattal segíti a nyelvet tanulókat. Ezek a feladatok fejezetenként, didaktikailag és nehézségi szempontból is jól összeválogatva végig kísérik a könyvet. Tondó és Gimpel nem kevesebbre vállalkozott, mint hogy a könyv 97 feladatának elkészíti a megoldásait. A kapott megoldásokat felhasználva minden tanuló ellenőrizheti a tudását, ill. elgondolkodhat azon, hogy vajon miért „profibb" a közölt megoldás a sajátjánál. Aki már elsajátított egy programozási nyelvet, az jól tudja, hogy a kész programok tanulmányozása adja a legtöbb segítséget a profi programozóvá váláshoz. A könyvben közölt megoldásokat a szerzők számos magyarázattal látták el, ill. sok helyen rámutatnak a program bővítési lehetőségeire, ill. a megoldás más módon való megközelítésére.
A könyvet minden, a C nyelvvel most ismerkedő szakember számára ajánljuk.

Tartalom:

ISBN 963 16 1067 5
9 789631 610673