Kedves Kollegina, Kolléga!

A jegyzetet Önnek készítettem azért, hogy referencia anyaga legyen a **Programnyelv** és a **Programfejlesztés** tárgyakhoz.

Szeretném a segítségét igénybe venni abból a célból, hogy a jegyzet minél pontosabb, megbízhatóbb legyen. Épp ezért arra kérem, ha az olvasás során valamilyen magyartalanságba, nem elégséges magyarázatba vagy uram bocsá' hibába ütközne, jelezze vissza nekem!

Ténykedését előre megköszönöm.

Győr, 2003. február

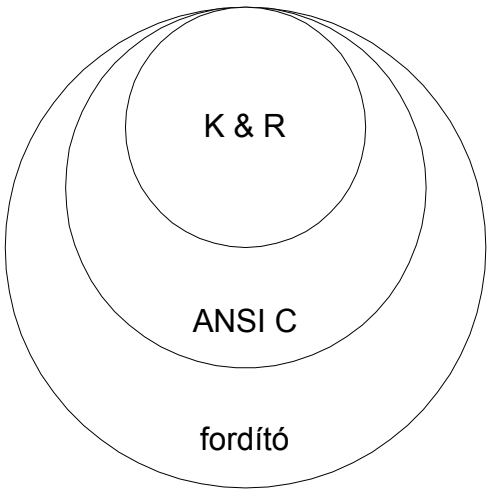
Bauer Péter
Varjasi Norbert

(B609) Tel.: (96) 503400/3254 e-mail: bauer@sze.hu
varjasin@sze.hu

1 BEVEZETÉS

A Széchenyi István Egyetem különféle informatika szakjai és szakirányai C programnyelvi jegyzetigényét hivatott kielégíteni ez a jegyzet. Az olvasóról feltételezi, hogy tisztában van a számítástechnikai alapfogalmakkal [1]. Alapos strukturált programozási ismereteket szerzett, és járatos az alapvető algoritmikus elemekben [2]. Jól ismeri a PASCAL programnyelvet és fejlesztő környezetet [3]. Magyarán ismer, és kezel ilyen fogalmakat, mint:

- Adatok és adattípusok.
- Konstansok, változók és azonosítók.
- Vezérlési szerkezetek: szekvencia, szelekció és iteráció.
- Tömbök és sztringek (karakterláncok).
- Programszerkezeti elemek: eljárások, függvények és programmodulok.
- Láncolt adatszerkezetek: listák és fák.
- Fájlok stb.



A C nyelvet tervezője, Dennis Ritchie, a Bell Laboratóriumban fejlesztette ki az 1970–es évek végén [4], és a UNIX operációs rendszer programnyelvének szánta. Ezt a változatot jelöli az ábrán a **K&R**. A C nyelv ezt követően praktikussága miatt széles körben elterjedt. Sokan készítettek sokféle C fordítót saját, vagy környezetük igényeinek megfelelően. A sokszínűségben amerikai nemzeti szabvánnyal (ANSI) teremtettek rendet az 1980–as évek végén [5]. Az **ANSI C** szabványt aztán Európában (ISO) is elfogadták néhány évvel később. Az ábrából látszik, hogy az **ANSI C** bővítette a **K&R** C halmazt.

További történeti áttekintéshez a [4] és az [5] bevezető részeit ajánljuk!

📖 Az ábrán a legbővebb C halmaz a **fordító**. Ha valamikor is valamilyen gondunk lenne azzal, hogy egy konkrét C utasítás, módosító stb. megfelel–e az **ANSI C** szabványnak, akkor fordítás előtt kapcsoljuk be az integrált programfejlesztő rendszer egy menüpontjával az **ANSI C** kompatibilis fordítást!

A C általános célú programnyelv, mely tömörségéről, hatékonyságáról, gazdaságosságáról és portabilitásáról (hordozhatóságáról) ismert. Nem tartalmaz túl sok vezérlési szerkezetet. Bőséges viszont az operátorkészlete, és több adattípus megléte jellemzi. Jól használható tehát műszaki–tudományos, vagy akár adatfeldolgozási problémák megoldására.

A C elég alacsony szintű – hardver közeli – programnyelv is egyben, hisz tervezője a UNIX operációs rendszert e nyelv segítségével készítette el néhány száz gépi kódú utasítás felhasználásával. A C programok gyakran ugyanolyan gyorsak, mint az ASSEMBLER nyelven készültek, de jóval könnyebben olvashatók és tarthatók karban.

Jegyzetünkben nem kívánunk elmerülni konkrét integrált programfejlesztő rendszerek, operációs rendszerek és processzorok részleteinek taglalásában. Teljes általánosságban azonban még sem célszerű a dolgokról beszélni, mert akkor ilyeneket kéne mondani, mint:

- Képezzük az operációs rendszernek megfelelő végrehajtható fájlt!
- Futtassuk a végrehajtható fájlt az operációs rendszerben!

Ehelyett rögzítsük azt, hogy fogalmainkkal az IBM PC kompatibilis személyi számítógépek területén maradunk! Erre a gépcsaládra is rengeteg cég készített C fordítót (compiler). Itt állapodjunk meg két fő gyártónál: a Borlandnál és a Microsoftnál! Az integrált programfejlesztő keretrendszer legyen menüvel irányítható, s ne parancssori paraméterként megadott kapcsolókkal kelljen vezérelni a fordítót és a kapcsoló–szerkesztőt (linker).

Az operációs rendszer számunkra jobbára csak olyan szempontból érdekes, hogy legyen karakteres szabványos bemenete (standard input), és létezzen karakteres szabvány kimenete (standard output), valamint szabványos hibakimenete (standard error output). A szabvány bemenet alapértelmezés szerint a billentyűzet, a kimenetek viszont a karakteres üzemmódú képernyőre, vagy a karakteres konzol ablakba dolgoznak. A karakteres képernyő, vagy konzol ablak felbontása természetesen változtatható, de mi minden példánál feltételezzük a 25 sorszor 80 oszlopot! A szabvány kimeneteken a mindenkori aktuális pozíciót kurzor jelzi.

2 JELÖLÉSEK

✍ Figyelem felkeltés. Valamely következtetés levonása az eddigiekből. Esetleg: merre találhatók további részletek a kérdéses témával kapcsolatban.

🕮 Lexikális ismeretek taglalása. Valamely folyamat pontosabb részletei. Egy fogalom precízebb definíciója.

💣 Valamilyen aránylag könnyedén elkövethető, de nehezen lokalizálható hiba.

👍 Egy alapvető, úgy nevezett „ököl” szabály.

`Forrásprogramok és képernyő tartalmak szövege.`

Valamilyen konkrétummal helyettesítendő szintaktikai egység.

Kulcsszó vagy valamilyen azonosító.

A fogalom első előfordulásának jelölésére szolgál.

A megoldandó feladatokat így jelöltük.

3 ALAPISMERETEK

3.1 Forrásprogram

Első közelítésben induljunk ki abból, hogy a C program (a forrásprogram) fájlazonosítójában **C** kiterjesztéssel rendelkező, ASCII kódú szövegfájl, mely előállítható, ill. módosítható

- akármilyen ASCII kódú szövegszerkesztővel, vagy
- a programfejlesztő rendszer beépített szövegszerkesztőjével.

📖 Az ASCII kódú szövegfájl sorokból áll. Egy sorban a szöveg sorbeli karaktereinek ASCII kódjai következnek rendre az egymás utáni bájtokban. A sorhoz végül még két, a soremelést leíró bájt tartozik, melyekben egy soremelés (Line Feed – 10) és egy kocsi vissza (Carriage Return – 13) vezérlő karakter van.

💣 Vigyázni kell ASCII kódú szövegfájlban a decimálisan 31 értékű bájt használatával, mert ez fájlvég jelzés a legtöbb operációs rendszerben!

Készítsük el első C programunkat, és mentsük el **PELDA1.C** néven!

```
/* PELDA1.C */
#include <stdio.h>
void main(void){
  printf("Ez egy C program!\n"); }
```

Fordítsuk le a programot, szerkesszük meg a végrehajtható fájlt, és futtassuk le!

A mindenki által gyanított végeredmény az a képernyőn, hogy megjelenik a szöveg, és a következő sor elején villog a kurzor:

```
Ez egy C program!
_
```

3.2 Fordítás

A fordító sikeres esetben a forrásprogramból egy vele azonos nevű (**OBJ** kiterjesztésű) tárgymodult állít elő,

1. ábra: Fordítás

és üzeneteket jelentet meg többek közt a hibákról. A hibaüzenetek legalább kétszintűek:

- (fatális) hibaüzenetek és

- figyelmeztető üzenetek.

💣 A (fatális) hibákat, melyek jobbára szintaktikai jellegűek, mindig kijavítja a programozó, mert korrekciójuk nélkül nem készíti el a tárgymodult a fordító. A figyelmeztető üzenetekkel azonban, melyek sok esetben a legsúlyosabb problémákat jelzik, nem szokott törődni, mert a fordító létrehozza a tárgymodult, ha csak figyelmeztető üzenetekkel zárul a fordítás.

A **PELDA1.C** programunk első sora megjegyzés (comment). A megjegyzés írásszabálya látszik a sorból, azaz:

- /* karakter párral kezdődik és
- */ karakter párral végződik.

A megjegyzés több forrássoron át is tarthat, minden sorba is írható egy, sőt akár egy soron belül több is megadható a szintaktikai egységek között.

💣 Egyetlen tilos dolog van: a megjegyzések nem ágyazhatók egymásba!

```
/* Ez a befoglaló megjegyzés eleje.
  /* Itt a beágyazott megjegyzés. */
     Ez meg a befoglaló megjegyzés vége. */
```

✍ Vegyük észre, hogy bármely hibás program rögtön hibátlanná válik, ha /*–ot teszünk az elejére, és a záró */–t elfelejtjük megadni a további forrásszövegben!

A fordítót egybeépítették egy speciális előfeldolgozóval (preprocessor), mely az „igazi” fordítás előtt

- elhagyja a forrásszövegből a megjegyzéseket,
- végrehajtja a neki szóló direktívákat, és
- ezeket is elhagyja a forrásszövegből.

2. ábra: Fordítás pontosabban

Az előfeldolgozó direktívák egy sorban helyezkednek el, és #–tel kezdődnek. Pontosabban # kell, legyen a sor első nem fehér karaktere.

🕮 Fehér karakterek a szóköz, a soremelés, a lapdobás karakter, a vízszintes és a függőleges tabulátor karakter. Meg kell említeni, hogy a meg-

jegyzés is fehér karakternek minősül. A fehér karakterek szolgálhatnak szintaktikai egységek elválasztására, de a felesleges fehér karaktereket elveti a fordító.

A **PELDA1.C** programunk második sora egy **#include** direktíva, melynek hatására az előfeldolgozó megkeresi és betölti a paraméter szövegfájlt a forrásszövegbe, és elhagyja belőle ezt a direktívát.

A pillanatnyilag betöltendő szövegfájl, az **stdio.h**, **H** kiterjesztésű. Az ilyen kiterjesztésű szövegfájlokat C környezetben fejfájloknak (header) nevezik. A fejfájl egy témával kapcsolatos adattípusok, konstansok definícióit és a vonatkozó függvények jellemzőit tartalmazza.

✍ Fedezzük fel, hogy az **stdio** a standard input output rövidítéséből származik, s így lényegében a szabvány kimenetet és bemenetet „kapcsoltuk" programunkhoz.

🕮 Nem volt még szó arról, hogy a paraméter *fájlazonosító* <> jelek között áll! A <> jel pár tájékoztatja az előfeldolgozót, hogy milyen könyvtárakban keresse a szövegfájlt. A programfejlesztő rendszer menüpontjai közt van egy, melynek segítségével megadhatók a fejfájlok (include fájlok) keresési útjai. *<fájlazonosító>* alakú paraméter hatására az **#include** direktíva csak a programfejlesztő rendszerben előírt utakon keresi a fájlt, és sehol másutt.

PELDA1.C programunk harmadik és negyedik sora a **main** (fő) függvény definíciója. A függvénydefiníció szintaktikai alakja:

típus függvénynév(*formális–paraméterlista*) { *függvény–test* }

- A visszatérési érték *típus*a **void**, mely kulcsszó éppen azt jelzi, hogy a függvénynek nincs visszaadott értéke.
- A *függvénynév* **main**. C környezetben a **main** az indító program.
- A *formális–paraméterlista* mindig ()–ben van. Pillanatnyilag itt is a **void** kulcsszó látható, ami e helyt azt rögzíti, hogy nincs formális paraméter.
- A *függvény–test*–et mindig {}–ben, úgy nevezett blokk zárójelekben, kell elhelyezni.

✍ PASCAL programozónak csak annyi említendő, hogy a { a BEGIN, és a } az END.

🕮 A forrásprogram több forrásmodulban (forrásfájlban) is elhelyezhető. Végrehajtható program létrehozásához azonban valamelyik forrásmodulnak tartalmaznia kell a **main**–t. Az indító program a végrehajtható

program belépési pontja is egyben. Ez az a hely, ahova a memóriába történt betöltés után átadja a vezérlést az operációs rendszer.

🕮 Az indító programnak természetesen a végrehajtható program „igazi" indítása előtt el kell látnia néhány más feladatot is, s a programozó által írt **main**–beli függvénytest csak ezután következik. A gyártók az indító programot rendszerint tárgymodul (**OBJ**) alakjában szokták rendelkezésre bocsátani.

A **PELDA1.C** programban a **main** függvény teste egyetlen függvényhívás. A függvényhívás szintaktikája:

függvénynév(*aktuális–paraméterlista*)

- A *függvénynév* a **printf**, mely függvény az aktuális paraméterét megjelenteti a szabvány kimeneten.
- A ()–ben álló *aktuális–paraméterlista* egytagú, és momentán egy karakterlánc konstans.

🕮 A karakterlánc konstans írásszabálya látszik: idézőjelek közé zárt karaktersorozat. A karakterlánc konstanst a memóriában meg képzeljük úgy el, hogy a fordító a szöveg karaktereinek ASCII kódjait rendre elhelyezi egymást követő bájtokban, majd végül egy tiszta zérustartalmú (minden bitje zérus) bájttal jelzi a karakterlánc végét!

✍ A példabeli karakterlánc konstans végén azonban van egy kis furcsaság: a \n!

Ha valaki áttanulmányozza az ASCII kódtáblát, akkor láthatja, hogy a lehetséges 256 kódpozíció nem mindegyikéhez tartozik karakterkép. Említsük csak meg a szóköz (32) alatti kódpozíciókat, ahol az úgy nevezett vezérlő karakterek is elhelyezkednek! Valahogyan azt is biztosítania kell a programnyelvnek, hogy ezek a karakterek, ill. a karakterkép nélküli kódpozíciók is megadhatók legyenek. A C programnyelvben erre a célra az úgynevezett escape szekvencia (escape jelsorozat) szolgál.

✍ Remélhetőleg világossá vált az előző okfejtésből, hogy az escape szekvencia helyfoglalása a memóriában egyetlen bájt!

🕮 Az escape szekvencia \ jellel kezdődik, s ezután egy karakter következik. A teljesség igénye nélkül felsorolunk itt néhányat!

Escape szekvencia	Jelentés
\b	visszatörlés (back space)
\n	soremelés vagy új sor (line feed)
\r	kocsi vissza (carriage return)
\t	vízszintes tabulátor (horizontal tab)
\”	egyetlen ” karakter
\\	egyetlen \ karakter
\0	karakterlánc záró bájt, melynek minden bitje zérus
\ooo	az *o*–k oktális számok

✍ Vegyük észre, hogy ha idézőjelet kívánunk a karakterlánc konstansba írni, akkor azt csak \” módon tehetjük meg! Ugyanez a helyzet az escape szekvencia kezdőkarakterével, mert az meg csak megkettőzve képez egy karaktert! Lássuk még be, hogy a *ooo* alakkal az ASCII kódtábla bármely karaktere leírható! Például a \012 azonos a \n–nel, vagy a \060 a 0 számjegy karakter.

🕮 A **printf** függvényhívás után álló ; utasításvég jelzés.

✍ **PELDA1.C** programunk „utolsó fehér foltja” a **printf**, mely egyike a szabványos bemenet és kimenet függvényeinek.

3.3 Kapcsoló–szerkesztés (link)

A gyártók a szabvány bemenet, kimenet és más témacsoportok függvényeinek tárgykódját statikus könyvtárakban (**LIB** kiterjesztésű fájlokban) helyezik el. Nevezik ezeket a könyvtárakat futásidejű könyvtáraknak (run time libraries), vagy szabvány könyvtáraknak (standard libraries) is. A könyvtárfájlokból a szükséges tárgykódot a kapcsoló–szerkesztő másolja hozzá a végrehajtható fájlhoz. Lássuk ábrán is a szerkesztést!

3. ábra: Kapcsoló–szerkesztés

💣 A programfejlesztő keretrendszerben a sikeres működéshez bizonyosan be kell állítani a statikus könyvtárfájlok (library) keresési útjait. Azt is meg kell adni természetesen, hogy hova kerüljenek a fordítás és a kapcsoló–szerkesztés során keletkező kimeneti fájlok.

3.4 Futtatás

A végrehajtható fájl a parancssorból azonosítójának begépelésével indítható. Valószínűleg a programfejlesztő rendszer menüjében is van egy pont, mellyel az aktuális végrehajtható fájl futtatható.

Többnyire létezik a három lépést (fordítás, kapcsoló–szerkesztés és futtatás) egymás után megvalósító egyetlen menüpont is.

💣 Ha programunk kimenete nem látható a képernyőn, akkor egy másik menüpont segítségével át kell váltani a felhasználói képernyőre (user screen), vagy aktuálissá kell tenni a futtatott végrehajtható fájl programablakát.

3.5 Táblázat készítése

Készítsük el a következő forint–euró átszámítási táblázatot! Egy euró pillanatnyilag legyen 244 forint 50 fillér!

Forint	Euró
100	0.41
200	0.82
300	1.23
. . .	. . .
1000	4.09

Adatstruktúra:

- A változók a valós típusú **euro**–n kívül, mind egészek. **also** lesz a tartomány alsó határa, **felso** a tartomány felső értéke, **lepes** a lépésköz, és **ft** a ciklusváltozó.

Algoritmus:

- Deklaráljuk a változókat!
- Ellátjuk őket – az **euro**–tól eltekintve – kezdőértékkel.
- Megjelentetjük a táblázat fejléc sorát és az aláhúzást.
- Működtetjük a ciklust, míg **ft** <= **felso**.
- A ciklusmagban kiszámítjuk az aktuális **euro** értéket, megjelentetjük az összetartozó **ft** – **euro** értékpárt, s végül léptetjük az **ft** ciklusváltozót.

Készítsük el a programot!

```
/* PELDA2.C: Forint-euró átszámítási táblázat */
#include <stdio.h>
```

```
void main(void){
  int also, felso, lepes, ft; /* Deklarációk */
  float euro;
  also = 100;          /* Végrehajtható utasítások */
  felso = 1000;
  lepes = 100;
  ft = also;
  printf("   Forint|     Euró\n"
         "---------+---------\n");
  while(ft <= felso){  /* Beágyazott (belső) blokk */
    euro = ft / 244.5;
    printf("%9d|%9.2f\n", ft, euro);
    ft = ft + lepes; } }
```

A **PELDA2.C**–ből kitűnően látszik a C függvény szerkezete, azaz az úgy nevezett blokkszerkezet:

- Előbb a deklarációs utasítások jönnek a blokkbeli változókra. A C szigorú szintaktikájú nyelv:
  - előzetes deklaráció nélkül nem használhatók benne a változók, és
  - kivétel nélkül deklarálni kell minden használatos változót!
- Aztán a végrehajtható utasítások következnek.

💣 A blokkszerkezet deklarációs és végrehajtható részre bontása a C–ben szigorú szintaktikai szabály, azaz egyetlen végrehajtható utasítás sem keveredhet a deklarációs utasítások közé, ill. a végrehajtható utasítások között nem helyezkedhet el deklarációs utasítás. Természetesen a végrehajtható utasítások közé beágyazott (belső) blokkban a szabály újra kezdődik.

✍ Vegyük észre a példaprogramunk végén elhelyezkedő beágyazott vagy belső blokkot! Figyeljük meg a **main** két első sorában, hogy a deklarációs utasítás szintaktikája:

típus azonosítólista;

Az *azonosítólista* azonosítók sorozata egymástól vesszővel elválasztva.

✍ Megfigyelhető még, hogy az azonosítók képzéséhez az angol ábécé betűi használhatók fel!

Foglalkozzunk kicsit a *típus*okkal!

Az **int** (integer) 16, vagy 32 bites, alapértelmezés szerint előjeles (**signed**), fixpontos belsőábrázolású egész típus. Vegyük, mondjuk, a 16 bites esetet! A legnagyobb, még ábrázolható pozitív egész binárisan és decimálisan:

$0111\ 1111\ 1111\ 1111_2 = 2^{15} - 1 = 32767$

A negatív értékek kettes komplemens alakjában tároltak. A legkisebb, még ábrázolható érték így:

$1000\ 0000\ 0000\ 0000_2 = -2^{15} = -32768$

Előjeltelen (**unsigned**) esetben nincsenek negatív értékek. A számábrázolási határok zérus és

$1111\ 1111\ 1111\ 1111_2 = 2^{16} - 1 = 65535$

közöttiek.

✍ Az előzők 32 bites esetre ugyanilyen könnyedén levezethetőek!

A **float** (floating point) 4 bájtos, lebegőpontos belsőábrázolású valós típus, ahol a mantissza és előjele 3 bájtot, s a karakterisztika előjelével egy bájtot foglal el. Az ábrázolási határok: $\pm 3.4*10^{-38} - \pm 3.4*10^{+38}$. Ez a mantissza méret 6 – 7 decimális jegy pontosságot tesz lehetővé.

🕮 Néhány programfejlesztő rendszer a lebegőpontos könyvtárakat (**LIB**) csak akkor kapcsolja be a kapcsoló–szerkesztő által keresésnek alávethető könyvtárak közé, ha a programban egyáltalán igény jelentkezik valamilyen lebegőpontos ábrázolás, vagy művelet elvégeztetésére.

A **PELDA2.C** végrehajtható részének első négy utasítása értékadás.

💣 Ki kell azonban hangsúlyozni, hogy a C–ben nincs értékadó utasítás, csak hozzárendelés operátor, s a hozzárendelésekből azért lesz utasítás, mert ;–t írtunk utánuk.

✍ A hozzárendelésre rögtön visszatérünk! Vegyük észre előbb az egész konstans írásszabályát! Elhagyható előjellel kezdődik, s ilyenkor pozitív, és ezután az egész szám jegyei következnek.

A fejléc sort és az aláhúzást egyetlen **printf** függvényhívással valósítottuk meg. Látszik, hogy a táblázat oszlopait 9 karakter szélességűre választottuk.

✍ Figyeljük meg, hogy a pontos pozícionálást segítendő a fejléc sort és az aláhúzást a **printf**–ben két egymás alá írt karakterlánc konstansként adtuk meg!

🕮 A C fordító a csak fehér karakterekkel elválasztott karakterlánc konstansokat egyesíti egyetlen karakterlánc konstanssá, s így a példabeli **printf**–nek végül is egyetlen paramétere van.

A **PELDA2.C**–ben az elöltesztelő ciklusutasítás következik, melynek szintaktikája:

while(*kifejezés*) *utasítás*

A *kifejezés* aritmetikai, azaz számértékű. Az elöltesztelő ciklusutasítás hatására lépésenként a következő történik:

1. Kiértékeli a fordító a *kifejezés*t. Ha hamis (zérus), akkor vége a ciklusnak, s a **while**-t követő utasítás jön a programban.
2. Ha a *kifejezés* igaz (nem zérus), akkor az *utasítás* végrehajtása, és aztán újból az 1. pont következik.

✍ Világos, hogy a *kifejezés* értékének „valahogyan" változnia kell az *utasítás*ban, különben a ciklusnak soha sincs vége. Az *utasítás* állhat több utasításból is, csak {}–be kell tenni őket. A {}–ben álló több utasítást összetett utasításnak nevezik. Az összetett utasítás szintaktikailag egyetlen utasításnak minősül.

A **PELDA2.C**–ben a **while** *kifejezés*e reláció. A relációjelek a szokásosak: kisebb (<), kisebb egyenlő (<=), nagyobb (>) és nagyobb egyenlő (>=). A reláció két lehetséges értéke: az igaz és a hamis logikai érték. A C–ben azonban nincsen logikai adattípus, így az igaz az 1 egész érték, és a zérus a hamis.

A példabeli belső blokk első és utolsó utasítása hozzárendelés, melynek szintaktikai alakja:

objektum = kifejezés

A hozzárendelés operátor (műveleti jel) bal oldalán valami olyan *objektum*nak kell állnia, ami értéket képes felvenni. A szaknyelv ezt módosítható balértéknek nevezi. Példánkban az összes hozzárendelés bal oldalán egy változó azonosítója áll. Az = jobb oldalán meghatározható értékű *kifejezés*nek (jobbértéknek) kell helyet foglalnia. A hozzárendelés hatására a *kifejezés* értéke - esetlegesen az *objektum* típusára történt konverzió után - felülírja az *objektum* értékét. Az egész „konstrukció" értéke az *objektum* új értéke, és típusa az *objektum* típusa.

✍ A legutóbbi mondat azt célozza, hogy ha a hozzárendelés kifejezés része, akkor ez az érték és típus vesz részt a kifejezés további kiértékelésében.

A beágyazott blokkbeli két hozzárendelés *kifejezés*e aritmetikai. Az aritmetikai műveleti jelek a szokásosak: összeadás (+), kivonás (–), szorzás (*) és az osztás (/).

✍ Vegyük észre, hogy az eddigi **printf** függvényhívásainknak egyetlen karakterlánc konstans paramétere volt, mely változatlan tartalommal jelent meg a karakteres képernyőn! A belső blokkbeli **printf**–ben viszont

három aktuális paraméter van: egy karakterlánc konstans, egy **int** és egy **float**. A gond ugye az, hogy az **int** és a **float** paraméter értékét megjelentetés előtt karakterlánccá kéne konvertálni, hisz bináris bájtok képernyőre vitelének semmiféle értelme nincs!

A **printf** első karakterlánc paramétere

- karakterekből és
- formátumspecifikációkból

áll. A karakterek változatlanul jelennek meg a képernyőn, a formátumspecifikációk viszont meghatározzák, hogy a **printf** további paramétereinek értékeit milyen módon kell karakterlánccá alakítani, s aztán ezt hogyan kell megjelentetni.

A formátumspecifikáció % jellel indul és típuskarakterrel zárul. A formátumspecifikációk és a **printf** további paraméterei balról jobbra haladva rendre összetartoznak. Sőt ugyanannyi formátumspecifikáció lehet csak, mint ahány további paraméter van.

Felsorolunk néhány típuskaraktert a következő táblázatban:

Típuskarakter	A hozzátartozó paraméter típusa	Megjelenítés
d	egész típusú	decimális egészként
f	lebegőpontos	tizedes tört alakjában
c	egy karakter	karakterként
s	karakterlánc	karakterláncként

A formátumspecifikáció pontosabb alakja:

%<*szélesség*><**.***pontosság*>*típuskarakter*

A <>–be tétel az elhagyhatóságot hivatott jelezni. A *szélesség* annak a mezőnek a karakteres szélességét rögzíti, amiben a karakterlánccá konvertált értéket – alapértelmezés szerint jobbra igazítva, és balról szóközfeltöltéssel – kell megjelentetni. Ha a *szélesség*et elhagyjuk a formátumspecifikációból, akkor az adat a szükséges szélességben jelenik meg. Maradjunk annyiban pillanatnyilag, hogy a *pontosság* lebegőpontos esetben a tizedes jegyek számát határozza meg!

✍ Lássuk be, hogy a `"%9d|%9.2f\n"` karakterlánc konstansból a `%9d` és a `%9.2f` formátumspecifikációk, míg a `|` és a `\n` sima karakte-

rek! Vegyük észre, hogy a „nagy” pozícionálgatás helyett táblázatunk fejléc sorának és aláhúzásának megjelentetését így is írhattuk volna:

```
printf("%9s|%9s\n---------+---------\n",
       "Forint", "Euró");
```

🕮 Foglalkoznunk kell még egy kicsit a műveletekkel! Vannak

- egyoperandusos (*operátor operandus*) és
- kétoperandusos (*operandus operátor operandus*)

műveletek. Az egyoperandusos operátorokkal kevés probléma van. Az eredmény típusa többnyire egyezik az *operandus* típusával, és az értéke az *operandus* értékén végrehajtva az *operátor*t. Például: *–változó*. Az eredmény típusa a *változó* típusa, és az eredmény értéke a *változó* értékének –1–szerese.

🕮 Problémák a kétoperandusos műveleteknél jelentkezhetnek, és hanyagoljuk el a továbbiakban az eredmény értékét! Ha kétoperandusos műveletnél a két *operandus* típusa azonos, akkor az eredmény típusa is a közös típus lesz. Ha a két *operandus* típusa eltér, akkor a fordító a rövidebb, pontatlanabb *operandus* értékét a hosszabb, pontosabb *operandus* típusára konvertálja, és csak ezután végzi el a műveletet. Az eredmény típusa természetesen a hosszabb, pontosabb típus. A `ft/244.5` osztásban a `ft` egész típusú és a `244.5` konstans lebegőpontos. A művelet elvégzése előtt a `ft` értékét lebegőpontossá alakítja a fordító, és csak ezután hajtja végre az osztást. Az eredmény tehát ugyancsak lebegőpontos lesz. Ezt implicit típuskonverziónak nevezik.

✍ Vegyük észre közben a valós konstans írásszabályát is! Elhagyható előjellel kezdődik, amikor is pozitív, és aztán az egész rész jegyei jönnek. Aztán egy tizedespont után a tört rész számjegyei következnek.

💣 A probléma a PASCAL programozó számára egészek osztásánál jelentkezik, hisz egészek osztásának eredménye is egész, és nincs semmiféle maradékmegőrzés, lebegőpontos átalakítás!

✍ Tételezzük fel, hogy 50 fillérrel csökkent az euró árfolyama! Alakítsuk csak át az `euro=ft/244.5` hozzárendelést `euro=ft/244`–re, s rögtön láthatjuk, hogy az eredményekben sehol sincs tört rész!

🕮 Felvetődik a kérdés, hogyan lehetne ilyenkor a helyes értéket meghatározni? A válasz: explicit típusmódosítás segítségével, melynek szintaktikai alakja:

(*típus*)*kifejezés*

Hatására a *kifejezés* értékét *típus* típusúvá alakítja a fordító. A konkrét esetben az osztás legalább egyik operandusát **float**–tá kéne módosítani, és ugye akkor a kétoperandusos műveletekre megismert szabály szerint a másik operandus értékét is azzá alakítaná a fordító a művelet tényleges elvégzése előtt, azaz:

```
euro = (float)ft / 244;
```

Megoldandó feladatok:

Készítsen programot, mely a képernyő 21 sorszor 21 oszlopos területén a csillag karakter felhasználásával megjelentet:

- Egy keresztet a 11. sor és 11. oszlop feltöltésével!
- A főátlót (bal felső sarokból a jobb alsóba menőt)!
- A mellékátlót (a másik átlót)!
- Egyszerre mindkét átlót, azaz egy X-et!

☞ A forint–euró átszámítási táblázatot elkészítő **PELDA2.C** megoldásunkkal az a „baj”, hogy túl sok változót használunk. Könnyen beláthatjuk, hogy az **ft**–től eltekintve a többi változó nem is az, hisz a program futása alatt nem változtatja meg egyik sem az értékét!

Készítsünk **PELDA3.C** néven egy jobb megoldást!

```
/* PELDA3.C: Forint-euró átszámítási táblázat */
#include <stdio.h>
void main(void){
  int ft;
  printf("%9s|%9s\n---------+---------\n",
         "Forint", "Euró");
  for(ft=100; ft<=1000; ft=ft+100)
    printf("%9d|%9.2f\n", ft, ft/244.5); }
```

PELDA3.C programunkban két új dolog látható. Az egyik a

for(<*init–kifejezés*>; <*kifejezés*>; <*léptető–kifejezés*>) *utasítás*

elöltesztelő, iteratív ciklusutasítás, melynek végrehajtása a következő lépések szerint történik meg:

1. A fordító végrehajtja az *init–kifejezés*t, ha van. Az elhagyhatóságot most is <> jelekkel szemléltetjük!
2. Kiértékeli a *kifejezés*t. Ha hamis (zérus), akkor vége a ciklusnak, s a **for**-t követő utasítás jön a programban. Látható, hogy a szintaktika szerint ez a *kifejezés* is elhagyható. Ilyenkor 1–nek (igaznak) minősül.

3. Ha a *kifejezés* igaz (nem zérus), akkor az *utasítás* végrehajtása jön. Az *utasítás* most is lehetne összetett utasítás is!
4. Végül az ugyancsak elhagyható *léptető–kifejezés*, majd újból a 2. pont következik.

Ha a **for** utasítást **while**-lal szeretnénk felírni, akkor azt így tehetjük meg:

<*init–kifejezés*>;
while(*kifejezés*) { *utasítás*; <*léptető–kifejezés*>; }

☞ A szintaktikai szabályt összefoglalva: a **for**-ból akár mindegyik kifejezés is elhagyható, de az első kettőt záró pontosvesszők nem!

A **PELDA3.C** programbeli másik új dolog az, hogy a **printf** aktuális paramétereként kifejezés is megadható.

☞ A **PELDA3.C** ugyan sokat rövidült, de ezzel a megoldással meg az a probléma, hogy tele van varázs–konstansokkal. Ha megváltoztatnánk átszámítási táblázatunkban a tartomány alsó, ill. felső határát, módosítanánk a lépésközt, vagy az euró árfolyamot, akkor ennek megfelelően át kellene írnunk varázs–konstansainkat is. Az ilyen átírogatás 8 soros programnál könnyen, és remélhetőleg hibamentesen megvalósítható. Belátható azonban, hogy nagyméretű, esetleg több forrásfájlból álló szoftver esetében, amikor is a varázs–konstansok rengeteg helyen előfordulhatnak, ez a módszer megbízhatatlan, vagy legalább is nagyon hibagyanús.

A C a probléma megoldására a szimbolikus konstansok, vagy más megnevezéssel: egyszerű makrók, használatát javasolja. A metódus a következő:

1. Definiálni kell a benne használatos szimbolikus állandókat egy helyen, egyszer, a forrásfájl elején.
2. Aztán a programban végig a konstansok helyett szisztematikusan a szimbolikus konstansokat kell használni.

A változtatás is nagyon egyszerűvé válik így:

- a megváltozott konstans értékét egy helyen át kell írni, s
- a többi felhasználása automatikusan módosul a következő fordításnál.

A szimbolikus állandó a

#define *azonosító helyettesítő–szöveg*

előfeldolgozó direktívával definiálható. A szimbolikus állandó – tulajdonképpen az *azonosító* – a direktíva helyétől a forrásszöveg végéig van érvényben. Az előfeldolgozó kihagyja a direktívát a forrásszövegből, majd végigmegy rajta, és az *azonosító* minden előfordulását *helyettesítő–szöveg*re cseréli.

Lássuk a „medvét”!

```
/* PELDA4.C: Forint-euró átszámítási táblázat */
#include <stdio.h>
#define ALSO 100       /* A tartomány alsó határa */
#define FELSO 1000     /* A tartomány felső értéke */
#define LEPES 100      /* A lépésköz */
#define ARFOLYAM 244.5 /* Ft/euró árfolyam */
void main(void){
  int ft;
  printf("%9s|%9s\n---------+---------\n",
         "Forint", "Euró");
  for(ft=ALSO; ft<=FELSO; ft=ft+LEPES)
    printf("%9d|%9.2f\n", ft, ft/ARFOLYAM); }
```

Szokás még – különösen több forrásmodulos esetben – a #**define** direktívákat (és még más dolgokat) külön fejfájlban elhelyezni, s aztán ezt minden forrásfájl elején #**include** direktívával bekapcsolni.

Készítsük csak el ezt a variációt is!

```
/* BEGEND.H: Fejfájl az átszámítási táblához */
#include <stdio.h>
#define ALSO 100       /* A tartomány alsó határa */
#define FELSO 1000     /* A tartomány felső értéke */
#define LEPES 100      /* A lépésköz */
#define ARFOLYAM 244.5 /* Ft/euró árfolyam */
#define begin {        /* {} helyett begin-end! */
#define end }
#define then           /* if utasításban then! */
#define LACI for       /* Kulcsszavak átdefiniálása
                          nem javasolt! */
```

```
/* PELDA5.C: Forint-euró átszámítási táblázat */
#include "BEGEND.H"
void main(void)
 begin
  int ft;
  printf("%9s|%9s\n---------+---------\n",
         "Forint", "Euró");
  LACI(ft=ALSO; ft<=FELSO; ft=ft+LEPES)
    printf("%9d|%9.2f\n", ft, ft/ARFOLYAM);
 end
```

✍ Vegyük észre, hogy az **#include** direktíva *fájlazonosító*ja nem <>–k, hanem ””–k között áll! Ennek hatására az előfeldolgozó a megadott azonosítójú fájlt először az aktuális mappában – abban a könyvtárban, ahol az a .C fájl is elhelyezkedik, melyben a **#include** direktíva volt – keresi. Ha itt nem találja, akkor továbbkeresi a programfejlesztő rendszerben beállított utakon.

3.6 Bemenet, kimenet

A kissé „lerágott csont” forint–euró átszámítási táblázatos példánkban nem volt bemenet. Megtanultuk már, hogy a szabvány bemenet és kimenet használatához be kell kapcsolni az **STDIO.H** fejfájlt:

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

Alapértelmezés szerint szabvány bemenet (**stdin**) a billentyűzet, és szabvány kimenet (**stdout**) a képernyő. A legtöbb operációs rendszerben azonban mindkettő átirányítható szövegfájlba is.

Egy karaktert olvas be a szabvány bemenetről az

int **getchar**(void);

függvény. Ezt aztán balról zérus feltöltéssel **int**–té típusmódosítja, és visszaadja a hívónak.

🕮 Azt, ahogyan az előbb a **getchar**–t leírtuk, függvény prototípusnak nevezik. A függvény prototípus teljes formai információt szolgáltat a szubrutinról, azaz rögzíti:

- a függvény visszatérési értékének típusát,
- a függvény nevét,
- paramétereinek számát, sorrendjét és típusát.

A **getchar** fájl végén, vagy hiba esetén **EOF**–ot szolgáltat.

✍ Az **EOF** az **STDIO.H** fejfájlban definiált szimbolikus állandó:

```
#define EOF (-1)
```

Tekintsük csak meg az **STDIO.H** fejfájlban! Nézegetés közben vegyük azt is észre, hogy a fejfájl tele van függvény prototípusokkal.

✍ Már csak az a kérdés maradt, hogy mi a fájlvég a szabvány bemeneten, ha az a billentyűzet? Egy operációs rendszertől függő billentyűkombináció: Ctrl+Z vagy Ctrl+D.

A paraméter karaktert kiviszi a szabvány kimenet aktuális pozíciójára az

int **putchar**(int *k*);

és sikeres esetben vissza is adja ezt az értéket. A hibát épp az jelzi, ha a **putchar** szolgáltatta érték eltér *k*–tól.

Készítsünk programot, ami a szabvány bemenetet átmásolja a szabvány kimenetre!

```
/* PELDA6.C: Bemenet másolása a kimenetre */
#include <stdio.h>
void main(void){
  int k;
  printf("Bemenet másolása a kimenetre:\n"
         "Gépeljen Ctrl+Z-ig sorokat!\n\n");
  k=getchar();
  while(k!=EOF){
    if(k!=putchar(k))
      printf("Hiba a kimeneten!\n");
    k=getchar(); } }
```

Fogalmazzuk meg minimális elvárásainkat egy programmal szemben!

A szoftver indulásakor jelezze ki, hogy mit csinál!

Ha valamilyen eredményt közöl, akkor azt lássa el tájékoztató szöveggel, mértékegységgel stb.!

Ha valamit bekér, akkor tájékoztasson róla, hogy mit kell megadni, milyen egységben stb.!

A bemenet ellenőrzendő! A hibás adat helyett – a hiba okát esetleg kijelezve – azonnal kérjen újat a program!

A <, <=, >, >= relációjelekről már szó volt! A C–ben != a nem egyenlő operátor és == az egyenlő műveleti jel. Az == és a != ráadásul a többi relációnál eggyel alacsonyabb prioritási szinten foglal helyet. Kifejezés kiértékelése közben előbb a magasabb prioritású műveletet végzi el a fordító, s csak aztán következik az alacsonyabb.

Vigyázat! Az egyenlő relációt az egymás után írt, két egyenlőség jel jelzi. Az egyetlen egyenlőség jel a hozzárendelés operátor!

A kétirányú szelekció szintaktikai alakja:

if(*kifejezés*) *utasítás1*
<**else** *utasítás2*>

Az elhagyhatóságot most is a <> jelzi. Ha a *kifejezés* igaz (nem zérus), akkor *utasítás1* végrehajtása következik. Ha a *kifejezés* hamis (zérus) és

van **else** rész, akkor az *utasítás2* következik. Mindkét utasítás összetett utasítás is lehet.

A **PELDA6.C** megoldásunk túlzottan PASCAL „ízű”. C–ben programunk utolsó 5 sorát így kéne megírni:

```
while((k=getchar())!=EOF)
  if(k!=putchar(k))
    printf("Hiba a kimeneten!\n");
```

A **while** kifejezése egy nem egyenlő reláció, melynek bal oldali operandusa egy külön zárójelben álló hozzárendelés. Előbb a hozzárendelés jobb oldalát kell kiértékelni. Lássuk csak sorban a kiértékelés lépéseit!

1. Meghívja a **getchar** függvényt a fordító.
2. A visszakapott értékkel felülírja **k** változó értékét.
3. A **getchar**–tól kapott értéket hasonlítja **EOF**–hoz.

A kifejezésből a hozzárendelés körüli külön zárójel nem hagyható el, mert a hozzárendelés alacsonyabb prioritású művelet a relációnál. Ha mégis elhagynánk, akkor a kiértékelés során a fordító:

1. Meghívná előbb a **getchar** függvényt.
2. A visszatérési értéket hasonlítaná **EOF**–hoz. Tehát kapna egy logikai igaz (1), vagy hamis (0) értéket!
3. A **k** változó felvenné ezt az 1, vagy 0 értéket.

Figyeljük meg a **PELDA6.C** futtatásakor, hogy a **getchar** a bemenetről olvasott karaktereket az operációs rendszer billentyűzet pufferéből kapja! Emlékezzünk csak vissza! A parancssorban a begépelt szöveget szerkeszthetjük mindaddig, míg Enter–t nem nyomunk. A billentyűzet pufferben levő karakterek tehát csak akkor állnak a **getchar** rendelkezésére, ha a felhasználó leütötte az Enter billentyűt.

Készítsünk programot, mely fájlvégig leszámlálja, hogy hány

- numerikus karakter,
- fehér karakter,
- más egyéb karakter és
- összesen hány karakter

érkezett a szabvány bemenetről!

Megoldásunkban az összes változó egész típusú. **k** tartalmazza a beolvasott karaktert. A **num**, a **feher** és az **egyeb** számlálók. Az algoritmus:

- Deklaráljuk a változókat, és az összes számlálót lássuk el zérus kezdőértékkel!
- Jelentessük meg a program címét, és tájékoztassunk a használatáról!
- Működtessük addig a ciklust, míg **EOF** nem érkezik a bemenetről!
- A ciklusmagban háromirányú szelekció segítségével el kell ágazni a három kategória felé, és ott meg kell növelni eggyel a megfelelő számlálót!
- A ciklus befejeződése után megjelentetendők a számlálók értékei megfelelő tájékoztató szövegekkel, és az is, hogy összesen hány karakter érkezett a bemenetről!

```
/* PELDA7.C: A bemenet karaktereinek
             leszámlálása kategóriánként */
#include <stdio.h>
void main(void){
  short k, num, feher, egyeb;
  num=feher=egyeb=0;
  printf("Bemeneti karakterek leszámlálása\n"
         "kategóriánként EOF-ig, vagy Ctrl+Z-ig.\n");
  while((k=getchar())!=EOF)
    if(k>='0'&&k<='9')++num;
    else if(k==' '||k=='\n'||k=='\t')++feher;
    else ++egyeb;
  printf("Karakter számok:\n"
         "----------------\n"
         "numerikus: %5hd\n"
         "fehér:     %5hd\n"
         "egyéb:     %5hd\n"
         "----------------\n"
         "össz: %10ld\n",
         num, feher, egyeb,
         (long)num+feher+egyeb); }
```

Pontosítani kell a deklarációs utasítás eddig megismert szintaktikáját!

<típusmódosítók> <alaptípus> azonosítólista;

Az elhagyható *alaptípus* alapértelmezés szerint **int**. Az ugyancsak elhagyható *típusmódosítók* az *alaptípus* valamilyen jellemzőjét változtatják meg. **int** típus esetén:

- Az egész alapértelmezés szerint előjeles (**signed**), és lehetne még előjeltelen (**unsigned**). A **signed** és az **unsigned** módosítók egymást kizáróak.
- Két, egymást kizáró hosszmódosítóval az egész belsőábrázolása

- bizonyosan 16 bites (**short**), ill.
- biztos 32 bites (**long**).

📖 Végül is a különféle **int** típusok méretei így összegezhetők:

short <= **int** <= **long**

✍ Vegyük észre, hogy az ismertetett szabályok szerint a **short**, a **short int** és a **signed short int** azonos típusok. A **short** és a **short int** írásmódnál figyelembe vettük, hogy **signed** az alapértelmezés. A **short** felírásakor még arra is tekintettel voltunk, hogy a meg nem adott *alaptípus* alapértelmezése **int**. Ugyanezek mondhatók el a **long**, a **long int** és a **signed long int** vonatkozásában is.

💣 Ugyan a szintaktika azt mutatja, de a deklarációs utasításban a *típusmódosítók* és az *alaptípus* egyszerre nem hagyhatók el!

Feltéve, hogy *a*, *b* és *c* balértékek, az

a=b=c=kifejezés

értelmezése megint abból fakad, hogy a hozzárendelés a C–ben operátor, azaz:

a=(*b*=(*c*=*kifejezés*))

📖 A fordító jobbról balra halad, azaz kiértékeli a *kifejezés*t, és visszafelé jövet beírja az eredményt a balértékekbe.

✍ A konstrukció hatására a fordító gyorsabb kódot is hoz létre. Ugyanis a

c=kifejezés;
b=kifejezés;
a=kifejezés;

írásmódnál háromszor kell kiértékelni ugyanazt a *kifejezés*t.

📖 C–ben a többágú (*N*) szelekcióra az egyik kódolási lehetőség:

if(*kifejezés1*)*utasítás1*
else if(*kifejezés2*)*utasítás2*
else if(*kifejezés3*)*utasítás3*
/* . . . */
else *utasításN*

Ha valamelyik **if** *kifejezés*e igaz (nem zérus) a konstrukcióban, akkor a vele azonos sorszámú *utasítás* végrehajtása következik, majd a konstrukciót követő utasítás jön. Ha minden *kifejezés* hamis (zérus), akkor viszont *utasításN* hajtandó végre.

✍ Fedezzük fel a karakter konstans írásszabályát: aposztrófok között karakter, vagy escape szekvencia.

🕮 A karakter konstans belsőábrázolása **int**, így az ASCII kód egész értéknek is minősül kifejezésekben.

✍ A **PELDA7.C**–ből látható, hogy a logikai és műveletet && jelöli, s a logikai vagy operátor a ||.

🕮 A kétoperandusos logikai operátorok prioritása alacsonyabb a relációkénál, és az && magasabb prioritású, mint a ||. Ha a kétoperandusos logikai művelet eredménye eldől a bal oldali operandus kiértékelésével, akkor a C bele sem kezd a másik operandus értékelésébe. Az és művelet eredménye eldőlt, ha a bal oldali operandus hamis. A vagy pedig akkor kész, ha az első operandus igaz.

A C–ben van inkrementálás (++) és dekrementálás (−−) egész értékekre. Mindkét művelet egyoperandusos, tehát nagyon magas prioritású. A ++ operandusa értékét eggyel növeli meg, s a −− pedig eggyel csökkenti, azaz:

++változó ≡ *változó*=*változó*+1

--változó ≡ *változó*=*változó*–1

💣 A problémák ott kezdődnek azonban, hogy mindkét művelet létezik előtag (prefix) és utótag (postfix) operátorként is!

🕮 Foglalkozzunk csak a ++ operátorral! A *++változó* és a *változó*++ hatására a *változó* értéke eggyel mindenképp megnövekedik. Kifejezés részeként előtag operátor esetén azonban a *változó* új értéke vesz részt a további kiértékelésben, míg utótag műveletnél a *változó* eredeti értéke számít be. Feltéve, hogy **a** és **b** egész típusú változók, és **b** értéke 6:

```
a = ++b;      /* a=7 és b=7 */
a = b++;      /* a=7 és b=8 */
```

✍ Figyeljünk fel rá, hogy a **PELDA7.C** utolsó **printf** utasításában hosszmódosítók állnak a **d** típuskarakterek előtt a formátumspecifikációkban! Látszik, hogy a **h** jelzi a **printf**–nek, hogy a formátumspecifikációhoz tartozó aktuális paraméter **short** típusú (2 bájtos), ill. **l** tudatja vele, hogy a hozzátartozó aktuális paraméter **long** (4 bájtos).

💣 A megfelelő hosszmódosítók megadása a formátumspecifikációkban elengedhetetlen, hisz nem mindegy, hogy a függvény a verem következő hány bájtját tekinti a formátumspecifikációhoz tartozónak!

✍ Vegyük azt is észre, hogy a típusokhoz a mezőszélességgel is felkészültünk: a maximális pozitív **short** érték bizonyosan elfér 5 pozíción, s **long** pedig 10–en!

✍ Látható még, hogy arra is vigyáztunk, hogy a három maximális **short** érték összege részeredményként se csonkuljon! Ezért az explicit **long**–gá módosítás a **printf** utolsó paraméterében:

```
(long)num+feher+egyeb
```

Megoldandó feladatok:

Készítsen programokat a **PELDA4.C** alapján a következőképpen:

- A forint 1000-től 100–ig csökkenjen 100–asával!
- Az euró növekedjék 1–től 10–ig egyesével!
- A forint 100–tól 2000–ig növekedjen 100–asával! Az eredményt a képernyőn fejléccél ellátva két oszlop párban oszlopfolytonosan haladva kell megjelentetni. A bal oldali oszlop pár 100–zal, a jobb oldali viszont 1100–zal kezdődjék!
- A feladat maradjon ugyanaz, mint az előbb, de a megjelentetés legyen sorfolytonos. A bal oldali oszlop pár kezdődjék 100–zal, a jobb oldali viszont 200–zal, s mindegyik oszlop párban 200 legyen a lépésköz!
- Maradva a sorfolytonos megjelentetésnél, kérjük be előbb a kijelzendő oszlop párok számát ellenőrzött inputtal! Az oszlop párok száma 1, 2, 3 vagy 4 lehet. A még kijelzendő felső érték ennek megfelelően 1000, 2000, 3000 vagy 4000. Az eredmény a képernyőn fejléccél ellátva az előírt számú oszlop párban jelenjen meg úgy, hogy 100 továbbra is a lépésköz!
- A forint 100–tól 10000–ig növekedjen 100–asával! A lista nem futhat el a képernyőről, azaz fejléccél ellátva lapozhatóan kell megjelentetni! Ez azt jelenti, hogy először kiíratjuk a lista egy képernyő lapnyi darabját, majd várunk egy gombnyomásra. A gomb leütésekor produkáljuk a lista következő lapját, és újból várunk egy gombnyomásra, és így tovább.
- Legyen ugyanaz a feladat, mint az előző pontban, de a lista a képernyőn fejléccél ellátva nem csak előre, hanem előre–hátra lapozhatóan jelenjen meg!

Készítsen programokat, melyek a szabvány bemenetet **EOF**–ig olvassák, és közben megállapítják, hogy:

- Hány sor volt a bemeneten? A bemenet karakterei közt a '\n'–eket kell leszámlálni. Az utolsó sor persze lehet, hogy nem '\n'–nel végződik, hanem **EOF**–fal.
- Hány szó volt a bemeneten? A szó nem fehér karakterekből áll. A szavakat viszont egymástól fehér karakterek választják el. Az utolsó szó lehet, hogy nem fehér karakterrel zárul, hanem **EOF**–fal.

3.7 Tömbök

Készítsünk programot, mely a szabvány bemenetet olvassa **EOF**-ig! Megállapítandó és kijelzendő, hogy hány A, B, C stb. karakter érkezett! A kis– és nagybetűk között nem teszünk különbséget! A betűkön kívüli többi karaktert tekintsük egy kategóriának, s ezek darabszámát is jelezzük ki!

Megoldásunkban az **elvalaszto** karakteres változó, a **k**, a **tobbi** és a **betu** viszont egész típusú. A **k** tartalmazza a beolvasott karaktert, és ciklusváltozói funkciókat is ellát. A **tobbi** és a **betu** számlálók. A **betu** annyi elemű tömb, mint ahány betű az angol ábécében van. A **tobbi** a betűkön kívüli többi karakter számlálója. Az **elvalaszto** karakteres változóra azért van szükség, mert az eredmény csak két oszlop páros listaként közölhető egy képernyőn. Az algoritmus:

- Deklaráljuk a változókat, és a tömböt! A számlálók nullázandók! Az **elvalaszto** induljon szóköz kezdőértékkel!
- Jelentessük meg a program címét, és tájékoztassunk a használatáról!
- Működtessük addig a ciklust, míg **EOF** nem érkezik a bemenetről!
- A ciklusmagban háromirányú szelekcióval el kell ágazni három kategória felé: nagybetű, kisbetű és más karakter. Megnövelendő eggyel természetesen a megfelelő számláló!
- A ciklus befejeződése után két oszlop páros táblázatban megjelentetendők a betűszámlálók értékei, és végül egy külön sorban a „többi karakter" kategória számlálója!

```
/* PELDA8.C: Betűszámlálás a bemeneten */
#include <stdio.h>
#define BETUK 26   /* Az angol ábécé betűszáma */
void main(void){
  char elvalaszto; /* Listelválasztó karakter. */
  int k,           /* Bemeneti kar. és ciklusváltozó. */
    tobbi,         /* Nem betűk számlálója. */
    betu[BETUK];   /* Betűszámlálók. */
  tobbi=0;         /* Kezdőérték adás. */
  for(k=0; k<BETUK; ++k) betu[k]=0;
```

```
elvalaszto=' ';
printf("Bemenet betűinek leszámlálása\n"
       "EOF-ig, vagy Ctrl+Z-ig.\n");
while((k=getchar())!=EOF)
  /* Nagybetűk: */
  if(k>='A'&&k<='Z')++betu[k-'A'];
  /* Kisbetűk: */
  else if(k>='a'&&k<='z')++betu[k-'a'];
  /* Más karakterek: */
  else ++tobbi;
/* Eredmények közlése: */
printf("\nBetű|Darab Betű|Darab\n"
       "----+----- ----+-----\n");
for(k=0; k<BETUK; ++k){
  printf("%4c|%5d%c", k+'A', betu[k], elvalaszto);
  if(elvalaszto==' ') elvalaszto='\n';
  else elvalaszto=' '; }
printf("\nTöbbi karakter: %5d\n", tobbi); }
```

📖 A **char** típusú változó egyetlen karakter tárolására alkalmas. A **char** ugyanakkor 8 bites, alapértelmezés szerint előjeles (**signed**), fixpontos belsőábrázolású egész típus is $0111\ 1111_2 = 2^7 - 1 = 127$ és $1000\ 0000_2 = -2^7 = -128$ ábrázolási határokkal. Az **unsigned char** 0 és 255 közötti ábrázolási lehetőségekkel rendelkezik.

A legtöbb programfejlesztő rendszerben az **unsigned char** alapértelmezésként is beállítható karakter típus.

A **PELDA8.C**–ből kitűnően látszik, hogy a tömb definíciója

típus tömbazonosító[*méret*]*;*

alakú. Pontosabban a deklarációs utasítás azonosítólistája nem csak egyszerű változók azonosítóiból állhat, hanem *tömbazonosító*[*méret*] konstrukciók is lehetnek köztük. A tömbdefinícióban a *méret* pozitív, egész értékű állandó kifejezés, és a tömb elemszámát határozza meg.

📖 Állandó kifejezés az, aminek fordítási időben kiszámítható az értéke.

A tömb egy elemének helyfoglalása típusától függ. Az egész tömb a memóriában összesen

sizeof(*tömbazonosító*) ≡ *méret****sizeof**(*típus*)

bájtot igényel. Például 16 bites **int**–et feltételezve a **sizeof**(**betu**) ≡ 26***sizeof**(**int**) pontosan 52.

📖 A magas prioritású, egyoperandusos **sizeof** operátor megadja a mögötte zárójelben álló *objektum*, vagy *típus* által elfoglalt bájtok számát.

A tömb egy elemére való hivatkozást indexes változónak is nevezik és szintaktikailag a következő:

tömbazonosító[*index*]

ahol az *index* nem negatív értékű egész kifejezés

0 <= *index* <= *méret*–1

értékhatárokkal.

💣 A tömbindexelés C–ben mindig zérustól indul, és a legnagyobb még létező indexérték a *méret*–1! Például a **betu** tömbnek létezik **betu**[0], **betu**[1], **betu**[2], és végül **betu**[BETUK–1] eleme, és ezek így helyezkednek el a memóriában:

betu[0]	**betu**[1]	**betu**[2]	**. . .**	**betu**[24]	**betu**[25]

✍ Vegyük észre, hogy a **betu**[0]–ban a program az A, a **betu**[1]–ben a B, …, és a **betu**[25]–ben a Z karaktereket számlálja! Tételezzük fel, hogy **k** értéke 68! Ez ugyebár a D betű ASCII kódja. Ilyenkor a **betu**[**k**–'A'] számláló nő eggyel. Az A ASCII kódja 65. Tehát **betu**[68–65]–ről, azaz **betu**[3] növeléséről van szó!

✍ Figyeljünk fel még arra, hogy az eredményeket közlő ciklusbeli **printf**–ben a **k**+'A' egész kifejezés értékét %c formátumspecifikációval jelentetjük meg, azaz rendre 65–öt, 66–ot, 67–et stb. íratunk ki karakteresen, tehát A–t, B–t, C–t stb. látunk majd.

✍ Lássuk még be, hogy az **elvalaszto** változó értéke szóköz és soremelés karakter közt váltakozik, s így két betű–darab pár képes megjelenni egy sorban. Tehát az **elvalaszto** változó segítségével produkáljuk a két oszlop páros eredménylistát.

👍 Listázni csak azt érdemes, ami valamilyen információt hordoz!

Tehát a zérus darabszámú betűk kijelzése teljesen felesleges! Magyarán a **for** ciklusbeli **printf**–et így kéne módosítani:

```
if(betu[k]>0) printf("%4c|%5d%c", k+'A',
                     betu[k], elvalaszto);
```

Megoldandó feladatok:

Fokozza úgy a **PELDA8.C**–ben megoldott feladatot, hogy megszámlálja a magyar ékezetes kis– és nagybetűket is!

Készítsen programot, mely a szabvány bemenetet **EOF**–ig olvassa! Számlálja meg és jelezze ki, hogy hány 0, 1, 2 stb. karakter érkezik! A

nem numerikus karaktereket tekintse egy kategóriának, és ezek számát is közölje!

3.8 Függvények

A függvényeket többféleképpen csoportosíthatnánk, de a legpraktikusabb úgy, hogy:

- Vannak előre megírtak. Könyvtárakban (**.LIB**), vagy tárgymodulokban (**.OBJ**) találhatók, s a kapcsoló-szerkesztő kapcsolja be őket a végrehajtható fájlba. Például: a **printf**, a **getchar**, a **putchar**, vagy a **main** stb. Minden végrehajtható programban kell lennie egy függvénynek, az indító programnak (a **main**-nek), mely az egész program belépési pontját képezi.
- Mi írjuk őket. Forrásfájlokban helyezkednek el, s kódjukat a fordító generálja.

A nyelv központi eleme a függvény. A más programozási nyelvekben szokásos eljárás (procedure) itt explicit módon nem létezik, mert a C szellemében az egy olyan függvény, aminek nincs visszaadott értéke:

void *eljárás*();

Jelezzük ki egy táblázatban az 1001 és 1010 közötti egész számok köbét!

```
/* PELDA9.C: Köbtáblázat */
#include <stdio.h>
#define TOL 1001            /* A tartomány kezdete. */
#define IG 1010             /* A tartomány vége. */
long kob(int);              /* Függvény prototípus. */
void main(void){
  int i;
  printf(" Szám|%11s\n-----+-----------\n", "Köbe");
  for(i=TOL; i<=IG; ++i)    /* Függvényhívás. */
    printf("%5d|%11ld\n", i, kob(i)); }
long kob(int a){            /* Függvénydefiníció. */
  return (long)a*a*a; }
```

A függvénydefiníció és a függvényhívás fogalmával megismerkedtünk már a **Kapcsoló–szerkesztés** fejezetben. A függvénydefinícióban van meg a függvény teste, azaz az a kód, amit a függvény meghívásakor végrehajt a processzor.

Egy függvényre a programban csak egyetlen definíció létezhet, és ennek nem mondhatnak ellent a prototípusok (deklarációk)!

A függvénydefinícióban előírt visszaadott érték típusának egyeznie kell ebből következőleg a programban bárhol előforduló, e függvényre vonat-

kozó prototípusokban (deklarációkban) megadott visszatérési érték típussal. A meghívott függvény akkor ad vissza értéket a hívó függvénynek a hívás pontjára, ha a processzor *kifejezés*sel ellátott **return** utasítást hajt végre benne. A „valamit” szolgáltató függvényben tehát lennie kell legalább egy **return** *kifejezés*; utasításnak, és rá is kell, hogy kerüljön a vezérlés. A visszaadott érték meghatározatlan, ha a processzor nem hajt végre **return** utasítást, vagy a **return** utasításhoz nem tartozott *kifejezés*.

💣 A visszaadott érték típusa bármi lehet végül is eltekintve a tömbtől és a függvénytől. Lehet valamilyen alaptípus, de el is hagyható, amikor is az alapértelmezés lesz érvényben, ami viszont **int**.

Nézzük a **return** szintaktikáját!

return <*kifejezés*>;

A fordító kiértékeli a *kifejezés*t. Ha a függvény visszatérési típusa *típus*, akkor a *kifejezés* típusának is ennek kell lennie, vagy implicit konverzióval ilyen típusúvá alakítja a *kifejezés* értékét a fordító, és csak azután adja vissza.

✍ Lássuk be, hogy a **PELDA9.C**–beli **return**–ben az explicit (**long**) típusmódosítás nem azért van, hogy megtakarítsuk a *kifejezés* értékének visszaadás előtti implicit konverzióját! Az igazi ok az, hogy egy 16 bites **int** köbe nem biztos, hogy elfér az **int**–ben! Gondoljunk 1000 köbére, ami 1000000000! Ez jóval meghaladja a 32767–es felsőábrázolási korlátot.

💣 C–ben az egész típusok területén nincs sem túlcsordulás, sem alulcsordulás! Pontosabban ami túlcsordul, vagy alulcsordul, az mindenféle üzenet nélkül elveszik.

A függvényhívás átruházza a vezérlést a hívó függvényből a hívottba úgy, hogy az aktuális paramétereket is átadja – ha vannak – érték szerint. A vezérlést a függvénytest első végrehajtható utasítása kapja meg. **void** visszatérésű függvény blokkjában aztán a végrehajtás addig folytatódik, míg *kifejezés* nélküli **return** utasítás nem következik, vagy a függvény blokkját záró }-re nem kerül a vezérlés. Ezután a hívási ponttól folytatódik a végrehajtás.

✍ Vegyük észre, hogy a **return** utasítás szintaktikájában az elhagyható *kifejezés* a paraméter nélküli **return**–t kívánta jelölni!

Belátható, hogy a függvény prototípusnak mindig meg kell előznie a hívást a forrásszövegben. A fordító így tisztában van a hívás helyén a függvény paramétereinek számával, sorrendjével és típusával, ill. ismeri a függvény visszatérési értékének típusát is.

A fordító a prototípus ismeretében implicit típuskonverziót is végrehajt az aktuális paraméter értékén a függvénynek történő átadás előtt, ha az aktuális paraméter típusa eltérő.

💣 Ha nincs prototípus, akkor nincs implicit konverzió, és csak a „csoda" tudja, hogy mi történik az átadott nem megfelelő típusú értékkel. Például a **kob**(3.0) hívás eredménye zérus, ami remélhetőleg kellően szemlélteti a prototípus megadásának szükségességét.

💣 Ha nincs prototípus, akkor a fordító azt feltételezi (tehát olyan hívási kódot generál), hogy a függvénynek az alapértelmezés miatt **int** visszaadott értéke van. Ez ugyebár eléggé érdekes eredményre vezet **void**, vagy lebegőpontos visszatérésű függvények esetében.

🕮 A nem **int** visszaadott értékű függvényt legalább deklaráni kell a hívó függvényben!

A függvénydeklaráció bemutatásához átírjuk a **PELDA9.C**–t:

```
/* PELDA9.C: Köbtáblázat */
#include <stdio.h>
#define TOL 1001          /* A tartomány kezdete. */
#define IG 1010           /* A tartomány vége. */
void main(void){
  int i;
  long kob();             /* Függvénydeklaráció. */
  printf(" Szám|%11s\n-----+-----------\n", "Köbe");
  for(i=TOL; i<=IG; ++i)  /* Függvényhívás: */
    printf("%5d|%11ld\n", i, kob(i)); }
long kob(int a){          /* Függvénydefiníció. */
  return (long)a*a*a; }
```

✍ Vegyük észre rögtön, hogy deklarációs utasításunk szintaktikája ismét módosult! Az *azonosítólista* nem csak egyszerű változók *azonosító*iból és *tömbazonosító*[*méret*] konstrukciókból állhat, hanem tartalmazhat

függvénynév()

alakzatokat is.

Természetesen a teljes függvény prototípus is beírható a deklarációs utasításba,

```
long kob(int);            /* Függvénydeklaráció. */
```

de ilyenkor a függvény prototípus csak ebben a blokkban lesz érvényben.

✍ Lássuk be, hogy az utóbbi módszer nem ajánlható olyan több függvénydefinícióból álló forrásfájlra, ahol a kérdéses függvényt több helyről is meghívják! Sokkal egyszerűbb a forrásszöveg elején megadni egyszer a

prototípust, mint minden őt hívó függvényben külön deklarálni a függvényt.

📖 A függvény definíciója prototípusnak is minősül, ha megelőzi a forrásszövegben a függvényhívást.

```
/* PELDA9.C: Köbtáblázat */
#include <stdio.h>
#define TOL 1001          /* A tartomány kezdete. */
#define IG 1010           /* A tartomány vége. */
long kob(int a){          /* Függvénydefiníció. */
  return (long)a*a*a; }
void main(void){
  int i;
  printf(" Szám|%11s\n-----+-----------\n", "Köbe");
  for(i=TOL; i<=IG; ++i)  /* Függvényhívás: */
    printf("%5d|%11ld\n", i, kob(i)); }
```

💣 C–ben tilos függvénydefiníción belül egy másikat kezdeni, azaz a függvénydefiníciók nem ágyazhatók egymásba!

📖 Ugyan a **Táblázat készítése** fejezetben már rögzítettük a függvény szerkezetét, vagyis a blokkszerkezetet, de itt újra kihangsúlyozzuk, hogy a függvénydefinícióban

- előbb a deklarációs utasítások jönnek, s
- a végrehajtható utasítások csak ezután következnek, és
- a két rész nem keveredhet egymással.

✍ Ebben a fejezetben csak az érték szerinti hívásról szóltunk, vagyis amikor a formális paraméterek értékét kapja meg a meghívott függvény. Van természetesen név (cím) szerinti hívás is a C–ben, de ezt most még nem tárgyaljuk!

3.9 Prodzsekt

Ha a végrehajtható program forrásszövegét témánként, vagy funkciónként külön–külön forrásfájlokban kívánjuk elhelyezni, akkor C–s programfejlesztő rendszerekben ennek semmiféle akadály sincs. Be kell azonban tartani a következő szabályokat:

- Egy és csak egy forrásmodulban szerepelnie kell az indító programnak (**main**).
- Prodzsektfájlt kell készíteni, melyben felsorolandók a program teljes szövegét alkotó forrásfájlok.

Szedjük szét két forrásmodulra: **FOPROG.C**–re és **FUGGV.C**–re, a **PELDA9.C** programunkat!

```
/* FOPROG.C: Köbtáblázat */
#include <stdio.h>
#define TOL 1001          /* A tartomány kezdete. */
#define IG 1010           /* A tartomány vége. */
long kob(int);            /* Függvény prototípus. */
void main(void){
  int i;
  printf(" Szám|%11s\n-----+-----------\n", "Köbe");
  for(i=TOL; i<=IG; ++i)  /* Függvényhívás: */
    printf("%5d|%11ld\n", i, kob(i)); }
```

```
/* FUGGV.C: A függvénydefiníció. */
long kob(int a){return (long)a*a*a; }
```

Hozzunk létre egy új prodzsektet! Soroljuk fel benne, vagy szúrjuk bele a két forrásfájlt, és mentsük el, mondjuk, **PRODZSI** fájlazonosítóval! A prodzsektfájl névadásánál csak arra vigyázzunk, hogy egyetlen benne felsorolt fájl azonosítójával se egyezzen meg a neve!

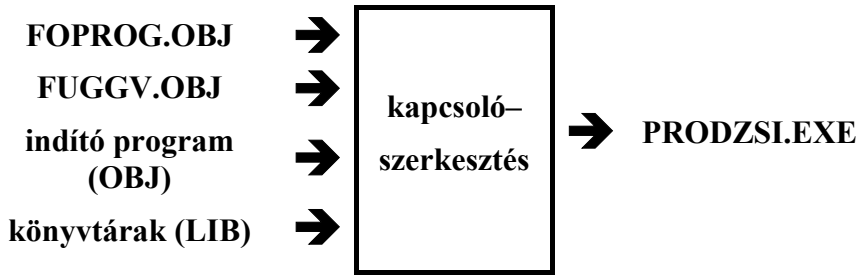
Azért nem konkretizáljuk a prodzsektfájl kiterjesztését, mert az programfejlesztő rendszerenként más–más lehet!

A programfejlesztő rendszerben kell lennie olyan menüpontoknak, melyekkel új prodzsektet hozhatunk létre, meglévőt tölthetünk be, nyithatunk meg, menthetünk el, törölhetünk, zárhatunk le stb.

Betöltött, vagy megnyitott prodzsekt esetén azonban a fejlesztő rendszer mindaddig a prodzsekt fordításával, kapcsoló–szerkesztésével és futtatásával foglalkozik, míg nem töröljük, nem zárjuk be. Akármilyen más forrásfájlokat is nyitogatnánk meg különféle ablakokban, a programfejlesztő rendszer az aktuális prodzsekt bezárásáig nem ezek fordításával, szerkesztésével, vagy futtatásával foglalkozik.

Lássuk a prodzsekt fordítását és kapcsoló–szerkesztését!

4. ábra: A PRODZSI prodzsekt fordítása

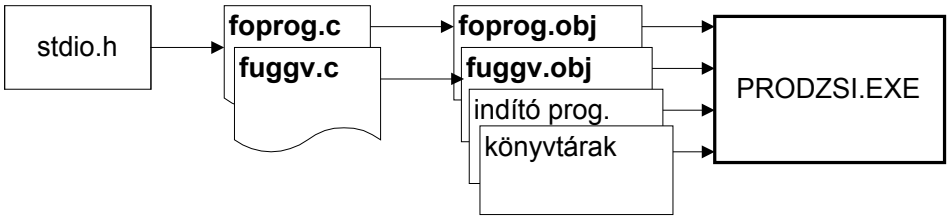


5. ábra: A PRODZSI prodzsekt kapcsoló–szerkesztése

✍ Fedezzük fel, hogy a végrehajtható fájl a prodzsekt nevét kapja meg!

🕮 A prodzsektet alkotó fájlok között implicit függőség van. Ez azt jelenti, hogy a prodzsekt futtatásakor csak akkor történik meg a tárgymodul alakjában is rendelkezésre álló forrásfájl fordítása, ha a forrásfájl utolsó módosításnak ideje (dátuma és időpontja) későbbi, mint a vonatkozó tárgymodulé. A kapcsoló–szerkesztés végrehajtásához az szükséges, hogy a tárgymodulok, ill. a könyvtárak valamelyikének ideje későbbi legyen a végrehajtható fájlénál. Az implicit függőség fennáll a forrásfájl és a bele **#include** direktívával bekapcsolt fájlok között is. A tárgymodult akkor is újra kell fordítani, ha valamelyik forrásfájlba behozott fájl ideje későbbi a tárgymodulénál.

✍ Bizonyos programfejlesztő rendszereknél előfordulhat, hogy az implicit függőségi mechanizmust úgy kell külön aktiválni (menüpont), ill. hogy a forrásfájlok, és a beléjük behozott fájlok közti függőséget explicit módon kell biztosítani.



6. ábra: Implicit függőség a PRODZSI prodzsektnél

🕮 A prodzsektfájlban a forrásmodulokon kívül megadhatók tárgymodulok (**OBJ**) és könyvtárak (**LIB**) fájlazonosítói is. A kapcsoló–szerkesztő a tárgymodulokat beszerkeszti a végrehajtható fájlba. A könyvtárakban pedig függvények tárgykódjait fogja keresni.

A prodzsektfájlban tulajdonképpen a gyári indító program és a szabvány könyvtárak is kicserélhetőek, de ennek pontos megvalósítása már „igazán” a programfejlesztő rendszertől függ.

3.10 Karaktertömb és karakterlánc

A karaktertömbök definíciója a **Tömbök** fejezetben ismertetettek szerint:

char *tömbazonosító*[*méret*];

Az egész tömb helyfoglalása:

*méret****sizeof**(**char**) ≡ *méret*

bájt. A tömbindexelés ebben az esetben is zérustól indul és *méret*–1–ig tart.

🕮 A C–ben nincs külön karakterlánc (sztring) adattípus. A karakterláncokat a fordítónak és a programozónak karaktertömbökben kell elhelyeznie. A karakterlánc végét az őt tartalmazó tömbben egy zérusértékű bájttal (’\0’) kell jelezni. Például a ”Karakterlánc” karakterláncot így kell letárolni a **tomb** karaktertömbben:

tomb	’K’	’a’	’r’	’a’	’k’	’t’	’e’	’r’	’l’	’á’	’n’	’c’	’\0’
	0	1	2	3	4	5	6	7	8	9	10	11	12

✍ Vegyük észre, hogy a karakterlánc első jele a **tomb**[0]–ban, a második a **tomb**[1]–ben, s a legutolsó a **tomb**[11]–ben helyezkedik el, és az ezt követő **tomb**[12] tartalmazza a lánczáró zérust!

✍ Figyeljünk fel arra is, hogy a karakterlánc hossza (12) megegyezik a lezáró ’\0’ karaktert magába foglaló tömbelem indexével!

✍ Fedezzük még rögtön fel, hogy a zérus egész konstans (0) és a lánczáró ’\0’ karakter értéke ugyanaz: zérus **int** típusban! Hiszen a karakter konstans belsőábrázolása **int**.

💣 A C–ben nincs külön karakterlánc adattípus, s ebből következőleg nem léteznek olyan sztring műveletek sem, mint a karakterláncok

- egyesítése,
- összehasonlítása,
- hozzárendelése stb.

Ezeket a műveleteket bájtról–bájtra haladva kell kódolni, vagy függvényt kell írni rájuk, mint ahogyan azt a következő példában bemutatjuk.

Készítsen programot, mely neveket olvas a szabvány bemenetről **EOF**–ig vagy üres sorig! Megállapítandó egy fordítási időben megadott névről, hogy hányszor fordult elő a bemeneten! A feladat megoldásához készítendő

- Egy **int strcmp**(**char** *s1*[], **char** *s2*[]) függvény, mely összehasonlítja két karakterlánc paraméterét! Ha egyeznek, zérust ad vissza. Ha az első hátrébb van a névsorban (nagyobb), akkor pozitív, egyébként meg negatív értéket szolgáltat.
- Egy **int getline**(**char** *s*[], **int** *n*) függvény, mely behoz a szabvány bemenetről egy sort! A sor karaktereit rendre elhelyezi az *s* karaktertömbben. A befejező soremelés karaktert nem viszi át a tömbbe, hanem helyette lánczáró '\0'–t ír a karakterlánc végére. A **getline** második paramétere az *s* karaktertömb méreténél eggyel kisebb egész érték, azaz a lánczáró zérus nélkül legfeljebb *n* karaktert tárol a tömbben a függvény. A **getline** visszatérési értéke az *s* tömbben végül is elhelyezett karakterlánc hossza.

```
/* PELDA10.C: Névszámlálás */
#include <stdio.h>
#define NEV "Jani" /* A számlált név. */
#define MAX 29     /* A bemeneti sor maximális mérete.
                      Most egyben a leghosszabb név is. */
int getline(char s[],int n);  /* Függvény prototípusok. */
int strcmp(char s1[], char s2[]);
void main(void){
  int db;                     /* Névszámláló. */
  char s[MAX+1];              /* Az aktuális név. */
  db=0;                       /* A számláló nullázása. */
  printf("A(z) %s név leszámlálása a bemeneten.\nAdjon\
 meg soronként egy nevet!\nProgramvég: üres sor.\n",NEV);
  /* Sorok olvasása üres sorig a bemenetről: */
  while(getline(s,MAX)>0)
    /* Ha a sor épp a NEV: */
    if(strcmp(s,NEV)==0) ++db;
  /* Az eredmény közlése: */
  printf("A nevek közt %d darab %s volt.\n",db,NEV); }
int strcmp(char s1[], char s2[]){
  int i;
  for(i=0; s1[i]!=0&&s1[i]==s2[i]; ++i);
  return(s1[i]-s2[i]);}
int getline(char s[],int n){
  int c,i;
  for(i=0;i<n&&(c=getchar())!=EOF&&c!='\n';++i) s[i]=c;
  s[i]='\0';
  return(i); }
```

🕮 Ha a forráskód egy sorát lezáró soremelés karaktert közvetlenül \ jel előzi meg, akkor mindkét karaktert elveti a fordító, s a két fizikai sort egyesíti, és egynek tekinti. Az ilyen értelemben egyesített sorokat már a másodiktól kezdve folytatássornak nevezik.

✍ A **main** első **printf**–jét folytatással készítettük el.

✍ A két elkészített függvény prototípusából és definíciójából vegyük észre, hogy a formális paraméter karaktertömb (és persze más típusú tömb is) méret nélküli!

Az **strcmp** megírásakor abból indultunk ki, hogy két karakterlánc akkor egyenlő egymással, ha ugyanazon pozícióikon azonos karakterek állnak, és ráadásul a lánczáró zérus is ugyanott helyezkedik el. Az **i** ciklusváltozót zérusról indítva, s egyesével haladva, végigindexeljük az **s1** karaktertömböt egészen a lánczáró nulláig (**s1**[**i**]!=0) akkor, ha közben minden **i**–re az **s1**[**i**]==**s2**[**i**] is teljesült. Ebből a ciklusból tehát csak két esetben van kilépés:

- Ha elértünk **s1** karaktertömb lánczáró zérusáig (**s1**[**i**]==0).
- Ha a két karakterlánc egyazon pozícióján nem egyforma érték áll (**s1**[**i**]!=**s2**[**i**]).

A visszaadott **s1**[**i**]–**s2**[**i**] különbség csak akkor:

- Zérus, ha **s1**[**i**] zérus és **s2**[**i**] is az, azaz a két karakterlánc egyenlő.
- Negatív, ha **s1** kisebb (előbbre van a névsorban), mint **s2**.
- Pozitív, ha **s1** nagyobb (hátrébb van a névsorban), mint **s2**.

🕮 Az **strcmp**–t már megírták. Tárgykódja benne van a szabvány könyvtárban. Nem fáradtunk azonban feleslegesen, mert a szabvány könyvtári változat ugyanúgy funkcionál, mint a **PELDA10.C**–beli. Használatához azonban be kell kapcsolni a prototípusát tartalmazó **STRING.H** fejfájlt.

✍ Írjuk be a **PELDA10.C** #**include** direktívája után a

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

, és töröljük ki a forrásszövegből az **strcmp** prototípusát és definícióját! Végül próbáljuk ki!

💣 Feltéve, hogy **s1** és **s2** karaktertömbök, lássuk be, hogy C programban, ugyan szintaktikailag nem helytelen, semmi értelme sincs az ilyen kódolásnak, hogy:

```
s1==s2, s1!=s2, s1>=s2, s1>s2, s1<=s2, s1<s2
```

E módon ugyanis **s1** és **s2** memóriabeli elhelyezkedését hasonlítjuk össze, azaz bizton állíthatjuk, hogy az **s1**==**s2** mindig hamis, és az **s1**!=**s2** pedig mindig igaz.

A **getline** ciklusváltozója ugyancsak zérusról indul, és a ciklus végéig egyesével halad. A ciklusmagból látszik, hogy a szabvány bemenetről érkező karakterekkel tölti fel az **s** karaktertömböt. A ciklus akkor áll le,

- ha az **s** tömb kimerült, és nem tölthető tovább (**i>=n**), vagy
- ha a beolvasott karakter **EOF** jelzés (**c==EOF**), vagy
- ha a bejövő karakter soremelés (**c**==’\n’).

Ezután kitesszük a tömb **i**–ik elemére a lánczáró zérust, s visszaadjuk **i**–t, azaz a behozott karakterlánc hosszát.

getline függvényünk jó próbálkozás az egy sor behozatalára a szabvány bemenetről, de van néhány problémája, ha a bemenet a billentyűzet:

1. A bemeneti pufferben levő karakterek csak soremelés (Enter) után állnak rendelkezésére, s addig nem.
2. Az **EOF** karakter (Ctrl+Z) érkezése nem jelenti tulajdonképpen a bemenet végét, mert utána még egy soremelést is végre kell hajtani, hogy észlelni tudjuk.
3. Ha a bemeneti pufferben az Enter megnyomásakor **n**–nél több karakter van, akkor a puffer (az egy sor) csak több **getline** hívással olvasható ki.

✍ Írjuk csak át a **PELDA10.C main**–jében a

```
while(getline(s,MAX)>0)
```

sort a következőre

```
while(getline(s,4)>0)
```

, és a program indítása után gépeljük be a

```
JenőJaniJojóJani↵
↵
```

sorokat!

Az első két probléma az operációs rendszer viselkedése miatt van, s jelenleg sajnos nem tudunk rajta segíteni. A második gondhoz még azt is hozzá kell fűznünk, hogy fájlvéget a billentyűzet szabvány bemeneten a **getline**–nak csak üres sor megadásával tudunk előidézni.

A harmadik probléma viszont könnyedén orvosolható, csak ki kell olvasni a bemeneti puffert végig a **getline**–ból való visszatérés előtt. Tehát:

```
int getline(char s[],int n){
  int c,i;
```

```
for(i=0;i<n&&(c=getchar())!=EOF&&c!='\n';++i) s[i]=c;
s[i]='\0';
while(c!=EOF&&c!='\n') c=getchar();
return(i); }
```

Megoldandó feladatok:

Készítse el a következő függvényeket, és próbálja is ki őket egy rövid programmal!

- A **void strcopy**(**char** *cél*[], **char** *forrás*[]) átmásolja a *cél* karaktertömbbe a *forrás* karakterláncot a lezáró nullájával egyetemben.
- A **void strct**(**char** *s1*[], **char** *s2*[]) egyesíti *s1*–be a két paraméter karakterláncot. Javasoljuk, hogy az algoritmus indexeljen előre *s1* lánczáró zérusáig, s ettől kezdve csak ide kell másolni *s2*–t!
- Az **unsigned strlen**(**char** *s*[]) visszaadja az *s* karakterlánc hosszát. Emlékezzünk rá, hogy a lánczáró zérus indexe a karaktertömbben egyben a karakterlánc hossza is!
- A **void strrv**(**char** *s*[]) megfordítja a saját helyén a paraméter karakterláncot. Például a ”Jani”–ból ”inaJ”–nak kell születnie.

Írjon programot, mely a **getline** segítségével sorokat olvas üres sorig a szabvány bemenetről, és megkeresi a legrövidebbet, s persze ki is jelzi a hosszával együtt! Javasoljuk, hogy a pillanatnyi minimum mentéséhez használja fel az **strcopy** függvényt!

3.11 Lokális, globális és belső, külső változók

Ha alaposan áttanulmányozzuk a **PELDA10.C**–t, akkor észrevehetjük, hogy mind az **strcmp**–ben, mind a **getline**–ban van egy–egy, **int** típusú **i** változó. Feltehetnénk azt a kérdést, hogy mi közük van egymáshoz? A rövid válasz nagyon egyszerű: semmi. A hosszabb viszont kicsit bonyolultabb:

- Mindkét **i** változó lokális a saját függvényblokkjára.
- A lokális változó hatásköre az a blokk, amiben definiálták.
- A lokális változó élettartama az az idő, míg saját függvényblokkja aktív, azaz benne van a vezérlés.

Tehát a két **i** változónak a névegyezésen túl nincs semmi köze sem egymáshoz.

📖 A változó hatásköre az a programterület, ahol érvényesen hivatkozni lehet rá, el lehet érni. Nevezik ezt ezért érvényességi tartománynak is.

✍ A lokális változó hatásköre az a blokk, és annak minden beágyazott blokkja, amiben definiálták. A blokkon belülisége miatt a lokális változót belső változónak is nevezik.

🕮 A változó élettartama az az időszak, amíg memóriát foglal.

✍ A lokális változó akkor jön létre (rendszerint a veremben), amikor a vezérlés bejut az őt definiáló blokkba. Ha a vezérlés kikerül onnét, akkor a memóriafoglalása megszűnik, helye felszabadul (a veremmutató helyrejön).

A függvényekben, így a **main**–ben is, definiált változók mind lokálisak arra a függvényre, amelyben deklarálták őket. Csak az adott függvényen belül lehet hozzájuk férni. Ez a lokális hatáskör. Futási időben a függvénybe való belépéskor jönnek létre, és kilépéskor meg is semmisülnek. Ez a lokális élettartam. A lokális változó kezdőértéke ebből következőleg „szemét”. Pontosabban az a bitkombináció, ami azokban a memóriabájtokban volt, amit most létrejövetelekor a rendszer rendelkezésére bocsátott. Tehát a belső változónak nincs alapértelmezett kezdőértéke. Az alapértelmezett kezdőértéket implicit kezdőértéknek is szokták nevezni.

✍ Mi dönti el, hogy a belső változó melyik blokkra lesz lokális? Nyilvánvalóan az, hogy az őt definiáló deklarációs utasítást melyik blokk elején helyezik el.

🕮 Tisztázzuk valamennyire a változó deklarációja és definíciója közti különbséget! Mindkét deklarációs utasításban megadják a változó néhány tulajdonságát, de memóriafoglalásra csak definíció esetén kerül sor.

🕮 A lokális változó az előzőekben ismertetetteken kívül **auto** tárolási osztályú is.

Egészítsük ki újból a deklarációs utasítás szintaktikáját!

<tárolási–osztály><típusmódosítók> <alaptípus> azonosítólista;

A *tárolási–osztály* a következő kulcsszavak egyike lehet: **auto**, **register**, **static**, vagy **extern**. E fejezet keretében nem foglalkozunk a **register** és a **static** tárolási osztállyal!

✍ A szintaktikát betartva most felírható, hogy

```
auto i;
```

, ami az **auto signed int** típusú, **i** azonosítójú változó definíciója.

✍ Hogyan lehet **auto** tárolási osztályú a lokális változó? Nyilván úgy, hogy a lokális helyzetű deklarációs utasításban az **auto** tárolási osztály

alapértelmezés. Az **auto** kulcsszó kiírása ezekben az utasításokban teljesen felesleges.

Foglaljuk össze a lokális változóval, vagy más néven belső változóval, kapcsolatos ismereteinket!

Neve:	Bármilyen azonosító.
Típusa:	Valamilyen típus.
Hatásköre:	Az a blokk, ahol definiálták.
Élettartama:	Amíg a blokk aktív, azaz a vezérlés benne van.
Alapértelmezett tárolási osztálya:	**auto**
Alapértelmezett kezdőértéke:	Nincs.
Deklarációs utasításának helye:	Annak a blokknak a deklarációs része, ahol a változót használni kívánjuk.
Memóriafoglalás:	Futási időben, többnyire a veremben.

🕮 Meg kell említeni, hogy a függvények formális paraméterei is lokális változóknak minősülnek, de

- deklarációjuk a függvénydefiníció fej részében és nem a blokkjában helyezkedik el, és
- értékük az aktuális paraméter értéke.

🕮 Az előfeldolgozáson átesett forrásmodult fordítási egységnek nevezik. A fordítási egység függvénydefiníciókból és külső deklarációkból áll. Külső deklaráció alatt a minden függvény „testén" kívüli deklarációt értjük. Az így definiált változót külső változónak, vagy globális változónak nevezik. Az ilyen változó:

- Hatásköre a fordítási egységben a deklarációs utasításának pozíciójától – az ún. deklarációs ponttól – indul, és a modul végéig tart. Ezt a hatáskört fájl hatáskörnek, vagy globális hatáskörnek nevezik.
- Élettartama a program teljes futási ideje. A memória hozzárendelés már fordítási időben megtörténik. Rendszerint az elsődleges adatterületen helyezkedik el, és biztos, hogy nem a veremben. Ez a statikus élettartam.

- Alapértelmezett (implicit) kezdőértéke: minden bitje zérus. Ez az aritmetikai típusoknál zérus. Karaktertömb esetében ez az üres karakterlánc, hisz rögtön a lánczáró nullával indul.
- Alapértelmezett tárolási osztálya **extern**.

💣 Bánjunk csínján az **extern** kulcsszó explicit használatával, mert deklarációs utasításban való megadása éppen azt jelenti, hogy az utasítás nem definíció, hanem csak deklaráció! Más fordítási egységben definiált külső változót kell ebben a forrásmodulban így deklarálni a rá való hivatkozás előtt.

```
/* Forrásmodul 1 */
extern int i;
extern char t[];
/* . . . */
```

```
/* Forrásmodul 2 */
int i;
char t[10];
/* . . . */
```

7. ábra: extern deklaráció

Prodzsektünk álljon e két forrásmodulból! Az **int** típusú **i** változót és a **t** karaktertömböt Forrásmodul 2–ben definiálták. Itt történt tehát meg a helyfoglalásuk. Ezek a változók Forrásmodul 1–ben csak akkor érhetők el, ha a rájuk történő hivatkozás előtt **extern** kulcsszóval deklarálják őket.

✍ Vegyük észre, hogy a Forrásmodul 1 elején elhelyezett deklarációs utasítások egyrészt globális szinten vannak, másrészt valóban csak deklarációk, hisz a tömb deklarációjában méret sincs!

💣 Persze azt, hogy a tömb mekkora, valahonnét a Forrásmodul 1–ben is tudni kell!

Írjuk át úgy a **PELDA10.C**–t, hogy a beolvasott sor tárolására használatos tömböt globálissá tesszük, majd nevezzük át **PELDA11.C**–re!

```
/* PELDA11.C: Névszámlálás */
#include <stdio.h>
#define NEV "Jani" /* A számlált név. */
#define MAX 29     /* A bemeneti sor maximális mérete.
                      Most egyben a leghosszabb név is. */
int getline(void); /* A függvény prototípusok. */
int strcmp(char s2[]);
void main(void){
  int db;          /* Névszámláló. */
  db=0;            /* A számláló nullázása. */
  printf("A(z) %s név leszámlálása a bemeneten.\nAdjon\
 meg soronként egy nevet!\nProgramvég: üres sor.\n",NEV);
  /* Sorok olvasása üres sorig a bemenetről: */
```

```
  while(getline()>0)
    /* Ha a sor épp a NEV: */
    if(strcmp(NEV)==0) ++db;
  /* Az eredmény közlése: */
  printf("A nevek közt %d darab %s volt.\n",db,NEV); }
char s[MAX+1];     /* Az aktuális név */
int strcmp(char s2[]){
  int i;
  for(i=0; s[i]!=0&&s[i]==s2[i]; ++i);
  return(s[i]-s2[i]);}
int getline(void){
  int c,i;
  for(i=0;i<MAX&&(c=getchar())!=EOF&&c!='\n';++i)
       s[i]=c;
  s[i]='\0';
  while(c!=EOF&&c!='\n') c=getchar();
  return(i); }
```

Vegyük észre, hogy a **PELDA10.C**–s verzióhoz képest a **getline** paraméter nélkülivé vált, és az **strcmp**–nek meg egyetlen paramétere maradt! Miután a globális **s** karaktertömb deklarációs pontja megelőzi a két függvény definícióját, a függvényblokkok **s** hatáskörében vannak. Belőlük tehát a tömb egyszerű hivatkozással elérhető, s nem kell paraméterként átadni. A **getline** második paramétere tulajdonképpen a **MAX** szimbolikus konstans volt, s ezt beépítettük a **for** ciklusba.

Elemezgessük egy kicsit a „külső változó, vagy paraméter” problémát!

- Globális változókat használva megtakarítható függvényhíváskor a paraméterek átadása, vagyis kevesebb paraméterrel oldható meg a feladat.
- Külső változókat manipuláló függvények másik programba viszont csak akkor másolhatók át és hívhatók meg változtatás nélkül, ha ezeket a változókat is velük visszük. Látszik, hogy az ilyen függvények mobilitását, újrafelhasználhatóságát csökkentik a járulékos, globális változók.
- A csak paramétereket és lokális változókat alkalmazó függvények átmásolás után változtatás nélkül használhatók más programokban. Legfeljebb az aktuális paraméterek lesznek mások a hívásban.

Világos, hogy nincs egyértelműen és általánosan ajánlható megoldás a problémára. A feladat konkrét sajátosságai döntik el, hogy mikor milyen függvényeket kell írni, kellenek–e külső változók stb.

Summázzuk a globális változóval, vagy más néven külső változóval, kapcsolatos ismereteinket!

Neve:	Bármilyen azonosító.
Típusa:	Valamilyen típus.
Hatásköre:	Globális, vagy fájl. A deklarációs ponttól a forrásmodul végéig tart.
Élettartama:	Statikus. A program teljes futási ideje alatt él.
Alapértelmezett tárolási osztálya:	**extern**
Alapértelmezett kezdőértéke:	Minden bitje zérus.
Deklarációs utasításának helye:	A forrásmodul minden függvénydefiníciójján kívül.
Memóriafoglalás:	Fordítási időben, vagy legalább is az indító program kezdődése előtt.

3.12 Inicializálás

Az inicializálás kezdőérték adást jelent a deklarációban, azaz az *inicializátor*ok kezdőértékkel látják el az objektumokat (változókat, tömböket stb.).

A statikus élettartamú objektumok egyszer a program indulásakor inicializálhatók. Implicit (alapértelmezett) kezdőértékük tiszta zérus, ami:

- Zérus az aritmetikai típusoknál.
- Üres karakterlánc karaktertömb esetén.

A lokális élettartamú objektumok inicializálása minden létrejövetelükkor megvalósul, de nincs implicit kezdőértékük, azaz „szemét” van bennük.

Módosítsuk újra a deklarációs utasítás szintaktikáját változókra és tömbökre!

típus azonosító<=*inicializátor*>;
típus tömbazonosító[<*méret*>]<={*inicializátorlista*}>;

, ahol az *inicializátorlista inicializátor*ok egymástól vesszővel elválasztott sorozata, és a *típus* a <*tárolási–osztály*><*típusmódosítók*> <*alaptípus*>–t helyettesíti.

A változókat egyszerűen egy kifejezéssel inicializálhatjuk, mely kifejezés opcionálisan {}-be is tehető. Az objektum kezdeti értéke a kifejezés értéke lesz. Ugyanolyan korlátozások vannak a típusra, és ugyanazok a

konverziók valósulnak meg, mint a hozzárendelés operátornál. Magyarán az *inicializátor hozzárendelés-kifejezés*. Például:

```
char y = 'z', k;      /* y 'z' értékű, és a k-nak meg */
                      /* nincs kezdőértéke. */
int a = 10000;        /* a 10000 kezdőértékű. */
```

C-ben a statikus élettartamú változók inicializátora csak konstans kifejezés lehet, ill. csak ilyen kifejezések lehetnek tömbök inicializátorlistájában. A lokális objektumok inicializátoraként viszont bármilyen legális kifejezés megengedett, mely hozzárendelés kompatibilis értékké értékelhető ki a változó típusára.

```
#define N 20
int n = N*2;        /* Statikus objektum inicializátora
                       csak konstans kifejezés lehet. */
/* . . . */
void fv(int par){
  int i = N/par;    /* A lokális objektumé viszont
                       bármilyen legális kifejezés.
   . . . */ }
```

✍ Emlékezzünk vissza, hogy globális változók csak egyszer kapnak kezdőértéket: a program indulásakor. A lokális objektumok viszont mindannyiszor, valahányszor blokkjuk aktívvá válik.

🕮 A szövegben használatos objektum fogalom nem objektum–orientált értelmű, hanem egy azonosítható memória területet takar, mely konstans vagy változó érték(ek)et tartalmaz. Minden objektumnak van azonosítója (neve) és adattípusa. Az adattípus rögzíti az objektumnak

- lefoglalandó memória mennyiségét és
- a benne tárolt információ belsőábrázolási formáját.

Tömbök esetén az inicializátorlista elemeinek száma nem haladhatja meg az inicializálandó elemek számát!

```
float tomb[3] = {0., 1., 2., 3.};   /* HIBÁS: több
                  inicializátor van, mint tömbelem. */
```

Ha az inicializátorlista kevesebb elemű, mint az inicializálandó objektumok száma, akkor a maradék objektumok a statikus élettartamú implicit kezdőérték adás szabályai szerint kapnak értéket, azaz nullázódnak:

```
float tmb[3] = {0., 1.};    /* tmb[0]==0.0, tmb[1]==1.0 és
                               tmb[2]==0.0. */
```

Az inicializálandó objektum lehet ismeretlen méretű is, ha az inicializátorlistából megállapítható a nagyság. Például:

```
int itmb[] = { 1, 2, 3};     /* Az itmb három elemű lesz. */
char nev[] = ”Lali”,         /* A lezáró ’\0’ karakter */
  csaladnev[] = ”Kiss”;      /* miatt 5 eleműek a tömbök.*/
```

Megoldandó feladatok:

Írja át az eddig elkészített **PELDA*n*.C**–ket, és a megoldott feladatok programjait úgy, hogy a változók kezdőértéket mindig inicializálással kapjanak!

4 TÍPUSOK ÉS KONSTANSOK

A fordító a forráskódot szintaktikai egységekre, vagy más elnevezéssel szimbólumokra, és fehér karakterekre tördeli. A több egymást követő fehér karakterből csak egyet tart meg. Ebből következőleg:

- Egyetlen C utasítás akár szimbólumonként külön–külön sorba írható.
- Egy sorban azonban több C utasítás is megadható.

✍ Pontosan hat szimbólum (int i ; float f ;) lesz a következőkből:

```
int
  i
  ;
float            f                      ;
```

vagy

```
int i; float f;
```

🕮 A karakter, vagy karakterlánc konstansokban előforduló, akárhány fehér karaktert változatlanul hagyja azonban a fordító.

Említettük már, hogy a programnyelv szimbólumokból (token) áll. Most ismertetjük a szimbólum definícióját módosított Backus–Naur, metanyelvi leírással:

szimbólum (token)*:*
  operátor
  kulcsszó
  elválasztó-jel
  azonosító
  konstans
  karakterlánc (string literal)

✍ Az értelmezés nagyon egyszerű: a felsorolt hat fogalom mindegyike szimbólum.

Az operátorokkal nem ebben a szakaszban foglalkozunk, hanem a **MŰVELETEK ÉS KIFEJEZÉSEK**ben!

A kulcsszavak:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**

default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Vannak ezeken kívül még nem szabványos kulcsszavak és más védett azonosítók, de ezek mindig megtudhatók a programfejlesztő rendszer segítségéből!

✍ Ilyenekre kell gondolni, mint a **cdecl**, a **pascal**, az **stdcall**, vagy egy–két aláhúzás karakterrel kezdődőkre, mint például az **__STDC__** stb.

🕮 ANSI C kompatibilis fordítást előírva mindig kideríthető, hogy az illető fordító mely kulcsszavai, operátorai, vagy elválasztó–jelei nem szabványosak.

4.1 Elválasztó-jel

A szintaktikai egységeket (a szimbólumokat) egymástól legalább egy fehér karakterrel el kell választani. Nincs szükség azonban az elválasztó fehér karakterre, ha a két nyelvi egység közé a szintaktikai szabályok szerint egyébként is valamilyen elválasztó-jelet kell írni. Az operátorok is elválasztó–jelnek minősülnek kifejezésekben.

elválasztó-jel: (a következők egyike!)
[ ] ( ) { } * , : = ; ... #

Nézzük meg néhány elválasztó–jel funkcióját!

Az utasítást záró pontosvessző minden példában benne van.

A kerek zárójeleknek csoportosító funkciója van kifejezésekben. Van, amikor a szintaktika része. Függvényhívásnál az aktuális paramétereket, függvénydeklarációban és definícióban a formális paramétereket ebbe kell tenni.

```
d = c*(a+b);
if(d==z) ++x;
fv(akt, par);
void fv2(int n);
```

A szögletes zárójelek tömbök deklarációjában és indexelő operátorként használatosak.

```
char kar, lanc[] = "Sztan és Pan.";
kar = lanc[3];
```

A kapcsos zárójelekbe tett több utasítás szintaktikailag egyetlen utasításnak minősül. A dolgot összetett utasításnak, blokknak nevezzük.

💣 Az összetett utasítás (blokk) záró kapcsos zárójele után tilos pontosvesszőt tenni!

```
if(a<b){        /* Illegális pontosvessző használat. */
  a=2; z=b+6; };
```

A csillag elválasztó-jelnek többféle szerepe van a nyelvben. Eddig csak a szorzás operátor funkcióját ismerjük!

```
a = 3.14*b;
```

Az egyenlőség jel hozzárendelés operátor, és deklarációs utasításban elválasztja a változót az inicializátortól, vagy inicializátorlistától.

```
char tomb[5] = {0, 1, 2, 3, 4};
int x=5, b, c=4;
b = x+c;
```

A kettős kereszt előfeldolgozó (preprocessor) direktíva kezdete. A sorbeli első nem fehér karakternek kell annak lennie.

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
```

4.2 Azonosító

azonosító:
  nem-számjegy
  azonosító nem-számjegy
  azonosító számjegy

nem-számjegy: (a következők egyike!)
  a ... z A ... Z _

számjegy: (a következők egyike!)
  0 1 2 3 4 5 6 7 8 9

Az azonosító változóknak, függvényeknek, felhasználó definiálta adattípusoknak stb. adott, a következő pontokban pontosított név:

- Kisbetűvel, nagybetűvel vagy aláhúzás (_) karakterrel köteles kezdődni.
- A további karakterek lehetnek számjegyek is.
- Az azonosító nem lehet kulcsszó vagy valamilyen előredefiniált, védett azonosító.
- Az azonosítók kis- és nagybetű érzékenyek, azaz az **Osszeg**, az **osszeg** vagy az **osszeG** három különböző azonosító.
- Kerülni kell a két és az egy aláhúzás karakterrel kezdődő nevek használatát is!

- Az azonosítók első, mondjuk, 31 karaktere szignifikáns. Ez azt jelenti, hogy hosszabb nevek is használhatók, de az első 31 karakterükben nem különböző azonosítók ugyanannak minősülnek.

✍ Az, hogy az azonosító első hány karaktere szignifikáns, a fordítótól függ. Másrészt a programfejlesztő rendszerben egy és ezen érték között változtatható is!

🕮 Bizonyos külső kapcsolódású azonosítókra, melyeket a kapcsoló–szerkesztő illeszt, eltérő megszorítások lehetnek érvényben. Például kevesebb karakterből állhatnak, vagy nem kis és nagybetű érzékenyek stb.

Megoldandó feladat:

A felsorolt példaazonosítók közül az első három hibás. Magyarázza meg, hogy mi a probléma velük!

- 6os_villamos
- Moszer Aranka
- Nagy_János
- Nagy_Jani
- puffer

4.3 Típusok és konstansok a nyelvben

A nyelvben összesen négy típuskategória van:

- A függvény a nyelv kódgeneráló egysége. Az összes többi kategória csak memóriát foglal az adatoknak.
- A **void** típus többnyire valaminek a meg nem létét jelzi. Például nincs paramétere és visszaadott értéke a **void main**(**void**) függvénynek.
- Skalár az aritmetikai típus, mely tovább bontható fixpontos egész és lebegőpontos valós ábrázolású típusokra. Ilyen a felsorolás (**enum**) típus és a mutató is.
- Aggregátum a tömb, a struktúra és az unió.

✍ A mutatókkal, a struktúrákkal és az uniókkal későbbi szakaszokban foglalkozunk!

A típusokat úgy is csoportosíthatnánk, hogy vannak

- alaptípusok és

- származtatott típusok

Az alaptípusok a **void**, a **char**, az **int**, a **float** és a **double**. A származtatás pedig a **short**, a **long**, a **signed** és az **unsigned** ún. típusmódosítókkal történhet. A **short** és a **long**, valamint a **signed** és az **unsigned** egymást kizáró módosító párok, de a két pár egyazon alaptípusra egyszerre is alkalmazható, azaz létezik

- **unsigned long int** vagy
- **signed short int** stb.

A **signed** és az **unsigned** módosító azonban csak egész típusokra (**char** és **int**) alkalmazható, lebegőpontos valós és a **void** alaptípusra nem.

A származtatott típusokba mindig beleértendők az ilyen típusú értékkel visszatérő függvények, az ilyen típust paraméterként fogadó függvények, a tömbök stb. Ha programunkban valamilyen azonosítót használni kívánunk, akkor előbb deklarálni (definiálni) kell. A deklaráció teremti meg a kapcsolatot az azonosító és az objektum között, és rögzíti legalább az objektum adattípusát. A származtatott típust is szokás egyszerűen típusnak nevezni.

„Készpénzre váltva” az előző bekezdésben mondottakat: ha a *típus* valamilyen nem **void** adattípus, akkor a deklarációk következőképp szemléltethetők:

```
típus t, t1, t2;   /* Három típus típusú objektum. */
típus f(void);     /* Típus típusú értéket visszaadó,
                      paraméter nélküli függvény. */
void fv(típus i); /*Típus típusú paramétert fogadó
                      eljárás. */
típus tt[10];      /* 10 elemű, típus  típusú tömb. Az
                      elemek rendre: tt[0], ..., tt[9]. */
```

🕮 Feltétlenül említést kell tennünk még két, definícióban használható módosítóról, melyek alaposan megváltoztatják a deklarált objektum tulajdonságait.

A **const** nem módosítható objektumot definiál, azaz meggátolja a hozzárendelést az objektumhoz, és az olyan mellékhatásokat, mint az inkrementálás, dekrementálás stb. Magyarán: nem engedi meg az azonosító elérését balértékként. A **const** a deklaráció elején bármilyen alaptípussal, aggregátum típussal stb. állhat, de tilos többtételes deklaráció első vesszője után kiírni:

```
float f = 4.5, const cf = 5.6;      /* HIBÁS. */
```

✍ Ha a **const** objektum nem lehet balérték, akkor hogyan lehet valamilyen értékkel ellátni?

🕮 A **const** típusú objektum értékkel való ellátásának egyetlen módja az inicializálás. Tehát az ilyen objektum definíciójában kötelező neki kezdőértéket adni, mert ez később már nem tehető meg. Például:

```
const float pi = 3.1415926;
const max2int = 32767;
```

✍ Az elő nem írt alaptípus miatt a deklaráció specifikátorként egyedül álló **const** tulajdonképpen **const int**, s ezért a **max2int** is az. E deklarációk után „botor dolgok” a következő utasítások:

```
pi = 3.;              /* Szintaktikai hiba. */
int i = max2int++;    /* Szintaktikai hiba. */
```

A **volatile** szó jelentése elpárolgó, illékony, állhatatlan. Módosítóként azt jelzi, hogy az illető változó értéke program végrehajtáson kívüli okból is megváltozhat.

✍ Mi lehet a program végrehajtásán kívüli ok? Például megszakítás, B/K port, konkurensen futó végrehajtási szál stb.

🕮 A **volatile** kulcsszó deklaráción belüli elhelyezésének szabályai egyeznek a **const**–éival. A fordító az ilyen változót nem helyezheti el regiszterben, ill. az ilyen objektumot is tartalmazó kifejezés kiértékelése során nem indulhat ki abból, hogy az érték közben nem változik meg. Szóval az ilyen változó minden hivatkozásához elérési kódot kell generálnia akkor is, ha az látszólag hatástalannak tűnik. Például:

```
volatile ticks;                /* volatile int a típus. */
void interrupt timer(void) {++ticks;}
void varj(int ennyit){
  ticks =0;
  while( ticks < ennyit ); }   /* Ne tegyen semmit. */
```

✍ A **while**-beli feltétel kiértékelésekor a ciklus minden ütemében tölteni kell **ticks** értékét.

🕮 Láttuk, hogy egy objektumot egész élettartamára **volatile**–lá tehetünk deklarációval, de explicit típusmódosító szerkezettel a változó egyetlen hivatkozását is **volatile**–lá minősíthetjük.

```
int ticks;
void interrupt timer(void) {++ticks;}
void varj(int ennyit){
  ticks =0;
  while( (volatile)ticks < ennyit ); }
```

📖 Egy objektum egyszerre lehet **const** és **volatile**, amikor is az őt birtokló program nem módosíthatja, de megváltoztathatja az értékét bármely aszinkron program vagy végrehajtási szál.

Az adattípusok és konstansok tárgyalásának megkezdése előtt lássuk a konstans definícióját!

konstans:
egész-konstans
enum-konstans
lebegőpontos-konstans
karakter-konstans

Tudjuk, hogy az objektum is lehet konstans, de itt most nem ilyen állandóról van szó. Az „igazi" konstans nem azonosítható memória területet takar, mely fix, a program futása alatt meg nem változtatható értéket tartalmaz. A konstansnak nincs szoftveresen is használható címe, de van adattípusa, mely meghatározza az állandónak

- lefoglalandó memória mennyiségét és
- a benne tárolt érték belsőábrázolási formáját.

4.3.1 Egész típusok és konstansok

Már említettük, hogy az egész típusok a **char** és az **int** alaptípusból a **signed - unsigned**, valamint a **short - long** módosító párok alkalmazásával állíthatók elő, azaz a deklaráció írásszabálya eltekintve a tárolási osztálytól és az inicializálástól:

<típusmódosítók> <alaptípus> azonosítólista;

A *típusmódosítók alaptípus* párost azonban a továbbiakban is *típus*nak fogjuk nevezni. A szabályok a következők:

- Mind az *alaptípus*, mind a *típusmódosítók* elhagyható. A kettő együtt azonban nem.
- Ha elhagyjuk az *alaptípus*t, alapértelmezés az **int**.
- A **short - long** módosítók elhagyásakor nincs rájuk vonatkozó alapértelmezés.
- Ha elhagyjuk a **signed - unsigned** módosítót, alapértelmezés a **signed**.

Típus	Méret bájtban	Minimális érték	Maximális érték
char, **signed char**	1	-128	127
unsigned char	1	0	255
short, short int, signed short int	2	-32768	+32767
unsigned short, unsigned short int	2	0	65535
int, signed int	2 vagy 4	**short** vagy **long**	**short** vagy **long**
unsigned, unsigned int	2 vagy 4	ugyanígy, de **unsigned**	ugyanígy, de **unsigned**
long, long int, signed long int	4	-2147483648	+2147483647
unsigned long, unsigned long int	4	0	4294967295

8. ábra: Egész típusok

- A **char** alaptípussal kapcsolatban kiegészítésre szorul a **signed char** alapértelmezés! A programfejlesztő rendszerben ugyanis **unsigned char** is beállítható a **char** típus alapértelmezéseként. A lehetséges karaktertípusok ilyenkor így változnak:

Típus	Méret bájtban	Minimális érték	Maximális érték
char, **unsigned char**	1	0	255
signed char	1	–128	127

- A táblázatból (8. ábra) is jól látszik, hogy az ANSI szabvány nem ír elő pontos méretet az **int** típusra, csupán csak annyit, hogy:

  short <= int <= long.

- A belsőábrázolás fixpontos, bináris egész, ezért **signed** típusokra a negatív számokat kettes komplemensük alakjában tárolja a fordító.

- A szabványos **LIMITS.H** fejfájlban találunk szimbolikus állandókat (**CHAR_MIN**, **UCHAR_MAX**, **SHRT_MIN**, stb.) az ábrázolási korlátokra!

💣 A nyelv szerint nincs sem egész túlcsordulás, sem alulcsordulás. Az egész konstansnál láthatunk erre is egy rövid példát!

egész-konstans:
decimális-konstans <*egész-utótag*>
oktális-konstans <*egész-utótag*>
hexadecimális-konstans <*egész-utótag*>

✍ A metanyelvben a < ... > az elhagyhatóságot jelöli!

egész-konstans:
decimális-konstans<*egész-utótag*>
oktális-konstans<*egész-utótag*>
hexadecimális-konstans<*egész-utótag*>

decimális-konstans:
nemzérus-számjegy
decimális-konstans számjegy

oktális-konstans:
0
oktális-konstans oktális-számjegy

hexadecimális-konstans:
0x*hexadecimális-számjegy*
0X*hexadecimális-számjegy*
hexadecimális-konstans hexadecimális-számjegy

nemzérus-számjegy: (a következők egyike!)
1 2 3 4 5 6 7 8 9

oktális-számjegy: (a következők egyike!)
0 1 2 3 4 5 6 7

hexadecimális-számjegy: (a következők egyike!)
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

egész-utótag:
unsigned-utótag<*long-utótag*>
long-utótag<*unsigned-utótag*>

unsigned-utótag: (a következők egyike!)
u U

long-utótag: (a következők egyike!)
l L

A definíció szerint:

- Az egész konstans lehet decimális, oktális és hexadecimális.
- Az egész konstansnak nincs előjele (pozitív), azaz a negatív egész konstans előtt egy egyoperandusos mínusz operátor áll.

- A decimális egész konstans **int**, **long int**, vagy **unsigned long int** belsőábrázolású a konstans értékétől függően alapértelmezés szerint. Az oktális és hexadecimális egész konstans az értékétől függően **int**, **unsigned int, long int**, vagy **unsigned long int** belsőábrázolású.

Decimális	Oktális	Hexadecimális	Típus
0–32767	0–077777	0X0–0X7FFF	**int**
	0100000–01777777	0X8000–0XFFFF	**unsigned**
32767–2147483647	02000000–017777777777	0X10000–0X7FFFFFFF	**long**
2147483648–4294967295	020000000000–037777777777	0X80000000–0XFFFFFFFF	**unsigned long**

9. ábra: Egész konstansok belsőábrázolása

Lássunk néhány példát a decimális, az oktális és a hexadecimális egész konstansokra!

```
int i = 10;
int j = 010;     /* j kezdetben decimálisan 8. */
int k = 0;       /* k decimálisan és oktálisan is 0. */
int l = 0XFF;    /* k decimálisan 255. */
```

- Explicit egész utótagot a konstans utána írva megváltoztathatjuk az alapértelmezett belsőábrázolást.

Az inicializátor konstansok típusegyeztetésével elkerülhető a szükségtelen közbenső konverzió. Például:

```
unsigned ui=2u;
long li=16l;
unsigned long uli=17lu;
```

Az egész konstans 0 és 4294967295 értékhatárok közt megengedett. Ez azonban a programozó felelőssége, mert a 4294967295-nél nagyobb érték egyszerűen csonkul. Legfeljebb egy „halk” figyelmeztető üzenet jöhet a fordítótól. Például a

```
#include <stdio.h>
void main(void){
  unsigned long pipipp;
  pipipp = 4300000000;
  printf("Pipipp = %ld\n", pipipp); }
```

hatására 5032704 jelenik meg, ami egy híján 4300000000 - 4294967295.

4.3.2 Felsorolás (enum) típus és konstans

Az **enum** (enumerate) adattípus mnemonikussá tesz egy sorozat egész értékre való hivatkozást. Például az

```
enum napok{
  vasar, hetfo, kedd, szerda, csut, pentek, szomb} nap;
```

deklaráció létrehoz egy egyedi **enum napok** egész típust. Helyet foglal egy ilyen típusú, **nap** azonosítójú változónak, és definiál hét, konstans egész értéket reprezentáló enumerátort: az enumerátor készletet.

📖 Felsorolás típusú változóban a típushoz definiált enumerátor készlet egyik értékét tarthatjuk. Az enumerátort enum konstansnak is nevezik. Felsorolás típusú változókat használhatunk index kifejezésben, minden aritmetikai és relációs operátor operandusaként, stb. Tulajdonképpen az **enum** a **#define** direktíva alternatívájának is tekinthető.

📖 ANSI C–ben az enumerátor értékét definiáló kifejezés csak egészértékű konstans kifejezés lehet, és típusa mindig **int**. Az **enum** változó tárolásához használt memória is annyi, mint az **int** típushoz használt. **enum** típusú konstans vagy érték a C–ben ott használható, ahol egész kifejezés is. Lássuk a szintaktikát!

enum-specifikátor:
  enum <*azonosító*> {*enumerátorlista*}
  enum *azonosító*

enumerátorlista:
  enumerátor
  enumerátorlista, enumerátor

enumerátor:
  enum-konstans
  enum-konstans = konstans-kifejezés

enum-konstans:
  azonosító

Az *enum-specifikátor* definíciójában az *enumerátorlistá*val definiált **enum** típus opcionális azonosítóját enum címkének (tag) nevezzük. Ha ezt elhagyjuk a definícióból, névtelen **enum** típust kapunk. Ennek az az ódiuma, hogy később nincs lehetőségünk ilyen felsorolás típusú változók definíciójára, hisz nem tudunk a névtelen **enum**–ra hivatkozni. Névtelen **enum** típusú változók tehát csak a névtelen **enum** deklarációjában definiálhatók, máshol nem. Az

```
enum {jan, feb, marc, apr, maj, jun, jul, aug, szep, okt,
     nov, dec};
```

így teljesen használhatatlan, hisz képtelenség ilyen típusú változót deklarálni. A probléma az azonosítólista megadásával elkerülhető:

```
enum {jan, feb, marc, apr, maj, jun, jul, aug, szep, okt,
    nov, dec} ho=jan, honap;
```

Foglaljuk össze az **enum** deklarációval kapcsolatos tapasztalatainkat:

enum <*enum címke*> <{*enumerátorlista*}> <*azonosítólista*>;

, ahol a < > most is az elhagyhatóságot jelöli. Az enum címke megadása tehát azt biztosítja, hogy az "**enum** *enum címke*" után azonosítólistát írva később ilyen felsorolás típusú változókat definiálhatunk. A fejezet elején említett deklarációs példában a **napok** azonosító az opcionálisan megadható enum címke, mely később **enum napok** típusú változók definícióiban használható fel. Például:

```
enum napok fizetes_nap, unnepnap;
```

Térjünk egy kicsit vissza az enumerátorokhoz! Láttuk, hogy az enum konstans a felsorolás típus deklarációjában definiált azonosító. Miután az enumerátor egész adattípusú, kifejezésekben ott használható, ahol egyébként egész konstans is megadható. Az enumerátor azonosítójának egyedinek kell lennie az **enum** deklaráció hatáskörében beleértve a normál változók neveit és más enumerátorlisták azonosítóit. Az enum konstansok az egyszerű változók névterületén helyezkednek el tehát. Az enum címkéknek viszont az **enum**, a struktúra és az uniócímkék névterületén kell egyedinek lenniük.

📖 A névterület olyan azonosító csoport, melyen belül az azonosítóknak egyedieknek kell lenniük. A C–ben több névterület van, amiből itt kettőt meg is említettünk. Két különböző névterületen létező, ugyanazon azonosítónak semmi köze nincs egymáshoz. A névterületeket majd egy későbbi fejezetben tárgyaljuk!

Lássunk példákat ezen közbevetés után!

```
int hetfo = 11;  /* A hetfo egyszerű int típusú változó.*/
{ enum napok{
    vasar, hetfo, kedd, szerda, csut, pentek, szomb};
  /* A hetfo enumerátor ebben a blokkban elrejti a hetfo
     int típusú változót. */
  double csut;  /* HIBÁS, mert ezen a névterületen van
                   már egy „csut” enumerátor.
  /* . . . */ }
hetfo+=2;          /* OK, mert itt már csak a hetfo int
                   változó létezik. */
```

Az **enum napok** deklarációban szereplő enum konstansok implicit módon kaptak értéket zérustól indulva és mindig eggyel növekedve. A **vasar** így 0, a **hetfo** 1, ..., és a **pentek** 6. Az enum konstansok azonban explici-

ten is inicializálhatók. Akár negatívak is lehetnek, ill. több enumerátor lehet azonos értékű is. Például:

```
enum ermek{ egyes=1, kettes, otos=5, tizes=2*otos,
           huszas=kettes*tizes, szazas=otos*huszas};
```

✍ Vegyük észre, hogy az enumerátor definiált már az enumerátorlista következő tagjához érve!

Mondottuk, hogy a felsorolás típusok mindenütt feltűnhetnek, ahol egyébként az egész típusok megengedettek. Például:

```
enum napok{
  vasar, hetfo, kedd, szerda, csut, pentek, szomb};
enum napok fizetes_nap = szerda, nap;
int i = kedd;      /* OK */
nap = hetfo;       /* OK */
hetfo = kedd;      /* HIBÁS, mert a hetfo konstans. */
```

✍ **enum** típusú változóhoz egész érték hozzárendelése megengedett, de explicit típusmódosító szerkezet használata javallott. Tehát a

```
fizetes_nap = 5;
```

helyett

```
fizetes_nap = (enum napok) 5;
```

írandó.

💣 A fordítóban nincs mechanizmus a konvertálandó egész érték érvényességének ellenőrzésére! Pontosabban a következő lehetetlenség elkerülése a programozó felelőssége:

```
n = (napok) 958;
```

Az **enum** típus egész (integral) típus. Implicit módon konvertál így egésszé bármely enumerátort a fordító, de a másik irányban az explicit típusmódosítás javasolt:

```
int i;
enum napok n = szomb;   /* OK */
i = n;                  /* OK */
n = 5;                  /* Nem javasolt! */
n = (napok) 5;          /* OK */
```

🕮 Egész típusnak az egész értékek tárolására alkalmas, aritmetikai adattípusokat (**char**, **short**, **int**, **long** és **enum)** nevezzük.

4.3.3 Valós típusok és konstans

Az IEEE lebegőpontos belsőábrázolású valós alaptípusok a **float** és a **double**. Egyedül a **long** módosító használata megengedett, s az is csak a **double** előtt, azaz van még **long double** származtatott típus. A **float** típusú, egyszeres pontosságú lebegőpontos ábrázolás 4 bájtot foglal, melyből 8 bit a 128 többletes, bináris exponens, 1 bit a mantissza előjele (1 a negatív!) és 23 bit a mantissza. A mantissza 1.0 és 2.0 közötti szám. Miután a mantissza legnagyobb helyiértékű bitje mindig egy, az ábrázolás ezt nem tárolja.

előjel	eltérített karakterisztika	mantissza
31. bit	30 – 23 bitek	22 - 0 bitek

A **double** exponens 11-, s a mantissza 52 bites.

előjel	eltérített karakterisztika	mantissza
63. bit	62 – 52 bitek	51 - 0 bitek

A lebegőpontos belsőábrázolás méretei, megközelítő határai és decimális jegyben mért pontossága a következő táblázatban látható:

Típus	Méret bájtban	Határok:	Pontosság:
float	4	$\pm 3.4*10^{-38}$ - $\pm 3.4*10^{+38}$	6–7 decimális jegy
double	8	$\pm 1.7*10^{-308}$ - $\pm 1.7*10^{+308}$	15–16 decimális jegy
long double	10	$\pm 1.2*10^{-4932}$ - $\pm 1.2*10^{+4932}$	19 decimális jegy

10. ábra: Lebegőpontos típusok

📖 Ehhez már csak annyit kell hozzáfűzni, hogy lebegőpontos zérus az, amikor a belsőábrázolás minden bitje zérus.

💣 A leírás az IEEE szabványos lebegőpontos ábrázolásról szól, de elképzelhető, hogy a konkrét fordító más formával dolgozik. A szabványos **FLOAT.H** fejfájlban azonban mindig megtalálhatók szimbolikus állandók (**FLT_MIN**, **DBL_MAX** stb.) alakjában az ábrázolási határok és más konkrétumok az aktuális belsőábrázolásról.

lebegőpontos-konstans:
tört-konstans <exponens-rész> <float-utótag>
számjegy-sor exponens-rész <float-utótag>

tört-konstans:
  <számjegy-sor> . számjegy-sor
  számjegy-sor .
exponens-rész:
  e *<előjel> számjegy-sor*
  E *<előjel> számjegy-sor*
előjel: (a következők egyike!)
  + -
számjegy-sor:
  számjegy
  számjegy-sor számjegy
float-utótag: (a következők egyike!)
  f l F L

- A mantisszából elhagyható a decimális egész rész vagy a tört rész, de a kettő együtt nem.
- Elhagyható a tizedes pont vagy az exponens rész, de mindkettő nem.
- A lebegőpontos konstans előjeltelen (pozitív). A negatív lebegőpontos konstans előtt egy egyoperandusos mínusz operátor áll.
- Float utótag nélkül a lebegőpontos konstans **double** belsőábrázolású. Az utótag megadásával kikényszeríthetjük a **float** (**f** vagy **F**), ill. a **long double** (**l** vagy **L**) belsőábrázolást.

double belsőábrázolású lebegőpontos konstansok:

```
-.5e35, 5., 5E-4, 3.4
```

Ugyanezek **float** és **long double** belsőábrázolásban:

```
-.5e35f, 5.L, 5E-4F, 3.4l
```

Az egyszeri programozó beírta a forrásszövegébe a

```
float v=143736120;
/* . . . */
printf("%9.0f\n", v);
```

sorokat, és meglepődött a 143736128–as eredményen. Ha **v** értékét eggyel csökkentette, akkor meg 143736112–őt kapott. Mi lehet a probléma?

A 143736120 (0X8913D38) binárisan

```
1000 1001 0001 0011 1101 0011◆1000
```

, de csak 24 bit fér el a belsőábrázolás mantisszájában. A fordító a legmagasabb helyiértékű, lecsorduló bit értékével megnöveli a mantisszát. Most, a karakterisztikával nem foglalkozva, az új érték:

```
1000 1001 0001 0011 1101 0100◆0000
```

, ami éppen 143736128 (0X8913D40).

A 143736119 (0X8913D37) binárisan

```
1000 1001 0001 0011 1101 0011◆0111
```

esetében az eredmény

```
1000 1001 0001 0011 1101 0011◆0000
```

lesz, ami143736112 (0X8913D30).

💣 Komolyan kell tehát venni a lebegőpontos ábrázolások decimális jegyben mért pontosságát, és az ábrázolási határokat. Még két dologra feltétlenül oda kell figyelni:

- A decimális véges tizedes tört többnyire nem véges bináris tört, az- az az átalakítás oda–vissza nem teljesen pontos.
- Matematikai iterációt, ami addig tart, míg két, számolt, valós érték különbsége zérus nem lesz, nem szabad egy az egyben megvalósítani, mert az esetek többségében végtelen ciklushoz vezet.

4.3.4 Karakter típus és konstans

A karakter típusról már szó esett az egész típusok tárgyalásánál. Tudjuk, hogy három karakter típus van: **char**, **signed char** és **unsigned char**.

A karakter típusú objektum helyfoglalása 1 bájt mindenképp. A karakter konstans típusára és helyfoglalására rögtön kitérünk definíciója után!

karakter-konstans:
'c-karakter-sor'
c-karakter-sor:
c-karakter
c-karakter-sor c-karakter
c-karakter:
bármilyen karakter aposztróf ('), fordított per jel (\) és soremelés (\n) kivételével
escape-szekvencia

Szekvencia	Érték	Karakter	Funkció
\0	0X00	NUL	karakterlánc vége
\a	0X07	BEL	fütty
\b	0X08	BS	visszatörlés
\t	0X09	HT	vízszintes tab
\n	0X0A	LF	soremelés
\v	0X0B	VT	függőleges tab
\f	0X0C	FF	lapdobás
\r	0X0D	CR	kocsi vissza
\”	0X22	”	macskaköröm
\’	0X27	’	aposztróf
\?	0X3F	?	kérdőjel
\\	0X5C	\	fordított per jel
\ooo	bármi	bármi	max. 3 oktális számjegy
\xhh	bármi	bármi	max. 3 hexadec. jegy
\Xhh	bármi	bármi	max. 3 hexadec. jegy

11. ábra: Escape szekvenciák

A definícióból következőleg a karakter konstans aposztrófok közt álló egy vagy több karakter, ill. escape szekvencia. Ezen az elven beszélhetünk egykarakteres és többkarakteres karakter konstansról.

- A karakter konstans adattípusa mindenképpen **int**. Tudjuk, hogy az **int** helyfoglalása 2 vagy 4 bájt, így maximum négy karakteres karakter konstans létezhet. Például:

```
’An’, ’\n\r’, ’Alfi’, ’tag’, ’\007\007\007’
```

Tudjuk, hogy az **int** belsőábrázolású karakter konstans esetében is érvényben van a **signed char** alapértelmezés. Ennek következtében az **int** méreténél kevesebb karakterből álló karakter konstansok előjel kiterjesztéssel alakulnak **int** belsőábrázolásúvá. Konkrét példaként tekintsük a 2 bájtos **int**-et, és a 852–es kódlapot! Az é betű kódja 130 (0X82) és az a betűé 97 (0X61). Az ’é’-ből így 0XFF82 és az ’a’-ból 0X0061 lesz a memóriában, s így igaz lesz a következő reláció:

```
'é' < 'a'
```

Ha **unsigned char** az alapértelmezés, akkor a felső bájt(ok)at bizonyosan 0X00-val tölti fel a fordító (az 'é'-ből 0X0082 lesz), és nem jön elő az előbb taglalt probléma.

A következő kis példa szemlélteti a tárgyalt karakter típus és karakter konstans méreteket!

```
#include <stdio.h>
#include <stdlib.h>
#define CH 'x'       /* Egykarakteres karakter konstans. */
void main(void) {
  char ch = CH;    /* Karakter típusú változó. */
  printf("Az int mérete:\t\t\t%d\n", sizeof(int));
  printf("A char mérete:\t\t\t%d\n", sizeof(char));
  printf("A karakter konstans mérete:\t   %d\n",
          sizeof(CH));
  printf("A karakter változó mérete:\t%d\n",
          sizeof(ch)); }
```

Szólnunk kell még néhány szót a pontosítás kedvéért az escape szekvenciáról! A \ jelet követheti egy a táblázatban (11. ábra) ismertetett karakter, vagy egy legfeljebb háromjegyű oktális szám, vagy x után hexadecimális számjegyek. Például a '\03' a Ctrl+C, a '\x3F' a ? karakter, stb. Szóval az escape szekvencia leírhat vezérlő és „igazi" karaktereket egyaránt. Nézzük még a szabályokat!

- Ha az escape szekvencia fordított per jelét nem legális karakter követi, akkor legfeljebb figyelmeztető üzenetet kapunk, és a fordító elhagyja a \ jelet, de megtartja az illegális karaktert. Például a \z–ből a z marad meg.
- Az oktálisan vagy hexadecimálisan adott escape szekvencia végét az első nem oktális, ill. hexadecimális karakter is jelzi. Például a '\518'-ból két karakteres karakter konstans lesz, ahol a karakterek rendre a '\51' és a '8', azaz ')8'.
- Vigyázzunk a nem egyértelmű megadásra is! Például a "\712 kőműves." karakterláncból látszik a programozó törekvése, azaz hogy a BEL karakter után a 12 kőműves szöveg következne. Ez azonban így szintaktikai hiba. A helyes eredményhez például "\7" "12 kőműves." módon juthatunk.

Miután az escape szekvenciával tulajdonképpen egyetlen karaktert adunk meg, a \ jelet követő oktális számérték nem haladhatja meg a 0377–et, a hexadecimális a 0XFF–et, vagyis a 255–öt!

✍ Igazából a karakter a végrehajtáskor érvényes gépi karakterkészletből vett érték. A könyvben végig bájtos (**char**) kódkészletet (ASCII) tételezünk fel, s nem foglalkozunk a széles karaktertípussal (**wchar_t**) és az Unicode kódkészlettel!

🕮 Az eddig tárgyalt adattípusokat összefoglaló névvel aritmetikai adattípusoknak is nevezhetjük. Az aritmetikai adattípusok „zavarba ejtő” bősége a nyelvben azonban nem arra szolgál, hogy a programozó

- csak úgy „ukk-mukk-fukk” kiválasszon valamilyen adattípust adatai és (rész)eredményei tárolására, hanem arra, hogy
- a számábrázolási korlátokat tekintetbe véve alaposan végiggondolja, hogy érték és pontosság vesztés nélkül milyen adattípust kell használnia az egyes adataihoz, (rész)eredményeihez. Esetleg milyen explicit konverziókat kell előírnia a részletszámítások végzése során a pontosság vesztés elkerüléséhez.

Készítsen szorzat piramist! A piramis alsó sora 16 darab 100–tól 115–ig terjedő egész szám! A következő sorokat úgy kapjuk, hogy az alattuk levő sor elemeit páronként összeszorozzuk. A második sor így néz ki tehát: 100*101 = 10100, 102*103 = 10506, …, 114*115 = 13110. A harmadik sorban a következők állnak: 10100*10506 = 106110600, 10920*11342 = 126854640, …, 12656*13110 = 165920160, és így tovább.

Figyeljünk arra, hogy az eredményül kapott számokat mindig a lehető legkisebb helyen tároljuk, de az értékeknek nem szabad csonkulniuk!

```
/* PELDA12.C: Szorzat piramis */
#include <stdio.h>
#define N 16
void main(void){
  int i;           /* Az alsó sor: 100-s nagyságrend. */
  char n[N]={100, 101, 102, 103, 104, 105, 106, 107,
             108, 109, 110, 111, 112, 113, 114, 115};
  short n1[N/2];   /* 2. sor: 10000-s nagyságrend. */
  long n2[N/4];    /* 3. sor: 100000000 kb. */
  long double n3[N/8]; /* 4. sor: 10 a 16-n kb. */
  printf("Szorzat piramis: első szint\n");
  for(i=0; i<N; i=i+2){
    n1[i/2]=(short)n[i]*(short)n[i+1];
    printf("%3d*%3d = %5hd\n", n[i], n[i+1], n1[i/2]); }
  printf("Szorzat piramis: második szint\n");
  for(i=0; i<N/2; i=i+2){
    n2[i/2]=(long)n1[i]*(long)n1[i+1];
    printf("%5hd*%5hd = %9ld\n", n1[i], n1[i+1],
           n2[i/2]); }
  printf("Szorzat piramis: harmadik szint\n");
```

```
for(i=0; i<N/4; i=i+2){
  n3[i/2]=(long double)n2[i]*(long double)n2[i+1];
  printf("%9ld*%9ld = %17.0Lf\n", n2[i], n2[i+1],
          n3[i/2]); }
printf("Szorzat piramis: csúcs\n");
printf("%17.0Lf*%17.0Lf ~ %33.0Lf\n", n3[0], n3[1],
        n3[0]*n3[1]);}
```

☞ Már a **PELDA7.C** példában találkoztunk az egészekre vonatkozó **h** és **l** hosszmódosítókkal a formátumspecifikációkban. Most a **long double** típusú paraméter jelzésére való **L** hosszmódosítót ismerhettük meg.

💣 Ha a szorzat piramis második szintjének kiszámítását végző

```
n2[i/2]=(long)n1[i]*(long)n1[i+1];
```

kifejezésből elhagyjuk a két, explicit (**long**) típusmódosítót, akkor a listarész a következőképp módosul:

```
Szorzat piramis: második szint
10100*10506 =      7816
10920*11342 =     -8400
. . .
```

Ez vitathatatlanul rossz, de miért?

- Az **n1** tömb **short** típusú.
- A kétoperandusos szorzási művelet operandusai azonos típusúak, tehát semmiféle implicit konverzió nem történik, és az eredmény típusa is **short**.
- A 32 bites regiszterben a 10100*10506–os szorzás végrehajtása után ugyan ott van a helyes eredmény, a 106110600 (0X6531E88)

  ```
  0000 0110 0101 0011 0001 1110 1000 1000
  ```

  , de a 16 bites **short** típus miatt csak az alsó két bájt képezi az eredményt. Az meg 0X1E88, vagyis 7816.

☞ Megemlítjük még, hogy a „csúcs” értékének (**n3**[0]***n3**[1]) pontosság vesztés nélküli tárolásához még a **long double** típus is kevés!

Megoldandó feladatok:

Készítsünk programot, mely megállapítja az **ÓÓ:PP:MM** alakú karakterláncról, hogy érvényes idő–e! Az óra zérus és 23, a perc, valamint a másodperc 0 és 59 közötti érték lehet csak. Jelezze ki a szoftver, ha a karakterlánc formailag hibás, és kérjen másikat! Ha érvényes az idő, akkor határozza meg és jelezze ki értékét másodpercben!

Egy másik szoftver fogadjon a szabvány bemenetről egész számokat mindaddig, míg üres sort nem adnak meg! Képezze rendre az egész számok és két szomszédjuk négyzetösszegét! A szám szomszédja a nálánál eggyel kisebb és eggyel nagyobb érték. A képernyőn jelenítse meg fejléccél ellátva, táblázatos formában a számhármasokat (a bevitt értékek álljanak középen) és a négyzetösszegeket! Legvégül adja meg a négyzetösszegek összegét is. A számok legyenek jobbra igazítottak!

4.4 Karakterlánc (string literal):

A karakterláncokról már szó volt a **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban! A definíció:

karakterlánc:
"*<s-karakter-sor>*"

s-karakter-sor:
s-karakter
s-karakter-sor s-karakter

s-karakter:
bármilyen karakter idézőjel ("), fordított per jel (\) és a soremelés (\n) kivételével
escape-szekvencia (11. ábra)

A karakterlánc adattípusa **char** tömb. A tömb mérete mindig eggyel hosszabb, mint ahány karakterből a karakterlánc áll, mert a nyelv a lánc végét '\0' karakterrel jelzi. A karakterlánc konstansokat mindig a statikus adatterületen helyezi el a fordító.

'L'	'a'	'l'	'i'	'\0'
0	1	2	3	4

Például a "Lali" karakterlánc egymást követő, növekvő című memória bájtokon így helyezkedik el.

A karakterlánc karaktereiként használható az escape szekvencia is. Például a

```
"\t\t\"Név\"\\\tCím\n\n
```

karakterláncot **printf** függvény paramétereként megadva

```
"Név"\    Cím
```

jelenik meg utána két soremeléssel.

A csak fehér karakterekkel elválasztott karakterlánc konstansokat a fordító elemzési fázisa alatt egyetlen karakterlánccá egyesíti:

```
"Egy " "kettő, " "három..." → "Egy kettő, három..."
```

Tudjuk, hogy a karakterlánc konstans a folytatássor (\) jelet használva több sorba is írható:

```
printf("Ez igazából egyetlen \
karakterlánc lesz.\n");
```

Írjunk meg néhány karakterláncot kezelő függvényt!

Az **unsigned strlen**(**char** *s*[]) a paraméter karakterlánc hosszát szolgáltatja.

Indítsunk egy változót zérustól, és indexeljünk előre vele a karakterláncon, míg a lánczáró nullát el nem érjük! Ez az index a karakterlánc hossza is egyben.

```
unsigned strlen(char s[]){
  unsigned i=0u;
  while(s[i]!='\0') ++i;
  return i; }
```

A **void strcopy**(**char** *cél*[], **char** *forrás*[]) a hozzárendelést valósítja meg a karakterláncok körében azzal, hogy a *forrás* karakterláncot átmásolja a *cél* karaktertömbbe.

Indexeljünk most is végig egy változóval a *forrás* karakterláncon a lezáró zérusig! Közben minden indexre rendeljük hozzá a *forrás* tömbelemet a *cél* tömbelemhez! Vigyázzunk, hogy a lánczáró nulla is átkerüljön!

```
void strcopy(char cél[], char forrás[]){
  int i=0;
  while((cél[i]=forrás[i])!=0) ++i; }
```

Javítsunk kicsit ezen a megoldáson! A **while**–beli reláció elhagyható, hisz a hozzárendelés is mindig igaz (nem zérus), míg a **forrás**[**i**] nem lánczáró zérus. Belátható, hogy a '\0' karakter is átkerül, mert a **while** csak a hozzárendelés végrehajtása után veszi csak észre, hogy kifejezése hamissá (zérussá) vált. Tehát:

```
  while(cél[i]=forrás[i]) ++i;
```

☞ Lássuk be, hogy a

```
  while(cél[i++]=forrás[i]);
```

is tökéletesen funkcionál, hisz

- Előállítja a **forrás** tömb **i**–ik elemét.
- Ezt hozzárendeli **cél** tömb **i**–ik eleméhez.
- Az utótag ++ miatt mellékhatásként közben **i** értéke is nő egyet.

A legjobb megoldásunk tehát:

```
void strcopy(char cél[], char forrás[]){
  int i=0;
```

```
  while(cél[i++]=forrás[i]); }
```

A **void strct**(**char** *s1*[], **char** *s2*[]) egyesíti a két paraméter karakterláncot *s1*–be.

Indexeljünk előre az *s1*–en a lánczáró nulláig, s ezután ide kell másolni az *s2* karakterláncot!

✍ Belátható, hogy a helyes megoldáshoz két indexváltozó szükséges, mert a másolásnál az egyiknek *s1* lánczáró nullájának indexétől kell indulnia, míg a másiknak *s2*–n zérustól.

```
void strct(char s1[], char s2[]){
  int i=0, j=0;
  while(s1[i]) ++i;
  while(s1[i++]=s2[j++]); }
```

A **void strup**(**char** *s*[]) nagybetűssé alakítja saját helyén az *s* karakterláncot.

Indexeljünk végig az *s* karaktertömbön a lánczáró zérusig! Egy–egy pozíción meg kell vizsgálni a tömbelem értékét. Ha nem kisbetű, akkor nem kell tenni semmit. Ha kisbetű, át kell írni nagybetűvé. Ha csak az angol ábécé betűivel foglalkozunk, akkor meg kell határozni a kisbetű tömbelem eltolását 'a'–hoz képest, s ezt az eltolást hozzá kell adni 'A'–hoz.

```
void strup(char s[]){
  int i;
  for(i=0; s[i]; ++i)
    if(s[i]>='a'&&s[i]<='z')s[i]=s[i]-'a'+'A'; }
```

🕮 Feltétlenül meg kell említeni, hogy a most megírt karakterlánc kezelő függvények kicsit más névvel, de azonos funkcióval, és paraméterezéssel léteznek a szabvány könyvtárban. Használatukhoz azonban a **STRING.H** fejfájlt be kell kapcsolni. A névlista:

```
strlen → strlen
strcopy → strcpy
strct   → strcat
strup   → strupr
```

Próbáljuk ki a megírt függvényeinket egy rövid programmal! Kérjünk be egy sort a szabvány bemenetről! Alakítsuk át nagybetűssé! Tegyük elé az ELEJE:, és mögé a :VÉGE szöveget! Jelentessük meg az eredményt! Írjuk még ki az eredeti és az egyesített karakterlánc hosszát!

✍ Helyszűke miatt a függvények teste helyett csak /* … */–eket közlünk. A valóságban oda kell másolni a függvénydefiníció forrásszövegét is.

```
/* PELDA13.C: Karakterlánc kezelése */
#include <stdio.h>
#define SOR 80       /* Bemeneti sor max. hossza. */
int getline(char s[],int n){ /* ... */ }
unsigned strlen(char s[]){ /* ... */ }
void strcopy(char cél[], char forrás[]){ /* ... */ }
void strct(char s1[], char s2[]){ /* ... */ }
void strup(char s[]){/* ... */ }
void main(void){
  int n;           /* A bemeneti sor hossza. */
  char sor[SOR+1], /* A bemeneti sor. */
     egy[SOR*2];  /* Az egyesített karakterlánc. */
  printf("Adjon meg egy sort!\n"
       "Nagybetűssé alakítva megjelentetjük\n"
       "ELEJE: :VÉGE szövegekbe zárva.\n"
       "Közöljük az eredeti és a végső hosszt is.\n");
  n=getline(sor,SOR);
  strcopy(egy, "ELEJE:");
  strct(egy, sor);
  strup(egy);
  strct(egy, ":VÉGE");
  printf("%s\nEredeti hossz:%3d\nMostani hossz:%3d\n",
         egy, n, strlen(egy)); }
```

Megoldandó feladatok:

Készítse el a következő függvényeket, és próbálja is ki őket egy rövid programmal!

- A **void strlw**(**char** *s*[]) kisbetűssé alakítja saját helyén az *s* karakterláncot.
- Az **int toup**(**int** *c*) nagybetűssé alakítva visszaadja *c* értékét.
- Az **int tolw**(**int** *c*) visszatérési értéke *c* kisbetűssé alakítva.
- A **void chdel**(**char** *s*[], **int** *c*) kitörli az *s* karakterláncból a saját helyén a benne előforduló *c* karaktereket. Másoljuk át az *s* karakterláncot egy segéd tömbbe úgy, hogy az aktuális karakter csak akkor kerüljön át, ha nem *c*! Az eredmény karakterlánc aztán visszamásolandó s–be! Sokkal jobb, ha a feladatot segéd tömb nélkül oldjuk meg, s a másolást a *c*–k kihagyásával rögtön az *s* tömbbe végezzük.

4.5 Deklaráció

A deklaráció megteremti a kapcsolatot az azonosító és az objektum között, ill. rögzíti például többek között az objektum adattípusát. Kétféle deklarációról beszélhetünk:

- A definíciós deklarációval (definíció) helyet foglaltatunk az objektumnak a memóriában, és esetleg kezdőértékkel is inicializáljuk. Ilyen deklaráció egy azonosítóra csak egyetlen egy létezhet az egész forrásprogramban.
- A referencia deklaráció (deklaráció) nem foglal memóriát az objektumnak, de tudatja a fordítóval, hogy van egy ilyen azonosítójú, adattípusú, stb. objektum a programban. Ebből a deklarációból - egymásnak ellent nem mondóan - több is létezhet ugyanarra az objektumra.
- Szólnunk kell még deklarációs pontról! Ez az azonosító deklarációjának helye a forrásprogramban. Az azonosító deklarációs pontja előtt legálisan nem érhető el a forráskódban.

Tudjuk azt is, hogy a deklaráció elhelyezése és maga a deklaráció meghatározza az objektum attribútumait:

- típus,
- tárolási osztály,
- hatáskör,
- élettartam stb.

A kísérleti definíció fogalmát az ANSI szabvány vezette be. Bármely külső adatdeklaráció, melynek nincs tárolási osztály specifikátora és nincs explicit inicializátora, kísérleti definíciónak tekintendő. Ha a deklarált azonosító aztán feltűnik egy későbbi definícióban, akkor a kísérleti definíciót **extern** tárolási osztályúnak kell venni. Más szóval, a kísérleti definíció egyszerű referencia deklaráció.

Ha a fordítási egység végéig sem találkozunk definícióval a kísérleti definíciós azonosítóra, akkor a kísérleti definíció teljes definícióvá válik tiszta zérus kezdőértékű objektummal.

```
int x;
int x;      /* OK legális, hisz nem mondtunk újat. */
int y;
int y = 4;  /* OK. Most specifikáltuk, hogy y-t 4-re kell
               inicializálni. */
int z = 4;
int z = 6;  /* HIBÁS, mert mindkettő inicializálna. */
```

A deklarálható objektumok a következők:

- változók,

- függvények,
- típusok,
- típusok tömbjei,
- enum konstansok és címkék.

📖 Deklarálhatók még a következő későbbi fejezetekben tárgyalt objektumok is: struktúra, unió és utasítás címkék, struktúra és uniótagok, valamint előfeldolgozó makrók.

A deklarátor szintaktika rekurzivitása miatt egészen komplex deklarátorok is megengedettek. Ezt el szokás kerülni típusdefinícióval (**typedef**). A deklaráció metanyelvi leírása a következő:

deklaráció:
  deklaráció-specifikátorok<init-deklarátorlista>
deklaráció-specifikátorok:
  tárolási-osztály-specifikátor<deklaráció-specifikátorok>
  típusspecifikátor<deklaráció-specifikátorok>
init-deklarátorlista:
  init-deklarátor
  init-deklarátorlista, init-deklarátor
init-deklarátor:
  deklarátor
  deklarátor=inicializátor
inicializátor:
  hozzárendelés-kifejezés
  {*inicializátorlista*}
  {*inicializátorlista ,*}
inicializátorlista:
  inicializátor
  inicializátorlista, inicializátor
tárolási-osztály-specifikátor:
  auto
  register
  extern
  static
  typedef
típusspecifikátor:
  void
  char
  short
  int
  long
  float
  double

signed
unsigned
const
volatile
struktúra-vagy-unió-specifikátor
enum-specifikátor
typedef-név

typedef-név:
azonosító

deklarátor:
<*mutató*>*direkt-deklarátor*

direkt-deklarátor:
azonosító
(*deklarátor*)
direkt-deklarátor [<*konstans-kifejezés*>]
direkt-deklarátor (*paraméter-típus-lista*)
direkt-deklarátor (<*azonosítólista*>)

konstans-kifejezés:
feltételes-kifejezés

azonosítólista:
azonosító
azonosítólista, azonosító

☞ A mutatókkal, a struktúrával, az unióval, a függvényekkel, az utasítás címkékkel és a makrókkal későbbi szakaszokban és fejezetekben foglalkozunk, így a vonatkozó metanyelvi leírások is ott találhatók.

A *tárolási-osztály-specifikátorok*at teljes részletességgel majd egy későbbi fejezetben taglaljuk, de közülük kettőről (**auto** és **extern**) már szó volt a **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban. A következő fejezet némi előzetes képet ad a **typedef**–ről.

A *típusspecifikátor*okat alaptípusokra és típusmódosítókra bontottuk korábban. A típusmódosítók és alaptípusok megengedett, együttes használatát e szakasz előző fejezeteiben tisztáztuk. A **const** és a **volatile** viszont bármely alaptípussal és típusmódosítóval együtt szinte korlátozás nélkül használható.

A *deklarátor* egyedi azonosítót határoz meg. Mikor az azonosító megjelenik egy vele egyezőtípusú kifejezésben, akkor a vele elnevezett objektum értékét eredményezi.

A *konstans-kifejezés* mindig fordítási időben is meghatározható konstans értékké értékelhető ki, azaz értéke bele kell hogy férjen a típus ábrázolási határaiba. A konstans kifejezés bárhol alkalmazható, ahol konstans egyébként használható. A konstans kifejezés operandusai lehetnek egész konstansok, karakter konstansok, lebegőpontos konstansok, enumeráto-

rok, típusmódosító szerkezetek, **sizeof** kifejezések, stb. A konstans kifejezés azonban a **sizeof** operátor operandusától eltekintve nem tartalmazhatja a következő operátorok egyikét sem:

- hozzárendelés,
- vessző,
- dekrementálás,
- inkrementálás és
- függvényhívás.

4.5.1 Elemi típusdefiníció (typedef)

Nem tartozik igazán ide a típusdefiníció tárgyalása, de miután a **typedef** kulcsszót a tárolási osztály specifikátor helyére kell írni, itt adunk róla egy rövid ismertetést.

A **typedef** kulcsszóval nem új adattípust, hanem új adattípus specifikátort definiálunk. Legyen szó például az

```
auto long int brigi;
```

definícióról, mely szerint a **brigi** egy 32 bites, **signed long int** típusú, lokális élettartamú objektum azonosítója. Használjuk most az **auto** kulcsszó helyett a **typedef**-et, és az azonosítót írjuk át nagybetűsre!

```
typedef long int BRIGI;
```

, ahol a **BRIGI** azonosító nem képez futásidejű objektumot, hanem egy új típusspecifikátor csak. A programban ezután a **BRIGI** típusspecifikátorként alkalmazható deklarációkban. Például az

```
extern BRIGI fizetni;
```

ugyanolyan hatású, mint az

```
extern long int fizetni;
```

Ez az egyszerű példa megoldható lenne a

```
#define BRIGI long
```

módon is, a **typedef**-fel azonban az egyszerű szöveghelyettesítésnél komplexebb alkalmazások is áthidalhatók.

A **typedef** nem hoz létre új típust tulajdonképpen, csak létező típusokra kreálható vele új kulcsszó. Komplexebb deklarációk egyszerűsítésére való.

💣 A típusdefiníció „megbonyolítására” a későbbiekben még visszatérünk! Meg kell jegyeznünk azonban annyit, hogy a **typedef**-fel létrehozott típusspecifikátor nem használható a deklarációban más típusspecifikátorokkal együtt! Legfeljebb a **const** és a **volatile** módosítók alkalmazhatók rá! Például

```
unsigned BRIGI keresni;   /* HIBÁS. */
const BRIGI kaba = 2;     /* OK */
```

5 MŰVELETEK ÉS KIFEJEZÉSEK

A műveleteket a nyelv operátorokkal (műveleti jelekkel) valósítja meg. A műveletek lehetnek:

- Egyoperandusosak. Alakjuk „operátor operandus”, ahol az operandus az a kifejezés, melyen az egyoperandusos műveletet el kell végezni. Például: –6, vagy a **sizeof**(**int**) stb.
- Kétoperandusosak, azaz „operandus1 operátor operandus2” formájúak. Például: **a** + **b**.
- Háromoperandusosak: A C-ben egyetlen ilyen művelet van, az ún. feltételes kifejezés. Például: (**a** > **b**) ? **a** : **b** értéke **a**, ha **a** > **b** és **b** máskülönben.

operátor: (a következők egyike!)
[ ] ( ) . -> ++ -- & * + - ~ ! **sizeof** / % << >> < > <= >= == != = ^ | && || ?: *= /= += -= %= <<= >>= &= ^= |= ,

A kifejezés operátorok, operandusok (és elválasztó-jelek) sorozata, mely az alábbi tevékenységek valamilyen kombinációját valósítja meg:

- Értéket számít ki.
- Objektumot vagy függvényt ér el.
- Mellékhatást generál.

A kifejezésbeli operandusokat elsődleges kifejezésnek nevezik.

elsődleges-kifejezés:
azonosító
konstans
karakterlánc
(*kifejezés*)

kifejezés:
hozzárendelés-kifejezés
kifejezés, hozzárendelés-kifejezés

A *konstans*okat, a *karakterlánc*ot tárgyaltuk a **TÍPUSOK ÉS KONSTANSOK** szakaszban, a *hozzárendelés-kifejezés*t definiálni fogjuk a hozzárendelés operátoroknál. Az *azonosító* lehet bármilyen egész vagy lebegőpontos típusú. Lehet **enum,** tömb, mutató, struktúra, unió, vagy függvény típusú. Lehet tehát:

- változó azonosító beleértve az indexelő operátort is, azaz az *azonosító*[*kifejezés*]-t is, és az unió, ill. a struktúratagokat, vagy

- függvényhívás, azaz azonosító a függvényhívás operátorral (*azonosító*()), melynek típusa mindig a függvény által visszaadott érték típusa lesz.

✍ Összesítve: az *azonosító*nak balértéknek vagy függvényhívásnak kell lennie.

A kifejezés kiértékelése bizonyos

- konverziós,
- csoportosító,
- asszociatív és
- prioritási (precedencia)

szabályokat követ, mely függ

- a használt operátoroktól,
- a ( ) párok jelenlététől és
- az operandusok adattípusától.

A kifejezések különfélék lehetnek:

- elsődleges kifejezés (primary),
- utótag kifejezés (postfix),
- egyoperandusos kifejezés (unary),
- előtag kifejezés (cast),
- hozzárendelés kifejezés stb.

✍ Figyeljük meg, hogy a kifejezések elnevezése - az elsődleges kifejezéstől eltekintve - a vele használatos operátorok szerint történik!

utótag-kifejezés:
elsődleges-kifejezés
utótag-kifejezés[*kifejezés*]
utótag-kifejezés(<*kifejezéslista*>)
utótag-kifejezés.azonosító
utótag-kifejezés->azonosító
utótag-kifejezés++
utótag-kifejezés—

kifejezéslista:
hozzárendelés-kifejezés
kifejezéslista , hozzárendelés-kifejezés

egyoperandusos-kifejezés:
  utótag-kifejezés
  ++ egyoperandusos-kifejezés
  -- egyoperandusos-kifejezés
  egyoperandusos-operátor előtag-kifejezés
  sizeof(*egyoperandusos-kifejezés*)
  sizeof(*típusnév*)

egyoperandusos-operátor: (a következők egyike!)
  & * + - ~ !

előtag-kifejezés:
  egyoperandusos-kifejezés
  (*típusnév*) *előtag-kifejezés*

típusnév:
  típusspecifikátor-lista<*absztrakt-deklarátor*>

absztrakt-deklarátor:
  mutató
  <*mutató*><*direkt-absztrakt-deklarátor*>

direkt-absztrakt-deklarátor:
  (*absztrakt-deklarátor*)
  <*direkt-absztrakt-deklarátor*>[<*konstans-kifejezés*>]
  <*direkt-absztrakt-deklarátor*>(<*paraméter-típus-lista*>)

A *típusnév* az adattípus típusneve. Szintaktikailag az adott típusú objektum olyan deklarációja, melyből hiányzik az objektum neve.

A *hozzárendelés-kifejezés*t, melyről most csak annyit jegyzünk meg, hogy nem balérték, majd a hozzárendelési műveleteknél ismertetjük!

5.1 Aritmetikai műveletek (+, -, *, / és %)

Közülük a legmagasabb prioritási szinten az egyoperandusos, jobbról balra kötő előjel operátorok vannak. Létezik a

- előtag-kifejezés

és a szimmetria kedvéért a

+ előtag-kifejezés.

Az előjel operátort követő előtag kifejezésnek aritmetikai típusúnak kell lennie, s az eredmény az operandus értéke (+), ill. annak -1-szerese (-). A + művelet egész operandusát egész–előléptetésnek (integral promotion) veti alá a fordító, s így az eredmény típusa az egész–előléptetés végrehajtása után képzett típus. A – műveletet megelőzheti implicit típuskonverzió, és egész operandus esetén az eredmény az operandus értékének kettes komplemense.

☞ Az egész–előléptetéssel az implicit típuskonverzió kapcsán rögtön foglalkozunk!

A többi aritmetikai operátor mind kétoperandusos, melyek közül a szorzás (*), az osztás (/) és a modulus (%) magasabb prioritási szinten van, mint az összeadás (+) és a kivonás (-). A szorzást, az osztást és a modulust multiplikatív operátoroknak, az összeadást és a kivonást additív operátoroknak is szokás nevezni.

5.1.1 Multiplikatív operátorok (*, / és %)

multiplikatív-kifejezés:
előtag-kifejezés
multiplikatív-kifejezés * *előtag-kifejezés*
multiplikatív-kifejezés / *előtag-kifejezés*
multiplikatív-kifejezés % *előtag-kifejezés*

Nézzük a műveletek pontos szabályait!

- A multiplikatív operátorok mind balról jobbra csoportosítanak.
- Mindhárom operátor operandusainak aritmetikai típusúaknak kell lenniük. A % operátor operandusai ráadásul csak egész típusúak lehetnek.
- Ha az operandusok különböző aritmetikai típusúak, akkor a művelet elvégzése előtt implicit konverziót hajt végre a fordító. Az eredmény típusa ilyenkor a konvertált típus. Miután a konverziónak nincsenek túl vagy alulcsordulási feltételei, értékvesztés következhet be, ha az eredmény nem fér el a konverzió utáni típusban.
- A / és a % második operandusa nem lehet zérusértékű, mert ez fordítási vagy futásidejű hibához vezet.
- Ha a / és a % mindkét operandusa egész, de a hányados nem lenne az, akkor:
  - Ha a két operandus - mondjuk *op1* és *op2* - értéke azonos előjelű vagy **unsigned**, akkor az *op1*/*op2* hányados az a legnagyobb egész, ami kisebb, mint az igazi hányados és az *op1*%*op2* osztási maradék *op1* előjelét örökli meg:

```
 3 / 2       → 1       3 % 2       →  1
(-3) / (-2) → 1       (-3) % (-2) → -1
```

  - Ha *op1* és *op2* ellenkező előjelű, akkor az *op1*/*op2* hányados az a legkisebb egész, ami nagyobb az igazi hányadosnál. Az *op1*%*op2* osztási maradék most is *op1* előjelét örökli meg:

```
(-3) / 2 → -1        (-3) % 2 → -1
3 / (-2) → -1        3 % (-2) →  1
```

Készítsünk programot, ami beolvas egy négyjegyű évszámot, és eldönti róla, hogy szökőév–e, vagy sem!

A Gergely–naptár szerint szökőév minden, néggyel maradék nélkül osztható év. Nem szökőév a kerek évszázad, de a 400–zal maradék nélkül oszthatók mégis azok.

1. Olvassunk be a szabvány bemenetről egy maximálisan négy karakteres sort!
2. Ha a bejött karakterlánc hossza nem pontosan négy, akkor kérjük be újra!
3. Ellenőrizzük le, hogy a karakterlánc minden pozíciója numerikus–e! Ha nem, újra bekérendő.

Írjunk **int nume**(**char** *s*[]) függvényt, mely 1–et (igazat) ad vissza, ha a paraméter karakterlánc tiszta numerikus, és zérust (hamisat), ha nem!

```
int nume(char s[]){
  int i;
  for(i=0; s[i]; ++i) if(s[i]<'0'||s[i]>'9') return 0;
  return 1; }
```

4. Át kéne konvertálni a numerikus karakterláncot fixpontos belsőábrázolású egésszé (**int n**–né)! A módszer a következő:

   ```
   n=(s[0]-'0')*1000+(s[1]-'0')*100+(s[2]-'0')*10+
      (s[3]-'0');
   ```

   Ezt ugye ciklusban, ahol **i** és **n** zérustól indul, és **i** egyesével haladva végigjárja a numerikus karakterláncot, így kéne csinálni:

   ```
   n=n*10+(s[i]-'0');
   ```

Írjunk **int atoi**(**char** *s*[]) függvényt, mely megvalósítja ezt a konverziót, s **n** lesz a visszaadott értéke! Az átalakítást végezze az első nem konvertálható karakterig! Engedjük meg, hogy a numerikus karakterlánc elején fehér karakterek és előjel is lehessen! Ha az előjelet elhagyják, akkor legyen a szám pozitív!

```
int atoi(char s[]){
  int i=0, n=0;
  int elojel=1;    /* Alapértelmezés: pozitív. */
  /* A karakterlánc eleji fehér karakterek átlépése: */
  while(s[i]==' '||s[i]=='\n'||s[i]=='\t') ++i;
  /* Előjel: */
  if(s[i]=='+'||s[i]=='-') if(s[i++]=='-') elojel=-1;
  /* Konverzió: */
  for(;s[i]>='0'&&s[i]<='9';++i) n=10*n+s[i]-'0';
  return(elojel*n); }
```

🕮 Megemlítendő, hogy pontosan ilyen prototípusú, nevű és funkciójú függvény létezik a szabvány könyvtárban is, de bekapcsolandó hozzá a szabványos **STDLIB.H** fejfájl. A könyvtárban van **atol** rutin, mely **long**–gá, és van **atof**, mely **double**–lé alakítja numerikus karakterlánc paraméterét.

☞ Jöhet a program, de helyszűke miatt a függvények definícióit nem ismételjük meg!

```
/* PELDA14.C: A négyjegyű évszám szökőév-e? */
#include <stdio.h>
#define MAX 80
int getline(char s[], int n);
int nume(char s[]);
int atoi(char s[]);
void main(void){
  int ev = 0;      /* Elfogadhatatlan értékről indul. */
  char s[MAX+1];   /* Az input puffer. */
  printf("A négyjegyű évszám szökőév-e?\n");
  while(ev<1000 || ev>9999){
    printf("Adjon meg egy évszámot!\n");
    if(getline(s,4)==4&&nume(s)) ev=atoi(s);
    else printf("Formailag hibás bemenet!\n"); }
  if(ev%4==0&&ev%100!=0||ev%400==0)
    printf("%4d szökőév.\n", ev);
  else printf("%4d nem szökőév.\n", ev); }
```

Megoldandó feladatok:

Készítsen számot leíró karakterláncok formai ellenőrzését végző függvényeket az **atoi** alapján, melyek helyes esetben 1–et (igaz) adnak vissza, és a hibát zérussal (hamis) jelzik! A lánc eleji fehér karaktereket át kell lépni. A szám végét a karakterlánc vége, vagy újabb fehér karakter következése mutatja.

- Az **int egesze**(**char** *s*[]) a decimális egész konstans írásszabályát ellenőrzi a paraméter karaktertömbön.
- Az **int hexae**(**char** *s*[]) megvizsgálja, hogy paramétere hexadecimális szám–e.
- Az **int ell210e**(**char** *s*[]) teszteli, hogy *s* 2 és 10 közötti alapú szám–e. A számrendszer alapját fordítási időben változtatni (#**define**) lehet!
- Az **int ellae**(**char** *s*[], **int** *alap*) ugyanazt teszi, mint az **ell210e**, de a 2 és 10 közötti *alap*ot futási időben paraméterként kapja meg.

- Az **int ella36e**(**char** *s*[], **int** *alap*) egyezik **ellae–**vel, de az *alap* 2 és 36 közötti lehet.

☞ A tíznél nagyobb alapú számrendszerek esetében a számjegyeket az angol ábécé betűivel jelöljük rendre, vagyis 10=A, 11=B stb. A 36–os korlátozás ebből fakad.

Készítsen konverziós függvényeket is a leellenőrzött karakterláncokra az **atoi** mintájára, és a konvertált érték legyen a rutinok visszatérési értéke!

- A **double atofix**(**char** *s*[]) az előjeles, legfeljebb egész és tört részből álló valós értéket alakítja **double**–lé.
- A **long atoh**(**char** *s*[]) hexadecimális karakterláncot alakít egésszé.
- A **long ato36**(**char** *s*[], **int** *alap*) a legfeljebb 36 *alap*ú számrendszerbeli láncot konvertálja egésszé.

5.1.2 Additív operátorok (+ és -)

additív-kifejezés:
multiplikatív-kifejezés
additív-kifejezés + multiplikatív-kifejezés
additív-kifejezés - multiplikatív-kifejezés

Az additív operátorok csoportosítása is balról jobbra történik. Operandusaik az aritmetikai értékeken túl mutatók is lehetnek.

☞ A mutatóaritmetikát majd a mutatók kapcsán ismertetjük!

Aritmetikai operandusok esetén az eredmény a két operandus értékének összege (+), ill. különbsége (–). Egész vagy lebegőpontos operanduson a művelet implicit típuskonverziót is végezhet, ha szükséges. Ilyenkor az eredmény típusa a konvertált típus. Miután a konverziónak nincsenek túl vagy alulcsordulási feltételei, értékvesztés következhet be, ha az eredmény nem fér el a konverzió utáni típusban.

5.1.3 Matematikai függvények

A matematikai függvények nem részei a C nyelvnek. Nyilvánvaló viszont, hogy kifejezések képzésekor szükség lehet rájuk. A C filozófiája szerint a matematikai függvények családját is a szabvány könyvtárban kell elhelyezni, mint ahogyan a szabvány bemenet és kimenet kezelését végző rutinokat.

Az ANSI szabvány pontosan rögzíti ezeket a könyvtári funkciókat, így bármilyen szabványos C fordító és operációs rendszer számára kompatibi-

lis formában léteznie kell. Magyarán: azok a programok, melyek az operációs rendszerrel való kapcsolatukat a szabvány könyvtáron át valósítják meg, minden változtatás nélkül átvihetők az egyik számítógépről a másikra, az egyik operációs rendszerből a másikba. Ezek az úgy nevezett portábilis programok.

A szabvány könyvtár függvényeit, típusait és makróit szabványos fejfájlokban deklarálták. Ezek közül néhánnyal már találkoztunk, másokkal meg még nem:

ASSERT.H	**CTYPE.H**	**ERRNO.H**	**FLOAT.H**	**ISO646.H**
LIMITS.H	**LOCALE.H**	**MATH.H**	**SETJMP.H**	**SIGNAL.H**
STDARG.H	**STDDEF.H**	**STDIO.H**	**STDLIB.H**	**STRING.H**
TIME.H	**WCHAR.H**	**WCTYPE.H**		

A matematikai függvények prototípusai a **MATH.H** fejfájlban helyezkednek el, így használatuk előtt ez a fejfájl bekapcsolandó!

```
#include <math.h>
```

Nem kívánjuk felsorolni és részletezni az összes fejfájlt, az összes függvényt, csak néhány fontosabbat említünk meg közülük. Az olvasótól azonban elvárjuk, hogy a programfejlesztő rendszere segítségéből a további fejfájlokról és rutinokról is tájékozódjék.

A matematikai függvények **double** értéket szolgáltatnak, s néhány kivételtől eltekintve, paramétereik is **double** típusúak. A matematikából ismeretes korlátozások természetesen érvényben maradnak rájuk. A trigonometrikus függvények paramétere, ill. inverzeik visszaadott értéke radiánban értendő.

Néhányat felsorolunk a teljesség igénye nélkül!

sin(*x*)	*x* szinusza.
cos(*x*)	*x* koszinusza.
tan(*x*)	*x* tangense.
asin(*x*)	–1<=*x*<=1 árkusz szinusza. Az értékkészlet: [–¶/2, ¶/2].
acos(*x*)	–1<=*x*<=1 árkusz koszinusza. Az értékkészlet: [0, ¶].
atan(*x*)	*x* árkusz tangense. Az értékkészlet: [–¶/2, ¶/2].
exp(*x*)	Az e^x exponenciális függvény.
log(*x*)	*x*>0 természetes alapú logaritmusa. (ln(*x*)).

log10(*x*)	*x*>0 tízes alapú logaritmusa. (lg(*x*)).
pow(*x*, *y*)	Az x^y hatványfüggvény. Hiba, ha *x*=0 és *y*<=0, ill. ha *x*<0 és *y* értéke nem egész.
sqrt(*x*)	*x*>=0 négyzetgyöke.
floor(*x*)	Az *x*–nél nem nagyobb, legnagyobb egész szám.
fabs(*x*)	Az *x* abszolút értéke.
fmod(*x*, *y*)	*y*!=0 estén *x*/*y* osztás lebegőpontos maradéka, mely *x*–szel egyező előjelű.

A szabvány könyvtári függvények, így a matematikaiak is, a hibát úgy jelzik, hogy valamilyen speciális értéket (**HUGE_VAL**, zérus stb.) adnak vissza, és beállítják a UNIX–tól örökölt, globális

```
extern int errno;
```

(hibaszám) változót a hiba kódjára. A hibakódok az **ERRNO.H** fejfájlban definiált, egész, nem zérusértékű szimbolikus állandók. A **HUGE_VAL** a legnagyobb, pozitív, még ábrázolható **double** érték.

A matematikai rutinok az értelmezési tartomány hibát **EDOM** értékű **errno**–val, és a fordítótól is függő függvény visszatérési értékkel jelzik. Értékkészlet probléma esetén az **errno ERANGE**. A függvény visszatérési érték túlcsorduláskor előjel helyes **HUGE_VAL**, ill. alulcsorduláskor zérus.

Az értelmezési tartomány hiba akkor fordul elő, ha a függvény aktuális paraméterének értéke nincs benn az értelmezési tartományban. Értékkészlet hiba egyértelműen az, ha az eredmény nem ábrázolható **double** értékként.

Például az **sqrt**(–1.) hatására az **errno EDOM**, és a visszakapott érték negatív **HUGE_VAL**.

Megoldandó feladatok:

Készítendő a középiskolás függvénytáblázatok mintájára lapozhatóan:

- egy logaritmustábla és
- egy szinusztábla.

5.2 Reláció operátorok (>, >=, <, <=, == és !=)

A reláció operátorok prioritása - eltekintve az egyoperandusos műveletektől - az aritmetikai és a logikai operátorok között helyezkedik el. A

reláció operátorok két prioritási szintet képeznek, ahol az „igazi” relációk (>, >=, < és <=) prioritása magasabb az egyenlőségi relációkénál (== és !=). Az összes reláció az első operandus értékét hasonlítja a másodikéhoz, és a reláció érvényességét vizsgálja. Az eredmény logikai érték (**int** típusú), mely 1, ha a reláció igaz és 0, ha nem. A definíciók:

relációs-kifejezés:
eltolás-kifejezés
relációs-kifejezés < *eltolás-kifejezés*
relációs-kifejezés > *eltolás-kifejezés*
relációs-kifejezés <= *eltolás-kifejezés*
relációs-kifejezés >= *eltolás-kifejezés*

egyenlőségi-kifejezés:
relációs-kifejezés
egyenlőségi-kifejezés == *relációs-kifejezés*
egyenlőségi-kifejezés != *relációs-kifejezés*

☞ Az *eltolás-kifejezés*t a bitenkénti eltolás operátoroknál definiáljuk!

A relációk operandusai egész, lebegőpontos, vagy mutató típusúak. Az operandusok típusa különbözhet. Az operátorok implicit típuskonverziót is végrehajthatnak aritmetikai operandusaikon a művelet elvégzése előtt.

☞ Ne feledjük, hogy a

```
kifejezés != 0
```

reláció mindig rövidíthető

```
kifejezés
```

módon, mert a nyelvben a nem zérus érték logikai igaznak minősül.

☞ Példaként tekintsük meg újra a korábbi szakaszokban ismertetett **atoi** és **getline** függvényeket!

5.3 Logikai műveletek (!, && és ||)

Közülük a legmagasabb prioritási szinten az egyoperandusos, jobbról balra kötő, logikai nem operátor van, melynek alakja:

! *előtag-kifejezés*

, ahol az *előtag-kifejezés* operandusnak egész, lebegőpontos, vagy mutató típusúnak kell lennie. Az eredmény mindenképpen **int** típusú, s az operandus logikai negációja. Az eredmény 0, ha az operandus értéke nem zérus, ill. 1, ha az operandus értéke zérus. Ez utóbbi mondatrész biztosítja, hogy a

```
kifejezés == 0
```

mindenkor rövidíthető

```
! kifejezés
```

módon. Például a multiplikatív operátoroknál ismertetett program részlet

```
if( ev%4 == 0 && ev%100 != 0 || ev%400 == 0)
```

utasítása így rövidíthető:

```
if( !(ev%4) && ev%100 || !(ev%400))
```

Két kétoperandusos logikai művelet van a nyelvben a logikai és (&&) és a logikai vagy (||), melyek prioritása alacsonyabb a relációkénál és a bit szintű műveleteknél. A logikai és prioritása ráadásul magasabb, mint a logikai vagyé. Mindkét művelet balról jobbra csoportosít. Egyik operátor sem hajt végre implicit típuskonverziót operandusain, ehelyett zérushoz viszonyítva értékeli ki őket. Az eredmény **int** típusú (1 - igaz és 0 - hamis).

logikai-és-kifejezés:
  vagy-kifejezés
  logikai-és-kifejezés && *vagy-kifejezés*

logikai-vagy-kifejezés:
  logikai-és-kifejezés
  logikai-vagy-kifejezés || *logikai-és-kifejezés*

✍ A *vagy-kifejezés* definícióját a bit szintű műveleteknél találjuk meg!

A *K1&&K2* kifejezés eredménye igaz (1), ha *K1* és *K2* egyike sem zérus. A *K1||K2* kifejezés igaz (1), ha *K1* és *K2* valamelyike is nem zérus. Máskülönben *K1&&K2* és *K1||K2* eredménye hamis (0).

💣 Mindkét operátor esetében garantált a balról jobbra történő végrehajtás. Először *K1*–et értékeli ki a fordító az esetleges összes mellékhatásával együtt, de:

- *K1&&K2* esetén, ha *K1* zérus, az eredmény hamis (0), és *K2* kiértékelése nem történik meg.
- *K1||K2* kifejezésnél, ha *K1* nem zérus, az eredmény igaz (1) lesz, és *K2* kiértékelése itt sem zajlik le.

✍ Ha valami előbbre való, vagy mindenképp szeretnénk, hogy megtörténjen, akkor azt a bal oldali operandusba kell beépíteni. Például a **PELDA10.C**–ben megírt **getline for** ciklusának feltétele nem véletlenül

```
i<n && (c=getchar())!=EOF && c!='\n'
```

sorrendű, hisz először azt kell biztosítani, hogy a paraméter karaktertömböt ne írhassa túl a függvény. Ez nem kerülhet hátrébb a kifejezésben. Aztán a következő karaktert előbb be kell olvasni a bemenetről, de min-

mindennek vége van fájlvég esetén. Itt sincs értelme a felcserélésnek, mert felesleges vizsgálgatni a fájlvéget, hogy soremelés–e. A relációk közti és műveletek miatt látszik, hogy balról jobbra történik az operandusok kiértékelése, és ha eközben az egyik hamis lesz, teljesen felesleges lenne továbbfolytatni a kiértékelést.

5.4 Implicit típuskonverzió és egész–előléptetés

Ha kétoperandusos (például aritmetikai) műveleteknél különbözik a két operandus típusa, akkor a művelet elvégzése előtt a fordító belső konverziót (átalakítást) hajt végre. Általában a pontosabb operandus típusára konvertálja a másikat. A kétoperandusos művelet eredményének típusa a konvertált típus lesz. Ezt a fajta konverziót szabványosnak, szokásosnak is nevezik. A szabályok nem prioritási sorrendben a következők:

1. Ha az egyik operandus típusa **long double**, akkor a másikat is **long double** típusúvá konvertálja a fordító.
2. Ha az előző pont nem teljesedik, s az egyik operandus **double**, akkor a másik is az lesz.
3. Ha az előző két feltétel egyike sem valósul meg, és az egyik operandus **float**, akkor a másikat is azzá konvertálja a fordító.
4. Ha az előző három feltétel egyike sem teljesül (egyik operandus sem lebegőpontos!), akkor egész–előléptetést hajt végre a fordító az operandusok értékén, ha szükséges, és aztán:

- Ha az egyik operandus **unsigned long**, akkor a másik is azzá alakul.
- Ha az előző pont nem teljesül, és az egyik operandus **long**, a másik pedig **unsigned int,** akkor mindkét operandus értékét **long**, vagy **unsigned long** típusúvá konvertálja a fordító. Ha az **unsigned int** teljes értéktartománya ábrázolható **long**–ként, akkor a választás **long**, máskülönben pedig **unsigned long**.

☞ Ha az **int** 16, s a **long** 32 bites, akkor –1L<1U. Hisz az elmondottak szerint az **unsigned int long**–gá alakul, s –1L<1L. Ha az **int** 32 bites, akkor –1L>1UL, mert a –1L $11111111111111111111111111111111_2$ binárisan, **unsigned long**–gá alakítva ugyanez marad, és ez sokkal nagyobb $00000000000000000000000000000001_2$–nél.

- Ha az előző pontok nem teljesülnek, és az egyik operandus **long**, akkor a másik is az lesz.
- Ha nem teljesülnek az előző pontok, és az egyik operandus **unsigned int**, akkor a másik is azzá alakul.

- Ha az előző pontok nem teljesülnek, akkor minkét operandus **int** az érvénybe lépett az egész–előléptetés (integral promotion) miatt. A **signed** vagy **unsigned char**, **short int**, vagy bitmező objektumok, ill. az **enum** típusúak használhatók kifejezésben ott, ahol bennük egész állhat. Ha az eredeti típus minden lehetséges értékét **int** képes reprezentálni, akkor az értéket **int** típusúvá konvertálja a fordító, máskülönben **unsigned int**-té. Az egész–előléptetési folyamat garantálja, hogy a konverzió előtti és utáni érték ugyanaz marad. A konvertált érték **unsigned** eredeti típusból 0X00 feltöltéssel, **signed** eredeti típusból viszont előjel kiterjesztéssel készül a felső bájtokba.

Típus	Konvertálva	Módszer
char	**int**	Az alapértelmezett **char** típustól függően előjel kiterjesztés van (**signed**) vagy 0X00 kerül a magasabb helyiértékű bájt(ok)ba (**unsigned**).
unsigned char	**int**	A felső bájt(ok) 0X00 feltöltésűek.
signed char	**int**	Előjel kiterjesztés van a felső bájt(ok)-ba.
short int	**int**	Ugyanaz az érték előjel kiterjesztéssel.
unsigned short	**unsigned int**	Ugyanaz az érték 0X00 feltöltéssel.
enum	**int**	Ugyanaz az érték.

12. ábra: Egész–előléptetés

Ne feledjük azonban el, hogy a konverzió mindig függvényhívást jelent (gépidő!), azaz szükségtelenül ne alkalmazzuk! Csak „értelmes” típuskonverziókat valósít meg a fordító. Például az **f** + **i** összeadás végrehajtása előtt - feltéve, hogy **f float** és **i int** típusú - **i** értéke (és nem **i** maga!) **float**-tá alakul.

Az „értelmetlen” lebegőpontos kifejezés indexben még csak megvalósul úgy, hogy a kifejezés értéke tört részét levágja a fordító

```
#include <stdio.h>
void main(void){
  int t[] = {2,3,4,5,6,7};
  float f=1.75;
  printf("%d\n",t[f]); }
```

, azaz 3 lesz az eredmény.

💣 Numerikus karakterlánc azonban sohasem alakul automatikusan aritmetikai értékké. Ehhez valamilyen konverziós függvényt kell használni. Az **STDLIB.H**–beli **atoi**–ról, **atol**–ról és **atof**–ról volt már szó!

5.5 Típusmódosító szerkezet

Az implicit (szokásos, szabványos stb.) konverziókon túl magunk is kikényszeríthetünk (explicit) típuskonverziót a

(*típusnév*) *előtag-kifejezés*

alakú, a **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban megismert típusmódosító szerkezet alkalmazásával. Látjuk, hogy a típusmódosító szerkezet egyoperandusos, s ez által magas prioritású művelet. A definícióban a *típusnév* a céltípus, és az *előtag-kifejezés* értékét erre a típusra kell konvertálni. Az *előtag-kifejezés*t úgy konvertálja a fordító, mintha az értéket egy *típusnév* típusú változó venné fel. Az explicit típuskonverzió tehát a hozzárendelési konverzió szabályait követi. A legális típusmódosítások:

Céltípus	**Potenciális források**
egész	bármilyen egész vagy lebegőpontos típus, vagy mutató
lebegőpontos	bármilyen aritmetikai típus
void	bármilyen típus

Példaként vegyük a matematikai függvények közül a négyzetgyököt, azaz:

```
#include <math.h>
double sqrt(double x);
```

Programunkban van egy **int** típusú **n** változó, akkor az **n**+26 pozitív gyökét az

```
sqrt(double(n+26))
```

függvényhívással kaphatjuk meg.

💣 Bármilyen azonosító, vagy kifejezés típusa módosítható **void**–dá. A típusmódosításnak alávetett azonosító, vagy kifejezés nem lehet azonban **void**. A **void** függvény hívását például hiába típusmódosítjuk **int-**re, a semmiből nem lehet egészet csinálni.

💣 A **void**–dá módosított kifejezés értéke nem képezheti hozzárendelés tárgyát. Hasonlóan az explicit típusmódosítás eredménye nem fogadható el balértékként hozzárendelésben.

Kifejezést csak olyan helyen módosíthatunk **void**-dá, ahol az értékére nincs szükség. Például nincs szükség a bejövő gombnyomásra:

```
printf("A folytatáshoz üssön Entert-t! "); (void)getchar();
```

5.6 sizeof operátor

A **BEVEZETÉS ÉS ALAPISMERETEK** szakaszból ismert **sizeof** egyoperandusos, jobbról balra kötő, magas prioritású művelet, mely mindig az operandusa tárolásához szükséges memória mennyiségét szolgáltatja bájtban. Az eredmény **size_t** típusú egész érték. Az **STDDEF.H** fejfájlban megtekintve a típust többnyire azt látjuk, hogy **unsigned int**.

✍ A **size_t** típus értelmezése fordítótól függ tulajdonképpen!

Két különböző alakja van az operátornak:

sizeof(*egyoperandusos-kifejezés*)
sizeof(*típusnév*)

sizeof(*egyoperandusos-kifejezés*) esetén az egyoperandusos kifejezés típusát a fordító a kifejezés kiértékelése nélkül határozza meg, azaz ha az operandus tömbazonosító, az egész tömb bájtokban mért helyfoglalásához jutunk. Például a **tomb** tömb elemszáma a következő konstrukcióval állapítható meg:

```
sizeof(tomb) / sizeof(tomb[0])
```

💣 A **sizeof** nem használható függvényre, vagy nem teljes típusú kifejezésre, ilyen típusok zárójelbe tett nevére, vagy olyan balértékre, mely bitmező objektumot jelöl ki.

✍ A **sizeof** azonban bátran alkalmazható előfeldolgozó direktívákban is!

```
#define MERET sizeof(int)*3
```

5.7 Inkrementálás (++), dekrementálás (--) és mellékhatás

Ezek az operátorok mind egyoperandusosak, s ezért magas prioritásúak. Mindkét operátor létezik utótag

utótag-kifejezés++
utótag-kifejezés--

és előtag műveletként:

++ *egyoperandusos-kifejezés*
-- *egyoperandusos-kifejezés*

Az inkrementáló és dekrementáló kifejezésben az *utótag*- vagy az *egyoperandusos-kifejezés*nek skalár (aritmetikai vagy mutató) típusúnak és módosítható balértéknek kell lenniük, de az eredmény nem balérték.

Az inkrementálásnál (++) a balérték eggyel nagyobb, dekrementálásnál (--) viszont eggyel kisebb lesz. Előtag operátornál a „konstrukció" értéke egyezik az új balértékkel, míg utótag operátornál a „konstrukció" értéke az inkrementálás vagy dekrementálás végrehajtása előtti érték. Az eredmény típusát az operandus típusa határozza meg. Például:

```
int x, i = 3, j = 4, n = 5;
x = n++;                    /* x == 5 és n == 6 */
x = ++n;                    /* x == 7 és n == 7 */
x = --( n - j + i +6);      /* x == 11 */
```

A kifejezés produkálhat

- balértéket,
- jobbértéket vagy
- nem szolgáltat értéket.

A kifejezésnek ezen kívül lehet mellékhatása is. Például a **TÍPUSOK ÉS KONSTANSOK** szakaszban megírt **strcopy** záró sorában

```
while(cél[i++]=forrás[i])
```

a hozzárendelés mellékhatásaként az **i** végrehajtás utáni értéke is eggyel nagyobb lesz. A mellékhatást a kifejezés kiértékelése okozza, s akkor következik be, ha megváltozik egy változó értéke. Minden hozzárendelés operátornak van mellékhatása. Láttuk, hogy a balértékre alkalmazott inkrementálási és dekrementálási műveletnek is van. Függvényhívásnak is lehet azonban mellékhatása, ha globális hatáskörű objektum értékét változtatja meg.

Írjuk meg a **void chdel**(**char** *s*[], **int** *c*)–t, mely saját helyen törli a benne előforduló *c* karaktereket az *s* karakterláncból!

Itt is másolni kell a forrásból a célba bájtról–bájtra haladva, de a *c* értékű karaktereket ki kell ebből hagyni. Két indexre van szükség. Az egyik az **i**, mely végigjárja a forrást. A másik a **j**, mely a célban mindig a következő szabad helyet éri el. A nem *c* értékű karaktert a következő szabad helyre kell másolni, s a célbeli indexnek az ezután következő szabad helyre kell mutatnia.

```
void chdel(char s[], int c){
  int i, j;
  for(i=j=0; s[i]; ++i) if(s[i] != c) s[j++] = s[i];
```

```
s[j] = 0; }
```

5.8 Bit szintű operátorok (~, <<, >>, &, ^ és |)

A bit szintű operátorok csak **signed** és **unsigned** egész típusú adatokra: **char**, **short**, **int** és **long** használhatók.

Legmagasabb prioritási szinten az egyoperandusos, jobbról balra kötő egyes komplemens operátor (~) van, melynek definíciója:

~ előtag-kifejezés

Az operátor előbb végrehajtja az egész–előléptetést, ha szükséges. Az eredmény típusa az operandus konverzió utáni típusa. Az eredmény maga a bit szintű egyes komplemens, azaz ahol az operandus bit 1 volt, ott az eredmény bit 0 lesz, és ahol az operandus bit 0 volt, ott az eredmény bit 1 lesz. Feltéve, hogy az egész–előléptetés 16 bites, és hogy:

```
unsigned short x = 0XF872,     /* 1111100001110010 */
          maszk = 0XF0F0;      /* 1111000011110000 */
```

, akkor a ~**x** 0000011110001101_2, és a ~**maszk** 0000111100001111_2.

A balról jobbra csoportosító eltolás operátorok (<< és >>) prioritása alacsonyabb az aritmetikai műveletekénél, de magasabb, mint a reláció operátoroké. Az eltolás operátorok első operandusuk értékét balra (<<) vagy jobbra (>>) tolják annyi bitpozícióval, mint amennyit a második operandus meghatároz. A definíció a következő:

eltolás-kifejezés:
additív-kifejezés
eltolás-kifejezés << additív-kifejezés
eltolás-kifejezés >> additív-kifejezés

A *K1<<K2* és a *K1>>K2* kifejezések esetében minkét operandus egész típusú kell, legyen. Az operátorok egész–előléptetést is megvalósíthatnak. Az eredmény típusát *K1* konvertált típusa határozza meg. Ha *K2* negatív vagy értéke nem kisebb *K1* bitszélességénél, akkor az eltolási művelet eredménye határozatlan.

Miután a C–ben nincs egész alul vagy túlcsordulás, a műveletek értékvesztést is okozhatnak, ha az eltolt eredmény nem fér el az első operandus konvertált típusában.

A *K1<<K2* balra tolja *K1* értékét *K2* bitpozícióval úgy, hogy jobbról 0 bitek jönnek be. *K1* túlcsordulás nélküli balra tolása ekvivalens $K1*2^{K2}$–vel. Ilyen értelemben aztán az eltolás aritmetikai műveletnek is tekinthető. Ez a gondolatsor igaz persze az összes bit szintű műveletre is! Ha *K1* valamilyen **signed** típus, akkor az eltolás eredménye csak „gonddal” szemlélhető az előjel bit esetleges kitolása miatt.

A *K1>>K2* művelet *K1* értékét *K2* bitpozícióval tolja jobbra. Ha *K1* valamilyen **unsigned** típusú, akkor balról 0 bitek jönnek be. Ha *K1* **signed**, akkor az operátor az előjel bitet sokszorozza. **unsigned**, nem negatív *K1* esetén a jobbra tolás $K1/2^{K2}$ hányados egész részeként is interpretálható.

Folytatva az egyes komplemensképzésnél megkezdett példát, az **x**<<2 1110000111001000_2, ill. a **maszk**>>5 0000011110000111_2.

A bit szintű logikai operátorok prioritásuk csökkenő sorrendjében az és (&), a kizáró vagy (^), valamint a vagy (|). A többi műveletre való tekintettel prioritásuk magasabb a kétoperandusos logikai operátorokénál, de alacsonyabb a relációkénál. Lássuk a definíciót!

és-kifejezés:
  egyenlőségi-kifejezés
  és-kifejezés & *egyenlőségi-kifejezés*
kizáró-vagy-kifejezés:
  és-kifejezés
  kizáró-vagy-kifejezés ^ *és-kifejezés*
vagy-kifejezés:
  kizáró-vagy-kifejezés
  vagy-kifejezés | *kizáró-vagy-kifejezés*

☞ Az *egyenlőségi-kifejezés* definíciója a relációknál megtalálható!

Ha szükséges, akkor a művelet elvégzése előtt a fordító implicit típuskonverziót hajt végre az egész típusú operandusok értékén. Az eredmény típusa az operandusok konverzió utáni típusa. A művelet bitről-bitre valósul meg az operandusok értékén, s egy bitre vonatkoztatva az eredmény így néz ki:

K1	*K2*	*K1 & K2*	*K1 ^ K2*	*K1 | K2*
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Befejezve az egyes komplemensnél megkezdett példát, az **x**|**maszk** értéke 1111100011110010_2. Állíthatjuk, hogy az eredményben minden olyan bit egy, ami a **maszk**–ban az volt. Az **x**^**x** eredménye biztosan tiszta zérus. Az **x**&~**maszk** értéke 0000100000000010_2. Megemlítjük, hogy az eredmény minden olyan bitpozícióján 0 van, ahol a **maszk** bitje 1. Tehát a kifejezés azokat a biteket bizonyosan törölte, ahol a **maszk** bit 1 volt.

💣 Nem szabad összekeverni a logikai vagy (||) és a bitenként vagy (|), ill. a logikai és (&&) és a bit szintű és (&) műveleteket! Feltéve, hogy két, egész típusú változó értéke: **a**=2 és **b**=4, akkor az

```
a && b  →  1 (igaz)
```

de az

```
a & b  →  0
```

🕮 Az ANSI szabvány szerint a bit szintű műveletek **signed** egészeken implementációfüggők. A legtöbb C fordító azonban **signed** egészeken ugyanúgy dolgozik, mint **unsigned**–eken. Például **short int**–ben gondolkodva a –16 & 99 eredménye 96, mert:

```
1111111111110000&0000000001100011 → 0000000001100000
```

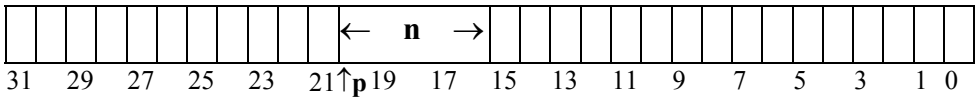
A fájl utolsó módosításának dátumát és idejét egy-egy szóban, azaz C nyelvi fogalmakkal: egy-egy **unsigned short int**-ben tároljuk. A két szó bitfelosztása legyen a következő:

dátum	év – 1980							hónap				nap				
idő	óra					perc						2 másodperc				
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A két szó azonosítója legyen **datum** és **ido**! Tételezzük fel, hogy az idő részeit az **ora**, a **perc** és az **mp unsigned short** változókban tároljuk! Az ugyanilyen **unsigned short** változók a dátum részeire: **ev**, **ho** és **nap**. Akkor az oda–visszaalakítás a következő:

```
/* Részeiből az idő szó előállítása: */
ido = ora << 11 | perc << 5 | mp >> 1;
/* Az idő szóból a részei: */
ora = ido >> 11;
perc = ido >> 5 & 0X3F;
mp = (ido & 0X1F) << 1;
/* Részeiből a dátum szó előállítása: */
datum = ev - 1980 << 9 | ho << 5 | nap;
/* A datum szóból a részei: */
ev = (datum >> 9) + 1980;
ho = datum >> 5 & 0XF;
nap = datum & 0X1F;
```

Írjunk **unsigned long invertal**(**unsigned long** *x*, **int** *p*, **int** *n*) függvényt egy rövid kipróbáló programmal, mely az **x** paramétere értékét a **p**.–ik bitpozíciójától **n** hosszban invertálja (az 1–eseket 0–kra, s a 0–kat 1–esekre cseréli)! A nem említett bitpozíciók értéke maradjon változatlan! Az invertált eredmény a függvény visszatérési értéke.



Az ábra **x** paramétert, a zérustól induló, 20 értékű **p** bitpozíciót és az **n**=5 bitszélességet szemlélteti.

Készítsünk előbb egy ugyancsak **unsigned long maszk**ot, mely **p** pozíciótól **n** szélességben 1–es biteket tartalmaz, és az ezen kívüli pozíciók mind nullák!

```
              ~0: 11111111111111111111111111111111
           ~0<<n: 11111111111111111111111111100000
       ~(~0<<n): 00000000000000000000000000011111
~(~0<<n)<<(p+1-n): 00000000000111110000000000000000
```

Ezután **x** egyes komplemenséből bitenkénti éssel kivágjuk a kérdéses invertált bitpozíciókat, azaz: **~x&maszk**. Ezt aztán bitenkénti vagy kapcsolatba hozzuk az eredeti **x** egy olyan változatával, melyben kinulláztuk az érdekes biteket, azaz: **x&~maszk**. Tehát:

```
/* PELDA15.C: Bitpozíciók invertálása. */
#include <stdio.h>
unsigned long invertal(unsigned long x, int p, int n){
  unsigned long maszk=~(~0<<n)<<(p+1-n);
  return ~x&maszk|x&~maszk; }
void binaris(unsigned long x){
  unsigned long maszk;
  for(maszk=0X80000000; maszk; maszk=maszk>>1)
    if(maszk&x) putchar('1'); else putchar('0'); }
void main(void){
  unsigned long x=0X4C59E9FA;
  int p=20, n=5;
  printf("Az eredeti értéket: ");
  binaris(x);
  printf("\nEzt invertáljuk %d bitpozíciótól %d "
         "bitszélességben.\n", p, n);
  printf("Az invertált érték: ");
  binaris(invertal(x, p, n));
  putchar('\n'); }
```

Vegyük észre, hogy a **binaris** függvény segítségével **unsigned long** értéket jelentetünk meg binárisan! A **maszk** változó 31–es bitpozíciójától indítunk egy 1–est, s a ciklusmag végrehajtása után mindig eggyel jobbra toljuk. A **maszk** és az érték bit szintű és kapcsolata akkor szolgáltat nem zérust (igazat), ha a kérdéses bitpozíción az értékben 1 van.

Megoldandó feladatok:

Készítse el a következő függvényeket, és próbálja is ki őket egy rövid programmal!

- Az **unsigned long getbits**(**unsigned long** *x*, **int** *p*, **int** *n*) *x* értékét szolgáltatja a *p*.–ik bitpozíciótól *n* bitszélességben.
- Az **unsigned long rotl**(**unsigned long** *x*) 1 bittel balra forgatja paramétere értékét. Tehát a 31. pozícióról kicsorgó bit lesz az eredmény 0. bitje.
- Az **unsigned long rotr**(**unsigned long** *x*) 1 bittel jobbra forgat.
- Az **unsigned long tobbrotl**(**unsigned long** *x*, **int** *n*) *n* bittel forgat körbe balra.
- Az **unsigned long tobbrotr**(**unsigned long** *x*, **int** *n*) *n* bittel forgat körbe jobbra.

5.9 Feltételes kifejezés (? :)

Ez a nyelvben az egyetlen három operandusos művelet, melynek prioritása alacsonyabb a kétoperandusos logikai operátorokénál. A definíció:

feltételes-kifejezés:
  logikai-vagy-kifejezés
  logikai-vagy-kifejezés ? *kifejezés* : *feltételes-kifejezés*
kifejezés:
  hozzárendelés-kifejezés
  kifejezés , *hozzárendelés-kifejezés*

A *logikai-vagy-kifejezés*nek egész, lebegőpontos vagy mutató típusúnak kell lennie, s kiértékelése zérushoz való hasonlítását jelenti:

- Ha *logikai-vagy-kifejezés* nem zérus, a *kifejezés*t értékeli ki a fordító. Ez azt jelenti, hogy a *kifejezés* kiértékelése csak akkor történik meg, ha a *logikai-vagy-kifejezés* igaz.
- Ha *logikai-vagy-kifejezés* zérus, a *feltételes-kifejezés*t határozza meg a fordító, azaz a *feltételes-kifejezés* kiértékelése csak akkor történik meg, ha a *logikai-vagy-kifejezés* hamis.

A *logikai-vagy-kifejezés*t mindenképpen kiértékeli a kód, de a *kifejezés* és a *feltételes-kifejezés* közül csak az egyik kiszámítása történik meg.

A *K1* ? *K2* : *K3* feltételes kifejezésben *K1* értékétől függően *K2*–t vagy *K3*–at értékeli ki a fordító. A konstrukció eredményének típusa ilyen alapon *K2* vagy *K3* operandusok típusától függ a következőképp:

- Ha mindkettő aritmetikai típusú, akkor az esetleg szükséges implicit típuskonverzió után az eredmény típusa a konvertált típus.
- Ha a két operandus ugyanolyan struktúra, unió, vagy mutató típusú, akkor az eredmény is a közös típusú.
- Ha mindkettő **void** típusú, akkor az eredmény is az.

Például az

```
if( a > b ) z = a;
else z = b;
```

utasítás helyettesíthető a

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

feltételes kifejezéssel. Ha például egy **int** típusú **a** tömb első **N** elemét szeretnénk megjelentetni úgy, hogy egy sorba egymástól szóközzel elválasztva 10 elem kerüljön, akkor ezt „tömör” kódot alkalmazva így is megtehetjük:

```
for(i=0; i<N; ++i)
  printf("%6d%c", a[i], (i%10==9|| i==N-1) ? '\n': ' ');
```

Javítsunk ki néhány eddig megírt függvényt a feltételes kifejezés felhasználásával!

- A **PELDA15.C binaris** függvényében az

  ```
  if(maszk&x) putchar('1'); else putchar('0');
  ```

  most így is írható:

  ```
  putchar((maszk&x)? '1': '0');
  ```

- A **PELDA14.C atoi** rutinjában az

  ```
  if(s[i]=='+'||s[i]=='-') if(s[i++]=='-') elojel=-1;
  ```

  sor átírható a következőre:

  ```
  if(s[i]=='+'||s[i]=='-') elojel=(s[i++]=='-')? -1: 1;
  ```

Mindkét példában a feltételes kifejezés *logikai-vagy-kifejezés*e köré zárójelet tettünk. Lássuk azonban be, hogy erre semmi szükség sincs, hisz ennél alacsonyabb prioritású művelet már csak kettő van: a hozzárendelés és a vessző operátor! A felesleges zárójel legfeljebb a jobban olvashatóságot biztosítja.

5.10 Hozzárendelés operátorok

A jobbról balra csoportosító hozzárendelés prioritása alacsonyabb, mint a feltételes kifejezésé, s ennél alacsonyabb prioritású művelet már csak a

vessző operátor. A művelet a jobb oldali operandus értékét rendeli a bal oldali operandushoz, melyből következőleg a bal oldali operandusnak módosítható balértéknek kell lennie. A hozzárendelés kifejezés értéke ugyan egyezik a bal oldali operandus hozzárendelés végrehajtása utáni értékével, de nem balérték. A jobb oldali operandus értékét a fordító a bal oldali operandus típusára konvertálja a bal operandusba való letárolás előtt a hozzárendelési konverzió szabályai szerint. A bal oldali operandus nem lehet tömb, függvény vagy konstans. Nem lehet természetesen nem teljes (még nem teljesen deklarált) típusú sem. A definíció:

hozzárendelés-kifejezés:
feltételes-kifejezés
egyoperandusos-kifejezés hozzárendelés-operátor hozzárendelés-kifejzés

hozzárendelés-operátor: (a következők egyike!)
= *= /= %= += -= &= ^= |= <<= >>=

Van tehát egyszerű hozzárendelés operátor (=) és vannak összetettek vagy kombináltak (ezek a többiek).

Foglalkozzunk előbb az egyszerű hozzárendeléssel!

A *K1 = K2* kifejezésben *K1*-nek módosítható balértéknek kell lennie. *K2* értéke - esetlegesen a *K1* típusára történt konverzió után - felülírja a *K1* által kijelölt objektum értékét. Az egész „konstrukció” értéke *K2* értéke az esetleg a *K1* típusára szükségessé vált hozzárendelési konverzió végrehajtása után.

✍ Reméljük, hogy nem felejtették még el, hogy a balérték (*K1*) és jobbérték (*K2*) fogalom éppen az egyszerű hozzárendelésből származik. A balérték a hozzárendelés operátor bal oldalán, a jobbérték pedig a jobb oldalán állhat.

✍ Tudjuk, ha egy kifejezésben hozzárendelés operátor is van, akkor annak a kifejezésnek bizonyosan van mellékhatása.

✍ Emlékezzünk vissza, hogy a definíció megengedi a hozzárendelés operátor

K1 = K2 = K3 = ... = Kn = kifejezés

formájú használatát is, amikor is a *kifejezés* kiértékelése után jobbról balra haladva az operandusok felveszik a *kifejezés* értékét. Az egész konstrukció értéke most is a *kifejezés* értéke lesz. Például

```
a = b= c = d + 6;
```

A kombinált hozzárendelés operátorok a

K1 = K1 operátor K2

kifejezést

K1 operátor = K2

módon rövidítik, és *K1* kiértékelése csak egyszer történik meg. A megengedett *operátor*ok a definícióban láthatók! Mindegyik megvalósítja azt a műveletet, konverziót és korlátozást, amit a kétoperandusos operátor egyébként realizál, és végrehajtja a hozzárendelést is. A kombinált hozzárendelés operátor operandusai egész vagy lebegőpontos típusúak lehetnek általában. A += és −= bal oldali operandusa mutató is lehet, amikor is a jobb oldali operandus köteles egész típusú lenni.

Például:

```
x = x * ( y + 6);  →  x *= y + 6;
```

Az összetett operátorokat használva kevesebbet kell írni. Például:

```
t[ t1[i3 + i4] + t2[i1 - i2]] += 56;
```

Használjunk kombinált hozzárendelés operátorokat néhány eddig már megírt függvényben!

- A **PELDA15.C binaris** rutinja:

```
void binaris(unsigned long x){
  unsigned long maszk;
  for(maszk=0X80000000; maszk; maszk>>=1)
    putchar((maszk&x)? '1': '0'); }
```

- A **PELDA4.C**–beli

```
for(ft=ALSO; ft<=FELSO; ft=ft+LEPES)
```

most így írható:

```
for(ft=ALSO; ft<=FELSO; ft+=LEPES)
```

5.11 Hozzárendelési konverzió

Hozzárendelésnél a hozzárendelendő érték típusát a hozzárendelést fogadó változó típusára konvertálja a fordító. A C megengedi a hozzárendelési konverziót lebegőpontos és egész típusok között azzal, hogy a konverziónál értékvesztés történhet. A használatos konverziós módszer követi az implicit típuskonverzió szabályait és ezen túl még a következőket:

- Konverzió **signed** egész típusokról: Nem negatív **signed** egész nem kisebb méretű **unsigned** egésszé alakításakor az érték változatlan. Például **signed char unsigned char**–rá válásakor a bitminta változatlan, de a legmagasabb helyiértékű bit elveszti az előjelbit funkcióját. A konverzió különben a **signed** egész előjel kiterjesztésével

történik. Például **signed char unsigned long**–gá úgy válik, hogy előjel kiterjesztéssel előbb **long** lesz belőle, s aztán ez a bitminta megtartásával, és előjelbit funkcióvesztéssel lesz **unsigned long**. Hosszabb egész típus rövidebbé alakulásakor az egész típus maradó alsó bitjei változatlanok, s a fölösleg egyszerűen levágódik még akkor is, ha értékvesztés történne. **long int** érték **float** lebegőpontossá alakításakor nincs értékvesztés, de pontosságvesztés lehet. Ilyen átalakításkor a rövidebb **signed** egészből előbb **long** lesz előjel kiterjesztéssel, s csak ezután jön a lebegőpontos konverzió.

✍ Miután az **enum** típus **int** definíció szerint, a felsorolás típusra és típusról való konverzió egyezik az **int**–évl.

- Konverzió **unsigned** egész típusokról: Rövidebb **unsigned** vagy **signed** egésszé alakításkor a maradó alsó bitek változatlanok, s a fölösleg egyszerűen levágódik még akkor is, ha értékvesztés történik. Az eredmény legfelső bitje felveszi az előjelbit funkciót, ha **signed**–dé konvertálás volt. Hosszabb **unsigned** vagy **signed** egésszé alakításkor a bejövő magasabb helyiértékű bitek nulla feltöltésűek. Lebegőpontos konverziónál a rövidebb **unsigned** egészből előbb **long** lesz nulla feltöltéssel, s csak ezután jön az igazi konverzió. Megállapíthatjuk itt is, hogy lehet pontosságvesztés **float**–tá alakításkor.

- Konverzió lebegőpontos típusokról: A rövidebb lebegőpontos ábrázolás hosszabbá konvertálásakor nem változik meg az érték. A hosszabb lebegőpontos ábrázolás **float**–tá alakítása is pontos, ha csak lehetséges. Pontosságvesztés is bekövetkezhet, ha értékes jegyek vesznek el a mantisszából. Ha azonban az eredmény a **float** ábrázolási korlátain kívül esne, akkor a viselkedés definiálatlan.

✍ Ez a definiálatlanság tulajdonképpen a fordítótól függő viselkedést takar!

A lebegőpontos értéket úgy konvertál egésszé a fordító, hogy levágja a törtrészt. Az eredmény előre megjósolhatatlan, ha a lebegőpontos érték nem fér be az egész ábrázolási korlátaiba. Különösen definiálatlan a negatív lebegőpontos érték **unsigned**–dé alakítása.

🕮 Konverzió más típusokról: Nincs konverzió a struktúra és az unió típusok között. Explicit típusmódosítással bármilyen érték konvertálható **void** típusúvá, de csak abban az értelemben, hogy a kifejezés értékét elvetjük. A **void** típusnak definíció szerint nincs értéke. Ebből következőleg nem konvertálható más típusúra, s más típus sem konvertálható **void**–ra hozzárendeléssel.

5.12 Vessző operátor

Ez a legalacsonyabb prioritású művelet.

kifejezés:
  hozzárendelés-kifejezés
  kifejezés , *hozzárendelés-kifejezés*

A *K1*, *K2*, ..., *Kn* esetén balról jobbra haladva kiértékeli a fordító a *kifejezés*eket úgy, hogy a bennük foglalt minden mellékhatás is megvalósul. Az első *n* - 1 *kifejezés* **void**–nak tekinthető, mert az „egész konstrukció” típusát és értékét a legjobboldalibb *kifejezés* típusa és értéke határozza meg. Például a

```
regikar = kar, kar = getchar()
```

esetén **regikar** felveszi **kar** pillanatnyi értékét, aztán **kar** és az egész kifejezés értéke a szabvány bemenetről beolvasott karakter lesz. Tipikus példa még a több kezdőérték adás és léptetés a **for** utasításban:

Írjuk meg egy rövid kipróbáló programmal a **void strrv**(**char** *s*[]) függvényt, mely saját helyén megfordítja a paraméter karakterláncot!

Az algoritmusról annyit, hogy a karakterlánc első karakterét meg kell cserélni az utolsóval, a másodikat az utolsó előttivel, és így tovább. Két indexet kell indítani a karaktertömbben: egyet alulról és egyet felülről. Az alsót minden csere után meg kell növelni eggyel, a felsőt pedig ugyanennyivel kell csökkenteni. A ciklus tehát addig mehet, míg az alsó index kisebb a felsőnél.

```
/* PELDA16.C: Karakterlánc megfordítása. */
#include <stdio.h>
#include <string.h>  /* Az strlen miatt! */
#define INP 66       /* Az input puffer mérete. */
void strrv(char s[]){
  int also, felso, csere;
  for(also=0, felso=strlen(s)-1; also<felso;
      ++also, --felso){
    csere=s[also]; s[also]=s[felso]; s[felso]=csere; } }
int getline(char s[],int n){
  int c,i;
  for(i=0;i<n&&(c=getchar())!=EOF&&c!='\n';++i) s[i]=c;
  s[i]='\0';
  while(c!=EOF&&c!='\n') c=getchar();
  return(i); }
void main(void){
  char s[INP+1];     /* Az input puffer. */
  printf("A szabvány bemenet sorainak megfordítása.\n"
         "Programvég: üres sor.\n\n");
  while(printf("Jöhet a sor! "), getline(s,INP)){
    strrv(s);
```

```
    printf("Megfordítva: %s\n", s); } }
```

🕮 A **STRING.H** fejfájlt bekapcsolva rendelkezésre áll az **strrv** szabvány könyvtári változata, mely ugyanilyen paraméterezésű és funkciójú, de **strrev** a neve, s más egy kicsit a visszatérési értékének a típusa.

☞ Vegyük észre, hogy a **main**–beli **while**–ban is vesszős kifejezést használtunk!

Olyan szövegkörnyezetben, ahol a vessző szintaktikai jelentésű, a vessző operátort csak zárójelbe tett csoporton belül szabad használni. Ilyen helyek: az inicializátorlista például, vagy a függvény paraméterlistája. A

```
fv(b, (a = 2, t += 3), 4);
```

kitűnően mutatja, hogy az **fv** függvény három paraméterrel rendelkezik. A hívásban a második paraméter vesszős kifejezés, ahol **a** előbb 2 értékű lesz, aztán **t** megnő hárommal, s ezt az értéket kapja meg a függvény is második aktuális paraméterként. Ha a függvény prototípusa mást nem mond, akkor a második paraméter típusa **t** típusa.

5.13 Műveletek prioritása

A műveletek prioritását nevezik precedenciának, vagy rendűségnek is. Mindhárom esetben arról van szó, hogy zárójel nélküli helyzetben melyik műveletet kell végrehajtani előbb a kifejezés kiértékelése során.

A következő táblázat csökkenő prioritással haladva mutatja az egyes operátorok asszociativitását. A többször is előforduló operátorok közül mindig az egyoperandusos a magasabb prioritású. Abban a rovatban, ahol több operátor van együtt, a műveletek azonos prioritásúak, és asszociativitásuknak megfelelően hajtja őket végre a fordító. Minden operátor kategóriának megvan a maga asszociativitása (balról jobbra vagy jobbról balra köt), mely meghatározza zárójel nélküli helyzetben a kifejezés csoportosítását azonos prioritású műveletek esetén.

Operátorok	Asszociativitás
() [] -> .	balról jobbra
! ~ + - ++ -- & * **sizeof**	jobbról balra
* / %	balról jobbra
+ -	balról jobbra
<< >>	balról jobbra
< <= > >=	balról jobbra
== !=	balról jobbra
&	balról jobbra
^	balról jobbra
|	balról jobbra
&&	balról jobbra
||	balról jobbra
?:	jobbról balra
= *= /= %= += -= &= ^= |= <<= >>=	jobbról balra
,	balról jobbra

13. ábra: Műveletek prioritása

🕮 A táblázatban felsoroltakon kívül van még a # és a ## operátor, melyek az előfeldolgozónak szólnak.

✍ A prioritási táblázat (13. ábra) természetesen tartalmaz eddig még nem ismertetett műveleteket is.

Vannak többjelentésű operátorok is, melyek értelmezése a helyzettől függ. Például:

```
cimke:              /* utasítás címke */
a?x:y               /* feltételes kifejezés */
a=(b+c)*d;          /* zárójeles kifejezés */
void fv(int n);     /* függvénydeklaráció */
a, b, c;            /* vesszős kifejezés */
fv(a, b, c);        /* függvényhívás */
```

A kifejezés kiértékelési sorrendje nem meghatározott ott, ahol a nyelvi szintaktika erről nem gondoskodik. A fordító a generált kód hatékonyságának javítása érdekében átrendezheti a kifejezést különösen az asszocia-

tív és kommutatív operátorok (*, +, &, ^ és |) esetén, hisz azt feltételezi, hogy a kiértékelés iránya nem hat a kifejezés értékére.

☞ Fedezzük fel, hogy itt nem arról van szó, hogy a fordító a következő fordításkor másként rendezi át a kifejezést, hisz a fordítás determinisztikus dolog, hanem arról, hogy különböző C fordítók más–más eredményre juthatnak!

Probléma lehet az olyan kifejezéssel,

- melyben ugyanazt az objektumot egynél többször módosítjuk, ill.
- melyben ugyanazon objektum értékét változtatjuk és felhasználjuk.

Például:

```
i = v[i++];   /* Döntsük el, mit is akarunk i-vel! */
i = a+++b[a]; /* b indexe attól függ, hogy az összeadás
                 melyik tagját értékeli ki előbb a fordító. */
```

A fentiek akkor is igazak, ha kifejezésünket „ jól összezárójelezzük”:

```
int osszeg=0;
sum = (osszeg=3)+(++osszeg);/* sum == 4 vagy 7 ? */
```

Segédváltozók bevezetésével a dolgok midig egyértelműsíthetők:

```
int seged, osszeg=0;
seged = ++osszeg;           /* seged == 1, osszeg == 1. */
sum = (osszeg=3)+seged;     /* sum == 4. */
```

Ha a szintaktika rögzíti a kiértékelési sorrendet (az &&, a ||, a ?: és a vessző operátor esetében ez így van), akkor ott „bármit” megtehetünk. Például:

```
sum = (i=3, i++, i++);      /* OK, sum == 4 és i == 5. */
```

A kifejezés kiértékelési sorrendjét ugyan () párokkal befolyásolhatjuk:

```
f = a*(b+c);
```

, de asszociatív és kommutatív operátorok operandusait még „agyonzárójelezve” is összecserélheti a fordító, hisz feltételezheti, hogy a kifejezés értékét ez nem befolyásolja:

```
f = (a+b) + (c+d);
```

Határozatlan a függvény aktuális paramétereinek kiértékelési sorrendje is:

```
printf(”%d  %d\n”, ++n, fv(n));
```

A bitenkénti operátorok prioritása alacsonyabb a relációkénál, így a

```
c & 0XF == 8
```

mindig hamis, mert az előbb kiértékelésre kerülő egyenlőségi reláció sohasem lehet igaz. A kifejezés helyesen:

```
(c & 0XF) == 8.
```

Vigyázzunk a hozzárendelés (=) és az egyenlő reláció (==) operátor felcserélésére, mert például az

```
if( i = 2 ) utasítás1; else utasítás2;
```

else ágára sohasem jut el a vezérlés tekintettel arra, hogy a 2 hozzárendelése „ebben az életben” sem válik hamissá.

Ügyeljünk azokkal a kifejezésekkel is, melyekben csak logikai és (&&) vagy csak logikai vagy (||) műveletek vannak, mert ezeket balról jobbra haladva

- && esetén csak az első hamis tagig, ill.
- || operátornál csak az első igaz tagig

fogja kiértékelni a fordító! Az

```
x && y++
```

kifejezésben **y** növelése csak akkor történik meg, ha **x** nem zérus.

Kifejezés kiértékelése közben adódhatnak „áthidalhatatlan” szituációk. Ilyenek

- a zérussal való osztás és
- a lebegőpontos túl vagy alulcsordulás.

Újra felhívjuk azonban a figyelmet arra, hogy a nyelvben nem létezik sem egész túl, sem alulcsordulás!

6 UTASÍTÁSOK

Az utasításokat – ha mást nem mondanak – megadásuk sorrendjében hajtja végre a processzor. Az utasításoknak nincs értéke. Végrehajtásuk hatással van bizonyos adatokra, vagy programelágazást valósítanak meg stb. Definíciójuk a következő:

utasítás:
összetett-utasítás
címkézett-utasítás
kifejezés-utasítás
szelekciós-utasítás
iterációs-utasítás
ugró-utasítás

6.1 Összetett utasítás

összetett-utasítás:
{ <*deklarációlista*><*utasításlista*> }
deklarációlista:
deklaráció
deklarációlista deklaráció
utasításlista:
utasítás
utasításlista utasítás

Az összetett utasítás utasítások (lehet, hogy üres) listája { }-be téve. Az összetett utasítást blokknak is nevezik, mely szintaktikailag egy utasításnak minősül, és szerepet játszik az azonosítók hatáskörében és láthatóságában.

📖 Ha a *deklarációlistá*ban előforduló azonosítót már korábban az összetett utasításon kívül is deklarálták, akkor a blokkra lokális azonosító elfedi a blokkon kívülit a blokk teljes hatáskörében. Tehát az ilyen blokkon kívüli azonosító a blokkban nem látható.

C blokkban előbb a deklarációs, s csak aztán a végrehajtható utasítások következnek.

Az összetett utasítások akármilyen mély szinten egymásba ágyazhatók, s az előbbi szerkezet a beágyazott blokkra is vonatkozik.

💣 Tudjuk, hogy az **auto** tárolási osztályú objektumok inicializálása mindannyiszor megtörténik, valahányszor a vezérlés a „fejen át” kerül be az összetett utasításba. Ez az inicializálás azonban elmarad, ha a vezérlés ugró utasítással érkezik a blokk „közepére”.

💣 Tilos ;–t írni az összetett utasítás záró }–e után!

6.2 Címkézett utasítás

címkézett-utasítás:
  azonosító : *utasítás*
  case *konstans-kifejezés* : *utasítás*
  default : *utasítás*

A **case** és a **default** formák csak **switch** utasításban használatosak. Ezek ismertetésével majd ott foglalkozunk! Az

  azonosító : *utasítás*

alakban az *azonosító* (címke) célja lehet például egy feltétlen elágaztató

  goto *azonosító*;

utasításnak. A címkék hatásköre mindig az őket tartalmazó függvény. Nem deklarálhatók újra, de külön névterületük van, és önmagukban nem módosítják az utasítások végrehajtási sorrendjét.

💣 C-ben csak végrehajtható utasítás címkézhető meg, azaz

```
{
/* . . .
cimke:      /* HIBÁS */
}
```

A megoldás helyessé válik, ha legalább egy üres utasítást teszünk a **cimke** után:

```
{
/* . . .
cimke: ;    /* OK */
}
```

6.3 Kifejezés utasítás

kifejezés-utasítás:
  <*kifejezés*>;

Ha a *kifejezés*t pontosvessző követi, kifejezés utasításnak minősül. A fordító kiértékeli a *kifejezés*t, s eközben minden mellékhatás érvényre jut, mielőtt a következő utasítás végrehajtása elkezdődne. Például az

```
x = 0, i++, printf("Hahó!\n")
```

*kifejezés*ek, s így válnak *kifejezés-utasítás*okká:

```
x = 0; i++; printf("Hahó!\n");
```

A legtöbb kifejezés utasítás hozzárendelés vagy függvényhívás.

Üres utasítást (null statement) úgy kapunk, hogy *kifejezés* nélkül pontosvesszőt teszünk. Hatására természetesen nem történik semmi, de ez is

utasításnak minősül, azaz szintaktikailag állhat ott, ahol egyébként utasítás állhat.

6.4 Szelekciós utasítások

szelekciós-utasítás:
if(*kifejezés*) *utasítás*
if(*kifejezés*) *utasítás* **else** *utasítás*
switch(*kifejezés*) *utasítás*

Látszik, hogy két szelekciós utasítás van: az **if** és a **switch**. A feltételesen elágaztató

if(*kifejezés*) *utasítás1* **else** *utasítás2*

utasításban a *kifejezés*nek aritmetikai, vagy mutató típusúnak kell lennie. Ha értéke zérus, akkor a logikai *kifejezés* hamis, máskülönben viszont igaz. Ha a kifejezés igaz, *utasítás1* következik. Ha hamis, és az **else** ág létezik, akkor *utasítás2* jön.

✍ Mind *utasítás1*, mind *utasítás2* lehet összetett utasítás is!

A nyelvben nincs logikai adattípus, de cserében minden egész és mutató típus annak minősül. Például:

```
if(ptr == 0)     /* Rövidíthető „if(!ptr)”-nek. */
if(ptr != 0)     /* Rövidíthető „if(ptr)”-nek. */
```

Az **else** ág elhagyhatósága néha problémákhoz vezethet. Például a

```
if( x == 1)
  if( y == 1 ) puts( ”x = 1 és y = 1\n”);
else
  puts( ”x != 1\n”);
```

forrásszövegből a programozó „elképzelése” teljesen világos. Sajnos azonban egyet elfelejtett. Az **else** mindig az ugyanazon blokk szinten levő, forrásszövegben megelőző, **else** ág nélküli **if**-hez tartozik. A megoldás helyesen:

```
if( x == 1)
  { if( y == 1 ) puts( ”x = 1 és y = 1\n”); }
else
  puts( ”x != 1\n”);
```

Az **if** utasítások tetszőleges mélységben egymásba ágyazhatók, azaz mind az **if**, mind az **else** utasítása lehet újabb **if** utasítás is. Például:

```
if( x == 1)
  { if( y == 1 ) puts( ”x = 1 és y = 1\n”);
    else puts( ”x = 1 és y != 1\n”);}
else
  { if( y == 1 ) puts( ”x != 1 és y = 1\n”);
```

```
    else puts( "x != 1 és y != 1\n");}
```

Többirányú elágazást (szelekciót) valósít meg a következő konstrukció:

```
if( kifejezés1 ) utasítás1
else if( kifejezés2 ) utasítás2
else if ( kifejezés3 ) utasítás3
/* . . . */
else utasításN
```

Ha valamelyik **if** *kifejezés*e igaz, akkor az azonos sorszámú *utasítás*t hajtja végre a processzor, majd a konstrukciót követő utasítás jön. Ha egyik **if** *kifejezés*e sem igaz, akkor viszont *utasításN* következik.

Ha egy **n** elemű, növekvőleg rendezett **t** tömbben keresünk egy **x** értéket, akkor ezt bináris keresési algoritmust alkalmazva így interpretálhatjuk:

```
int binker(int x, int t[], int n){
  int also=0, felso=n-1, kozep;
  while(also<=felso){
    kozep=(also+felso)/2;
    if(x<t[kozep]) felso=kozep-1;
    else if(x>t[kozep]) also=kozep+1;
    else return kozep; }      /* Megvan. */
    return (-1); }            /* Nincs meg. */
```

A **binker** háromirányú elágazást realizál. **x t** tömbbeli előfordulásának indexét szolgáltatja, ill. –1–et, ha nincs is **x** értékű elem a tömbben.

A bináris keresés növekvőleg rendezett tömbben úgy történik, hogy először megállapítjuk, hogy a keresett érték a tömb alsó, vagy felső felébe esik. A módszert aztán újraalkalmazzuk az aktuális félre, s így tovább. A dolognak akkor van vége, ha a kereső tartomány semmivé szűkül (ilyenkor nincs meg a keresett érték), vagy a felező tömbelem egyezik a keresett értékkel.

Rendezett sorozatban egy érték keresésének ez a módja, és nem a minden elemhez történő hasonlítgatás!

Megoldandó feladatok:

Készítsen olyan **binker** függvényt, mely:

- csökkenőleg rendezett tömbben keres!
- pótlólagos paraméterben kapja meg, hogy csökkenő vagy növekvő a rendezettség!

- plusz paraméter nélkül is eldönti, hogy csökkenőleg, vagy növekvőleg keressen a tömbben!

A többirányú programelágaztatás másik eszköze a

switch(*kifejezés*) *utasítás*

, melyben a *kifejezés*nek egész típusúnak kell lennie. Az *utasítás* egy olyan speciális összetett utasítás, mely több **case** címkét

case *konstans-kifejezés* : *utasítás*

és egy elhagyható **default** címkét

default : *utasítás*

tartalmazhat. A vezérlést azon **case** címkét követő *utasítás* kapja meg, mely *konstans-kifejezés*ének értéke egyezik a **switch**–beli *kifejezés* értékével. A végrehajtás aztán itt addig folytatódik, míg **break** utasítás nem következik, vagy vége nincs a **switch** blokkjának.

A **case** címkebeli *konstans-kifejezés*nek is egész típusúnak kell lennie. Ez a **TÍPUSOK ÉS KONSTANSOK** szakasz **Deklaráció** fejezetében írottakon túl további korlátozásokat ró a konstans kifejezésre. Operandusai csak egész, felsorolás, karakteres és lebegőpontos állandók lehetnek, de a lebegőpontos konstanst explicit típuskonverzióval egésszé kell alakítani. Operandus lehet még a **sizeof** operátor is, aminek operandusára természetesen nincsenek ilyen korlátozások.

A végrehajtás során a *kifejezés* és a **case** *konstans-kifejezés*ek értékén is végbemegy az egész–előléptetés. A *kifejezés* az összes esetleges mellékhatásával egyetemben valósul meg, mielőtt az értékhasonlítás megkezdődne.

Ha nincs a **switch** *kifejezés* értékével egyező **case** címke, akkor a vezérlést a **default** címke utasítása kapja meg. Ha nincs **default** címke sem, akkor vége van a **switch**–nek.

Az utasítás használatához még két megjegyzést kell fűzni:

- Több **case** címke is címkézhet egy utasítást.
- Egyazon **switch** utasításban viszont nem fordulhat elő két azonos értékű **case** *konstans-kifejezés*.

A **switch** utasítások egymásba is ágyazhatók, s ilyenkor a **case** és a **default** címkék mindig az őket közvetlenül tartalmazó **switch**–hez tartoznak.

A **switch** utasítást szemléltetendő írjuk át a szabvány bemenet karaktereit kategóriánként leszámláló **PELDA7.C**–t!

```
/* PELDA17.C: A bemenet karaktereinek
              leszámlálása kategóriánként */
#include <stdio.h>
void main(void){
  short k, num=0, feher=0, egyeb=0;
  printf("Bemeneti karakterek leszámlálása\n"
         "kategóriánként EOF-ig, vagy Ctrl+Z-ig.\n");
  while((k=getchar())!=EOF) switch(k){
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      ++num;
      break;
    case ' ':
    case '\n':
    case '\t':
      ++feher;
      break;
    default:
      ++egyeb; }
  printf("Karakter számok:\n----------------\n"
         "numerikus: %5hd\nfehér:     %5hd\n"
         "egyéb:     %5hd\n----------------\n"
         "össz: %10ld\n", num, feher, egyeb,
         (long)num+feher+egyeb); }
```

☞ Belátható, hogy a **switch** utasítás használhatóságát szegényíti, hogy a *kifejezés* csak egész típusú lehet és, hogy a **case** címkék csak egészértékű *konstans-kifejezés*ek lehetnek. Az **if**–nél említett többirányú elágaztatási konstrukció így jóval általánosabban használható.

6.5 Iterációs utasítások

iterációs-utasítás:
while(*kifejezés*) *utasítás*
do *utasítás* **while**(*kifejezés*);
for(<*kifejezés*>; <*kifejezés*>; <*kifejezés*>) *utasítás*

Szemmel láthatóan háromféle iterációs utasítás van, amiből kettő elöltesztelő ciklusutasítás. A

while(*kifejezés*) *utasítás*

*kifejezés*éről ugyanaz mondható el, mint az **if** utasítás *kifejezés*éről. Lépésenként a következő történik:

1. Kiértékeli a fordító a *kifejezés*t, melynek során minden esetleges mellékhatás is megvalósul. Ha hamis (zérus) az értéke, akkor vége a ciklusnak, s a **while**-t követő utasítás jön a programban.
2. Ha a *kifejezés* igaz (nem zérus), akkor az *utasítás* végrehajtása, és újból az 1. pont következik.

✍ Világos, hogy a *kifejezés* értékének valahogyan változnia kell az *utasítás*ban, különben a ciklusnak soha sincs vége. Az *utasítás* állhat több utasításból is, azaz összetett utasítás is lehet. Ha az utasítások közt ugró utasítás is van, akkor ezen mód is kiléphetünk a ciklusból.

A

for(<*init-kifejezés*>; <*kifejezés*>; <*léptető-kifejezés*>) *utasítás*

elöltesztelő, iteratív ciklusutasítás, melynek megvalósulása a következő lépésekből áll:

1. A fordító végrehajtja az *init-kifejezés*t, ha van. Többnyire egy vagy több változó értékadásáról lehet itt szó.
2. Kiértékeli a *kifejezés*t. Ha hamis (zérus), akkor vége a ciklusnak, s a **for**-t követő utasítás jön a programban. Látható, hogy a szintaktika szerint ez a *kifejezés* is elhagyható. Ilyenkor 1 (igaz) kerül a helyére, azaz a végtelen ciklus könnyedén felírható:

   `for( ; ; );` ➔ `for( ; 1; );`
3. Ha a *kifejezés* igaz (nem zérus), akkor az *utasítás* jön,
4. aztán a *léptető-kifejezés*, majd újból a 2. pont következik.

Ha a **for** utasítást **while**-lal szeretnénk felírni, akkor azt így tehetjük meg, ha a ciklusmagban nincs **continue**:

<*init-kifejezés*>;

while(<*kifejezés*>) { *utasítás*; <*léptető-kifejezés*>; }

✍ A szintaktikai szabályt összefoglalva: a **for**-ból akár mindegyik kifejezés is elhagyható, de az első kettőt záró pontosvesszők nem!

✍ A vessző operátor alkalmazásával mind az *init*-, mind a *léptető-kifejezés* több kifejezés is lehet. Ezt szemlélteti az előző szakaszban a **PELDA16.C**–ben megírt **strrv** függvény.

Írjunk **void rendez**(**int** *a*[], **int** *n*) függvényt, mely növekvőleg rendezi saját helyén az *n* elemű *a* tömböt!

- Indulva az első elemtől megkeressük a tömb minimális elemét, és kicseréljük az első elemmel.
- A 2. elemtől kezdve a hátra levő tömbrészben keressük meg a legkisebb elemet, s ezt kicseréljük a 2. elemmel, s így tovább.
- Látszik, hogy minimumkeresést utoljára a tömb utolsó előtti és utolsó elemén kell elvégezni, s az itteni esetleges csere után rendezett is az egész tömb.

```
void rendez(int a[],int n){
  int i, j, m, cs;
  for(i=0; i<n-1; ++i){
     for(j=i+1, m=i; j<n; ++j) if(a[j]<a[m]) m=j;
     if(i!=m){ cs=a[i]; a[i]=a[m]; a[m]=cs;} } }
```

Ez ugyebár példa az egymásba ágyazott ciklusokra!

Készítsük el az **int egesze**(**char** *s*[]) rutint, mely ellenőrzi a decimális egész konstans írásszabályát a paraméter karaktertömbön. 1–et szolgáltat, ha a dolog rendben van, s zérust, ha nem!

```
/* A decimális számjegyek száma maximálisan: */
#define HSZ sizeof(int)/sizeof(short)*5
int egesze(char s[]){
  int i = 0, kezd;
  /* A karakterlánc eleji fehér karakterek átlépése: */
  while(s[i]==' ' || s[i]=='\n' || s[i]=='\t') ++i;
  /* Az előjel átlépése: */
  if(s[i]=='+' || s[i]=='-') ++i;
  kezd=i;     /* A számjegyek itt kezdődnek. */
  /* Előre a következő nem numerikus karakterre: */
  while(s[i]>='0' && s[i]<='9' && i-kezd<HSZ) ++i;
  /* Döntés: */
  if(kezd==i || s[i]!='\n' && s[i]!='\t' &&
     s[i]!=' ' && s[i]!=0) return 0;
  else return 1; }
```

Látszik, hogy **HSZ** 16 bites **int** esetén 5 (32767), és 32 bitesnél 10 (2147483647). Arra való, hogy ennél több decimális számjegyet ne fogadjon el a rutin.

Vegyük észre, hogy a függvény visszautasítja, ha egyetlen számjegy karaktert sem tartalmaz a numerikus karakterlánc! Elveti azt is, ha a numerikus lánc rész nem fehér karakterrel, vagy lánczáró zérussal végződik. Az ellenőrzés persze nem tökéletes, hisz a numerikus karakterlánc ábrázolási határok közé férését nem biztosítja. Például 16 bites **int**–nél minden ötje-

gyű számot érvényesnek tekint, holott a –99999 és –32769, valamint a 32768 és 99999 közötti számok ábrázolhatatlanok. Ez az ellenőrzés azonban csak akkor lenne könnyen megvalósítható, ha tesztelés közben konvertálnánk is a számot. Ekkor azonban már „többe kerülne a leves, mint a hús”.

Készítsük programot, mely egész számokat kér be. Meghatározza az átlagukat, a minimumukat, maximumukat, és növekvőleg rendezi is őket! Az egészek darabszáma csak futás közben dől el.

```
/* PELDA18.C: Egész számok átlaga, minimuma, maximuma és
              rendezése */
#include <stdio.h>
#include <stdlib.h>       /* Az atoi miatt! */
#include <limits.h>       /* INT_MIN és INT_MAX végett! */
#define MAX 100           /* A tömb max. elemszáma. */
#define INP 20            /* Az input puffer mérete. */
int getline(char s[],int lim);
/* A decimális számjegyek száma maximálisan: */
#define HSZ sizeof(int)/sizeof(short)*5
int egesze(char s[]);
void rendez(int a[],int n);
void main(void){
  int n=0;         /* A rendezendő elemek száma. */
  char sor[INP+1]; /* Az input puffer. */
  int a[MAX];      /* A egészeket tároló tömb. */
  int min, max;    /* Minimum, maximum. */
  double osszeg=0.;/* Az összeg */
  int i;
  while(n<1||n>MAX){
    printf("\nHány egész számot rendezünk(1-%d)? ",MAX);
    getline(sor,INP);
    if(egesze(sor)) n=atoi(sor);}
  printf("\n\nKérem a rendezendő számokat %d és +%d”
         ” között!\n", INT_MIN, INT_MAX);
  for(i=0;i<n;++i){
    printf("%3d: ",i+1);
    if(getline(sor,INP)>0 && egesze(sor))
      a[i]=atoi(sor);
    else --i; }
  min=max=a[0];
  for(i=0;i<n;osszeg+=a[i],++i)
    if(a[i]<min) min=a[i];
    else if(a[i]>max) max=a[i];
    printf("\nA számsorozat\tminimuma:%14d.\n"
           "\t\tmaximuma:%14d.\n"
           "\t\tátlaga:  %17.2f\n",
           min, max, osszeg/n);
  printf("\nA rendezett számok:\n");
  rendez(a, n);
```

```
for(i=0; i<n; i++){
   printf("%13d",a[i]);
   if(!((i+1)%6)) putchar('\n'); }
putchar('\n'); }
```

Nem rendezett sorozatban a minimum, vagy maximum megkeresésének az a módja, hogy vesszük a sorozat egy létező elemét (többnyire az elsőt), és aztán hasonlítgatjuk a többiekhez, hogy vajon van–e nálánál kisebb, ill. nagyobb. Ha van, akkor attól kezdve azzal folytatjuk a hasonlítást.

El kell ismerni természetesen, hogy a minimum és maximum megkeresése teljesen felesleges volt, mert a rendezés után ezeket az értékeket **a**[0] és **a**[**n**–1] amúgy is tartalmazza.

Vegyük még észre, hogy az egészek összegét a **for** ciklus *léptető-kifejezés*ében számítjuk ki!

Megoldandó feladatok:

Alakítsa át a **PELDA18.C**–ben megvalósított programot a következőképp:

- Ne kérje be előre a rendezendő számok darabszámát, hanem az érték bekérésekor jelentse üres sor a bemenet végét!
- Dolgozzék nem egész, hanem valós számokkal a program!

A

do *utasítás* **while**(*kifejezés*);

hátultesztelő ciklusutasítás. A *kifejezés*re ugyanazon megkötések érvényesek, mint a **while**–nál. A dolog lényege az, hogy fordító a *kifejezés* értékétől függetlenül egyszer biztosan

1. végrehajtja az *utasítás*t.
2. Kiértékeli a *kifejezés*t. Ha hamis (zérus), akkor vége a ciklusnak, s a **while**-t követő utasítás jön a programban. Ha viszont igaz, akkor az 1. pont következik.

Összesítve: Az *utasítás*t egyszer mindenképp végrehajtja a processzor, s ezt követően mindaddig ismétli, míg a *kifejezés* hamis nem lesz.

A szemléltető példában az **itoa** függvény az **int n** paraméterét karakterlánccá konvertálja, és elhelyezi a paraméter **s** karaktertömbben:

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

```
void itoa(int n, char s[]){
  int i=0, elojel, j=0;
  if(n==INT_MIN) { ++n; ++j; }
  if((elojel=n)<0) n=-n;
  do s[i++]=n%10+'0'; while(n/=10);
  if(elojel<0) s[i++]='-';
  s[i]=0;
  s[0]+=j;
  strrev(s); }
```

A belsőábrázolási formából karakterlánccá alakító **itoa** rutinban az **i** ciklusváltozó. A **j** logikai változó, s induló értéke 0. Egyetlen egy esetben egyértékű, ha **n** éppen **INT_MIN**. A probléma az ugye, hogy mindenképp pozitív értéket kívánunk konvertálni az esetleges negatív előjelet megjegyezve, de az **INT_MIN** –1–szerese nem ábrázolható **int**–ként, hisz éppen eggyel nagyobb **INT_MAX**–nál. Ha ilyenkor megnöveljük eggyel **n** értékét, akkor –1–szerese épp a felső ábrázolási korlát lesz, amivel már nincs probléma. A **j** logikai változó tehát akkor 1, ha volt növelés. Az **elojel** változóra **n** eredeti előjelének megtartása végett van szükség, és azért, hogy negatív **n** esetén a keletkező karakterlánc végére ki lehessen tenni a mínuszjelet.

Az algoritmus éppen megfordítva állítja elő a karakterláncot. Nézzük csak asztali teszttel a 16 bites **int** esetét! Tegyük fel, hogy **n** értéke az ominózus –32768!

Amig a **do**–**while** utasításig nem érünk, **j** 1 lesz, **n** 32767 és **i** zérus marad. Lejátszva a ciklust a következő történik:

j:	1						
n:	32767	3276	327	32	3	0	0
i:	0	1	2	3	4	5	6
s:	'7'	'6'	'7'	'2'	'3'	'–'	'\0'

A táblázatban az az állapot látszik, amikor a **do**–**while**–t követő két utasítás is lezajlott. Az **s**[0]–ból, vagyis '7'–ből, '8' lesz, és aztán a szabvány könyvtári **strrev** megfordítja a saját helyén az eredmény karakterláncot.

6.6 Ugró utasítások

ugró-utasítás:
break;
continue;
goto *azonosító*;
return <*kifejezés*>;

Csak iterációs (**while**, **do-while** és **for**) vagy **switch** utasításon belül használható a

break;

, mely befejezi ezeket az utasításokat, azaz hatására a vezérlés feltétel nélkül kilép belőlük. Több egymásba ágyazott iterációs utasítás esetén a **break** csak abból a ciklusból léptet ki, amelyben elhelyezték, azaz a **break** csak egyszintű kiléptetésre képes.

Készítendő egy **int trim**(**char** *s*[]) függvény, mely a paraméter karaktertömb elejéről és végéről eltávolítja a fehér karaktereket a saját helyén, s visszatér az így letisztított karakterlánc hosszával!

```
#include <string.h>
int trim(char s[]){
  int i=0, n;
  /* A fehér karakterek eltávolítása a lánc végéről: */
  for(n=strlen(s)-1; n>=0; --n)
    if(s[n]!=' '&&s[n]!='\n'&&s[n]!='\t') break;
  s[++n]='\0';
  /* A fehér karakterek eltávolítása a lánc elejéről: */
  while(s[i]==' '||s[i]=='\n'||s[i]=='\t') ++i;
  if(i) { n=0; while(s[n++]=s[i++]); --n;}
  return(n); }
```

Csak iterációs utasításokban (**while**, **do-while** és **for**) alkalmazható a

continue;

, mely a vezérlést a *kifejezés* kiértékelésére viszi **while** és **do-while** estén, ill. hatására a *léptető-kifejezés* kiértékelése következik **for** utasításnál. Egymásba ágyazott iterációs utasítások esetén ez is mindig csak az őt magába foglaló iterációs utasításra vonatkozik.

💣 Ugyebár **switch**–ben csak iterációs utasításon belül használható a **continue**!

Feltétlen vezérlésátadást hajt végre az *azonosító*val címkézett utasításra a

goto *azonosító*;

Az utasítást bármilyen mély blokk szintről is végrehajtja a processzor, de a cél címkézett utasításnak ugyanabban a függvényben kell lennie, ahol a **goto** is van.

A **void** visszatérésűektől eltekintve a függvény testében lennie kell legalább egy

return <*kifejezés*>;

utasításnak. Ha a rutin visszatérési típusa *típus*, akkor a *kifejezés* típusának is ennek kell lennie, vagy hozzárendelési konverzióval ilyen *típus*úvá alakítja a *kifejezés* értékét a fordító. A **return** hatására visszakapja a vezérlést a hívó függvény, s átveszi a visszatérési értéket is.

A *típus* visszatérésű függvényt hívó kifejezés

```
fv(aktuális-paraméterlista)
```

típus típusú jobbérték, s nem balérték:

```
típus t;
t=fv(aktuális-paraméterlista);      /* OK */
t=++fv(aktuális-paraméterlista);    /* OK */
```

A függvényhívás hatására beindult végrehajtás a **return** utasítás bekövetkeztekor befejeződik. Ha nincs **return**, akkor a függvény testét záró }-ig megy a végrehajtás.

Ha a függvény által visszaadott érték típusa **void**, és a függvényt nem a testét záró }-nél szeretnénk befejezni, akkor a függvény belsejébe a kívánt helyre **return** utasítást kell írni.

7 ELŐFELDOLGOZÓ (PREPROCESSOR)

A fordító első menete során mindig meghívja az előfeldolgozót a forrásfájlra.

Ha szeretnénk tudni, hogy fest az előfeldolgozáson átesett forrásfájl (a fordítási egység), akkor a programfejlesztő rendszerben utána kell nézni, hogy tehetjük láthatóvá az előfeldolgozás eredményét. Az előfeldolgozott forrásfájlban aztán megtekinthetjük:

- a makrók kifejtését,
- a behozott (include) fájlokat,
- a feltételes fordítást,
- a szomszédos karakterlánc konstansok egyesítését,
- a direktívák elhagyását és
- a megjegyzések kimaradását (helyettesítését egyetlen szóköz karakterrel).

A nem karakter, vagy karakterlánc konstansban előforduló, egymást követő, több fehér karaktert mindig eggyel helyettesíti az előfeldolgozó.

Az előfeldolgozó direktívák írásszabálya, mely független a C nyelv többi részétől, a következő:

- A sor első, nem fehér karakterének #-nek kell lennie.
- A #-et követheti aztán fehér karakter a soremelést kivéve. A sorokra tördelés nagyon lényeges elem, mert az előfeldolgozó sorokra bontva elemzi a forrásszöveget.
- A karakter konstansban, a karakterlánc konstansban és a megjegyzésben levő # karakter nem minősül előfeldolgozó direktíva kezdetének.

A direktívákat - miután nem C utasítások - tilos pontosvesszővel lezárni!

- Ha a direktívában a soremelést \ karakter előzi meg, akkor a következő sor folytatássornak minősül, azaz az előfeldolgozó elhagyja a \–t és a soremelést, s egyesíti a két sort.
- Az előfeldolgozó direktívákba megjegyzés is írható.

- Az előfeldolgozó direktívák bárhol elhelyezkedhetnek a forrásfájlban, de csak a forrásfájl ezt követő részére hatnak, egészen a fájl végéig.

A teljes szintaktika az előfeldolgozó direktívákra a feltételes fordítástól eltekintve úgy, hogy a már megismert fogalmakat újra nem definiáljuk, a következő:

csoport:
  csoport-rész
  csoport csoport-rész

csoport-rész:
  előfeldolgozó-szimbólumok újsor
  feltételes-fordítás
  vezérlő-sor

előfeldolgozó-szimbólumok:
  előfeldolgozó-szimbólum
  előfeldolgozó-szimbólumok előfeldolgozó-szimbólum

előfeldolgozó-szimbólum:
  <*fájlazonosító*> (csak #**include** direktívában)
  "*fájlazonosító*" (csak #**include** direktívában)
  azonosító (nincs kulcsszó megkülönböztetés)
  konstans
  karakterlánc-konstans
  operátor
  elválasztó-jel
  bármilyen nem fehér karakter, mely az előzőek egyike sem

vezérlő-sor:
  #include *előfeldolgozó-szimbólumok újsor*
  #define *azonosító <előfeldolgozó-szimbólumok> újsor*
  #define *azonosító*(*azonosítólista*) *előfeldolgozó-szimbólumok újsor*
  #undef *azonosító újsor*
  #line *előfeldolgozó-szimbólumok újsor*
  #error <*előfeldolgozó-szimbólumok***>** *újsor*
  #pragma <*előfeldolgozó-szimbólumok***>** *újsor*
  # *újsor*

újsor:
  soremelés

Az *előfeldolgozó-szimbólum* definíciójában a *fájlazonosító* körüli <> a szintaktika része, és nem az elhagyhatóságot jelöli, mint máskor. Látszik, hogy a feldolgozásban bármely karakter, ami nem foglalt az előfeldolgozónak, szintén szintaktikai egységet képez.

7.1 Üres (null) direktíva

A csak egy # karaktert tartalmazó sor. Ezt a direktívát elhagyja az előfeldolgozó, azaz hatására nem történik semmi.

7.2 #include direktíva

Az #**include** direktíva lehetséges alakjait:

#include <*fájlazonosító*> *újsor*

#include ”*fájlazonosító*” *újsor*

már a **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban részleteztük. Tudjuk, hogy a *vezérlő-sor*t az előfeldolgozó a megadott fájl teljes tartalmával helyettesíti. Magát a fájlt ”*fájlazonosító*” esetén előbb az aktuális könyvtárban (abban, ahonnét azt a fájlt töltötte, amelyikben ez az **#include** direktíva volt), majd és <*fájlazonosító*>–s esetben a programfejlesztő rendszerben előírt utakon keresi.

✍ A <> és az idézőjelek között nincs makróhelyettesítés.

💣 Ha a *fájlazonosító*–ban >, ”, ’, \, vagy /* karakterek vannak, az előfeldolgozás eredménye meghatározatlan.

💣 Ha macskakörmös esetben a *fájlazonosító* elérési utat is tartalmaz, akkor a fájlt a preprocesszor csak abban a könyvtárban keresi, és sehol másutt.

A direktíva

#include *előfeldolgozó-szimbólum újsor*

formáját előbb feldolgozza az előfeldolgozó, de a helyettesítésnek itt is <>, vagy ”” alakot kell eredményeznie, s a hatás ennek megfelelő. Például:

```
#define ENYIM ”\Cfajlok\Munka16\Pipo.h”
/* . . . */
#include ENYIM
```

🕮 Az #**include** direktívák egymásba ágyazhatók, azaz a behozott fájl újabb #**include**–okat tartalmazhat, s azok megint újabbakat, és így tovább.

💣 Az egymásba ágyazgatásokkal azonban vigyázni kell, mert egyes programfejlesztő rendszerek ezek szintjét – például 10–ben – korlátozhatják!

7.3 Egyszerű #define makró

A #**define** direktíva makrót definiál (makródefiníció). A makró szimbólumhelyettesítő mechanizmus függvényszerű formális paraméterlistával vagy anélkül.

Előbb a paraméter nélküli esettel foglalkozunk! Ilyenkor a direktíva alakja:

#define *azonosító <előfeldolgozó-szimbólumok> újsor*

Hatására az előfeldolgozó a forráskódban ez után következő minden makró*azonosító* előfordulást helyettesít a lehet, hogy üres *előfeldolgozó-szimbólumok*kal. Ha üres *előfeldolgozó-szimbólumok*kal történik a helyettesítés, a makró*azonosító* akkor is definiált, azaz **#if defined** vagy **#ifdef** direktívával "rákérdezhetünk" az azonosítóra, de a makró*azonosító* minden forrásszövegbeli előfordulásának eltávolítását jelenti tulajdonképp. Nem történik meg a helyettesítés, ha a makró*azonosító* karakter vagy karakterlánc konstansban, vagy megjegyzésben, vagy más makró*azonosító* részeként található meg. Ezt a folyamatot makrókifejtésnek (expansion) nevezik. A *előfeldolgozó-szimbólumok*at szokás makrótest névvel is illetni. Például:

```
#define HI    ”Jó napot!”
#define begin {
#define end   }
#define then
#define NIL   ””
#define EGY   1
int main(void)
  begin           /* Helyettesítés {-re. */
  int i=8, k=i+EGY;/* Csere k=i+1-re.*/
  puts(HI);       /* puts(”Jó napot!”); lesz belőle */
  puts(NIL);      /* puts(””); lesz a sorból. */
  puts(”then”);   /* Nincs helyettesítés, mert a makróa-
                zonosító karakterlánc konstansban van. */
  if(++i<k)
  /* A sor eleji then semmire helyettesítése, de a */
  then puts(”Ez a then-ág!\n”);          /* másik marad.*/
  else puts(”Ez az else-ág!\n”);
  return 0;
  end                                    /* Csere }-re. */
```

A makrókifejtés utáni makróazonosítókat is helyettesíti a preprocesszor, azaz a makrók is egymásba ágyazhatók.

💣 A makróazonosító újradefiniálása csak akkor nem hiba, ha az *előfeldolgozó-szimbólumok* tökéletesen, pozíció–helyesen azonosak. Ettől eltérő újradefiniálás csak a rávonatkozó #**undef** direktíva után lehetséges.

✍ A nyelv kulcsszavait is alkalmazhatjuk makródefiníciókban, legfeljebb kissé értelmetlennek tekinthető az eljárásunk:

```
#define LACI for
#define PAPI while
#define int long
```

, de a következő fejezetben ismertetett, szabványos, előredefiniált makrók nem jelenhetnek meg sem #**define**, sem #**undef** direktívákban.

✍ A programfejlesztő rendszer segítségében célszerű utána nézni, hogy még milyen más, védett makróazonosítók használata tiltott!

7.4 Előredefiniált makrók

Néhány makró előredefiniált az előfeldolgozó rendszerben, s kifejtésükkel speciális információ képezhető. Ezek a makrók mind **defined** típusúak. Nem tehetők definiálatlanná, és nem definiálhatók újra. A szabványos, előredefiniált makrók és jelentésük a következő:

__DATE__: HHH NN ÉÉÉÉ alakú karakterlánc, s az aktuális forrásfájl előfeldolgozása kezdetének dátuma. A HHH hárombetűs hónapnév rövidítés (Jan, Feb stb.). Az NN 1 és 31 közötti napszám, s így tovább.

__FILE__: Karakterláncként a feldolgozás alatt álló forrásfájl azonosítóját tartalmazza. A makró változik #**include** és #**line** direktíva hatására, valamint ha a forrásfájl fordítása befejeződik.

__LINE__: Decimális értékként a feldolgozás alatti forrásfájl aktuális sorának sorszáma. A sorszámozás 1-től indul. Módosíthatja például a #**line** direktíva is.

__STDC__: Definiált és 1, ha ANSI kompatibilis fordítás van, máskülönben definiálatlan.

__TIME__: OO:PP:MM alakú karakterlánc, s a forrásfájl feldolgozása megkezdésének idejét tartalmazza.

7.5 #undef direktíva

#undef *azonosító újsor*

A direktíva definiálatlanná teszi a makró*azonosító*t, azaz a továbbiakban nem vesz részt a makrókifejtésben.

Azt, hogy egy makróazonosító definiált-e vagy sem a forráskódban, megtudhatjuk a

```
#ifdef azonosító
#ifndef azonosító
```

direktívák segítségével, azaz a makródefiníciónál ajánlható a következő stratégia:

```
#ifndef MAKROKA
  #define MAKROKA 162
#endif
```

🕮 Az ismeretlen makróazonosítóra kiadott #**undef** direktívát nem tekinti hibának az előfeldolgozó.

A definiálatlanná tett makróazonosító később újradefiniálható akár más *előfeldolgozó-szimbólumok*kal. A #**define**-nal definiált és közben #**undef**-fel definiálatlanná nem tett makróazonosító definiált marad a forrásfájl végéig. Például:

```
#define BLOKK_MERET 512
/* . . . */
puff = BLOKK_MERET*blkszam; /* Kifejtés: 512*blkszam. */
/* . . . */
#undef BLOKK_MERET
/* Innét a BLOKK_MERET ismeretlen makróazonosító. */
/* . . . */
#define BLOKK_MERET 128
/* . . . */
puff = BLOKK_MERET*blkszam; /* Kifejtés: 128*blkszam. */
```

7.6 Paraméteres #define direktíva

A direktíva alakja ilyenkor:

#define *azonosító*(*azonosítólista*) *előfeldolgozó-szimbólumok újsor*

Az *azonosítólista* egymástól vesszővel elválasztott formális paraméterazonosítókból áll. A makrót hívó aktuális paraméterlistában ugyanannyi paraméternek kell lennie, mint amennyi a formális paraméterlistában volt, mert különben hibaüzenetet kapunk.

✍ Ugyan a makróra is a függvénnyel kapcsolatos fogalmakat használjuk képszerűségük végett, de ki kell hangsúlyozni, hogy a makró nem függvény!

💣 A makróazonosító és a paraméterlistát nyitó kerek zárójel közé semmilyen karakter sem írható, hisz rögtön egyszerű #**define**-ná tenné a paraméteres direktívát!

Az előfeldolgozó előbb a makróazonosítót helyettesíti, s csak aztán a zárójelbe tett paramétereket:

```
Definíció:    #define KOB(x) ((x)*(x)*(x))
Forrássor:    n = KOB(y);
Kifejtve:     n = ((y)*(y)*(y));
```

A látszólag redundáns zárójeleknek nagyon fontos szerepük van:

```
Definíció:    #define KOB(x) (x*x*x)
Forrássor:    n = KOB(y + 1);
Kifejtve:     n = (y + 1*y + 1*y + 1);
```

A külső zárójel pár a kifejezésekben való felhasználhatóságot biztosítja:

```
Definíció:    #define SUM(a,b) (a)+(b)
Forrássor:    n = 14.5 * SUM(x*y, z-8);
```

```
Kifejtve:     n = 14.5 * (x*y)+(z-8);
```

A zárójelbe, vagy aposztrófok, idézőjelek közé tett vesszők nem minősülnek a listában azonosító elválasztónak:

```
Definíció:    #define SUM(a,b) ((a)+(b))
Forrássor:    return SUM(f(i,j), g(k,l));
Kifejtve:     return ((f(i,j))+(g(k,l)));
Definíció:    #define HIBA(x,lanc) hibaki(”Hiba: ”,x,lanc)
Forrássor:    HIBA(2,”Üssön Enter-t, s Esc-t!”);
Kifejtve:     hibaki(”Hiba: ”,2,”Üssön Enter-t, s Esc-t!”);
```

Új azonosító generálási céllal a *szimbólum1##szimbólum2* alakból *szimbólum1szimbólum2*-t állít elő az előfeldolgozó:

```
Definíció:    #define VAR(i,j) (i##j)
Forrássor:    VAR(a,8)
Kifejtve:     (a8)
```

A formális paraméter elé írt # (úgy nevezett karakterláncosító operátor) az aktuális paramétert karakterlánc konstanssá konvertálja:

```
Definíció:    #define NYOM(jel) printf(#jel ”=%d\n”, jel)
Forrássor:    int kilo = 100; NYOM(kilo);
Kifejtve:     int kilo = 100; printf(”kilo” ”=%d\n”, kilo);
```

Folytatássor most is sorvégi \ jellel képezhető:

```
Definíció:    #define FIGYU ”Ez igazából egyetlen \
              sornak minősül!”
Forrássor:    puts(FIGYU);
Kifejtve:     puts(”Ez igazából egyetlen sornak minősül!”);
```

A makró nem függvény, tehát semmilyen ellenőrzés sincs a paraméterek adattípusára! Ha az aktuális paraméter kifejezés, akkor kiértékelése többször is megtörténik:

```
int kob(int x){ return x*x*x;}
#define KOB(x) ((x)*(x)*(x))
/* . . . */
int b = 0, a = 3;
b = kob(a++); /* b == 27 és a == 4. */
a = 3;
b = KOB(a++); /* Kifejtve: ((a++)*(a++)*(a++)),
                 azaz b == 60 és a == 6. */
```

7.7 Karaktervizsgáló függvények (makrók)

Megismerkedtünk az előző fejezetekben a makrók előnyös, és persze hátrányos tulajdonságaival. Mindezek dacára a makrók használata a C–ben elég széleskörű.

✍ Nézzük csak meg a szabványos **STDIO.H** fejfájlt, s látni fogjuk, hogy a szabvány bemenetet és kimenetet kezelő **getchar** és **putchar** rutinok makrók.

A szabványos **STDIO.H** fejfájlban deklarált függvények karakterosztályozást végeznek. A rutinok *c* paramétere **int** ugyan, de az értékének **unsigned char** típusban ábrázolhatónak, vagy **EOF**–nak kell lennie. A visszatérési értékük ugyancsak **int** logikai jelleggel, azaz nem zérus (igaz), ha a feltett kérdésre adott válasz igen, ill. zérus (hamis), ha nem.

A teljesség igénye nélkül felsorolunk néhány karakterosztályozó függvényt!

Függvény	Kérdés
islower(*c*)	*c* kisbetű–e?
isupper(*c*)	*c* nagybetű–e?
isalpha(*c*)	**islower**(*c*) | **isupper**(*c*)
isdigit(*c*)	*c* decimális számjegy–e?
isalnum(*c*)	**isalpha**(*c*) | **isdigit**(*c*)
isxdigit(*c*)	*c* hexadecimális számjegy–e?
isspace(*c*)	*c* fehér karakter–e? (szóköz, soremelés, lapemelés, kocsi vissza, függőleges vagy vízszintes tabulátor)
isprint(*c*)	*c* nyomtatható karakter–e?

Meg kell még említeni két konverziós rutin is:

int **tolower**(*c*);

int **toupper**(*c*);

, melyek *c* értékét kisbetűvé (**tolower**), ill. nagybetűvé (**toupper**) alakítva szolgáltatják, ha *c* betű karakter, de változatlanul adják vissza, ha nem az.

Írjuk át **PELDA17.C**–t úgy, hogy karaktervizsgáló függvényeket használjon!

```
/* PELDA19.C: A bemenet karaktereinek leszámlálása kategó-
              riánként az is… függvények segítségével. */
#include <stdio.h>
#include <ctype.h>
void main(void){
  short k, num=0, feher=0, egyeb=0;
  printf("Bemeneti karakterek leszámlálása\n"
         "kategóriánként EOF-ig, vagy Ctrl+Z-ig.\n");
```

```
  while((k=getchar())!=EOF)
    if(isdigit(k)) ++num;
    else if (isspace(k)) ++feher;
    else ++egyeb;
  printf("Karakter számok:\n----------------\n"
         "numerikus: %5hd\nfehér:     %5hd\n"
         "egyéb:     %5hd\n----------------\n"
         "össz: %10ld\n", num, feher, egyeb,
         (long)num+feher+egyeb); }
```

Írjuk még át ugyanebben a szellemben a **PELDA18.C egesze** függvényét is!

```
#include <ctype.h>
#define HSZ sizeof(int)/sizeof(short)*5
int egesze(char s[]){
  int i = 0, kezd;
  while(isspace(s[i])) ++i;
  if(s[i]=='+' || s[i]=='-') ++i;
  kezd=i;     /* A számjegyek itt kezdődnek. */
  while(isdigit(s[i]) && i-kezd<HSZ) ++i;
  if(kezd==i || !isspace(s[i]) && s[i]!=0) return 0;
  else return 1; }
```

A **PELDA13.C**–beli **strup** rutin így módosulna:

```
#include <ctype.h>
void strup(char s[]){
  int i;
  for(i=0; s[i]; ++i) s[i]=toupper(s[i]); }
```

Milyen előnyei vannak a karakterosztályozó függvények használatának?

- A kód rövidebb, és ez által gyorsabb is.
- A program portábilis lesz, hisz függetlenedik az ASCII (vagy más) kódtábla sajátosságaitól.

☞ Az olvasó utolsó logikus kérdése már csak az lehet, hogy miért pont a makrók között tárgyaljuk a karaktervizsgáló rutinokat?

Több C implementáció makróként valósítja meg ezeket a függvényeket. Erre mutatunk itt be egy szintén nem teljes körű példát azzal a feltétellel, hogy a **CHAR_BIT** (bitek száma a **char** típusban – lásd **LIMITS.H**–t!) makró értéke 8.

```
/* Bitmaszk értékek a lehetséges karaktertípusokra: */
#define _UPPER    0x1    /* Nagybetű. */
#define _LOWER    0x2    /* Kisbetű. */
#define _DIGIT    0x4    /* Decimális számjegy. */
#define _SPACE    0x8    /* '\t','\r','\n','\v','\f' */
#define _PUNCT    0x10   /* Elválasztó-jel. */
```

```
#define _CONTROL0x20      /* Vezérlő karakter. */
#define _BLANK    0x40    /* Szóköz. */
#define _HEX    0x80      /* Hexadecimális számjegy. */
/* Globális tömb, melyben a rendszer mindenegyes kódtábla
  pozícióra beállította ezeket a biteket: */
extern unsigned char _ctype[];
/* Néhány makró: */
#define islower(_c)     (_ctype[_c]&_LOWER)
#define isupper(_c)     (_ctype[_c]&_UPPER)
#define isalpha(_c)     (_ctype[_c]&(_UPPER|_LOWER))
#define isdigit(_c)     (_ctype[_c]&_DIGIT)
#define isalnum(_c)     (_ctype[_c]&(_UPPER|_LOWER|_DIGIT))
#define isxdigit(_c)    (_ctype[_c]&_HEX)
#define isspace(_c)     (_ctype[_c]&(_SPACE|_BLANK))
#define isprint(_c)     (_ctype[_c]&(_BLANK|_PUNCT|_UPPER|\
                                     _LOWER|_DIGIT))
```

Megoldandó feladatok:

Készítse el a következő függvények makró változatát!

- A fejezetben említett **tolower**–ét és **toupper**–ét.
- A **TÍPUSOK ÉS KONSTANSOK** szakaszban megírt **strcopy**–ét, és más karakterlánc kezelőkét.

☞ Ha az olvasóban felmerült volna az a gondolat, hogy mi van akkor, ha ugyanolyan nevű makró és függvény is létezik, akkor arra szeretnénk emlékeztetni, hogy:

- Az előfeldolgozás mindig a fordítás előtt történik meg, s így mindenből makró lesz.
- Ha #**undef** direktívával definiálatlanná tesszük a makrót, akkor attól kezdve csak függvény lesz a forrásszövegben.
- Ha a hívásban redundáns zárójelbe zárjuk a makró vagy a függvény nevét, akkor az előfeldolgozó ezt nem fejti ki, tehát bizonyosan függvényhívás lesz belőle.

  `...(`*`makrónév`*`)(`*`paraméterek`*`)...`

7.8 Feltételes fordítás

feltételes-fordítás:
  if-csoport <*elif-csoportok*> <*else-csoport*> *endif-sor*

if-csoport:
  #if *konstans-kifejezés újsor* <*csoport*>
  #ifdef *azonosító újsor* <*csoport*>
  #ifndef *azonosító újsor* <*csoport*>

elif-csoportok:
  elif-csoport
  elif-csoportok elif-csoport
elif-csoport:
  #elif *konstans-kifejezés újsor <csoport>*
else-csoport:
  #else *újsor <csoport>*
endif-sor:
  #endif *újsor*
újsor:
  soremelés

A feltételes direktívák szerint kihagyandó forrássorokat az előfeldolgozó törli a forrásszövegből, s a feltételes direktívák sorai maguk pedig kimaradnak az eredmény fordítási egységből. A feltételes direktívák által képzett konstrukciót - melyet rögtön bemutatunk egy általános példán - mindenképpen be kell fejezni abban a forrásfájlban, amelyben elkezdték.

#if *konstans-kifejezés1*
        <szekció1>
*<***#elif** *konstans-kifejezés2*
        <szekció2>>
/ . . . */*
*<***#elif** *konstans-kifejezésN*
        <szekcióN>>
*<***#else**
        <végső-szekció>>
#endif

Lássuk a kiértékelést!

1, Ha a *konstans-kifejezés1* értéke nem zérus (igaz), akkor a preprocesszor a *szekció1* sorait feldolgozza, és az eredményt átadja a fordítónak. A *szekció1* természetesen üres is lehet. Ezután az ezen **#if**-hez tartozó összes többi sort a vonatkozó #**endif**-fel bezárólag kihagyja, s az #**endif**-t követő sorral folytatja a munkát az előfeldolgozó.

2, Ha a *konstans-kifejezés1* értéke zérus (hamis), akkor a preprocesszor a *szekció1*-t teljes egészében elhagyja. Tehát nincs makrókifejtés, és nem adja át a feldolgozott darabot a fordítónak! Ezután viszont a következő #**elif** *konstans-kifejezés*e kiértékelésébe fog, s így tovább.

3, Összesítve az #**if**-en és az #**elif**-eken lefelé haladva az a *szekció* kerül előfeldolgozásra, s ennek eredménye fordításra, melynek *konstans-kifejezés*e igaznak bizonyul. Ha egyik ilyen *konstans-kifejezés* sem igaz, akkor az #**else** *végső-szekció*jára vonatkoznak az előbbiekben mondottak.

✍ Az #**if** . . . #**endif** konstrukciók tetszőleges mélységben egymásba ágyazhatók.

💣 Az #**if** . . . #**endif** szerkezetbeli *konstans-kifejezés*eknek korlátozott, egész típusúaknak kell lenniük! Konkrétabban egész konstanst, karakter állandót tartalmazhat a kifejezés, és benne lehet a **defined** operátor is. Tilos használni viszont benne explicit típuskonverziót, **sizeof** kifejezést, enumerátort és lebegőpontos konstanst, mint normál egész típusú konstans kifejezésekben! Az előfeldolgozó közönséges makróhelyettesítési menettel dolgozza fel a *konstans-kifejezés*eket.

7.8.1 A defined operátor

Makróazonosítók definiáltságának ellenőrzésére való, s csak #**if** és #**elif** *konstans-kifejezés*eiben szerepelhet. A

defined(*azonosító*)

vagy a

defined *azonosító*

alak a makróazonosító definiáltságára kérdez rá. Miután a válasz logikai érték a **defined** szerkezetek logikai műveletekkel is kombinálhatók a *konstans-kifejezés*ekben. Például:

```
#if defined(makro1) && !defined(makro2)
```

Ha biztosítani szeretnénk azt, hogy a fordítási egységbe egy bizonyos fejfájl (legyen **HEADER.H**) csak egyszer épüljön be, akkor a fejfájl szövegét következőképp kell direktívákba foglalni:

```
#if !defined(_HEADERH)
  #define _HEADERH
  /* Itt van a fejfájl szövege. */
#endif
```

Ilyenkor akárhány #**include** is jön a forrásfájlban a **HEADER.H** fejfájlra, a behozatala csak először történik meg, mert a további bekapcsolásokat az **_HEADERH** makró definiáltsága megakadályozza.

✍ Nézzünk csak bele néhány szabványos fejfájlba, ott is alkalmazzák ezt a konstrukciót!

7.8.2 Az #ifdef és az #ifndef direktívák

Az #**ifdef** direktíva egy makróazonosító definiáltságára, s az #**ifndef** viszont a definiálatlanságára kérdez rá, azaz:

#**ifdef** *azonosító* ≡ #**if defined**(*azonosító*)

#**ifndef** *azonosító* ≡ #**if** !**defined**(*azonosító*)

7.9 #line sorvezérlő direktíva

#**line** *egész-konstans* <”*fájlazonosító*”> *újsor*

Jelzi az előfeldolgozónak, hogy a következő forrássor *egész-konstans* sorszámú, és a *fájlazonosító* nevű fájlból származik. Miután az aktuálisan feldolgozás alatt álló forrásfájlnak is van azonosítója a *fájlazonosító* paraméter elhagyásakor a #**line** az aktuális fájlra vonatkozik.

A makrókifejtés a #**line** paramétereiben is megtörténik.

Vegyünk egy példát!

```
/* PELDA.C: a #line direktívára: */
#include <stdio.h>
#line 4 ”PIPI.C”
void main(void) {
  printf(”\nA(z) %s fájl %d sorában vagyunk!”, __FILE__,
         __LINE__);
  #line 12 ”PELDA.C”
  printf(”\n”);
  printf(”A(z) %s fájl %d sorában vagyunk!”, __FILE__,
         __LINE__);
  #line 8
  printf(”\n”);
  printf(”A(z) %s fájl %d sorában vagyunk!\n”, __FILE__,
         __LINE__);  }
```

Az előállított standard kimenet a következő lehet:

```
#line 1 "pelda.c"
#line 1 "c:\\msdev\\include\\stdio.h"
. . .
#line 524 "c:\\msdev\\include\\stdio.h"
#line 3 "pelda.c"
#line 4 "PIPI.C"
void main(void) {
  printf(”\nA(z) %s fájl %d sorában vagyunk!”, ”PIPI.C”,
         6);
  #line 12 "PELDA.C”
  printf(”\n”);
  printf(”A(z) %s fájl %d sorában vagyunk!”, ”PELDA.C”,
         13);
  #line 8 "PELDA.C"
  printf(”\n”);
  printf(”A(z) %s fájl %d sorában vagyunk!\n”,”PELDA.C”,
         9); }
```

✍ A **#line** direktíva tulajdonképpen a **__FILE__** és a **__LINE__** előredefiniált makrók értékét állítja. Ezek a makróértékek a fordító hibaüzeneteiben jelennek meg. Szóval a direktíva diagnosztikai célokat szolgál.

7.10 #error direktíva

#error <*hibaüzenet*> *újsor*

direktíva üzenetet generál, és befejeződik a fordítás. Az üzenet alakja lehet a következő:

`Error: fájlazonosító sorszám: Error directive: hibaüzenet`

Rendszerint #**if** direktívában használatos. Például:

```
#if (SAJAT!=0 && SAJAT!=1)
  #error A SAJAT 0-nak vagy 1-nek definiálandó!
#endif
```

7.11 #pragma direktívák

#pragma <*előfeldolgozó-szimbólumok***>** *újsor*

A direktívák gép és operációs rendszerfüggők. Bennük a **#pragma** kulcsszót követő szimbólumok mindig objektumai a makrókifejtésnek, és tulajdonképpen speciális fordítói utasítások, s ezek paraméterei. Az előfeldolgozó a fel nem ismert #**pragma** direktívát figyelmen kívül hagyja.

8 OBJEKTUMOK ÉS FÜGGVÉNYEK

Az azonosítók „értelmét” a deklarációk rögzítik. Tudjuk, hogy a deklaráció nem jelent szükségképpen memóriafoglalást. Csak a definíciós deklaráció ilyen.

deklaráció:
  deklaráció-specifikátorok<init-deklarátorlista>
init-deklarátorlista:
  init-deklarátor
  init-deklarátorlista, init-deklarátor
init-deklarátor:
  deklarátor
  deklarátor=inicializátor

Az *inicializátor*okkal és *inicializátorlistá*kkal a **BEVEZETÉS ÉS ALAPISMERETEK** szakasz **Inicializálás** fejezetében foglalkoztunk. A *deklarátor*ok a deklarálandó neveket tartalmazzák. A *deklaráció-specifikátorok* típus és tárolási osztály specifikátorokból állnak:

deklaráció-specifikátorok:
  tárolási-osztály-specifikátor<deklaráció-specifikátorok>
  típusspecifikátor<deklaráció-specifikátorok>
  típusmódosító<deklaráció-specifikátorok>

A *típusspecifikátor*okat a **TÍPUSOK ÉS KONSTANSOK** szakaszban tárgyaltuk, s ezek közül kivettük a **const** és a **volatile** *típusmódosító*kat.

8.1 Objektumok attribútumai

Az objektum egy azonosítható memória területet, mely konstans vagy változó érték(ek)et tartalmaz. Az objektum egyik attribútuma (tulajdonsága) az adattípusa. Tudjuk, hogy van ezen kívül azonosítója (neve) is. Az objektum adattípus attribútuma rögzíti az objektumnak

- allokálandó (lefoglalandó) memória mennyiségét és
- a benne tárolt információ belsőábrázolási formáját.

Az objektum neve nem attribútum, hisz különféle hatáskörben több különböző objektumnak is lehet ugyanaz az azonosítója. Az objektum további attribútumait (tárolási osztály, hatáskör, láthatóság, élettartam, stb.) a deklarációja és annak a forráskódban elfoglalt helye határozza meg.

✍ Emlékeztetőül: a lokális, globális és a belső, külső változókat taglaltuk már a **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban!

8.1.1 Tárolási osztályok

Az objektumokhoz rendelt azonosítóknak van legalább

- tárolási osztály és
- adattípus

attribútuma. Szokás e kettőt együtt is adattípusnak nevezni. A tárolási osztály specifikátor definíciója:

tárolási-osztály-specifikátor:
auto
register
extern
static
typedef

A tárolási osztály meghatározza az objektum élettartamát, hatáskörét és kapcsolódását. Egy adott objektumnak csak egy tárolási osztály specifikátora lehet egy deklarációban. A tárolási osztályt a deklaráció forráskódbeli elhelyezése implicit módon rögzíti, de a megfelelő tárolási osztály kulcsszó expliciten is beleírható a deklarációba.

☞ A kapcsolódással külön fejezetben foglalkozunk.

Tárolási osztály kulcsszó nélküli deklarációk esetében a blokkon belül deklarált

- objektum mindig **auto** definíció, és
- a függvény pedig **extern** deklaráció.

A függvénydefiníciók és az ezeken kívüli objektum és függvénydeklarációk mind **extern**, statikus tárolási osztályúak.

Kétféle tárolási osztály van.

8.1.1.1 Automatikus (auto, register) tárolási osztály

Az ilyen objektumok lokális élettartamúak, és lokálisak a blokk egy adott példányára. Az ilyen deklarációk definíciók is egyben, azaz megtörténik a memóriafoglalás is. Ismeretes, hogy a függvényparaméterek is automatikus tárolási osztályúaknak minősülnek. Rekurzív kód esetén az automatikus objektumok garantáltan különböző memória területen helyezkednek el mindenegyes blokkpéldányra.

A C az automatikus objektumokat a program vermében tárolja, s így alapértelmezett kezdőértékük "szemét". Expliciten inicializált, lokális automatikus objektum esetében a kezdőérték adás viszont mindannyiszor

megtörténik, valahányszor bekerül a vezérlés a blokkba. A blokkon belül definiált objektumok **auto** tárolási osztályúak, hacsak ki nem írták expliciten az **extern** vagy a **static** kulcsszót a deklarációjukban. Csak lokális hatáskörű objektumok deklarációjában használható az **auto** tárolási osztály specifikátor.

💣 Az **auto** kulcsszót tilos külső deklarációban vagy definícióban alkalmazni!

Az automatikus tárolási osztályú objektumok lokális élettartamúak és nincs kapcsolódásuk. Miután ez az alapértelmezés az összes lokális hatáskörű objektum deklarációjára, nem szokás és szükségtelen expliciten kiírni.

💣 Az **auto** tárolási osztály specifikátor függvényre nem alkalmazható!

Az automatikus tárolási osztály speciális válfaja a regiszteres. A **register** kulcsszó a deklarációban azt jelzi a fordítónak, hogy

- a változót nagyon gyakran fogjuk használni, és
- kérjük, hogy az illető objektumot regiszterben helyezze el, ha lehetséges.

✍ A regiszteres tárolás rövidebb gépi kódú programot eredményez, hisz elmarad a memóriából regiszterbe (és vissza) töltögetés. Emiatt, és mert a regiszter a memóriánál jóval kisebb elérési idejű, a szoftver futása is gyorsul.

A hardvertől függ ugyan, de valójában csak kevés objektum helyezkedhet el regiszterben, és csak meghatározott típusú változók kerülhetnek oda. A fordító elhagyja a **register** kulcsszót a felesleges és a nem megfelelő típusú deklarációkból, azaz az ilyen változók csak normál, automatikus tárolási osztályúak lesznek.

A regiszteres objektumok lokális élettartamúak, és ekvivalensek az automatikus változókkal. Csak lokális változók és függvényparaméterek deklarációjában alkalmazható a **register** kulcsszó.

💣 Külső deklarációban vagy definícióban a **register** kulcsszót tilos alkalmazni!

💣 Függetlenül attól, hogy a **register** változó igazán regiszterben helyezkedik el, vagy sem, tilos a címére hivatkozni!

🕮 A globális optimalizálást bekapcsolva a fordító figyelmen kívül hagyja a programozó **register** igényeit, s saját maga választ regiszter ki-

osztást, de minden más a **register** kulcsszóhoz kapcsolódó szemantikát tekintetbe vesz.

Írjunk **int prime**(**int** *x*) függvény, mely eldönti pozitív egész paraméteréről, hogy prímszám–e!

A prímszám csak 1–gyel és önmagával osztható maradék nélkül. Próbáljuk meg tehát egész számok szorzataként előállítani. Ha sikerül, akkor nem törzsszámról van szó. A szám prím viszont, ha ez nem megy. Indítunk tehát 2–ről mindig növelgetve egy **osz** változót, és megpróbáljuk, hogy osztható–e vele maradék nélkül az **x**.

Meddig növekedhet az **osz**? **x** négyzetgyökéig, mert a két szorzótényezőre bontásnál fordított arányosság van a két tényező között.

```
int prime(register x){
  register osz = 2;
  if(x < 4) return 1;
  while(osz*osz <= x){
    if(!(x%osz)) return 0;
    ++osz;
    if(!(osz&1)) ++osz; }
  return 1; }
```

✍ A **prime** paramétere és az **osz** lokális változó **register int** típusú programgyorsítási céllal. A függvény utolsó előtti sorából látszik, hogy legalább a páros számokat nem próbáljuk ki osztóként, miután 2–vel nem volt maradék nélkül osztható az **x**.

Az olvasóra bízzuk, hogy kísérje meg még gyorsítani az algoritmust!

Készítsünk programot, mely megállapítja egy valós számsorozat átlagát és azt, hogy hány átlagnál kisebb és nagyobb eleme van a sorozatnak! A valós számokat a szabvány bemenetről kell beolvasni! Egy sorban egyet! A sorozat megadásának végét jelentse üres sor érkezése a bemenetről!

Kezdjük a kódolást az **int lebege**(**char** *s*[]) függvénnyel, mely megállapítja karakterlánc paraméteréről, hogy formálisan helyes lebegőpontos szám–e! A karaktertömb elején levő fehér karaktereket át kell lépni, és a numerikus rész ugyancsak fehér, vagy lánczáró zérus karakterrel zárul. A lebegőpontos számnak ki kell egyébként elégítenie a lebegőpontos konstans írásszabályát!

```
#include <ctype.h>
int lebege(char s[]){
  int i=0, kezd;
  /* Fehér karakterek átlépése a lánc elején: */
  while(isspace(s[i])) ++i;
  /* A mantissza előjele: */
```

```
  if(s[i]=='+'||s[i]=='-') ++i;
  kezd=i;                  /* A szájegyek itt kezdődnek. */
  /* A mantissza egész része: */
  while(isdigit(s[i])) ++i;
  /* A mantissza tört része: */
  if(s[i]=='.') ++i;
  while(isdigit(s[i])) ++i;
  /* Nincs számjegy, vagy csak egy . van: */
  if(i==kezd||kezd+1==i&&s[kezd]=='.') return 0;
  /* Kitevő rész: */
  if(toupper(s[i])=='E'){
    ++i;
    if(s[i]=='+'||s[i]=='-')++i;
    /* Egy számjegynek lennie kell a kitevőben! */
    if(!isdigit(s[i])) return 0;
    while(isdigit(s[i])) ++i;}
  /* Vége: */
  if(isspace(s[i])||!s[i]) return 1;
  else return 0; }
```

8.1.1.2 Statikus (static, extern) tárolási osztály

A statikus tárolási osztályú objektumok kétfélék:

- blokkra lokálisak, vagy
- blokkokon át külsők.

Az ilyen tárolási osztályú objektumok statikus élettartamúak. Bárhogyan is: megőrzik értéküket az egész program végrehajtása során, akárhányszor is hagyja el a vezérlés az őket tartalmazó blokkot, és tér oda vissza. Függvényen, blokkon belül úgy definiálható statikus tárolási osztályú változó, hogy a deklarációjába ki kell expliciten írni a **static** kulcsszót. A **static** kulcsszavas deklaráció definíció.

Készítsünk **double gyujto**(**double** *a*) függvényt, mely gyűjti aktuális paraméterei értékét! Mindig az eddig megállapított összeget adja vissza. Ne legyen „bamba”, azaz ne lehessen vele „áttörni” az ábrázolási határokat!

☞ A lebegőpontos túl, vagy alulcsordulás futásidejű hiba!

```
#include <float.h>
double gyujto(double a){
  static double ossz=0.0;
  if(a<0.0&&-(DBL_MAX+a)<ossz || DBL_MAX-a>ossz)
    ossz+=a;
  return ossz; }
```

✍ Látszik, hogy nincs összegzés, ha az **ossz** –**DBL_MAX**–hoz, vagy +**DBL_MAX**–hoz **a** távolságon belülre kerül.

Rekurzív kódban a statikus objektum állapota garantáltan ugyanaz minden függvénypéldányra.

Explicit inicializátorok nélkül a statikus változók minden bitje zérus kezdőértéket kap. Az implicit és az explicit inicializálás még lokális statikus objektum esetén is csak egyszer történik meg, a program indulásakor.

✍ A **gyujto**–ben teljesen felesleges zérussal inicializálni a statikus **ossz** változót, hisz implicit módon is ez lenne a kezdőértéke.

A függvénydefiníciókon kívül elhelyezett, tárolási osztály kulcsszó nélküli deklarációk külső, statikus tárolási osztályú objektumokat definiálnak, melyek globálisak az egész programra nézve.

A külső objektumok statikus élettartamúak. Az explicit módon **extern** tárolási osztályúnak deklaráltak olyan objektumokat deklarálnak, melyek definíciója nem ebben a fordítási egységben van, vagy befoglaló hatáskörben található.

```
extern int MasholDefinialt; /* Más ford. egységben */
void main(){
  int IttDefiniált;
  {
    extern int IttDefiniált;
    /* A befoglaló hatáskörbeli IttDefiniált-ra való
       hivatkozás. */ } }
```

A külső kapcsolódást jelölendő az **extern** függvény és objektum fájl és lokális hatáskörű deklarációiban használható. Fájl hatáskörű változók és függvények esetében ez az alapértelmezés, tehát expliciten nem szokás kiírni.

💣 Az **extern** kulcsszó explicit kiírása tilos a változó definiáló deklarációjában!

A külső objektumok és a függvények is deklarálhatók **static**–nek, amikor is lokálissá válnak az őket tartalmazó fordítási egységre, és minden ilyen deklaráció definíció is egyben.

Folytatva a példánkat: a szabvány bemenetről érkező, valós számokat valahol tárolni kéne, mert az átlag csak az összes elem beolvasása után állapítható meg. Ez után újra végig kell járni a számokat, hogy kideríthessük, hány átlag alatti és feletti van köztük.

A változatosság kedvéért, és mert a célnak tökéletesen megfelel, használjunk vermet a letároláshoz, melyet és kezelő függvényeinek definícióit

helyezzük el a **DVEREM.C** forrásfájlban, és a más fordítási egységből is hívható függvények prototípusait tegyük be a **DVEREM.H** fejfájlba!

☝ Egy témakör adatait és kezelő függvényeit egyébként is szokás a C–ben külön forrásfájlban (úgy nevezett implementációs fájlban) elhelyezni, vagy a lefordított változatot külön könyvtárfájlba tenni.

A dologhoz mindig tartozik egy fejfájl is, mely tartalmazza legalább a témakör más forrásmodulból is elérhető adatainak deklarációit, és kezelő függvényeinek prototípusait. Implementációs fájl esetén a fejfájlban még típusdefiníciók, szimbolikus állandók, makrók, stb. szoktak lenni.

```
/* DVEREM.H: double verem push, pop és clear
             függvényenek prototípusai. */
int clear(void);
double push(double x);
double pop(void);
```

```
/* DVEREM.C: double verem push, pop és clear
             függvényekkel. */
#define MERET 128      /* A verem mérete. */
static int vmut;       /* A veremmutató. */
static double v[MERET];/* A verem. */
int clear(void){
  vmut=0;
  return MERET; }
double push(double x){
  if(vmut<MERET) return v[vmut++]=x;
  else return x+1.; }
double pop(void){
  if(vmut>0) return v[--vmut];
  else return 0.; }
```

✍ A **vmut** veremmutató, és a **v** verem statikus ugyan, de lokális a **DVEREM.C** fordítási egységre. Látszik, hogy a **push x** paraméterével tölti a vermet, és sikeres esetben ezt is adja vissza. Ha a verem betelt, más érték jön vissza tőle. A **pop** visszaszolgáltatja a legutóbb betett értéket, ill. az üres veremből mindig zérussal tér vissza. A **clear** törli a veremmutatót, s ez által a vermet, és visszaadja a verem maximális méretét.

Kódoljuk le végre az eredetileg kitűzött feladatot!

```
/* PELDA20.C: Valós számok átlaga, és az ez alatti, ill.
              feletti elemek száma. */
#include <stdio.h>
#include <stdlib.h>    /* Az atof miatt! */
#define INP 60         /* Az input puffer mérete. */
int getline(char s[],int lim);
#include <ctype.h>
int lebege(char s[]);
```

```
#include <float.h>
double gyujto(double a);
#include "DVEREM.H"
void main(void){
  int max=clear(),      /* A verem max. mérete. */
    i=0, alatt, felett;
  char s[INP+1];        /* Az input puffer. */
  double a;
  printf("Számsorozat átlaga alatti és feletti "
         "elemeinek száma.\nA megadást üres "
         "sorral kell befejezni!\n\n");
  while( printf("%4d. elem: ", i+1), i<max &&
         getline(s,INP)>0)
       if(lebege(s)){
         push(a=atof(s));
         printf("Az összeg:%30.6f\n\n", a=gyujto(a));
         ++i;}
  printf("\nAz átlag: %30.6f\n", a/=i);
  for(max=alatt=felett=0; max<i; ++max){
    double b=pop();
    if(b<a) ++alatt;
    else if(b>a) ++felett; }
  printf("Az átlag alattiak száma: %8d.\n"
         "Az átlag felettiek száma:%8d.\n",
         alatt, felett); }
```

Vigyázat: a példa a **PELDA20.C**–ből és a **DVEREM.C**–ből képzett prodzsekt segítségével futtatható csak!

8.1.2 Élettartam (lifetime, duration)

Az élettartam attribútum szorosan kötődik a tárolási osztályhoz, s az a periódus a program végrehajtása közben, míg a deklarált azonosítóhoz objektumot allokál a fordító a memóriában, azaz amíg a változó vagy a függvény létezik. Megkülönböztethetünk

- fordítási idejű és
- futásidejű objektumokat.

A változók például a típusoktól és a típusdefinícióktól eltérően futás időben valós, allokált memóriával rendelkeznek. Három fajta élettartam van.

8.1.2.1 Statikus (static vagy extern) élettartam

Az ilyen objektumokhoz a memória hozzárendelés a program futásának megkezdődésekor történik meg, s az allokáció marad is a program befejeződéséig. Minden függvény statikus élettartamú objektum bárhol is defi-

niálják őket. Az összes fájl hatáskörű változó is ilyen élettartamú. Más változók a **static** vagy az **extern** tárolási osztály specifikátorok explicit megadásával tehetők ilyenné. A statikus élettartamú objektumok minden memória bitje (a függvényektől eltekintve) zérus kezdőértéket kap explicit inicializálás hiányában.

💣 Ne keverjük össze a statikus élettartamot a fájl (globális) hatáskörrel, ui. egy objektum lokális hatáskörrel is lehet statikus élettartamú, csak deklarációjában meg kell adni expliciten a **static** tárolási osztály kulcsszót.

8.1.2.2 Lokális (auto vagy register) élettartam

Ezek az objektumok akkor jönnek létre (allokáció) a veremben vagy regiszterben, amikor a vezérlés belép az őket magába foglaló blokkba vagy függvénybe, s meg is semmisülnek (deallokáció), mihelyt kikerül a vezérlés innét. A lokális élettartamú objektumok lokális hatáskörűek, és mindig explicit inicializálásra szorulnak, hisz létrejövetelük helyén „szemét” van.

✍ Ne feledjük, hogy a függvényparaméterek is lokális élettartamúak!

Az **auto** tárolási osztály specifikátor deklarációban való kiírásával expliciten lokális élettartamúvá tehetünk egy változót, de erre többnyire semmi szükség sincs, mert blokkon vagy függvényen belül deklarált változók esetében az alapértelmezett tárolási osztály amúgy is az **auto**.

A lokális élettartamú objektum egyben lokális hatáskörű is, hisz az őt magába foglaló blokkon kívül nem létezik. A dolog megfordítása nem igaz, mert lokális hatáskörű objektum is lehet statikus élettartamú.

Ha egy regiszterben is elférő változót (például **char**, **short**, stb. típusút) expliciten **register** tárolási osztályúnak deklarálunk, akkor a fordító ehhez hozzáérti automatikusan az **auto** kulcsszót is, hisz a változókat csak addig tudja regiszterben elhelyezni, míg azok el nem fogynak, s ezután a veremben allokál nekik memóriát.

8.1.2.3 Dinamikus élettartam

Az ilyen objektumokhoz a C–ben például a **malloc** függvénnyel rendelhetünk memóriát a heap–en, amit aztán a **free**–vel felszabadíthatunk. Miután a memóriaallokáláshoz könyvtári függvényeket használunk, és ezek nem részei a nyelvnek, így a C–ben nincs is dinamikus élettartam igazából.

🕮 A C++–ban ugyanezen funkciókra megalkották a **new** és a **delete** operátorokat, melyek részei a nyelvnek.

8.1.3 Hatáskör (scope) és láthatóság (visibility)

A hatáskör – érvényességi tartománynak is nevezik – az azonosító azon tulajdonsága, hogy vele az objektumot a program mely részéből érhetjük el. Ez is a deklaráció helyétől és magától a deklarációtól függő attribútum. Felsoroljuk őket!

8.1.3.1 Blokk (lokális, belső) hatáskör

A deklarációs ponttól indul és a deklarációt magába foglaló blokk végéig tart. Az ilyen hatáskörű változókat szokás belső változóknak is nevezni. Lokális hatáskörűek a függvények formális paraméterei is, s hatáskörük a függvénydefiníció teljes blokkja. A blokk hatáskörű azonosító hatásköre minden a kérdéses blokkba beágyazott blokkra is kiterjed. Például:

```
int fv(float lo){
  double szamar;       /* A lokális hatáskör itt indul. */
  /* . . . */
  long double oszver;  /* Az oszver hatásköre innét
       (deklarációs pont) indul és a függvénydefiníció
       végéig tart. Szóval nem lehetne a szamar változót
       az oszver-rel inicializálni. */
    if (/* feltétel */){
      char lodarazs = 'l'; /* A lodarazs hatásköre
                              ez a belső blokk. */
      /* . . . */
      }      /* A lodarazs hatáskörének vége. */
    /* . . . */
  }    /* A lo, szamar és oszver hatáskörének vége. */
```

8.1.3.2 Függvény hatáskör

Ilyen hatásköre csak az utasítás címkének van. Az utasítás címke ezen az alapon: függvényen belüli egyedi azonosító, melyet egy olyan végrehajtható utasítás elé kell írni kettőspontot közbeszúrva, melyre el kívánunk ágazni. Például:

```
int fv(float k){
  int i, j;
  /* . . . */
  cimke: utasítás;
  /* . . . */
  if (/* feltétel */) goto cimke;
  /* . . . */ }
```

8.1.3.3 Függvény prototípus hatáskör

Ilyen hatásköre a prototípusban deklarált paraméterlista azonosítóinak van, melyek tehát a függvény prototípussal be is fejeződnek. Például a

következő függvénydefinícióban az **i**, **j** és **k** azonosítóknak van függvény prototípus hatásköre:

```
void fv(int i, char j, float k);
```

☞ Az **i**, **j** és **k** ilyen megadásának semmi értelme sincs. Az azonosítók teljesen feleslegesek. A

```
void fv(int, char, float);
```

ugyanennyit „mondott” volna. Függvény prototípusban neveket akkor célszerű használni, ha azok leírnak valamit. Például a

```
double KamatOsszeg(double osszeg, double kamat, int evek);
```

az **osszeg** kamatos **kamat**át közli **evek** évre.

8.1.3.4 Fájl (globális, külső) hatáskör

A minden függvény testén kívül deklarált azonosítók rendelkeznek ilyen hatáskörrel, mely a deklarációs pontban indul és a forrásfájl végéig tart. Ez persze azt is jelenti, hogy a fájl hatáskörű objektumok a deklarációs pontjuktól kezdve minden függvényből és blokkból elérhetők. A globális változókat szokás külső változóknak is nevezni. Például a **g1**, **g2** és **g3** változók ilyenek:

```
int g1 = 7;        /* g1 fájl hatásköre innét indul. */
void main(void) { /* ... */ }
float g2;          /* g2 fájl hatásköre itt startol. */
void fv1(void) { /* ... */ }
double g3 = .52E-40; /* Itt kezdődik g3 hatásköre */
void fv2(void) { /* ... */ }
/* Itt van vége a forrásfájlnak és a g1, g2 és g3 külső
   változók hatáskörének. */
```

8.1.3.5 Láthatóság

A forráskód azon régiója egy azonosítóra vonatkozóan, melyben legális módon elérhető az azonosítóhoz kapcsolt objektum. A hatáskör és a láthatóság többnyire fedik egymást, de bizonyos körülmények között egy objektum ideiglenesen rejtetté válhat egy másik ugyanilyen nevű azonosító feltűnése miatt. A rejtett objektum továbbra is létezik, de egyszerűen az azonosítójával hivatkozva nem érhető el, míg a másik ugyanilyen nevű azonosító hatásköre le nem jár. Például:

```
{ int i; char c = 'z'; /* Az i és c hatásköre indul. */
  i = 3;               /* int i-t értük el. */
  /* . . . */
  { double i = 3.5e3;  /* double i hatásköre itt kezdő-
                          dik, s elrejti int i-t, bár */
```

```
    /* . . . */        /* hatásköre nem szűnik meg. */
    c = 'A';           /* char c látható és hatásköre
                          itt is tart. */
  }         /* A double i hatáskörének vége */
            /* int i és c hatáskörben és láthatók. */
  ++i;
}           /* int i és char c hatáskörének vége. */
```

8.1.3.6 Névterület (name space)

Az a „hatáskör”, melyen belül az azonosítónak egyedinek kell lennie. Más névterületen konfliktus nélkül létezhet ugyanilyen azonosító, a fordító képes megkülönböztetni őket.

A névterület fajtákat a **STRUKTÚRÁK ÉS UNIÓK** tárgyalása után tisztázzuk majd!

8.1.4 Kapcsolódás (linkage)

A kapcsolódást csatolásnak is nevezik. A végrehajtható program úgy jön létre, hogy

- több, különálló fordítási egységet fordítunk,
- aztán a kapcsoló-szerkesztővel (linker) összekapcsoltatjuk az eredmény .**OBJ** fájlokat, más meglévő tárgymodulokat és a könyvtárakból származó tárgykódokat.

Probléma akkor van, ha ugyanaz az azonosító különböző hatáskörökkel deklarált - például más-más forrásfájlban - vagy ugyanolyan hatáskörrel egynél többször is deklarált.

A kapcsoló-szerkesztés az a folyamat, mely az azonosító minden előfordulását korrekt módon egy bizonyos objektumhoz vagy függvényhez rendeli. E folyamat során minden azonosító kap egy kapcsolódási attribútumot a következő lehetségesek közül:

- külső (external) kapcsolódás,
- belső (internal) kapcsolódás vagy
- nincs (no) kapcsolódás.

Ezt az attribútumot a deklarációk elhelyezésével és formájával, ill. a tárolási osztály (**static** vagy **extern**) explicit vagy implicit megadásával határozzuk meg.

Lássuk a különféle kapcsolódások részleteit!

A külső kapcsolódású azonosító minden példánya ugyanazt az objektumot vagy függvényt reprezentálja a programot alkotó minden forrásfájlban és könyvtárban. A belső kapcsolódású azonosító ugyanazt az objektumot vagy függvényt jelenti egy és csak egy fordítási egységben (forrásfájlban). A belső kapcsolódású azonosítók a fordítási egységre, a külső kapcsolódásúak viszont az egész programra egyediek. A külső és belső kapcsolódási szabályok a következők:

- Bármely objektum vagy függvényazonosító fájl hatáskörrel belső kapcsolódású, ha deklarációjában expliciten előírták a **static** tárolási osztályt.
- Az explicit módon **extern** tárolási osztályú objektum vagy függvényazonosítónak ugyanaz a kapcsolódása, mint bármely látható fájl hatáskörű deklarációjának. Ha nincs ilyen látható fájl hatáskörű deklaráció, akkor az azonosító külső kapcsolódású lesz.
- Ha függvényt explicit tárolási osztály specifikátor nélkül deklarálnak, akkor kapcsolódása olyan lesz, mintha kiírták volna az **extern** kulcsszót.
- Ha fájl hatáskörű objektumazonosítót deklarálnak tárolási osztály specifikátor nélkül, akkor az azonosító külső kapcsolódású lesz.

💣 A fordítási egység belső kapcsolódásúnak deklarált azonosítójához egy és csak egy külső definíció adható meg. A külső definíció olyan külső deklaráció, mely az objektumhoz vagy függvényhez memóriát is rendel. Ha külső kapcsolódású azonosítót használunk kifejezésben (a **sizeof** operandusától eltekintve), akkor az azonosítónak csak egyetlen külső definíciója létezhet az egész programban.

A kapcsolódás nélküli azonosító egyedi entitás. Ha a blokkban az azonosító deklarációja nem vonja maga után az **extern** tárolási osztály specifikátort, akkor az azonosítónak nincs kapcsolódása, és egyedi a függvényre. A következő azonosítóknak nincs kapcsolódása:

- Bármely nem objektum vagy függvénynévvel deklarált azonosítónak. Ilyen például a típusdefiníciós (**typedef**) azonosító.
- A függvényparamétereknek.
- Explicit **extern** tárolási osztály specifikátor nélkül deklarált, blokk hatáskörű objektumazonosítóknak.

8.2 Függvények

A függvényekkel kapcsolatos alapfogalmakat tisztáztuk már a **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban, de fussunk át rajtuk még egyszer!

A függvénynek kell legyen definíciója, és lehetnek deklarációi. A függvény definíciója deklarációnak is minősül, ha megelőzi a forrásszövegben a függvényhívást. A függvénydefinícióban van a függvény teste, azaz az a kód, amit a függvény meghívásakor végrehajt a processzor.

A függvénydefiníció rögzíti a függvény nevét, visszatérési értékének típusát, tárolási osztályát és más attribútumait. Ha a függvénydefinícióban a formális paraméterek típusát, sorrendjét és számát is előírják, függvény prototípusnak nevezzük. A függvény deklarációjának meg kell előznie a függvényhívást, melyben aktuális paraméterek vannak. Ez az oka annak, hogy a forrásfájlban a szabvány függvények hívása előtt behozzuk a prototípusaikat tartalmazó fejfájlokat (#**include**).

📖 A függvényparamétereket argumentumoknak is szokták nevezni.

A függvényeket a forrásfájlokban szokás definiálni, vagy előrefordított könyvtárakból lehet bekapcsoltatni (linkage). Egy függvény a programban többször is deklarálható, feltéve, hogy a deklarációk kompatibilisek. A függvény prototípusok használata a C-ben ajánlatos (a C++ meg úgy is kötelezően előírja), mert a fordítót így látjuk el elegendő információval ahhoz, hogy ellenőrizhesse

- a függvény nevét (a függvények adott azonosító),
- a paraméterek számát, típusát és sorrendjét (típuskonverzió lehetséges), valamint
- a függvény által visszaadott érték típusát (típuskonverzió itt is lehet).

A függvényhívás átruházza a vezérlést a hívó függvényből a hívott függvénybe úgy, hogy az aktuális paramétereket is – ha vannak – átadja érték szerint. Ha a hívott függvényben **return** utasításra ér a végrehajtás, akkor visszakapja a vezérlést a hívó függvény egy visszaadott értékkel együtt (ha megadtak ilyet!).

💣 Egy függvényre a programban csak egyetlen definíció lehetséges. A deklarációk (prototípusok) kötelesek egyezni a definícióval.

8.2.1 Függvénydefiníció

A függvénydefiníció specifikálja a függvény nevét, a formális paraméterek típusát, sorrendjét és számát, valamint a visszatérési érték típusát, a függvény tárolási osztályát és más attribútumait. A függvénydefinícióban van a függvény teste is, azaz a használatos lokális változók deklarációja, és a függvény tevékenységét megszabó utasítások. A szintaktika:

fordítási-egység:
  külső-deklaráció
  fordítási-egység külső-deklaráció
külső-deklaráció:
  függvénydefiníció
  deklaráció
függvénydefiníció:
  <deklaráció-specifikátorok> deklarátor <deklarációlista> összetett-utasítás
deklarátor:
  <mutató> direkt-deklarátor
direkt-deklarátor:
  direkt-deklarátor(*paraméter-típus-lista*)
  direkt-deklarátor(*<azonosítólista>*)
deklarációlista:
  deklaráció
  deklarációlista deklaráció

A *külső-deklaráció*k hatásköre a fordítási egység végéig tart. A *külső-deklaráció* szintaktikája egyezik a többi *deklaráció*éval, de függvényeket csak ezen a szinten szabad definiálni, azaz:

💣 tilos függvényben másik függvényt definiálni!

✍ A *deklaráció* és az *azonosítólista* definíciók a **TÍPUSOK ÉS KONSTANSOK** szakasz **Deklaráció** fejezetében megtalálhatók. A *mutató*kat a következő szakasz tartalmazza.

A *függvénydefiníció*beli *összetett-utasítás* a függvény teste, mely tartalmazza a használatos lokális változók deklarációit, a külsőleg deklarált tételekre való hivatkozásokat, és a függvény tevékenységét megvalósító utasításokat.

Az opcionális *deklaráció-specifikátorok* és a kötelezően megadandó *deklarátor* együtt rögzítik a függvény visszatérési érték típusát és nevét. A *deklarátor* természetesen függvénydeklarátor, azaz a függvénynév és az őt követő zárójel pár. Az első *direkt-deklarátor*(*paraméter-típus-lista*) alak a függvény új (modern) stílusú definícióját teszi lehetővé. A *deklarátor* szintaktikában szereplő *direkt-deklarátor* a modern stílus szerint a definiálás alatt álló függvény nevét rögzíti, és a kerek zárójelben álló *pa-*

raméter-típus-lista specifikálja az összes paraméter típusát. Ilyen *deklarátor* tulajdonképpen a függvény prototípus is. Például:

```
char fv(int i, double d){
  /* . . . */ }
```

A második *direkt-deklarátor*(<*azonosítólista*>) forma a régi stílusú definíció:

```
char fv(i, d)
  int i;
  double d; {
  /* . . . */ }
```

A továbbiakban csak az új stílusú függvénydefinícióval foglalkozunk, s nem emlegetjük tovább a régit!

deklaráció-specifikátorok:
tárolási-osztály-specifikátor <*deklaráció-specifikátorok*>
típusspecifikátor <*deklaráció-specifikátorok*>
típusmódosító <*deklaráció-specifikátorok*>

típusmódosító: (a következők egyike!)
const
volatile

☞ A *tárolási-osztály-specifikátorok* és a *típuspecifikátor*ok definíciói a **TÍPUSOK ÉS KONSTANSOK** szakasz **Deklaráció** fejezetében megtekinthetők!

8.2.1.1 Tárolási osztály

*Függvénydefiníció*ban két tárolási osztály kulcsszó használható: az **extern** vagy a **static**. A függvények alapértelmezés szerint **extern** tárolási osztályúak, azaz normálisan a program minden forrásfájljából elérhetők, de explicit módon is deklarálhatók **extern–**nek.

Ha a függvény deklarációja tartalmazza az **extern** tárolási osztály specifikátort, akkor az azonosítónak ugyanaz a kapcsolódása, mint bármely látható, fájl hatáskörű ugyanilyen külső deklarációnak, és ugyanazt a függvényt jelenti. Ha nincs ilyen fájl hatáskörű, látható deklaráció, akkor az azonosító külső kapcsolódású. A fájl hatáskörű, tárolási osztály specifikátor nélküli azonosító mindig külső kapcsolódású. A külső kapcsolódás azt jelenti, hogy az azonosító minden példánya ugyanarra a függvényre hivatkozik, azaz az explicit vagy implicit módon **extern** tárolási osztályú függvény a program minden forrásfájljában látható.

💣 Az **extern–**től különböző tárolási osztályú, blokk hatáskörű függvénydeklaráció hibát generál.

A függvény explicit módon deklarálható azonban **static**-nek is, amikor is a rá való hivatkozást az őt tartalmazó forrásfájlra korlátozzuk, azaz a függvény belső kapcsolódású, és csak a definícióját tartalmazó forrásmodulban látható. Az ilyen függvény legelső bekövetkező deklarációjában (ha van ilyen!) és definíciójában is ki kell írni a **static** kulcsszót.

Akármilyen esetről is van szó azonban, a függvény mindig a definíciós vagy deklarációs pontjától a forrásfájl végéig látható magától.

8.2.1.2 A visszatérési érték típusa

A visszatérési érték típusa meghatározza a függvény által szolgáltatott érték méretét és típusát. A *függvénydefiníció* metanyelvi meghatározása az elhagyható *deklaráció-specifikátorok*kal kezdődik. Ezek közül tulajdonképpen a *típusspecifikátor* felel meg a visszatérési érték típusának.

E meghatározásokat nézegetve látható, hogy a visszaadott érték típusa bármi lehet eltekintve a tömbtől és a függvénytől (az ezekre mutató mutató persze megengedett). Lehet valamilyen aritmetikai típusú, lehet **void** (nincs visszaadott érték), de el is hagyható, amikor is alapértelmezés az **int**. Lehet struktúra, unió vagy mutató is, melyekről majd későbbi szakaszokban lesz szó.

A függvénydefinícióban előírt visszaadott érték típusának egyeznie kell a programban bárhol előforduló, e függvényre vonatkozó deklarációkban megadott visszatérési érték típussal. A meghívott függvény akkor ad vissza értéket a hívó függvénynek a hívás pontjára, ha a processzor *kifejezés*sel ellátott **return** utasítást hajt végre. A fordító természetesen előbb kiértékeli a *kifejezés*t, és konvertálja – ha szükséges – az értéket a visszaadott érték típusára. A **void** visszatérésűnek deklarált függvénybeli *kifejezés*sel ellátott **return** figyelmeztető üzenetet eredményez, és a fordító nem értékeli ki a *kifejezés*t.

💣 Vigyázat! A függvény típusa nem azonos a visszatérési érték típusával. A függvény típusban ezen kívül benne van még a paraméterek

- száma,
- típusai és
- sorrendje is!

8.2.1.3 Formális paraméterdeklarációk

A *függvénydefiníció* metanyelvi meghatározásából következően a modern stílusú *direkt-deklarátor*(*paraméter-típus-lista*) alakban, a zárójelben

álló *paraméter-típus-lista* vesszővel elválasztott paraméterdeklarációk sorozata.

paraméter-típus-lista:
paraméterlista
paraméterlista, ...
paraméterlista:
paraméterdeklaráció
paraméterlista, paraméterdeklaráció
paraméterdeklaráció:
deklaráció-specifikátor deklarátor
deklaráció-specifikátor <absztrakt-deklarátor>
absztrakt-deklarátor:
mutató
<mutató><direkt-absztrakt-deklarátor>
direkt-absztrakt-deklarátor:
(*absztrakt-deklarátor*)
<direkt-absztrakt-deklarátor>[*<konstans-kifejezés>*]
<direkt-absztrakt-deklarátor>(*<paraméter-típus-lista>*)

A *paraméterdeklaráció* nem tartalmazhat más *tárolási-osztály-specifikátor*t, mint a **register**–t. A *deklaráció-specifikátor* szintaktikabeli *típus-specifikátor* elhagyható, ha a típus **int**, és egyébként megadják a **register** tárolási osztály specifikátort. Összesítve a formális paraméterlista egy elemének formája a következő:

<**register**> *típusspecifikátor <deklarátor>*

💣 Az **auto**–nak deklarált függvényparaméter fordítási hiba!

A C szabályai szerint a paraméter lehet bármilyen aritmetikai típusú. Lehet akár tömb is, de függvény nem (az erre mutató mutató persze megengedett). A paraméter lehet természetesen struktúra, unió vagy mutató is, melyekről majd későbbi szakaszokban lesz szó. A paraméterlista lehet **void** is, ami nincs paraméter jelentésű.

📖 A formális paraméterazonosítók nem definiálhatók át a függvény testének külső blokkjában, csak egy ebbe beágyazott belső blokkban, azaz a formális paraméterek hatásköre és élettartama a függvénytest teljes legkülső blokkja. Az egyetlen rájuk is legálisan alkalmazható tárolási osztály specifikátor a **register**. Például:

```
int f1(register int i){/* ... */}/* Igény regiszteres
                                    paraméter átadásra. */
```

A **const** és a **volatile** módosítók használhatók a formális paraméter deklarátorokkal. Például a

```
void f0(double p1, const char s[]){
```

```
/* . . . */
s[0]='A';                    /* Szintaktikai hiba. */}
```

const–nak deklarált formális paramétere nem lehet balérték a függvény testében, mert hibaüzenetet okoz.

Ha nincs átadandó paraméter, akkor a paraméterlista helyére a definícióban és a prototípusban a **void** kulcsszó írandó:

```
int f2(void){/* ... */}        /* Nincs paraméter. */
```

Ha van legalább egy formális paraméter a listában, akkor az ,...-ra is végződhet:

```
int f3(char str[], ...){/* ... */}/* Változó számú vagy
                                     típusú paraméter. */
```

Az ilyen függvény hívásában legalább annyi aktuális paramétert meg kell adni, mint amennyi formális paraméter a ,... előtt van, de természetesen ezeken túl további aktuális paraméterek is előírhatók. A ,... előtti paraméterek típusának és sorrendjének ugyanannak kell lennie a függvény deklarációiban (ha egyáltalán vannak), mint a definíciójában.

A függvény aktuális paraméterei típusának az esetleges szokásos konverzió után hozzárendelés kompatibilisnek kell lennie a megfelelő formális paraméter típusokra. A ,... helyén álló aktuális paramétereket nem ellenőrzi a fordító.

📖 Az **STGARG.H** fejfájlban vannak olyan makrók, melyek segítik a felhasználói, változó számú paraméteres függvények megalkotását! A témára visszatérünk még a **MUTATÓK** kapcsán!

8.2.1.4 A függvény teste

A függvény teste elhagyható deklarációs és végrehajtható utasításokból álló összetett utasítás, azaz az a kód, amit a függvény meghívásakor végrehajt a processzor.

összetett-utasítás:
{<*deklarációlista*> <*utasításlista*>}

A függvénytestben deklarált változók lokálisak, **auto** tárolási osztályúak, ha másként nem specifikálták őket. Ezek a lokális változók akkor jönnek létre, mikor a függvényt meghívják, és lokális inicializálást hajt rajtuk végre a fordító. A függvény meghívásakor a vezérlést a függvénytest első végrehajtható utasítása kapja meg. **void**-ot visszaadó függvény blokkjában aztán a végrehajtás addig folytatódik, míg **return** utasítás nem következik vagy a függvény blokkját záró }-re nem kerül a vezérlés. Ezután a hívási ponttól folytatódik a program végrehajtása.

A „valamit” szolgáltató függvényben viszont lennie kell legalább egy **return** *kifejezés* utasításnak, és visszatérés előtt rá is kell, hogy kerüljön a vezérlés. A visszaadott érték meghatározatlan, ha a processzor nem hajt végre **return** utasítást, vagy a **return** utasításhoz nem tartozott *kifejezés*. A *kifejezés* értékét szükséges esetben hozzárendelési konverziónak veti alá a fordító, ha a visszaadandó érték típusa eltér a *kifejezés*étől.

8.2.2 Függvény prototípusok

A függvénydeklaráció megelőzi a definíciót, és specifikálja a függvény nevét, a visszatérési érték típusát, tárolási osztályát és a függvény más attribútumait. A függvénydeklaráció akkor válik prototípussá, ha benne megadják az elvárt paraméterek típusát, sorrendjét és számát is.

✍ Összegezve: a függvény prototípus csak abban különbözik a definíciótól, hogy a függvény teste helyén egy ; van.

👍 C–ben ugyan nem kötelező, de tegyük magunknak kötelezővé a függvény prototípus használatát, mert ez a következőket rögzíti:

- A függvény **int**–től különböző visszatérési érték típusát.
- Ezt az információt a fordító a függvényhívások paramétertípus és szám megfeleltetés ellenőrzésén túl konverziók elvégzésére is felhasználja.

A paraméterek konvertált típusa határozza meg azokat az aktuális paraméter értékeket, melyek másolatait a függvényhívás teszi ki a verembe. Gondoljuk csak meg, hogyha az **int**–ként kirakott aktuális paraméter értéket a függvény **double**–nek tekintené, akkor nem csak e paraméter félreértelmezéséről van szó, hanem az összes többi ezt követő is "elcsúszik"!

A prototípussal a fordító nem csak a visszatérési érték és a paraméterek típusegyezését tudja ellenőrizni, hanem az attribútumokat is. Például a **static** tárolási osztályú prototípus hatására a függvénydefiníciónak is ilyennek kell lennie.

📖 A függvénydefiníció módosítóinak egyeznie kell a függvénydeklarációk módosítóival!

A prototípusbeli azonosító hatásköre a prototípus. Prototípus adható változó számú paraméterre, ill. akkor is, ha paraméter egyáltalán nincs.

A komplett paraméterdeklarációk (**int a**) vegyíthetők az *absztrakt-deklarátor*okkal (**int**) ugyanabban a deklarációban. Például:

```
int add(int a, int);
```

A paraméter típusának deklarálásakor szükség lehet az adattípus nevének feltüntetésére, mely a *típusnév* segítségével érhető el. A *típusnév* az objektum olyan deklarációja, melyből hiányzik az azonosító. A metanyelvi leírás:

típusnév:
  típusspecifikátor-lista<*absztrakt-deklarátor*>
absztrakt-deklarátor:
  mutató
  <*mutató*><*direkt-absztrakt-deklarátor*>
direkt-absztrakt-deklarátor:
  (*absztrakt-deklarátor*)
  <*direkt-absztrakt-deklarátor*>[<*konstans-kifejezés*>]
  <*direkt-absztrakt-deklarátor*>(<*paraméter-típus-lista*>)

Az *absztrakt-deklarátor*ban mindig megállapítható az a hely, ahol az azonosítónak lennie kellene, ha a konstrukció deklaráción belüli deklarátor lenne. A következő típusnevek jelentése: **int**, 10 elemű **int** tömb és nem meghatározott elemszámú **int** tömb:

```
int, int [10], int []
```

✍ Lássuk be, hogy a függvény prototípus a kód dokumentálására is jó! Szinte rögtön tudunk mindent a függvényről:

```
void strcopy(char cel[], char forras[]);
```

A

<*típus*> fv(void);

olyan függvény prototípusa, melynek nincsenek paraméterei.

Normál esetben a függvény prototípus olyan függvényt deklarál, mely fix számú paramétert fogad. Lehetőség van azonban változó számú vagy típusú paraméter átadására is. Az ilyen függvény prototípus paraméterlistája ...-tal végződik:

<*típus*> fv(int, long, ...);

A fixen megadott paramétereket fordítási időben ellenőrzi a fordító, s a változó számú vagy típusú paramétert viszont a függvény hívásakor típusellenőrzés nélkül adja át a veremben.

🕮 Az **STGARG.H** fejfájlban vannak olyan makrók, melyek segítik a felhasználói, változó számú paraméteres függvények megalkotását! A témára visszatérünk még a **MUTATÓK** kapcsán!

Nézzünk néhány példa prototípust!

```
int f();          /* int-et visszaadó függvény, melynek
                     paramétereiről nincs információnk. */
```

```
int f1(void);      /* Olyan int-et szolgáltató függvény,
                      melynek nincsenek paraméterei. */
int f2(int, long); /* int-et visszaadó függvény, mely
                      elsőnek egy int, s aztán egy long
                      paramétert fogad. */
int pascal f3(void);/*Paraméter nélküli, int-et
                      szolgáltató pascal függvény. */
int printf(const char [], ...);/* int-tel visszatérő
                     függvény egy fix, és nem meghatározott
                     számú vagy típusú paraméterrel. */
```

Készítsünk programot, mely megállapítja az **ÉÉÉÉ.HH.NN** alakú karakterláncról, hogy érvényes dátum–e!

```
/* PELDA21.C: Dátumellenőrzés. */
#include <stdio.h>
#include <stdlib.h>  /* Az atoi miatt! */
#define INP 11       /* Az input puffer mérete. */
#include <ctype.h>   /* Az isdigit miatt! */
int getline(char [], int);
int datume(const char []);
void main(void){
  char s[INP+1];     /* Az input puffer. */
  printf("Dátumellenőrzés.\nBefejezés üres sorral!\n");
  while( printf("\nDátum (ÉÉÉÉ.HH.NN)? "),
         getline(s,INP)>0)
           if(datume(s)) printf("Érvényes!\n");
           else printf("Érvénytelen!\n"); }
```

Az **ÉÉÉÉ.HH.NN** alakú dátum érvényességét a kérdésre logikai értékű választ szolgáltató, **int datume**(**const char** *s*[]) függvény segítségével érdemes eldöntetni.

Követelmények:

- A karakterláncnak 10 hosszúságúnak kell lennie.
- A hónapot az évtől elválasztó karakter nem lehet numerikus, és azonosnak kell lennie a hónapot a naptól elválasztóval.
- A láncbeli összes többi karakter csak numerikus lehet.
- Csak 0001 és 9999 közötti évet fogadunk el.
- Az évszám alapján megállapítjuk a február hónap napszámát.
- A hónapszám csak 01 és 12 közötti lehet.
- A napszám megengedett értéke 01 és a hónapszám maximális napszáma között van.

```
int datume(const char s[]){
```

```
static int honap[ ] =
            { 0,31,28,31,30,31,30,31,31,30,31,30,31 };
int i, ho;
if(!s[10] && !isdigit(s[4]) && s[4]==s[7]){
  for(i=0; i<10; ++i){
    if(i==4||i==7) ++i;
    if(!isdigit(s[i])) return 0; }
  if((i=atoi(s))>=1){
    honap[2]=28+(!(i%4)&&i%100 || !(i%400));
    if((ho=10*(s[5]-'0')+s[6]-'0')>=1&&ho<=12&&
       (i=10*(s[8]-'0')+s[9]-'0')>=1&&
       i<=honap[ho]) return 1; } }
return 0; }
```

Megoldandó feladatok:

Változtasson úgy az **int datume**(**const char** *s*[]) függvényen, hogy akár egyjegyű is lehessen:

- az évszám, majd
- a hónap és a napszám is!

int indexe(**char** *s*[], **char** *t*[]) és **int indexu**(**char** *s*[], **char** *t*[]) függvények készítendők, melyek meghatározzák és visszaadják a *t* paraméter karakterlánc *s* karaktertömbbeli első, illetve utolsó előfordulásának indexét! Próbálja is ki egy rövid tesztprogrammal a függvényeket!

Írjon olyan szoftvert, mely megkeresi egy próba karakterlánc összes előfordulását a szabvány bemenetről érkező sorokban!

8.2.3 Függvények hívása és paraméterkonverziók

A függvényt aktuális paraméterekkel hívjuk meg. Ezek sorrendjét és típusát a formális paraméterek határozzák meg. A függvényhívás operátor alakja

utótag-kifejezés(<*kifejezéslista*>)

kifejezéslista:
hozzárendelés-kifejezés
kifejezéslista, *hozzárendelés-kifejezés*

, ahol az *utótag-kifejezés* egy függvény neve, vagy függvénycímmé értékeli ki a fordító, s ezt hívja meg. A zárójelben álló, elhagyható *kifejezéslista* tagjait egymástól vessző választja el, és tudjuk, hogy ezek azok az aktuális paraméter kifejezések, melyek értékmásolatait a hívott függvény kapja meg.

- Ha az *utótag-kifejezés* nem deklarált azonosító az aktuális hatáskörben, akkor a fordító implicit módon a függvényhívás blokkjában

  extern int *azonosító*();

módon tekinti deklaráltnak.

A függvényhívás kifejezés értéke és típusa a függvény visszatérési értéke és típusa. Az értéket vissza nem adó függvényt **void–**nak kell deklarálni, ill. **void** írandó a *kifejezéslista* helyére, ha a függvénynek nincs paramétere.

- Ha a prototípus paraméterlistája **void**, akkor a fordító zérus paramétert vár mind a függvényhívásban, mind a definícióban. E szabály megsértése hibaüzenethez vezet.

- Az aktuális paraméter *kifejezéslista* kiértékelési sorrendje nem meghatározott, pontosabban a konkrét fordítótól függ. A más paraméter mellékhatásától függő paraméter értéke így ugyancsak definiálatlan. A függvényhívás operátor egyedül azt garantálja, hogy a fordító a paraméterlista minden mellékhatását realizálja, mielőtt a vezérlést a függvényre adná.

- Függvénynek tömb és függvény nem adható át paraméterként, de ezekre mutató mutató persze igen.

A paraméter lehet aritmetikai típusú. Lehet struktúra, unió vagy mutató is, de ezekkel későbbi szakaszokban foglalkozunk. A paraméter átadása érték szerinti, azaz a függvény az értékmásolatot kapja meg, melyet természetesen el is ronthat a hívás helyén levő eredeti értékre gyakorolt bármiféle hatás nélkül. Szóval a függvény módosíthatja a formális paraméterek értékét.

A fordító kiértékeli a függvényhívás *kifejezéslistá*ját, és szokásos konverziót (egész–előléptetést) hajt végre minden aktuális paraméteren. Ez azt jelenti, hogy a **float** értékből **double** lesz, a **char** és a **short** értékből **int**, valamint az **unsigned char** és az **unsigned short** értékből **unsigned int** válik.

Ha van vonatkozó deklaráció a függvényhívás előtt, de nincs benne információ a paraméterekre, akkor a fordító kész az aktuális paraméterek értékével.

Ha deklaráltak előzetesen függvény prototípust, akkor az eredmény aktuális paraméter típusát hasonlítja a fordító a prototípusbeli megfelelő paraméter típusával. Ha nem egyeznek, akkor a deklarált formális paraméter típusára alakítja az aktuális paraméter értékét hozzárendelési kon-

verzióval, és újra a szokásos konverzió következik. A nem egyezés másik lehetséges végkifejlete diagnosztikai üzenet.

A hívásnál a *kifejezéslista*beli paraméterek számának egyeznie kell a függvény prototípus vagy definíció paramétereinek számával. Kivétel az, ha a prototípus ,...-tal végződik, amikor is a fordító a fix paramétereket az előző pontban ismertetett módon kezeli, s a ,... helyén levő aktuális paramétereket úgy manipulálja, mintha nem deklaráltak volna függvény prototípust.

8.2.4 Nem szabványos módosítók, hívási konvenció

A deklaráció deklarátorlistájában a megismert szabványos alaptípusokon, típusmódosítókon kívül minden fordítóprogram rendelkezik még speciális célokat szolgáló, a deklarált objektum tulajdonságait változtató, nem szabványos módosítókkal is.

✍ Az olvasónak javasoljuk, hogy nézzen utána ezeknek a programfejlesztő rendszere segítségében!

Teljességre való törekvés nélkül felsorolunk itt néhány ilyen módosítót, melyek közül egyik–másik ki is zárja egymást!

módosító:
cdecl
pascal
interrupt
fastcall
stdcall
export
near
far
huge

Az első néhány módosító a függvény hívási konvencióját határozza meg. Az alapértelmezett hívási konvenció C programokra **cdecl**.

✍ Tekintsünk csak bele bármelyik szabványos fejfájlba, mindegyik függvény prototípusa elején ott találjuk a **cdecl** módosítót!

Ha egy azonosító esetében biztosítani kívánjuk a kis-nagybetű érzékenységet, az aláhúzás karakter (_) név elé kerülését, ill. függvénynévnél a paraméterek jobbról balra való verembe rakását, akkor az azonosító deklarációjában írjuk ki expliciten a **cdecl** módosítót! Ez a hívási konvenció biztosítja az igazi változó paraméteres függvények írását, hisz a vermet a hívó függvénynek kell rendbe tennie.

✍ Változó paraméteres függvények írásával majd a **MUTATÓK** kapcsán foglalkozunk, most vegyünk egy példát az elmondottakra!

```
void fv(short s, int i, double d){}
void main(void){
  static short s=5;
  static int i=7;
  static float f=3.14f;
  /* . . . */
  fv(s, i, f);
  /* . . . */ }
```

Hívás	A hívott függvény	Verem helyreállítás
A paraméter értékeket jobbról balra haladva rakja ki a verembe a kód a konverziók elvégzése után, majd meghívja az **fv**–t. A veremmutató **SP**! **SP+12:** 3.14 (**double**) **SP+8:** 7 (**int**) **SP+4:** 5 (**int**) **SP:** visszatérési cím	Hozzáfér az értékmásolatokhoz a veremben, de közben az **SP** nem változik meg. Elvégzi a dolgát a függvény, és visszatér a hívó függvénybe. **SP+8:** 3.14 (**double**) **SP+4:** 7 (**int**) **SP:** 5 (**int**)	A hívó függvényben 16–tal csökkenti a kód az **SP** értékét. **SP:** ✍ Változó számú paraméter átadása könnyen lehetséges, hiszen ha még a hívás helyén sem ismerjük az aktuális paraméterek számát, akkor nem fogjuk megtudni sohasem.

Az **stdcall** kis–nagybetű érzékeny. Standard hívási konvencióval kell hívni a WIN32 API függvények többségét. A **fastcall** módosító azt jelenti, hogy a fordító regiszterekben igyekszik átadni a paramétereket a hívott függvénynek a verembe rakás sorrendjében. Ha nincs elég regiszter, akkor a maradékot vermen át juttatja el a hívóhoz. A visszatérési érték átadása ugyancsak regiszteren át történik, ha lehetséges stb.

A **cdecl** nagyobb végrehajtható kódot generál, mint a **fastcall**, vagy az **stdcall,** hisz a hívásit, vermet rendbe tevő kódnak kell követnie a hívó függvényben. Már mondottuk, hogy **cdecl** az alapértelmezett hívási konvenció C programokra. A **main**–nek mindenképp **cdecl**-nek kell lennie, mert az indító kód C hívási konvenciók szerint hívja meg.

✍ A legtöbb programfejlesztő rendszerben beállítható az általánosan használandó hívási konvenció, mely mindig felülbírálható a kívánt módosító explicit kiírásával.

Ha például nem **cdecl** hívási sorrendű programban szeretnénk használni a **printf** függvényt, akkor:

```
extern cdecl printf(const char format[], ...);
void egeszekki(int i, int j, int k);
void cdecl main(void){ egeszekki( 1, 4, 9); }
void egeszekki(int i, int j, int k){
  printf("%d  %d  %d\n", i, j, k); }
```

módon kell dolgozni. Elismerjük, hogy #**include** <**stdio.h**>-t alkalmazva semmi szükség sincs az

```
extern cdecl printf(const char format[], ...);
```

kiírására, hisz ez a prototípus amúgy is benne van az **STDIO.H** fejfájlban.

8.2.5 Rekurzív függvényhívás

Bármely függvény meghívhatja önmagát közvetlenül vagy közvetve. A rekurzív függvényhívásoknak egyedül a verem mérete szab határt.

☞ A verem méretét befolyásoló, például kapcsoló-szerkesztő opció, beállítást megtudhatjuk programfejlesztő rendszerünk segítségéből.

Valahányszor meghívják a függvényt, új tároló területet allokál a rendszer az aktuális paramétereknek, az **auto** és a nem regiszterben tárolt **register** változóknak. A paraméterek és a lokális változók tehát a veremben jönnek létre a függvénybe való belépéskor, és megszűnnek, mihelyt a vezérlés távozik a függvényből, azaz:

- valahányszor meghívjuk a függvényt, saját lokális változó és aktuális paraméter másolatokkal rendelkezik, ami
- biztosítja, hogy a függvény „baj nélkül" meghívhatja önmagát közvetlenül vagy közvetetten (más függvényeken át).

☞ A rekurzívan hívott függvények tulajdonképpen dolgozhatnak dinamikusan kezelt, globális vagy **static** tárolási osztályú lokális változókkal is. Azt azonban ilyenkor ne felejtsük el, hogy a függvény összes híváspéldánya ugyanazt a változót éri el.

Ha az lenne a feladatunk, hogy írjunk egy olyan függvényt, mely meghatározza **n** faktoriálisát, akkor valószínűleg így járnánk el:

```
long double faktor(int n){
  long double N = n<1? 1.L: n;
  while(--n) N*=n;
  return N; }
```

✍ Látható, hogy a **long double** ábrázolási formát választottuk, hogy a lehető legnagyobb szám faktoriálisát legyünk képesek meghatározni a C számábrázolási lehetőségeivel. Az algoritmus az egynél kisebb egészek faktoriálisát egynek tekinti, és az ismételt szorzást fordított sorrendben hajtja végre, vagyis: **N**=**n***(**n**–1)*(**n**–2)*…*3*2*1.

Írjunk egy rövid keretprogramot, mely bekéri azt az 1 és **MAXN** (fordítási időben változtatható) közti egész számot, melynek megállapíttatjuk a faktoriálisát!

```
/* PELDA22.C: Rekurziós példaprogram: faktoriális. */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 40            /* Az input puffer mérete. */
#define MAXN 40           /* A max. szám. */
long double faktor(int n);
int getline(char s[],int n);
/* A decimális számjegyek száma maximálisan: */
#define HSZ sizeof(int)/sizeof(short)*5
int egesze(char s[]);
void main(void){
  int n=0;                /* A szám. */
  char sor[MAX+1];        /* A bemeneti puffer. */
  printf("\n\t\tEgész szám faktoriálisa.\n");
  while(n<1||n>MAXN){
    printf("\nMelyik egész faktoriálisát számítjuk”
           ”(1-%d)? ",MAXN);
    getline(sor,MAX);
    if(egesze(sor)) n=atoi(sor); }
  printf("%d! = %-20.0Lf \n",n, faktor(n)); }
```

✍ Ismeretes, hogyha a formátumspecifikációban előírjuk a mezőszélességet, akkor a kijelzés alapértelmezés szerint jobbra igazított. A balra igazítás a szélesség elé írt – jellel érhető el.

A faktoriális–számítás rekurzív megoldása a következő lehetne:

```
long double faktor(int n){
  if(n <= 1) return 1.L;
  else return (n*faktor(n-1)); }
```

✍ Látható, hogy a **faktor** egynél nem nagyobb **n** paraméter esetén azonnal **long double** 1–gyel tér vissza. Más esetben viszont meghívja önmagát **n**–nél eggyel kisebb paraméterrel. Az itt visszakapott értéket megszorozza **n**–nel, és ez lesz a majdani visszaadott értéke. Nézzük meg asztali teszttel, hogyan történnek a hívások, feltéve, hogy a **main** a 0–s hívási szint **faktor**(5)–tel indult!

n:	5	4	3	2	1
Hívási szint:	1 →	2 →	3 →	4 →	5 ↓
Hívás:	**faktor**(4)	**faktor**(3)	**faktor**(2)	**faktor**(1)	↓
Visszatérési szint:	0	← 1	← 2	← 3	← 4
Visszatérési érték:	120	24	6	2	1

☞ Cseréljük ki a keretprogramban a **faktor** függvényt a rekurzív változatra, és próbáljuk ki!

Megoldandó feladatok:

Írja meg a következő függvények rekurzív változatai:

- **void strrv**(**char** *s*[]), mely megfordítja a saját helyén a paraméter karakterláncot.
- Az **itoa** függvény, mely az **int** paraméterét karakterlánccá konvertálja, és elhelyezi az ugyancsak paraméter karaktertömbben.

9 MUTATÓK

A nyelvben a mutatóknak két fajtája van:

- (adat) objektumra mutató mutatók és
- függvényre (kód) mutató mutatók.

Bármilyenek is legyenek, a mutatók memória címek tárolására szolgálnak. Előbb az adatmutatókkal fogunk foglalkozni. A mutató mindaddig, míg erről külön nem szólunk, jelentsen adatmutatót!

A *típus* típusú mutató *típus* típusú objektum címét tartalmazhatja. A mutatók **unsigned** egészek, de saját szabályaik és korlátozásaik vannak a hozzárendelésre, a konverzióra és az aritmetikára.

Miután a mutató is skalár objektum, létezik a mutatóra mutató mutató is. A mutatók azonban többnyire más skalár vagy **void** objektumokra (változókra), ill. aggregátumokra (tömbökre, struktúrákra és uniókra) mutatnak.

9.1 Mutatódeklarációk

A mutatódeklaráció elnevezi a mutató változót, és rögzíti annak az objektumnak a típusát, melyre ez a változó mutathat.

Tehát csak egy bizonyos *típus*ra (előredefiniáltra - beleértve a **void**-ot is - vagy felhasználó definiáltára) mutató mutató deklarálható

*típus *azonosító;*

módon, amikor is az *azonosító* nevű mutató *típus* típusú objektum címét veheti fel.

✍ Vegyük észre, hogy *típus* * az *azonosító* típusa! Például az

```
int *imut;
int *fv(char *);
```

deklarációk alapján: **imut int** típusú objektumra mutató mutató, **fv int** típusú címet visszaadó, és egy **char** objektumra mutató paramétert fogadó függvény.

Ha a definiált mutató statikus élettartamú, akkor tiszta zérus kezdőértéket kap implicit módon. A fordító tiszta zérus címen nem helyez el sem objektumot, sem függvényt, s így a zérus cím (az ún. **NULL** mutató) speciális felhasználású. Magát a **NULL** mutatót a szabványos fejfájlokban (például az **STDIO.H**–ban) definiálja a nyelv.

Bármilyen típusú mutató **NULL**-hoz hasonlítása mindenkor helyes eredményre vezet, ill. bármilyen típusú mutatóhoz hozzárendelhetjük a **NULL** mutatót.

Ha lokális a definiált mutató, akkor a definíció hatására a fordító akkora memóriaterületet foglal, hogy abban egy cím elférjen, de a lefoglalt bájtok „szemetet” tartalmaznak.

💣 Az ilyen mutató tehát nem használható mindaddig „értelmesen” semmire, míg valamilyen módon érvényes címet nem teszünk bele.

Hogy lehet elérni valamilyen objektum címét?

9.1.1 Cím operátor (&)

Egyoperandusos, magas prioritású művelet, mely definíció szerint:

& előtag-kifejezés

alakú. Az *előtag-kifejezés* operandusnak vagy függvényt kell kijelölnie, vagy olyan objektumot elérő balértéknek kell lennie, mely nem **register** tárolási osztályú, és nem bitmező. Lehet tehát például változó, vagy tömbelem.

✍ A bitmezőkkel a struktúrák kapcsán később foglalkozunk!

Ha az operandus típusa *típus*, akkor az eredmény mutató a *típus* típusra.

Cím csak mutatónak adható át. A lehetséges módszerek szerint vagy hozzárendeljük, vagy inicializátort alkalmazunk a deklarációban. Például:

```
típus val1, val2, *ptr = &val1;     /* A ptr mutatónak van
    kezdőértéke, de a vele elért objektumnak nincs. */
ptr = &val2;         /* Hozzárendeljük a másik változó
    címét a mutatóhoz. */
```

Ha

```
T1 *ptr1; T2 *ptr2;
```

különböző típusú objektumokra mutató mutatók, akkor a

```
ptr1 = ptr2;
```

vagy a

```
ptr2 = ptr1;
```

hozzárendelés figyelmeztető vagy hibaüzenetet okoz. Explicit típusmódosító szerkezetet alkalmazva viszont „gond nélkül” mehet a dolog:

```
ptr1 = (T1 *) ptr2;
ptr2 = (T2 *) ptr1;
```

💣 Teljesen illegális dolog azonban a függvény és az adatmutatók összerendelése!

Ha a mutató már érvényes címet tartalmaz, az

9.1.2 Indirekció operátor (*)

segítségével elérhetjük a mutatott értéket. Az indirekció operátor ugyancsak egyoperandusos, magas prioritású művelet, melynek definíciója:

** előtag-kifejezés*

Az *előtag-kifejezés* operandusnak *típus* típusra mutató mutatónak kell lennie, ahol a *típus* bármilyen lehet. Az indirekció eredménye az *előtag-kifejezés* mutatta címen levő, *típus* típusú érték. Az indirekció eredménye egyben balérték is. Például:

```
típus t1, t2;
/* . . . */
típus *ptr = &t1;  /* A mutatót inicializáltuk is. */
*ptr = t2;         /* Ekvivalens a t1 = t2-vel. */
```

💣 Ne kíséreljük meg azonban a cím operátorral kifejezés vagy konstans címét előállítani, vagy indirekció operátort nem címmé kiértékelhető operandus elé odaírni,

```
ptr = &(t1 + 6);   /* HIBÁS */
ptr = &8;          /* HIBÁS */
t2 = *t1;          /* HIBÁS */
```

mert hibaüzenethez jutunk!

Meghatározatlan az indirekció eredménye akkor is, ha a mutató:

- **NULL** mutató,
- vagy olyan lokális objektum címét tartalmazza, mely a program adott pontján nem látható,
- vagy a végrehajtható program által nem használható címet tartalmaz.

👍 A mutatókkal végzett munka során „ököl-szabályunk” szerint: ahol objektum lehet egy kifejezésben, ott kerek zárójelbe téve az indirekció műveletét követően ilyen típusú objektumra mutató mutató is állhat. Ha:

```
típus val, *ptr = &val;
```

, akkor ahol **val** szerepelhet egy kifejezésben, ott állhat (***ptr**) is.

Például:

```
int y, val=0, *ptr = &val;
/* . . . */
printf("val címe (%p) van ptr-ben (%p).\n", &val, ptr);
y = *ptr + 3;          /* y = val +3 tulajdonképpen. */
y = sqrt((double) ptr);/* y = sqrt(val). */
*ptr += 6;             /* val += 6. */
(*ptr)++;              /* Zárójel nélkül a ptr-t
                          inkrementáltatnánk. */
printf("val értéke %d.\n", val);
printf("A ptr címen levő érték %d.\n", *ptr); }
```

Fedezzük fel, hogy a cím, vagy mutatótartalom kijelzéséhez használatos típuskarakter a **p** a formátumspecifikációban, és hogy a **printf** paraméterlistájában cím értékű kifejezés is állhat, persze akár indirekcióval is!

Vigyázzunk nagyon! Ha egy változónak nem adunk kezdőértéket, és mondjuk, hozzáadogatjuk egy tömb elemeit, akkor a végeredmény „zöldség”, s a dolog szarvashiba. Ha mutatónak nem adunk kezdőértéket, és a benne levő „szemétre”, mint címre, írunk ki értéket indirekcióval, akkor az duplán szarvashiba. A „szeméttel”, mint címmel valahol „pancsolunk” a memóriában, és felülírjuk valami egészen más objektum értékét, s a hiba is egészen más helyen jelentkezik, mint ahol elkövettük.

9.1.3 void mutató

Külön kell említenünk a

```
void *vptr;
```

mutató típust, ami nem semmire, hanem meghatározatlan típusra mutató mutató. Explicit típusmódosító szerkezet alkalmazása nélkül bármilyen típusú mutató vagy cím hozzárendelhető a **void** mutatóhoz. A dolog megfordítva is igaz, azaz bármilyen típusú mutatóhoz hozzárendelhetünk **void** mutatót. Például:

```
típus ertek, *ptr=&ertek;
vptr=ptr;       /* OK */
vptr=&ertek;    /* OK */
ptr=vptr;       /* OK */
```

void mutatóval egyetlen művelet nem végezhető csak: az indirekció, hisz meghatározatlan a típus, és a fordító nem tudja, hogy hány bájtot és milyen értelmezésben kell elérnie.

```
ertek=*vptr;    /* HIBÁS */
```

9.1.4 Statikus és lokális címek

Miután a statikus élettartamú objektum minden bitjét zérusra inicializálja alapértelmezés szerint a fordító, de címe nem változik, kezdőértéke lehet akár statikus mutatónak is. Az **auto** változók címe viszont nem lehet statikus inicializátor, hisz a cím más–más lehet a blokk különböző végrehajtásakor.

```
int GLOBALIS;
int fv(void){
  int LOKALIS;
  static int *slp = &LOKALIS;  /* HIBÁS. */
  static int *sgp = &GLOBALIS; /* OK. */
  register int *rlp = &LOKALIS;/* OK. */
  long a = 10000001, *lp = &a;/* a kezdőértéke
           1000000 és lp kezdetben rá mutat. */
  register int *pi = 0;        /* pi regiszteres mutató
           értéke NULL. */
  const int i = 26;            /* Ez az egyetlen hely, ahol
           i értéket kaphatott. */
  /* . . . */ }
```

9.1.5 Mutatódeklarátorok

deklarátor:
  <*mutató*>*direkt-deklarátor*

direkt-deklarátor:
  azonosító
  (*deklarátor*)
  direkt-deklarátor [<*konstans-kifejezés*>]
  direkt-deklarátor (*paraméter-típus-lista*)
  direkt-deklarátor (<*azonosítólista*>)

mutató:
  *<*típusmódosító-lista*>
  *<*típusmódosító-lista*>*mutató*

típusmódosító-lista:
  típusmódosító
  típusmódosító-lista típusmódosító

típusmódosító: (a következők egyike!)
  const
  volatile

A *deklarátor*ok szerkezetileg az indirekcióhoz, a függvényhez és a tömbkifejezéshez hasonlítanak, s csoportosításuk is azonos. A *deklarátor*t követheti egyenlőségjel után *inicializátor*, de mindig *deklaráció-specifikátorok* előzik meg. A *deklaráció-specifikátorok tárolási-osztály-specifikátor* és *típusspecifikátor* sorozat tulajdonképp, és igazából nem csak egyetlen *deklarátor*ra vonatkoznak, hanem ezek listájára.

✍ Megkérjük az olvasót, hogy lapozzon vissza egy pillanatra a **TÍPUSOK ÉS KONSTANSOK Deklaráció** fejezetéhez!

A *direkt-deklarátor* definíció első lehetősége szerint a *deklarátor* egy egyedi *azonosító*t deklarál, melyre a tárolási osztály egy az egyben vonatkozik, de a típus értelmezése kicsit függhet a *deklarátor* alakjától is. A *deklarátor* tehát egyedi *azonosító*t határoz meg, s mikor az *azonosító* feltűnik egy tőle típusban nem eltérő kifejezésben, akkor a vele elnevezett objektum értékét eredményezi.

Összesítve, és csak a lényeget tekintve a *deklaráció*

típus deklarátor

alakú. Ezt nem változtatja meg az sem, ha a *direkt-deklarátor* második alternatíváját tekintjük, mert a zárójelezés nem módosítja a típust, csak összetettebb *deklarátor*ok kötésére lehet hatással.

✍ A függvénydeklarátorokat már tárgyaltuk, s a tömbdeklarátorokra még ebben a szakaszban visszatérünk!

A mutatódeklaráció így módosul:

típus * *<típusmódosító-lista> deklarátor*

, ahol a * *<típusmódosító-listá>*–val változtatott *típus* a *deklarátor* típusa. A * operátor után álló típusmódosító magára a mutatóra, és nem a vele megcímezhető objektumra vonatkozik.

✍ Foglalkozzunk például a **const** módosítóval!

9.1.6 Konstans mutató

Mind a mutató, mind a mutatott objektum deklarálható **const**-nak. Bármely **const**-nak deklarált „valami” ugyebár nem változtathatja meg az értékét. Az sem mehet persze, hogy olyan mutatót kreáljunk, mellyel megsérthetnénk a **const** objektum érték megváltoztathatatlanságát.

```
int i;
int *pi;             /* pi int objektumra mutató
                        inicializálatlan mutató. */
int * const cp = &i; /* cp konstans mutató int-re, de
                        amire mutat, az nem konstans. */
const int ci = 7;    /* ci 7 értékű konstans int. */
const int *pci;      /* pci konstans int-re mutat. A
                        mutató tehát nem konstans. */
const int * const cpc = &ci;  /* cpc konstans int-re
                        mutató konstans mutató. */
```

A következő hozzárendelések legálisak:

```
i = ci;  /* const int hozzárendelése int-hez. */
*cp = ci;/* const int hozzárendelése konstans mutató
            mutatta nem konstans int-hez. */
++pci;   /* const int-re mutató mutató inkrementálása.*/
pci = cpc;/*Konstans int-re mutató, konstans mutató hoz-
            zárendelése const int-re mutató mutatóhoz.*/
```

A következő hozzárendelések illegálisak:

```
ci = 0;  /* Értékhozzárendelési kísérlet const int-hez.*/
ci--;    /* const int dekrementálási kísérlete. */
*pci = 3;/* Értékadási kísérlet a const int-re mutató
            mutatóval megcímzett objektumnak. */
cp = &ci;/* Értékadási kísérlet konstans mutatónak. */
cpc++;   /* Konstans mutató inkrementálási kísérlete. */
pi = pci;/* Ha ez a hozzárendelés legális lenne, akkor
            *pi = ... módon módosíthatnánk azt a const
            int-et, amire pci mutat. */
```

Ahhoz, hogy a „kígyó megharapja a farkát” kell, hogy **const** objektumra mutató mutatót ne lehessen hozzárendelni nem **const**-ra mutató mutatóhoz. Ha ez menne, akkor ugyan „kerülő úton”, de a mutatott **const** érték megváltoztatható lenne.

9.2 Mutatók és függvényparaméterek

Eddig kiemelten csak az ún. érték szerinti paraméter átadással foglalkoztunk a függvényhívás kapcsán. Ezt úgy interpretálhatjuk, hogy a fordító függvényhíváskor az aktuális paraméterek (esetleg típuskonverzión átesett) értékét helyezi el, például a veremben, s a meghívott függvény nem az aktuális paraméterek értékéhez, hanem annak csak egy másolatához fér hozzá.

Rövidsége és találósága miatt átvesszük [4] vonatkozó mintapéldáját!

Tegyük fel, hogy a programozó azt a feladatot kapta, hogy írjon olyan függvényt, mely megcseréli két, **int** paramétere értékét!

```
csere(paraméter1, paraméter2);
```

módon hívható első kísérlete a következő volt:

```
void csere(int x, int y){
  int seged = x;
  x = y;
  y = seged; }
```

Barátunk próbálkozása „professzionális” olyan értelemben, hogy gondolt arra, hogy a csere végrehajtásához szüksége van segédváltozóra, de a

függvényt hívó programjában „meglepetten” tapasztalta, hogy semmiféle értékcsere nem történt.

☞ A csere tulajdonképpen lezajlott az aktuális paraméterek másolatain a veremben, de ennek semmilyen hatása sincs az aktuális paraméterek hívó programbeli értékeire. A függvény visszatérése miatt a veremmutató is visszaállt a hívás előtti értékére, s így a verembeli, felcserélt értékmásolatok is elérhetetlenné váltak.

A megoldás a cím szerinti paraméter átadásban rejlik, azaz a **csere** függvényt

```
csere(&paraméter1, &paraméter2);
```

módon kell meghívni, s a függvénydefiníció pedig így módosul:

```
void csere(int *x, int *y){
  int seged = *x;
  *x = *y;
  *y = seged; }
```

☞ A rutin most az aktuális paraméterek címmásolatait kapja meg a veremben, s az indirekció műveletét alkalmazva így az aktuális paraméterek értékén dolgozik.

Megoldandó feladatok:

Írja át a következő, formai érvényességet ellenőrző függvényeket úgy, hogy igaz (1) visszatérési érték mellett a vizsgált karakterlánc konvertált eredményét is szolgáltassák! Ha az érvényességellenőrzés viszont hibával zárul, akkor a visszaadott hamis (0) értéken kívül semmiféle értékváltozást nem okozhat a függvény.

- A **PELDA18.C**–ben definiált **egesze** legyen **int egesze**(**const char** *s*[], **int** **ertek*) prototípusú!
- A **PELDA20.C**–beli **lebege** átalakítandó **int lebege**(**const char** *s*[], **double** **ertek*)–ké!
- A **PELDA21.C datume** függvényéből váljék **int datume**(**const char** *s*[], **int** **ev*, **int** **ho*, **int** **nap*)!

9.3 Tömbök és mutatók

☞ A **BEVEZETÉS ÉS ALAPISMERETEK** szakaszban külön fejezet foglalkozik a tömbökkel, és az **Inicializálás** rész tárgyalja az egydimenziós tömbök kezdőérték adását is.

Tömb létesíthető aritmetikai típusokból, de definiálható

- mutatóból (külön későbbi fejezet),
- struktúrából és unióból (a következő szakasz), valamint
- tömbből (önálló fejezete van a többdimenziós tömböknek).

💣 Bármilyen típusból is hozzuk azonban létre a(z egydimenziós) tömböt, a típusnak teljesnek kell lennie. Nem lehet félig kész, nem teljesen definiált, felhasználói típusból tömböt kreálni.

🕮 A tömbök és a mutatók között nagyon szoros kapcsolat van. Ha kifejezés, vagy annak része *típus* tömbje, akkor a (rész)kifejezés értékét a tömb első elemét megcímző, konstans mutatóvá alakítja a fordító, és a (rész)kifejezés típusa *típus* * **const** lesz. Nem hajtja végre a fordító ezt a konverziót, ha a (rész)kifejezés cím operátor (&), ++, −−, vagy a pont (.) szelekciós operátor, vagy a **sizeof** operandusa, vagy hozzárendelési művelet bal oldalán áll.

🔗 A . operátorral a következő szakasz foglalkozik. Elemezzük az előző bekezdésben mondottakat egy példa tükrében!

Legyen a következő tömb és mutató!

```
#define MERET 20   /* Tömbméret. */
/* . . . */
float tomb[MERET], *pt;
/* . . . */
```

A tömböt a fordító, mondjuk, a 100–as címtől kezdve helyezte el a memóriában. Egy tömbelem helyfoglalása **sizeof**(**tomb**[0]) ≡ **sizeof**(**float**) ≡ 4, általánosságban **sizeof**(*típus*). Az elemek

```
tomb[0], tomb[1], ..., tomb[MERET - 1]
```

sorrendben, növekvő címeken helyezkednek el a tárban. Tehát a **tomb**[0] a 100–as, a **tomb**[1] a 104–es, a **tomb**[2] a 108–as és így tovább címen van. A 180–as memóriacím már nem tartozik a tömbhöz.

Ha valamilyen kifejezésben meglátja a fordító a **tomb** azonosítót, akkor rögtön helyettesíteni fogja a **float * const** 100–as címmel. Tehát, ha a **pt**–t fel kívánjuk tölteni **tomb** kezdőcímével, akkor nem kell ilyen

```
pt = &tomb[0];
```

hosszadalmasan kódolni, tökéletesen elég a

```
pt = tomb;
```

9.3.1 Index operátor

A **MŰVELETEK ÉS KIFEJEZÉSEK** szakasz elején definiált *utótag-kifejezés* második alternatívája az

utótag-kifejezés[*kifejezés*]

az indexelő operátor. Nevezik ezt indexes változónak is, vagyis mindenképpen hivatkozás ez a tömb egy meghatározott elemére.

📖 A szabályok szerint az *utótag-kifejezés* és a *kifejezés* közül az egyiknek mutatónak, a másiknak egész típusúnak kell lennie, és hatásukra a fordító a

*((*utótag-kifejezés*) + (*kifejezés*))

műveletet valósítja meg. A fordító alkalmazza a következő fejezetben tárgyalt konverziós szabályokat a + műveletre és a tömbre. Ha az *utótag-kifejezés* a tömb és a *kifejezés* az egész típusú, akkor a konstrukció a tömb *kifejezés*edik elemére hivatkozik.

Mi van a típussal? A külső zárójel párban még mutató típus (*típus* *) van, s ezen hajtja végre a fordító az indirekciót. Tehát az eredmény típusát a mutató dönti el.

Vegyük a **tomb**[6] indexes változót! A mondottak szerint ebből

```
*((tomb) + (6))
```

lesz, ami nem 6, hanem 6–nak, mint indexnek, a hozzáadását jelenti a **tomb** kezdőcíméhez, azaz:

```
*((float *)100 + 6*sizeof(float))
```

Az összeadás elvégzése után

```
*((float *)124)
```

, ami az indirekció végrehajtása után a **tomb** 6–os indexű elemét eredményezi.

✍ A + kommutatív művelet, így az indexelés is az. Egydimenziós tömbökre a következő négy kifejezés teljesen ekvivalens feltéve, hogy **p** mutató és **i** egész:

```
p[i] ≡ i[p] ≡ *(p + i) ≡ *(i + p)
```

Folytassuk az index operátor ismertetésének megkezdése előtt elkezdett gondolatmenetet, azaz a **pt** mutató legyen **tomb** értékű!

```
pt = tomb;
```

Ilyenkor:

```
*pt ≡ tomb[0],
*(pt + 1) ≡ tomb[1]
```

és általában

```
*(pt + i) ≡ tomb[i].
```

✍ Vegyük észre, hogy a **pt**-re szükség sincs a tömbelemek és címeik előállításához, hisz:

```
tomb[i] ≡ *(tomb + i)
```

és

```
&tomb[i] ≡ tomb + i.
```

💣 Fontos különbség van azonban a **tomb** és a **pt** között. A **pt** mutató változó. A **tomb** pedig mutató konstans. Ebből következőleg nem megengedettek a következők:

```
tomb = pt;    /* Mintha 3=i-t írtunk volna fel. */
tomb++;       /* Mint 3++. */
pt = &tomb;   /* Mintha &3-at akarnánk előállítani. */
```

A mutató változóra természetesen megengedettek ezek a műveletek:

```
pt = tomb;
pt++;
```

Ha **pt** mutató, akkor azt kifejezés indexelheti a tanultak szerint, azaz

```
pt[i] ≡ *(pt + i)
```

✍ Meg kell említeni még, hogy amikor egy függvényt tömbazonosító aktuális paraméterrel hívtunk meg, akkor is cím szerinti paraméter átadás történt, azaz a függvény a tömb kezdőcím konstans másolatát kapta meg, például, a veremben. Ez a címmásolat aztán persze a függvényben már nem konstans, el is lehet rontani stb.

✍ Vegyük észre, hogy a cím szerinti paraméter átadást szinte minden példánkban, már a kezdetek óta használjuk! Eddig a függvényparaméter tömböt mindig

típus azonosító[]

alakban adtuk meg, de legújabb ismereteink szerint a

típus * *azonosító*

forma használandó, hisz ez egyértelműen mutatja, hogy a paraméter mutató. A függvény testén belül ettől függetlenül szabadon dönthet a programozó, hogy

- tömbként,

- mutatóként, vagy
- vegyesen

kezeli az ilyen paramétert.

☝ A gépi kódú utasítások operandusai többnyire memória címek. Ebből következőleg minél inkább mutatókat használva írjuk meg programjainkat, annál közelebb kerülünk a gépi kódhoz, s ez által annál gyorsabb lesz a szoftverünk.

Írjuk át ennek szellemében a **PELDA21.C datume** függvényét!

```
int datume(const char *s){
  static int honap[ ] =
              { 0,31,28,31,30,31,30,31,31,30,31,30,31 };
  int i, ho;
  if(!*(s+10) && !isdigit(*(s+4)) && *(s+4)==*(s+7)){
    for(i=0; i<10; ++i){
      if(i==4||i==7) ++i;
      if(!isdigit(*(s+i))) return 0; }
    if((i=atoi(s))>=1){
      *(honap+2)=28+(!(i%4)&&i%100 || !(i%400));
      if((ho=atoi(&s[5])>=1&&ho<=12&&(i=atoi(&s[8])>=1&&
        i<=*(honap+ho)) return 1; } }
  return 0; }
```

✍ Lássuk be, hogy az ilyen fajta függvény átírásnak, amikor **s**[**i**]–ből ***(s+i)**–t csinálgatunk, semmi értelme sincs, hisz ezt a fordító magától is megteszi. Az átalakítás egyetlen előnye, hogy a formális paraméter jobban szemlélteti, hogy **const** karakterláncra mutat, ill. az észre vehető, hogy az **atoi** függvényt nem csak a karaktertömb kezdetével szabad meghívni.

Foglalkozzunk még egy kicsit a tömbdeklarátorokkal!

9.3.2 Tömbdeklarátor és nem teljes típusú tömb

A **Mutatódeklarátorok** fejezetben a *direkt-deklarátor* definíció harmadik változata

direkt-deklarátor [<*konstans-kifejezés*>]

a tömbdeklarátor. A tömbdeklaráció tehát

típus deklarátor [<*konstans-kifejezés*>]<={*inicializátorlista*<,>}>

, ahol az elhagyható *konstans-kifejezés*nek egész típusúnak és zérusnál nagyobb értékűnek kell lennie, s ez a tömb mérete. Ez a **TÍPUSOK ÉS KONSTANSOK** szakasz **Deklaráció** fejezetében írottakon túl további korlátozásokat ró a konstans kifejezésre, hogy egész típusúnak kell lennie.

Operandusai ebben az esetben csak egész, felsorolás, karakteres és lebegőpontos állandók lehetnek, de a lebegőpontos konstanst explicit típuskonverzióval egésszé kell alakítani. Operandus lehet még a **sizeof** operátor is, aminek operandusára természetesen nincsenek ilyen korlátozások.

Tudjuk, hogyha elmarad a tömbméret, akkor a fordító az *inicializátorlista* elemszámának megállapításával rögzíti azt, s a típus így válik teljessé. Megadott méretű tömb esetében a lista *inicializátor*ainak száma nem haladhatja meg a tömb elemeinek számát. Ha az *inicializátor*ok kevesebben vannak a tömbméretnél, akkor a magasabb indexű, fennmaradó tömbelemek zérus kezdőértéket kapnak. Tudjuk azt is, hogy tömb inicializátorai csak állandó kifejezések lehetnek.

💣 Ne felejtkezzünk meg róla, hogy ugyan a karaktertömb *inicializátorlistá*ja karakterlánc konstans, de az előbb felsorolt korlátozások ugyanúgy vonatkoznak rá is! Vagyis rögzített méretű tömb esetén a karakterlánc hossza sem haladhatja meg a méretet. Ha a lánchossz és a tömbméret egyezik, nem lesz lánczáró zérus a karaktertömb végén.

Ha a tömbdeklarációból hiányzik a tömbméret és *inicializátorlista* sincs, akkor a deklaráció nem teljes típusú tömböt határoz meg. Lássuk a lehetséges eseteket!

- Ez csak előzetes deklaráció és a tömb méretét majd egy későbbi definíció rögzíti. Például:

```
float t[];
/* . . . */
float t[20];
```

- Ez csak előzetes deklaráció és a tömbméretet egy későbbi inicializátorlistás deklaráció tisztázza. Például:

```
char lanc[];
/* . . . */
char lanc[]="Ne tudd ki!";
```

✍ Természetesen az együtt tömbméretes és inicializátorlistás definiáló deklaráció is megengedett!

- A globális tömböt nem ebben a forrásmodulban definiálták, itt csak egyszerűen a rá való hivatkozás előtt deklarálták. Például:

```
extern double dtomb[];
```

✍ Persze a **dtomb** méretét valamilyen módon ebben a forrásmodulban is ismernünk kell!

- A tömb függvény paramétere. Például:

```
void fv(float t[], int n) { /* . . . */ }
```

✍ A tömb méretét valahonnan a függvényben is tudni kéne! Például úgy, hogy további paraméterként átadják neki.

9.4 Mutatóaritmetika és konverzió

A mutató vagy címaritmetika az inkrementálásra, a dekrementálásra, az összeadásra, a kivonásra és az összehasonlításra szorítkozik. A *típus* típusú objektumra mutató mutatón végrehajtott aritmetikai műveletek automatikusan figyelembe veszik a *típus* méretét, azaz az objektum tárolására elhasznált bájtok számát. A mutatóaritmetika ezen kívül feltételezi, hogy a mutató a *típus* típusú objektumok tömbjére mutat, azaz például:

```
int i = 6;
float ftomb[50], *fptr = ftomb;
```

esetén

```
fptr += i;
```

hatására az **fptr**-beli cím

```
sizeof(float)*i
```

-vel (általánosságban **sizeof**(*típus*)***egész*-szel) nő, azaz a példa szerint **ftomb**[6]-ra mutat.

Ha **ptr1** a *típus* típusú tömb második és **ptr2** a tizedik elemére mutat, akkor a két mutató különbsége

```
ptr2 - ptr1 ⇒ 8.
```

✍ Figyeljük meg, hogy a mutatókhoz indexértéket adunk hozzá vagy vonunk ki belőle és a mutatók különbsége is indexérték! Igazából a két mutató különbsége **ptrdiff_t** típusú egész indexkülönbség. A **ptrdiff_t** az **STDDEF.H** fejfájlban definiált, s többnyire **signed int**.

9.4.1 Összeadás, kivonás, inkrementálás és dekrementálás

- Összeadás egyik operandusa lehet mutató, ha a másik operandus *egész* típusú. Az eredmény ilyenkor a címaritmetika szabályait követő mutató, azaz olyan cím, mely *egész****sizeof**(*típus*)–sal nagyobb, mint az eredeti. A *típus* a mutató által mutatott nem **void** típus.
- Kivonásnál az első operandus lehet valamilyen objektumra mutató mutató, ha ilyenkor a második *egész* típusú. Az eredmény most is a mutatóaritmetika szabályait követő mutató, azaz olyan cím, mely *egész****sizeof**(*típus*)–sal kisebb, mint az eredeti. A *típus* a mutató által mutatott nem **void** típus.

- Kivonásnál mindkét operandus lehet ugyanazon objektum részeire mutató mutató. Az eredmény az előző két pont szellemében a mutatóaritmetika szabályainak megfelelő egész (indexkülönbség). Az egész típusa a szabvány **STDDEF.H** fejfájlban definiált **ptrdiff_t (signed int).**

Ha az operandus mutató, akkor az eggyel növelésben (++) vagy csökkentésben (−−) a címaritmetika szabályai érvényesek, azaz az eredmény mutató a következő, vagy a megelőző elemre fog mutatni.

💣 Ha **p** mutató a tömb utolsó elemére mutat, akkor a ++**p** még legális érték: a tömb utolsó utáni elemének címe, de minden ezután következő mutatónövelés definiálatlan eredményre vezet. Hasonló probléma van akkor is, ha **p** a tömb kezdetére mutat. Ilyenkor a mutatócsökkentés - már a --**p** is - definiálatlan eredményt okoz.

✍ Vegyük észre, hogy az egész mutatóaritmetikának csak akkor van értelme, ha a mutatók egyazon tömb elemeire mutatnak!

💣 Minden nem tömbre alkalmazott címaritmetikai művelet eredménye definiálatlan. Ugyanez mondható el akkor is, ha a tömbre alkalmazott mutatóaritmetikai művelet eredménye a tömb legelső eleme elé, vagy a legutolsó utánin túlra mutat.

9.4.2 Relációk

A következő operandus típuskombinációk használhatók relációkban:

- Mindkét operandus lehet ugyanazon típusú objektumra mutató mutató, amikor is a két objektum memória címének összehasonlításáról lesz szó.
- A mutató összehasonlítás csak egyazon objektum részeire definiált, és a mutatók nem tartalmazhatnak ezen objektumon kívülre irányuló címet sem egyetlen kivétellel, tömb esetén megengedett az utolsó létező elem utáni címére való hivatkozás. Ha a mutatók tömb elemekre hivatkoznak, akkor az összehasonlítás az indexek összehasonlításával ekvivalens. A nagyobb indexű elem címe magasabb.
- Az egyenlőségi relációkat használva a mutató hasonlítható a konstans 0 értékhez, azaz a **NULL** mutatóhoz is.
- Csak az egyenlőségi relációk esetében a mutató hasonlítható a konstans, 0 értékű, egész kifejezéshez, vagy **void–**ra mutató mutatóhoz. Ha mindkét mutató **NULL** mutató, akkor egyenlők.

9.4.3 Feltételes kifejezés

Ha a *K1* ? *K2* : *K3*–ban *K2*, *K3* egyike–másika mutató, akkor a *K2* vagy *K3* operandusok típusától függő konstrukció eredményének típusa a következő:

- Ha az egyik operandus valamilyen típusú objektumra mutató és a másik **void** mutató, akkor az eredmény (esetleges konverzió után) ugyancsak **void** mutató.
- Ha az egyik operandus mutató, s a másik operandus 0 értékű konstans kifejezés, akkor az eredmény típusa a mutató típusa lesz.

Mutatók típusának összehasonlításakor a **const** vagy **volatile** típusmódosítók nem szignifikánsak, de az eredmény típusa megörökli mindkét oldal módosítóit.

Írjuk át a címaritmetika alkalmazásával a **PELDA20.C**–ben használt **DVEREM.C**–t!

```
/* DVEREMUT.C: double verem push, pop és clear
               függvényekkel. */
#define MERET 128           /* A verem mérete. */
static double v[MERET];     /* A verem. */
static double *vmut=v;      /* A veremmutató. */
int clear(void){
  vmut=v;
  return MERET; }
double push(double x){
  if(vmut<v+MERET) return *vmut++=x;
  else return x+1.; }
double pop(void){
  if(vmut>v) return *--vmut;
  else return 0.; }
```

☞ A **vmut** most valóban veremmutató a **double** veremben (és nem a processzorében). **v** kezdőértékkel indul, és mindig a következő szabad helyre irányul. Ha kisebb, mint a veremhez már nem tartozó cím (**v**+**MERET**), akkor a **push** kiteszi rá a paraméterét, és mellékhatásként előbbre is lépteti eggyel a veremmutatót a következő szabad helyre. A **pop** csak akkor olvas a veremből, ha van benne valami (**vmut**>**v**). Kiszedéskor előbb vissza kell állítani a veremmutatót (az előtag −−), s csak aztán érhető el indirekcióval a legutoljára kitett érték, s most ez lesz a következő szabad hely is egyben a következő **push** számára.

9.4.4 Konverzió

A fordító által automatikusan elvégzett, implicit konverziókat már megismertük a címaritmetika műveleteinél.

Az explicit típuskonverziós

(*típusnév*) *előtag-kifejezés*

szerkezetben a (*típusnév*) többnyire (*típus* *) alakú lesz, és ilyen *típus*ú objektumra mutató mutatóvá konvertálja az *előtag-kifejezés* értékét. Például:

```
char *lanc;
int *ip;
/* . . . */
lanc = (char *) ip;
```

Nullaértékű konstans, egész kifejezés, vagy ilyen (**void** *)–gal típusmódosítva konvertálható explicit típusmódosítással, hozzárendeléssel vagy összehasonlítással akármilyen típusú mutatóvá. Ez **NULL** mutatót eredményez, mely megegyezik az ugyanilyen típusú **NULL** mutatóval, de eltér bármely más objektumra vagy függvényre mutató mutatótól.

Egy bizonyos típusú mutató konvertálható más típusú mutatóvá. Az eredmény azonban címzés hibás lehet, ha nem megfelelő tárillesztésű objektumot érne el. Csak azonos, vagy kisebb szigorúságú tárillesztési feltételekkel bíró adattípus mutatójává konvertálható az adott mutató, és onnét vissza.

🕮 A tárillesztés hardver sajátosság, s azt jelenti, hogy a processzor bizonyos típusú adatokat csak bizonyos határon levő címeken helyezhet el. A legkevésbé megszorító a **char** típus szokott lenni. A **short** csak szóhatáron (2–vel maradék nélkül osztható címen) kezdődhet, a **long** viszont dupla szóhatáron (4–gyel maradék nélkül osztható címen) helyezkedhet el, s így tovább.

✍ Felkérjük az olvasót, hogy programfejlesztő rendszere segítségében feltétlenül nézzen utána a konkrétumoknak!

void mutató készíthető akármilyen típusú mutatóból, és megfordítva korlátozás és információvesztés nélkül. Ha az eredményt visszakonvertáljuk az eredeti típusra, akkor az eredeti mutatót állítjuk újra elő.

Ha ugyanolyan, de más, **const** vagy **volatile** módosítójú típusra konvertálunk, akkor az eredmény ugyanaz a mutató a módosító által előidézett megszorításokkal. Ha a módosítót aztán elhagyjuk, akkor a további műveletek során az eredetileg az objektum deklarációjában szereplő **const** vagy **volatile** módosítók maradnak érvényben.

A mutató mindig konvertálható a tárolásához elegendően nagy, egész típussá. A mutató mérete, és az átalakító függvény persze nem gépfüggetlen. Leírunk egy, több fejlesztő rendszerben is használatos mutató–egész és egész–mutató konverziót.

A mutató–egész konverzió módszere függ a mutató és az egész típus méretétől, valamint a következő szabályoktól:

- Ha a mutató mérete nem kisebb az egész típusénál, akkor a mutató érték **unsigned–**ként viselkedik azzal a megkötéssel, hogy nem konvertálható lebegőpontossá.
- Ha a mutató mérete kisebb az egész típusénál, akkor a mutatót előbb az egésszel megegyező méretűvé alakítja a fordító, s aztán konvertálja egésszé.

Az egész–mutató átalakítás sem portábilis, de a következő szabályok szerint mehet például:

- Ha az egész típus ugyanolyan méretű, mint a mutató, akkor az egész értéket **unsigned–**ként mutatónak tekinti a fordító.
- Ha az egész típus mérete különbözik a mutatóétól, akkor az egész típust az előző pontokban ismertetett módon konvertálja előbb mutató méretű, egész típusúvá, majd **unsigned–**ként mutatónak tekinti.

9.5 Karaktermutatók

Tudjuk, hogy a karakterlánc konstans karaktertömb típusú, s ebből következőleg mögötte ugyancsak egy cím konstans van, hisz például a

```
printf(”Ez egy karakterlánc konstans.”);
```

kitűnően működik, holott a **printf** függvény első paramétere **const char *** típusú. Ez a konstans mutató azonban a tömbtől eltérően nem rendelkezik azonosítóval sem, tehát később nincs módunk hivatkozni rá. Ennek elkerülésére, azaz a cím konstans értékének megőrzésére, a következő módszerek ajánlhatók:

```
char *uzenet;
uzenet = ”Kész a kávé!\n”;
```

vagy

```
const char *uzenet = ”Kész a kávé!\n”;
```

9.5.1 Karakterlánc kezelő függvények

A rutinok prototípusai a szabványos **STRING.H** fejfájlban helyezkednek el. Egyik részüknek **str**–rel, másik csoportjuknak **mem**–mel kezdődik

a neve. Az **str** kezdetűek karakterláncokkal (**char** *), míg a **mem** nevűek memóriaterületekkel bájtonként haladva (**void** * és nincs feltétlenül lánczáró zérus a bájtsorozat végén) foglalkoznak. A **char** *, vagy **void** * visszaadott értékű függvények mindig az eredmény lánc kezdőcímét szolgáltatják.

💣 A **memmove**–tól eltekintve, a többi rutin viselkedése definiálatlan, ha egymást a memóriában átfedő karaktertömbökre használják őket.

Néhányat – teljességre való törekvés nélkül – felsorolunk közülük!

char ***strcat**(char **cel*, const char **forras*);

char ***strncat**(char **cel*, const char **forras*, size_t *n*);

A függvények a *cel* karakterlánchoz fűzik a *forras*–t (**strcat**), vagy a *forras* legfeljebb első, *n* karakterét (**strncat**), és visszatérnek az egyesített *cel* karakterlánc címével. Nincs hibát jelző visszaadott érték! Nincs túlcsordulás vizsgálat a karakterláncok másolásakor és hozzáfűzésekor.

✍ A **size_t** többnyire az **unsigned int** típusneve.

Írjuk csak meg a saját **strncat** függvényünket!

```
char *strncat(char *cel, const char *forras, size_t n){
  char *seged = cel;
  /* Pozícionálás a cel lezáró '\0' karakterére: */
  while(*cel ) ++cel;
  /* forras cel végére másolása a záró '\0'-ig, vagy
     legfeljebb n karakterig: */
  while(n-- && (*cel++ = *forras++));
  /* Vissza az egyesített karakterlánc kezdőcíme: */
  return seged; }
```

char ***strchr**(const char **string*, int *c*);

void ***memchr**(const void **string*, int *c*, size_t *n*);

A rutinok a *c* karaktert keresik *string*–ben, ill. *string* első *n* bájtjában, és az első előfordulás címével térnek vissza, ill. **NULL** mutatóval, ha nincs is *c* a *string*–ben, vagy az első *n* bájtjában. A láncáró zérus is lehet *c* paraméter. Az

char ***strrchr**(const char **string*, int *c*);

ugyanazt teszi, mint az **strchr**, csak *c string*–beli utolsó előfordulásának címével tér vissza, ill. **NULL** mutatóval, ha nincs is *c* a *string*–ben.

int **strcmp**(const char **string1*, const char **string2*);

int **strncmp**(const char **string1*, const char **string2*, size_t *n*);

int **memcmp**(const void **string1*, const void **string2*, size_t *n*);

A függvények **unsigned char** típusú tömbökként összehasonlítják *string1* és *string2* karakterláncokat, és negatív értéket szolgáltatnak, ha *string1* < *string2*. Pozitív érték jön, ha *string1* > *string2*. Az egyenlőséget viszont a visszaadott zérus jelzi.

Az **strncmp** és a **memcmp** a hasonlítást legfőljebb az első *n* karakterig, ill. bájtig végzik.

📖 A legtöbb fejlesztő rendszerben nem szabványos **stricmp** és **strnicmp** is szokott lenni, melyek nem kis–nagybetű érzékenyen hasonlítják össze a karakterláncokat.

A saját **strcmp**:

```
int strcmp(const char *s1, const char *s2 ){
  for( ; *s1 == *s2; ++s1, ++s2)
    if(!(*s1)) return 0;  /* s1 == s2 */
  return(*s1 - *s2); }    /* s1 < s2 vagy s1 > s2 */
```

char ***strcpy**(char **cel*, const char **forras*);

char ***strncpy**(char **cel*, const char **forras*, size_t *n*);

void ***memcpy**(void **cel*, const void **forras*, size_t *n*);

void ***memmove**(void **cel*, const void **forras*, size_t *n*);

Az **strcpy** a *forras* karakterláncot másolja lánczáró karakterével együtt a *cel* karaktertömbbe, és visszatér a *cel* címmel. Nincs hibát jelző visszatérési érték. Nincs túlcsordulás ellenőrzés a karakterláncok másolásánál.

Az **strncpy** a *forras* legfeljebb első *n* karakterét másolja. Ha a *forras n* karakternél rövidebb, akkor *cel* végét '\0'–ázza *n* hosszig a rutin. Ha az *n* nem kisebb, mint a *forras* mérete, akkor nincs zérus a másolt karakterlánc végén.

A **memcpy** és a **memmove** mindenképpen *n* bájtot másolnak. Egyetlenként a karakterlánc kezelő függvények közül, a **memmove** akkor is biztosítja az átlapoló memóriaterületeken levő, eredeti *forras* bájtok felülírás előtti átmásolását, ha a *forras* és a *cel* átfedné egymást.

A saját **strcpy**:

```
char *strcpy(char *cel, const char *forras ){
  char *seged = cel;
  /* forras cel-ba másolása lezáró '\0' karakterével: */
  while(*cel++ = *forras++);
  /* Vissza a másolat karakterlánc kezdőcíme: */
  return seged; }
```

size_t **strlen**(const char **string*);

A rutin a *string* karakterlánc karaktereinek számával tér vissza a lánczárót nem beszámítva. Nincs hibát jelző visszaadott érték!

char ***strpbrk**(const char **string*, const char **strCharSet*);

A függvények az *strCharSet*–beli karakterek valamelyikét keresik a *string*–ben, és visszaadják az első előfordulás címét, ill. **NULL** mutatót, ha a két paraméternek közös karaktere sincs. A keresésbe nem értendők bele a lánczáró zérusok.

A példában számokat keresünk az **s** karakterláncban.

```
#include <string.h>
#include <stdio.h>
void main(void){
  char s[100] = "3 férfi és 2 fiu 5 disznót ettek.\n";
  char *er = s;
  int i=1;
  while(er = strpbrk(er, "0123456789"))
    printf("%d: %s\n", i++, er++); }
```

A kimenet a következő:

```
1: 3 férfi és 2 fiu 5 disznót ettek.
2: 2 fiu 5 disznót ettek.
3: 5 disznót ettek.
```

char ***strset**(char **string*, int *c*);

char ***strnset**(char **string*, int *c*, size_t *n*);

void ***memset**(void **string*, int *c*, size_t *n*);

A **memset** feltölti *string* első *n* bájtját *c* karakterrel, és visszaadja *string* kezdőcímét.

📖 A legtöbb fejlesztő rendszerben létező, nem ANSI szabványos **strset** és **strnset** a *string* minden karakterét – a lánczáró zérus kivételével – *c* karakterre állítják át, és visszaadják a *string* kezdőcímét. Nincs hibát jelző visszatérési érték. Az **strnset** azonban a *string* legfeljebb első *n* karakterét inicializálja *c*–re.

size_t **strspn**(const char **string*, const char **strCharSet*);

size_t **strcspn**(const char **string*, const char **strCharSet*);

Az **strspn** annak a *string* elején levő, maximális alkarakterláncnak a méretét szolgáltatja, mely teljes egészében csak az *strCharSet*–beli karakterekből áll. Ha a *string* nem *strCharSet*–beli karakterrel kezdődik, akkor

zérust kapunk. Nincs hibát jelző visszatérési érték. A keresésbe nem értendők bele a lánczáró zérus karakterek.

A függvény visszaadja tulajdonképpen az első olyan karakter indexét a *string*–ben, mely nincs benn az *strCharSet* karakterlánccal definiált karakterkészletben.

Az **strcspn** viszont annak a *string* elején levő, maximális alkarakterláncnak a méretét szolgáltatja, mely egyetlen karaktert sem tartalmaz az *strCharSet*–ben megadott karakterekből. Ha a *string strCharSet*–beli karakterrel kezdődik, akkor zérust kapunk. Például a:

```
#include <string.h>
#include <stdio.h>
void main(void){
  char string[] = "xyzabc";
  printf("%s láncban az első a, b vagy c indexe %d\n",
         string, strcspn(string, "abc")); }
```

eredménye:

```
xyzabc láncban az első a, b vagy c indexe 3
```

char ***strstr**(const char **string1*, const char **string2*);

A függvény *string2* karakterlánc első előfordulásának címét szolgáltatják *string1*–ben, ill. **NULL** mutatót kapunk, ha *string2* nincs meg *string1*–ben. Ha *string2* üres karakterlánc, akkor a rutin *string1*–gyel tér vissza. A keresésbe nem értendők bele a lánczáró zérus karakterek.

char ***strtok**(char **strToken*, const char **strDelimit*);

A függvény a következőleg megtalált, *strToken*–beli szimbólum címével tér vissza, ill. **NULL** mutatóval, ha nincs már további szimbólum az *strToken* karakterláncban. Mindenegyes hívás módosítja az *strToken* karakterláncot, úgy hogy '\0' karaktert tesz a bekövetkezett elválasztójel helyére. Az *strDelimit* karakterlánc az *strToken*–beli szimbólumok lehetséges elválasztó karaktereit tartalmazza.

Az első **strtok** hívás átlépi a vezető elválasztójeleket, visszatér az *strToken*–beli első szimbólum címével, és ezelőtt a szimbólumot '\0' karakterrel zárja. Az *strToken* maradék része további szimbólumokra bontható újabb **strtok** hívásokkal. Mindenegyes **strtok** hívás módosítja az *strToken* karakterláncot, úgy hogy '\0' karaktert tesz az aktuálisan visszaadott szimbólum végére. Az *strToken* következő szimbólumát az *strToken* paraméter helyén **NULL** mutatós **strtok** hívással lehet elérni. A **NULL** mutató első paraméter hatására az **strtok** megkeresi a következő szimbólumot a módosított *strToken*–ben. A lehetséges elválasztójeleket tartalma-

zó *strDelimit* paraméter, s így maguk az elválasztó karakterek is, változhatnak hívásról–hívásra.

A példaprogramban ciklusban hívjuk az **strtok** függvényt, hogy megjelentethessük az **s** karakterlánc összes szóközzel, vesszővel, stb. elválasztott szimbólumát:

```
#include <string.h>
#include <stdio.h>
void main( void ){
  char s[] = "Szimbólumok\tkarakterlánca\n ,, és néhány"
             "  további szimbólum";
  char selv[]   = " ,\t\n", *token;
  printf("%s\nSzimbólumok:\n", s);
  /* Míg vannak szimbólumok a karakterláncban, */
  token = strtok(string, selv);
  while(token != NULL){
    /* addig megjelentetjük őket, és */
    printf("\t%s\n", token);
    /* vesszük a következőket: */
    token=strtok(NULL, selv); } }
```

A kimenet a következő:

```
Szimbólumok karakterlánca
 ,, és néhány  további szimbólum
Szimbólumok:
  Szimbólumok
  karakterlánca
  és
  néhány
  további
  szimbólum
```

Megoldandó feladatok:

Készítse el az alább felsorolt, ismert C függvények mutatókat használó változatát! A **char *** visszatérésű rutinok az eredmény címével térnek vissza.

- **int getline**(**char ***, **int**): sor beolvasása a szabvány bemenetről.
- **char *squeeze**(**char ***s*, int *c*): *c* karakter törlése az *s* karakterláncból a saját helyén.
- **int *binker**(**int** *x*, **int ***t*, **int** *n*): a növekvőleg rendezett, *n* elemű, *t* tömbben megkeresendő *x* csak mutatókat használó, bináris keresési algoritmussal! Ha megvan, vissza kell adni a *t*–beli előfordulás címét. Ha nincs, **NULL** mutatót kell szolgáltatni.
- **int atoi**(**char ***): karakterlánc egésszé konvertálása.

- **char *strupr(char *)**: karakterlánc nagybetűssé alakítása a saját helyén.
- **char *strrev(char *)**: karakterlánc megfordítása a saját helyén.
- **char *strset(char *s, int c)**: *s* karakterlánc feltöltése *c* karakterrel.
- **char *strstr(const char *, const char *)**: a második karakterlánc keresése az elsőben. A pontos ismertetés kicsit előbbre megtalálható!

Készítsen **char *strstrnext(const char *, const char *)** függvényt is, mely ugyanazt teszi, mint az **strstr**, de **static** mutató segítségével az első hívás után a második karakterlánc következő elsőbeli előfordulásának címével tér vissza, és ezt mindaddig teszi, míg **NULL** mutatót nem kell szolgáltatnia.

9.5.2 Változó paraméterlista

Az **OBJEKTUMOK ÉS FÜGGVÉNYEK** szakaszból tudjuk, hogy a függvény paraméterlistája …–tal is végződhet, amikor is a fordító a ,… előtti fix paramétereket a szokásos módon kezeli, de a ,.. helyén álló aktuális paramétereket úgy manipulálja, mintha nem adtak volna meg függvény prototípust.

A szabványos **STDARG.H** fejfájlban definiált típus és függvények (makrók) segítséget nyújtanak az ismeretlen paraméterszámú és típusú paraméterlista feldolgozásában. A típus és az ANSI kompatibilis makródefiníciók a következők lehetnek például:

```
typedef char *va_list;
#define _INTSIZEOF(x) ((sizeof(x)+sizeof(int)-1)&\
                      ~(sizeof(int) -1))
#define va_start(ap, utsofix)  (ap=(va_list)&utsofix+\
                       _INTSIZEOF(utsofix))
#define va_arg(ap, tipus)      (*(tipus   *)((ap+=_INTSIZEOF
                       (tipus)) - _INTSIZEOF(tipus)))
#define va_end(ap) ap = (va_list)0
```

A **va_...** szerkezettel csak olyan változó paraméteres függvények írhatók, ahol a változó paraméterek a paraméterlista végén helyezkednek el. A **va_...** makrókat a következőképp kell használni:

1. A szerkezetet használó függvényben deklarálni kell egy **va_list** típusú, mondjuk, **param** nevű változót:

   ```
   va_list param;
   ```

, ami a **va_arg** és a **va_end** által igényelt információt hordozó mutató.

2. Meg kell hívni a **va_start** függvényt:

   ```
   va_start(param, utsofix);
   ```

   a **va_arg** és a **va_end** függvények használata előtt. A **va_start** a **param** mutatót a változó paraméterlistával meghívott függvény első változó paraméterére állítja. Az **utsofix** paraméterben elő kell írni a változó paraméterlistával meghívott függvény azon utolsó formális paraméterének nevét, amely még rögzített. A **va_start**–nak nincs visszaadott értéke. A **va_start** első paraméterének **va_list** típusúnak kell lennie. Ha a második paraméter **register** tárolási osztályú, akkor a makró viselkedése nem meghatározott.

3. Eztán **va_arg** hívások következnek:

   ```
   (tipus) va_arg(param, tipus);
   ```

   , melyek rendre szolgáltatják a változó paraméterlista következő aktuális paraméterének értékét. A makróból látszik, hogy a **tipus** megadása fontos, hisz eszerint lépteti a **param** a mutatót előre, ill. ilyen típusú értéket szolgáltat a változó paraméterlistában soron következő paraméterből.

💣 Azt aztán, hogy a változó paraméterlistának mikor van vége, a programozónak kell tudnia!

4. Ha a **va_arg** kiolvasta a változó paraméterlista minden elemét, vagy más miatt kell abbahagynunk a további feldolgozást, akkor meg kell hívni a

   ```
   va_end(param);
   ```

   makrót, hogy a változó paraméterlistával meghívott függvényből biztosítsuk a normális visszatérést. A **va_end**–nek nincs visszaadott értéke, és a makródefinícióból látszik, hogy **NULL**ázza a **param** mutatót.

Tekintsünk meg a következő egyszerű példát, melyben zérus jelzi a paraméterlista végét!

```
#include <stdio.h>
#include <stdarg.h>
void sum(char *uzen, ...) {
  int osszeg = 0, tag;
  va_list param;
  va_start(param, uzen);
  while(tag = va_arg(param, int)) osszeg += tag;
```

```
  printf(uzen, osszeg); }
void main(void) {
  sum("1+2+3+4 = %d\n", 1, 2, 3, 4, 0); }
```

A **param char** * típusú. Tételezzük fel a 32 bites **int** és cím esetét! Akkor a **va_start**(**param**, **uzen**) hatására az

```
_INTSIZEOF(uzen)=
(sizeof(uzen)+sizeof(int)-1)&~(sizeof(int) -1)=
 (4+4-1)&~(4-1)=7&~3=4
```

és így a

```
param=(char *)&uzen+4
```

, ami azt jelenti ugye, hogy **param** az első változó paraméter címét fogja tartalmazni.

A **tag=va_arg**(**param**, **int**) következtében

```
tag=(*(int *)((param+=_INTSIZEOF(int))-_INTSIZEOF(int)))
```

, azaz:

```
tag=*(int *)((param+=4)-4)
```

Tehát a **tag** változó felveszi a következő, **int**, aktuális paraméter értékét, és eközben **param** már a következő aktuális paraméterre mutat.

🕮 A következő függvények a *parlist* változó paraméterlistától eltekintve ugyanúgy dolgoznak, mint nevükben **v** nélküli párjaik. A felsorolt **printf** függvények a 3. pont **va_arg** hívásait maguk intézik, és a *format* karakterlánc alapján a változó paraméterlista végét is látják. A *parlist* az aktuális változó paraméterlista első elemére mutató mutató.

int **vfprintf**(FILE **stream*, const char **format*, va_list *parlist*);

int **vprintf**(const char **format*, va_list *parlist*);

int **vsprintf**(char * *puffer*, const char * *format*, va_list *parlist*);

9.6 Mutatótömbök

Mutatótömb a

*típus *azonosító*[<*konstans-kifejezés*>] <={*inicializátorlista*}>;

deklarációval hozható létre. Az *azonosító* a mutatótömb neve: konstans mutató, melynek típusa (*típus* **), vagyis *típus* típusú objektumra mutató mutatóra mutató mutató. Tehát olyan konstans mutató, mely (*típus* *) mutatóra mutat.

✍ Vegyük észre a * mutatóképző operátor rekurzív használatát!

A tömb egy eleme viszont (*típus* *) típusú változó mutató.

Kezdjünk el egy példát magyar kártyával!

A színeket kódoljuk a következőképp: makk (0), piros (1), tök (2) és zöld (3). A kártyákat egy színen belül jelöljük így: hetes (0), nyolcas (1), kilences (2), tízes (3), alsó (4), felső (5), király (6) és ász (7). A konkrét kártya kódja úgy keletkezik, hogy a színkód után leírjuk a kártya kódját. Ilyen alapon a 17 például piros ász.

Készítsünk **char *KartyaNev(int** *kod*) függvényt, mely visszaadja a paraméter *kod*ú kártya megnevezését!

```
/* KARTYA.H fejfájl. */
/* Szimbolikus állandók: */
#define SZINDB  4  /* Színek száma. */
#define KTYDB   8  /* Kártyák száma egy színen belül. */
#define OSZTO   10 /* Szín és kártyakód szétválasztásához*/
#define PAKLI SZINDB*KTYDB  /* Kártyaszám a pakliban. */
#define MAXKOD  (SZINDB-1)*OSZTO+KTYDB /* Max. kártykód.*/
/* Prototípusok: */
char * KartyaNev(int);
void UjPakli(void);
int  Mennyi(void);
int  UjLap(void);
```

```
/* KARTYA.C: Adat és függvénydefiníciók. */
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "KARTYA.H"
static char *szin[SZINDB]={"makk ", "piros ", "tök ",
                           "zöld "};
static char *kartya[KTYDB]={"VII", "VIII", "IX", "X",
                     "alsó", "felső", "király", "ász"};
char * KartyaNev(int kod){
  static char szoveg[20];
  if(kod>=0 && kod/OSZTO<SZINDB && kod%OSZTO<KTYDB){
    strcpy(szoveg, szin[kod/OSZTO]);
    strcat(szoveg, kartya[kod%OSZTO]); }
  else strcpy(szoveg, "zöldség");
  return szoveg; }
```

Vigyázzunk nagyon a típusokkal! A **szin** és a **kartya** statikus élettartamú, belső kapcsolódású, fájl hatáskörű, karakterláncokra mutatók tömbjeinek kezdeteire mutató mutató (**char** **) konstansok nevei. A **szin**[1] a piros karakterlánc kezdetének címét tartalmazó, karakteres mutatótömb elem (**char** *). Igazak például a következők:

```
*szin ≡ szin[0]
**szin ≡ *szin[0] ≡ 'm'
```

A **szoveg** statikus élettartamú karaktertömb, így nem szűnik meg a memóriafoglalása a függvényből való visszatéréskor, csak lokális hatáskörű azonosítójával nem érhető el!

Írjunk rövid, kipróbáló programot!

```
/* PELDA23.C: Kártya megnevezések kiíratása. */
#include <stdio.h>
#include "KARTYA.H"
void main(void){
  int i;
  printf("Magyar kártya megnevezések rendre:\n");
  for(i=-1; i<=MAXKOD; ++i)
    printf("%-40s", KartyaNev(i)); }
```

9.7 Többdimenziós tömbök

A többdimenziós tömböket a tömb típus tömbjeiként konstruálja meg a fordító. A deklaráció

típus azonosító[<*méret1*>][*méret2*]. . .[*méretN*] <={*inicializátorlista*}>;

alakú. Az elhelyezésről egyelőre annyit, hogy sorfolytonos, azaz a jobbra álló index változik gyorsabban. Például a

```
double matrix[2][3];
```

sorfolytonosan a következő sorrendben helyezkedik el a tárban:

```
matrix[0][0], matrix[0][1], matrix[0][2], matrix[1][0],
matrix[1][1], matrix[1][2]
```

Egy elem elérése például

azonosító[index1][index2]. . .[indexN]

módon megy, ahol az indexekre igazak a következők:

0 <= *index1* < *méret1*

0 <= *index2* < *méret2*

. . .

0 <= *indexN* < *méretN*

Vigyázzunk az indexek külön–külön []–be írására is! Véve mondjuk a **matrix**[**i**][**j**]–t, a **matrix**[**i**,**j**] kifejezés is „értelmes” a C–ben. Persze nem **matrix**[**i**][**j**]–t jelenti, hanem a vessző operátor végrehajtása után a **matrix**[**j**] tömböt.

A **BEVEZETÉS ÉS ALAPISMERETEK** szakasz **Inicializálás** fejezetében a tömbök inicializálásáról mondottak most is érvényben vannak, de

a többdimenziós tömb további tömbökből áll, s így az inicializálási szabályok is rekurzívan alkalmazandók. Az *inicializátorlista* egymásba ágyazott, egymástól vesszővel elválasztott *inicializátor*ok és *inicializátorlistá*k sorozata a sorfolytonos elhelyezési rend betartásával. Például a 4x3–as **t** tömb első sorának minden eleme 1 értékű, második sorának minden eleme 2 értékű, s így tovább.

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

Ha az *inicializátorlistá*ban nincs beágyazott *inicializátorlista*, akkor az ott felsorolt értékek az alaggregátumok, s azon belül az elemek sorrendjében veszik fel az aggregátum elemei. Az előző példával azonos eredményre vezetne a következő inicializálás is:

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

Kapcsos zárójelek akár az egyes inicializátorok köré is tehetők, de ha a fordítót nem kívánjuk „becsapni”, akkor célszerű őket az aggregátum szerkezetét pontosan követve használni! Az

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

hatására a **t** mátrix utolsó sorára (a **t**[3] tömbre) nem jut inicializátor, így a **t**[3][0], **t**[3][1] és **t**[3][2] elemek mind zérus kezdőértéket kapnak. Az

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

eredményeként a **t**[0][0] 1, a **t**[1][0] 2, a **t**[2][0] 3 lesz, és a tömb összes többi eleme zérus.

Tudjuk, hogyha nem adjuk meg, akkor az *inicializátorlista* elemszámából állapítja meg a tömbméretet a fordító. Többdimenziós tömb deklarációjában ez azonban csak az első méretre vonatkozik, a további méreteket mind kötelező előírni.

```
int a[][]={{1, 1}, {2, 2}, {3, 3}};            /* HIBÁS */
int t[][3]={{1, 1, 1}, {2, 2, 2}, {3, 3, 3}};  /* OK */
```

Folytassuk tovább a **PELDA23.C**–ben megkezdett magyar kártyás példánkat! Újabb programunknak legyen az a feladata, hogy megkeveri a paklit, s kioszt öt lapot! Aztán újra kioszt öt lapot, s így tovább mindaddig, míg a kártya el nem fogy. Újabb keverés következik és újabb osztások. A szoftvernek akkor van vége, ha már nem kérnek több lapot.

Azt, hogy milyen kártyákat osztott már ki a pakliból, statikus élettartamú, csak a **KARTYA.C** forrásfájlban elérhető, kétdimenziós, **SZINDB*** **KTYDB**–s, **kty** tömbben tartja nyilván a program. A megfelelő tömbelem zérus, ha még pakliban van a lap, és 1–gyé válik, ha kiosztják. A statikus,

a **KARTYA.C** forrásmodulra ugyancsak lokális, **ktydb** változóban a pakli aktuális kártyaszámát őrzi a szoftver.

A következő sorokkal mindig a **KARTYA.C** bővítendő!

```
static int kty[SZINDB][KTYDB];
static int ktydb=SZINDB*KTYDB;
```

Az **UjPakli** függvény ugyanezt az állapotot állítja elő.

```
void UjPakli(void){
  int i, j;
  for(i=0; i<SZINDB; ++i)
     for(j=0; j<KTYDB; ++j) kty[i][j]=0;
     ktydb=SZINDB*KTYDB; }
```

A **Mennyi** rutinnal lekérdezhető, hogy még hány kártya van a pakliban.

```
int Mennyi(void){
  return ktydb; }
```

Az **int UjLap(void)** függvény egy lapot ad a pakliból, azonban ezt véletlenszerűen teszi a következő technikával:

- Ha nincs kártya a pakliban, –1–et szolgáltat.
- Ha egyetlen lapból áll a pakli, akkor azt adja.
- Ha **ktydb**<**KTYDB**, akkor előállít egy 1 és **ktydb** közti véletlenszámot. Végigjárja a paklit, és visszaadja a véletlenszámadik, még nem kiosztott kártyát.
- Ha **ktydb**>=**KTYDB**, akkor véletlen színt és véletlen kártyát választ. Kiadja a lapot, ha még eddig nem osztotta ki. Ha a kártya már nincs a pakliban, újat választ, és így tovább.

☞ Látszik, hogy szükségünk lesz véletlenszám generátorra!

9.7.1 Véletlenszám generátor

Használatához az **STDLIB.H** fejfájlt kell bekapcsolni. Egész számok pszeudóvéletlen sorozatát generálja 0 és **RAND_MAX** között az

int **rand**(void);

függvény. A rutinnak nincs hibás visszatérése. A véletlenszám generálás *kezd*őértékét lehet beállítani a

void **srand**(unsigned int *kezd*);

függvénnyel. Az alapértelmezett induló érték 1, így ilyen értékű *kezd* paraméterrel mindig újrainicializáltathatjuk a véletlenszám generátort, azaz

a **rand** hívások ugyanazt a véletlenszám sorozatot produkálják **srand**(1) után, mintha mindenféle **srand** megidézés nélkül használtuk volna a **rand**–ot.

Az **srand**–ot a véletlenszerű indulást is biztosítandó rendszerint a **TIME.H**–ban helyet foglaló

time_t **time**(time_t **timer*);

függvény *kezd* paraméterrel szokták meghívni. A **time** rutin az aktuális rendszer időt 1970. január elseje éjfél óta eltelt másodpercek számában, **time_t** (**long**) típusban szolgáltatja. Nincs hibás visszatérés. A visszatérési értéket a *timer* címen is elhelyezi, ha a paraméter nem **NULL** mutató. **NULL** mutató aktuális paraméter esetén viszont nem tesz ilyet.

Az **srand** függvény szokásos hívása tehát:

```
srand(time(NULL))
```

Véletlenszám generátorral kockadobás értéket a következőképp produkálhatunk:

```
rand()%6 + 1
```

Ha a fejlesztő rendszerben nincs lebegőpontos véletlenszámot generáló függvény, akkor 0 és 1 közötti véletlenszámokat a következő kifejezéssel állíthatunk elő:

```
(double)rand()/RAND_MAX
```

Folytassuk az **UjLap** függvényt!

```
int UjLap(void){
  int i, j, db;
  if(ktydb==SZINDB*KTYDB) srand(time(NULL));
  if(ktydb){
    if(ktydb>=KTYDB)
      while(kty[i=rand()%SZINDB][j=rand()%KTYDB]);
    else{
      db=ktydb>1?rand()%ktydb+1:1;
      for(i=0; i<SZINDB; ++i){
        for(j=0; db&&j<KTYDB; ++j)
          if(!kty[i][j]&&!(--db)) break;
        if(!db) break; } }
    --ktydb;
    kty[i][j]=1;
    return i*OSZTO+j; }
  else return (-1); }
```

Elkészültünk a kibővített **KARTYA.C**–vel, s most írjuk meg a működtető **PELDA24.C** programot!

```
/* PELDA24.C: Öt lap osztása. */
#include <stdio.h>
#include <ctype.h>
#include "KARTYA.H"
#define LAPDB 5
void main(void){
  int i, c;
  printf("Zsugázás: %d lap leosztása:\n", LAPDB);
  while(printf("Ossza már (I/N)? "),
        (c=toupper(getchar()))!='N'){
    if(c=='I'){
      putchar('\n');
      if(Mennyi()<LAPDB) UjPakli();
      for(i=0; i<LAPDB; ++i)
        printf("%-15s", KartyaNev(UjLap()));
      printf("\n\n"); }
    while(c!=EOF&&c!='\n') c=getchar(); } }
```

Megoldandó feladatok:

Javítson a közölt programon a következő módon:

- A leosztott öt lap megjelentetése történjék színek szerint, és azon belül kártyák szerint rendezetten!
- A kapott öt lapból legyen legfeljebb 3 cserélhető!
- A játék működjék francia kártyával!

Ahhoz, hogy a többdimenziós tömbök belső szerkezetét megértsük, hozzunk létre dinamikusan egy mátrixot!

9.7.2 Dinamikus memóriakezelés

A C a memóriát három részre osztja. Az elsődleges adatterületen helyezi el a fordító a konstansokat és a statikus objektumokat. A lokális objektumokat és a függvényparamétereket a verembe (stack) teszi. A harmadik memóriaterületet – nevezzük heap–nek, bár más névvel is szokták illetni – futás közben éri el a C program, és változó méretű memória blokkok dinamikus allokálására való. Például fák, listák, tömbök vagy bármi más helyezhető el rajta. Az ismertetett, ANSI szabványos függvények prototípusai az **STDLIB.H** fejfájlban találhatók.

void ***calloc**(size_t *tetelek*, size_t *meret*);

A **calloc** *tetelek***meret* méretű memória blokkot foglal, feltölti 0X00-val és visszaadja kezdőcímét. Tulajdonképpen *tetelek* elemszámú tömbnek foglal helyet, ahol egy elem mérete *meret* bájt.

Ha nincs elég hely, vagy a *tetelek*meret* szorzat értéke zérus, **NULL** mutatót kapunk vissza.

void ***malloc**(size_t *meret*);

A **malloc** legalább *meret* bájtos memória blokkot foglal, nem tölti fel semmivel és visszaadja kezdőcímét. A blokk nagyobb lehet *meret* bájtnál a tárillesztéshez igényelt, plusz terület és a karbantartási információ elhelyezése miatt.

Ha nincs elég hely a heap-en, **NULL** mutatót kapunk vissza a függvénytől. Ha a *meret* 0, a **malloc** zérusméretű blokkot allokál, és érvényes mutatót ad vissza erre a területre.

🕮 Jó néhány szabványos függvény is hívja a **malloc**–ot. Például a **calloc**, a **getchar** stb.

🕮 A **calloc**–kal, a **malloc**–kal lefoglalt, vagy a rögtön ismertetendő **realloc**–kal újrafoglalt terület tárillesztése olyan, hogy bármilyen típusú objektum elhelyezésére alkalmas. A függvényektől visszakapott cím explicit típusmódosító szerkezettel bármilyen típusúvá átalakítható.

Tegyük fel, hogy a program futása közben derül ki egy **double** tömb mérete! Ezt az értéket az **int** típusú, **N** változó tartalmazza. Hogyan lehet létrehozni, kezelni, s végül felszabadítani egy ilyen tömböt?

```
/* . . . */
int N;
double *dtomb;
/* . . . */
/* Itt kiderül, hogy mennyi N. */
/* . . . */
/* Ettől kezdve szükség van a tömbre. */
if((dtomb=(double *)malloc(N*sizeof(double)))!=NULL){
  /* Sikeres a memóriafoglalás, azaz használható a tömb.
     Például a 6. eleme dtomb[5] módon is elérhető. */
  /* . . . */
  /* Nincs szükség a továbbiakban a tömbre. */
  free(dtomb);
  /* . . . */ }
else /* Hibakezelés. */
/* . . . */
```

void ***realloc**(void **blokk*, size_t *meret*);

A **realloc** *meret* méretűre szűkíti vagy bővíti a korábban **malloc**, **calloc** vagy **realloc** hívással allokált memória blokkot, melynek kezdőcímét megkapja a *blokk* paraméterben. Sikeres esetben visszaadja az átméretezett memória blokk kezdőcímét. Ez a cím nem feltétlenül egyezik meg a

blokk paraméterben átadottal. Címeltérés esetén a függvény a korábbi memória blokk tartalmát átmozgatja az újba. Az esetleges rövidüléstől eltekintve elmondható, hogy az új *blokk* megőrzi a régi tartalmát.

Ha az újraallokálás sikertelen memória hiány miatt, ugyancsak **NULL** mutatót kapunk, de az eredeti *blokk* változatlan marad.

void **free**(void **blokk*);

A **free** deallokálja vagy felszabadítja a korábban **malloc**, **calloc** vagy **realloc** hívással allokált memóriaterületet, melynek kezdőcímét megkapja a *blokk* paraméterben. A felszabadított bájtok száma egyezik az allokációkor (vagy a **realloc** esetén) igényelttel. Ha a *blokk* **NULL**, a mutatót elhagyja a **free,** és rögtön visszatér.

💣 Az érvénytelen mutatós (nem **calloc**, **malloc**, vagy **realloc** függvénnyel foglalt memória terület címének átadása) felszabadítási kísérlet befolyásolhatja a rákövetkező allokációs kéréseket, és fatális hibát is okozhat.

Folytassuk a mátrixos feladatot!

```
/* PELDA25.C: Kétdimenziós tömb létrehozása dinamikusan.*/
#include <stdio.h>
#include <stdlib.h>
typedef long double TIPUS;
typedef TIPUS **OBJEKTUM;
int m=3, n=5;             /* Sorok és oszlopok száma. */
int main(void) {
  OBJEKTUM matrix;
  int i, j;
  /* A sorok létrehozása: */
  printf("%d*%d-s, kétdimenziós tömb létrehozása "
         "dinamikusan.\n", m, n);
  if(!(matrix=(OBJEKTUM)calloc(m, sizeof(TIPUS *)))){
    printf("Létrehozhatatlanok a mátrix sorai!\n");
    return 1; }
  /* Az oszlopok létrehozása: */
  for(i = 0; i < m; ++i)
    if(!(matrix[i]=(TIPUS *)malloc(n*sizeof(TIPUS)))){
      printf("Létrehozhatatlan a mátrix %d. sora!\n",
             i);
      while(--i>=0) free(matrix[i]);
      free(matrix);
      return 1; }
  /* Mesterséges inicializálás: */
  for(i = 0; i < m; ++i)
    for(j = 0; j < n; ++j)
      matrix[i][j] = rand();
  /* Kiírás: */
```

```
  printf("A mátrix tartalma:\n");
  for(i = 0; i < m; ++i) {
    for(j = 0; j < n; ++j)
      printf("%10.0Lf", matrix[i][j]);
    printf("\n"); }
  /* Az oszlopok felszabadítása: */
  for(i = 0; i < m; ++i) free(matrix[i]);
  /* Sorok felszabadítása: */
  free(matrix);
  return 0; }
```

✍ Vegyük észre, hogy a **main**–nek van visszaadott értéke! Zérust szolgáltat, ha minden rendben megy, és 1–et, ha memóriafoglalási probléma lép fel.

✍ Figyeljük meg azt is, hogy a memória felszabadítása foglalásával éppen ellenkező sorrendben történik, hogy a heap–en ne maradjanak foglalt „szigetek”!

🕮 A heap C konstrukció, s ha a program befejeződik, akkor maga is felszabadul, megszűnik létezni.

Összesítve: A **matrix** azonosító a tömb kezdetére mutató változó mutató. A tömb kezdete viszont **m** változó mutatóból álló mutatótömb. A mutatótömb egy-egy eleme **n** elemű, **long double** típusú tömb kezdetére mutat. A példa a részek memóriabeli elhelyezkedését is szemlélteti, azaz:

- előbb az **m** elemű mutatótömböt allokáljuk, aztán
- a mátrix első sora (0-s indexű) **n** elemének foglalunk helyet.
- A mátrix második sora (1-s indexű) **n** elemének helyfoglalása következik.
- . . .
- Legvégül a mátrix utolsó (**m**-1 indexű) sorának **n** elemét helyezzük el a memóriában.

💣 A fordító ugyanezzel a módszerrel dolgozik, de az általa létrehozott mátrixban a mutatótömb konstans mutatókat tartalmaz, s a mátrix azonosítója is konstans mutató.

A többdimenziós tömbök többféleképpen is szemlélhetők. A fordító által létrehozott

```
long double matrix[m][n];
```

mátrix példájánál maradva:

- A **matrix** egy **m** elemű vektor (tömb) azonosítója. E vektor minden eleme egy **n** elemű, **long double** típusú vektor.
- A **matrix**[**i**] (**i** = 0, 1, ..., **m**-1) az **i**–edik, **n long double** elemű vektor azonosítója.
- A **matrix**[**i**][**j**] (**i** = 0, 1, ..., **m**-1 és **j** = 0, 1, ..., **n**-1) a mátrix egy **long double** típusú eleme.

A dolgokat más oldalról tekintve!

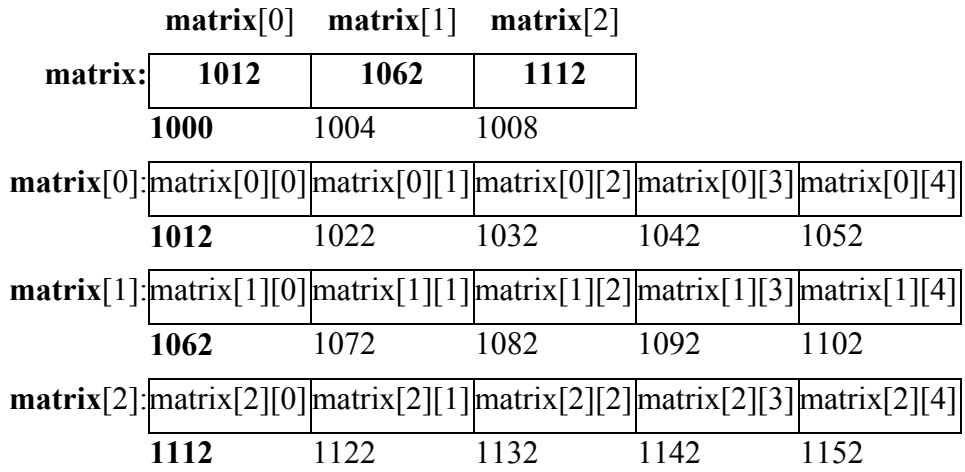
- A **matrix** konstans mutató, mely az **m** elemű, **matrix**[], konstans mutatótömb kezdőcímét tartalmazza (tehát **matrix**[0]-ét).
- A **matrix**+**i** e tömb **i**. elemének címe. E tömb elemeinek tartalmát megszemlélve láthatjuk, hogy **n long double** típusú változó méretével térnek el egymástól.
- A **matrix**+**i** cím tartalma ugye ***(matrix**+**i**) vagy **matrix**[**i**].
- A **matrix**[**i**] tehát az **i**-edik, **n** darab **long double** elemből álló tömb azonosítója: konstans mutató, mely az **i**-edik, **n long double** elemű tömb kezdetére mutat. A **matrix** konstans mutató e konstans mutatótömb kezdőcímét tartalmazza.
- A **matrix**[**i**]+**j** vagy ***(matrix**+**i**)+**j** az **i**–edik, **n long double** elemű tömb **j**-edik elemének címe. E cím tartalma elérhető a következő hivatkozásokkal:

  ```
  matrix[i][j], *(matrix[i]+j) vagy *(*(matrix+i)+j).
  ```

- A &**matrix**[0][0], a **matrix**[0], a &**matrix**[0] és a **matrix** ugyanaz az érték, azaz a mátrix kezdetének címe, de a **matrix**[0][0] egy **long double** azonosítója, a **matrix**[0] egy **n long double** elemű tömb azonosítója, és a **matrix** a tömbökből álló tömb azonosítója. Így:

  ```
  &matrix[0][0] + 1 ≡ &matrix[0][1],
  matrix[0] + 1 ≡ *matrix + 1 ≡ &matrix[0][1],
  &matrix[0] + 1 ≡ matrix + 1 és
  matrix + 1 ≡ &matrix[1] ≡ &matrix[1][0].
  ```

32 bites címeket feltételezve, 1000–ről indulva, **m**=3 és **n**=5 esetén a **matrix** a következőképpen helyezkedhet el a memóriában:



☞ A háromdimenziós tömböt úgy valósítja meg a fordító, hogy létrehoz előbb egy mutatótömböt, melynek mindenegyes eleme egy, az előzőekben ismertetett szerkezetű mátrixra mutat.

☞ Néhány szó még a többszörösen alkalmazott index operátorról! A *kifejezés1*[*kifejezés2*][*kifejezés3*]...–ban az index operátorok balról jobbra kötnek, így a fordító először a legbaloldalibb *kifejezés1*[*kifejezés2*] kifejezést értékeli ki. A született mutató értékhez aztán hozzáadva *kifejezés3* értékét új mutató kifejezést képez, s ezt a folyamatot a legjobboldalibb index kifejezés összegzéséig végzi. Ha a végső mutató érték nem tömb típust címez, akkor az indirekció művelete következik. Tehát például:

```
matrix[2][3] ≡ (*(matrix+2))[3] ≡ *(*(matrix+2)+3)
```

9.8 Tömbök, mint függvényparaméterek

Ha van egy

```
float vektor[100];
```

tömbünk, és kezdőcímével meghívjuk az

```
fv(vektor)
```

függvényt, akkor a függvény definíciójának

```
void Fv(float *v) { /* . . . */ }
```

vagy

```
void Fv(float v[]){ /* . . . */ }
```

módon kell kinéznie.

☞ Az utóbbi alakról tudjuk, hogy a fordító rögtön és automatikusan átkonvertálja az előző (a mutatós) formára.

✍ Ne feledjük, hogy ugyan a **vektor** konstans mutató a hívó függvényben, de **v** (címmásolat) már változó mutató a meghívott függvényben. Vele tehát elvégezhető például a **v**++ művelet. A ***v**, a ***(v+i)** vagy a **v[i]** balérték alkalmazásával a **vektor** tömb bármelyik eleme módosítható a meghívott függvényben. Emlékezzünk arra is, hogy a meghívott függvényt is tájékoztatni kell valahogyan a tömb méretéről. Például úgy, hogy a méretet is átadjuk paraméterként. Az itt elmondottak többdimenziós tömbök vonatkozásában is igazak, de ott már nem ismételjük meg!

Ha van egy

```
float matrix[10][20];
```

definíciónk, és az azonosítóval meghívjuk

```
Fvm(matrix)
```

módon az **Fvm** függvényt, akkor hogyan kell az **Fvm** definíciójának kinéznie? A

```
void Fvm(float **m) { /* . . . */ }
```

próbálkozás rossz, mert a formális paraméter **float** mutatóra mutató mutató. A

```
void Fvm(float *m[20]) { /* . . . */ }
```

változat sem jó, mert így a formális paraméter 20 elemű, **float** típusú objektumokra mutató mutatótömb. Ennél a kísérletnél az az igazi probléma, hogy a [] operátor prioritása nagyobb, mint *-é. Nekünk formális paraméterként 20 **float** elemű tömbre mutató mutatót kéne átadni. Tehát a helyes megoldás:

```
void Fvm(float (*m)[20]) { /* . . . */ }
```

vagy a „tradicionális” módszer szerint:

```
void Fvm(float m[][20]) { /* . . . */ }
```

, amiből rögtön és automatikusan előállítja a fordító az előző (a mutatós) alakot.

💣 Meg kell még említenünk, hogyha a többdimenziós tömböt dinamikusan hozzuk létre, akkor az előzőleg ajánlott megoldás nyilvánvalóan helytelen. A mátrix „horgonypontját” ebben az esetben

```
float **matrix;
```

módon definiáljuk, ami ugye **float** mutatóra mutató mutató. Tehát ilyenkor a

```
Fvmd(matrix)
```

módon hívott **Fvmd** függvény helyes formális paramétere:

```
void Fvmd(float **m) { /* . . . */ }
```

Megoldandó feladatok:

Készítsen programot két mátrix összeadására! A mátrixoknak ne dinamikusan foglaljon helyet a memóriában! A mátrixok mérete azonban csak futás időben válik konkréttá.

Írjon szoftvert két mátrix összeadására és szorzására! A mátrixok mérete itt is futás közben dől el! A programban használjon függvényeket

- a mátrix méretének
- és elemeinek bekéréséhez, valamint
- a két mátrix összeszorzásához!

A két utóbbi függvény paraméterként kapja meg a mátrixokat!

9.9 Parancssori paraméterek

Minden C programban kell lennie egy a programot elindító függvénynek, mely konzol bázisú alkalmazások esetében a **main** függvény.

Most és itt csak a **main** paramétereivel és visszatérési értékével szeretnénk foglalkozni! A paraméterekről állíthatjuk, hogy:

- elhagyhatóak és
- nem ANSI szabványosak.

A **main** legáltalánosabb alakja:

```
int main(int argc, char *argv[]);
```

📖 A paraméterek azonosítói bizonyos, C nyelvet támogató környezetekben ettől el is térhetnek, de funkciójuk akkor is változatlan marad.

Az **argc** a **main**-nek átadott parancssori paraméterek száma, melyben az indított végrehajtandó program azonosítója is benne van, és értéke így legalább 1.

Az **argv** a paraméter karakterláncokra mutató mutatótömb, ahol az egyes elemek rendre:

- **argv[**0**]:** A futó program (meghajtónévvel és) úttal ellátott azonosítójára mutató mutató.
- **argv[**1**]:** Az első parancssori paraméter karakterláncára mutató mutató.

- **argv[**2**]:** Az második paraméter karakterlánc kezdőcíme.
- **. . .**
- **argv[argc** - 1**]:** Az utolsó parancssori paraméter karakterláncára mutató mutató.
- **argv[argc]: NULL** mutató.

☞ Megjegyezzük, hogy az **argc** és az **argv main** paraméterek elérhetők az

```
extern int _argc;
extern char **_argv;
```

globális változókon át is (**STDLIB.H**)!

A **main** lehetséges alakjai a következők:

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

Vegyünk egy példát!

```
/* PELDA26.C: Parancssori paraméterek. */
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]){
  int i=0;
  printf("Parancssori paraméterek:\n");
  printf("Argc értéke %d.\n", argc);
  printf("Az átadott parancssori paraméterek:\n");
  for(i=0; i<argc; ++i)
    printf("\targv[%d]: %s\n", i, *argv++);
  return 0; }
```

Tételezzük fel, hogy a programot a következő parancssorral indítottuk:

```
PELDA26 elso_par "sodik par" 3 4 stop!
```

Ekkor a megjelenő kimenet a következő lehet:

```
Parancssori paraméterek:
Argc értéke 6.
Az átadott parancssori paraméterek:
  argv[0]: C:\C\PELDA26.EXE
  argv[1]: elso_par
  argv[2]: sodik par
  argv[3]: 3
  argv[4]: 4
  argv[5]: stop!
```

✍ Beszéljünk kicsit a **printf** utolsó, ***argv++** kifejezéséről! Az **argv**–t a **main** paraméterként kapja, tehát csak címmásolat, vagyis a **main**–ben akár el is rontható. Az **argv** típusa **char ****, és funkcionálisan a parancssori paraméter karakterláncok kezdőcímeit tartalmazó mutatótömb kezdetének címe. A rajta végrehajtott indirekcióval a típus **char *** lesz, s épp a mutatótömb első elemét (**argv**[0]) érjük el. Az utótag ++ operátor miatt eközben az **argv** már a második mutatótömb elemre (**argv**[1]) mutat. Elérjük ezt is, és mellékhatásként az **argv** megint előbbre mutat egy tömbelemmel. Tehát a ciklusban rendre végigjárjuk az összes parancssori paramétert.

💣 Jusson eszünkbe, hogy a **main**-nek átadott parancssor maximális hosszát korlátozhatja az operációs rendszer!

A legtöbb operációs rendszerben léteznek

változó=érték

alakú, ún. környezeti változók, melyek definiálják a környezetet (információt szolgáltatnak) az operációs rendszer és a benne futó programok számára. Például a **PATH** környezeti változó szokta tartalmazni az alapértelmezett keresési utakat a végrehajtható programokhoz, a parancsinterpreter helyét írja elő a **COMSPEC**, és így tovább. Az operációs rendszer természetesen lehetőséget biztosít ilyen környezeti változók törlésére, megadására, és *érték*ük módosítására.

C–ből a környezeti változók általában az **STDLIB.H**–ban deklarált, nem ANSI szabványos

extern char ****_environ**;

globális változóval érhetők el. Ez karakterláncokra mutató mutatótömb, és a mutatott karakterláncok a *változó=érték* alakú környezeti változókat írják le. Az utolsó utáni környezeti változó karakterláncára mutató tömbelem itt is **NULL** mutató, mint az **argv**–nél.

✍ A globális változó nevét azért nem árt pontosítani a programfejlesztő rendszer segítségéből!

A másik lehetőség az **STDLIB.H**–ban deklarált, szabványos, nem kis–nagybetű érzékeny

char ***getenv**(const char **valtozo*);

függvény, mely visszaad az aktuális környezet alapján a *valtozo* nevű környezeti változó értékére mutató mutatót. Ha nincs ilyen változó az aktuális környezeti táblában, **NULL** mutatót kapunk. A visszakapott nem **NULL** mutatóval azonban nem célszerű és nem biztonságos dolog a kör-

nyezeti változó értékét módosítani. Ehhez a nem szabványos **putenv** rutin használata ajánlott.

💣 A környezeti változó nevének a végére nem kell kitenni az = jelet, azaz például a **PATH** környezeti változót a **getenv**("PATH") hívással kérdezhetjük le!

Megoldandó feladatok:

Készítsen programot, mely a **JANI**, fordítási időben változtatható azonosítójú, környezeti változóról megállapítja, hogy létezik–e! Ha létezik, akkor eldönti, hogy értéke „kicsi", „nagy", vagy más. A feladat fokozható egyrészt úgy, hogy a változó lehetséges értékei is legyenek fordítási időben módosíthatók, másrészt úgy, hogy ne rögzítsük kettőben a lehetséges értékek darabszámát!

Írjon szoftvert, mely a környezeti változó azonosítóját és lehetséges értékeit parancssori paraméterekként kapja meg, és megállapításai az előző példában megfogalmazottakkal azonosak! Ha a programot paraméter nélkül indítják, akkor tájékoztasson használatáról!

Ha expliciten nem deklaráljuk **void**-nak, akkor a **main**-nek **int** típusú státuszkóddal kell visszatérnie az őt indító programhoz (process), rendszerint az operációs rendszerhez. Konvenció szerint a zérus visszaadott érték (**EXIT_SUCCESS**) hibátlan futást, s a nem zérus státuszkód (**EXIT_FAILURE** 1) valamilyen hibát jelez. Magát a **main**-ből való visszatérést (mondjuk 1–es státuszkóddal) megoldhatjuk a következő módok egyikével:

```
return 1;
exit(1);
```

Foglalkozzunk kicsit a programbefejezéssel is!

9.9.1 Programbefejezés

A **return** 1 csak a **main**–ben kiadva fejezi be a program futását. Az **STDLIB.H** bekapcsolásakor rendelkezésre álló, mindegyik operációs rendszerben használható

void **exit**(int *statusz*);

void **abort**(void);

függvények mind befejezik annak a programnak a futását, amiben meghívják őket akármilyen mély rutin szintről is. A *statusz* paraméter értékét visszakapja a befejezettet indító (várakozó szülő) program, mint kilépési állapotot (exit status). A *statusz* értéket átveszi persze az operációs rend-

szer is, ha ő volt a befejezett program indítója. Zérus (**EXIT_SUCCESS**) állapottal szokás jelezni a normál befejezést. A nem zérus állapot valamilyen hibát közöl (**EXIT_FAILURE** 1).

Az **exit** függvény a program befejezése előtt meghív minden regisztrált (lásd **atexit**!) kilépési függvényt, kiüríti a kimeneti puffereket, és lezárja a nyitott fájlokat.

Az **abort** alapértelmezés szerint befejezi az aktuális programot. Megjelenteti például az

```
Abnormal program termination
```

üzenetet az **stderr**–en, és aztán **SIGABRT** (abnormális programbefejezés) jelet generál. Ha nem írtak kezelőt (**signal**) a **SIGABRT** számára, akkor az alapértelmezett tevékenység szerint az **abort** 3–as státuszkóddal visszaadja a vezérlést a szülő programnak. Szóval nem üríti a puffereket, és nem hív meg semmilyen kilépési függvényt (**atexit**) sem.

✍ Az **stderr** a szabvány hibakimenet. Az **atexit** és a **signal** függvényekről rögtön szó lesz a következő fejezetben!

Megoldandó feladatok:

Készítsen programot, mely neveket olvas a szabvány bemenetről! Egy sorban egy név érkezik, s az üres sor a bemenet végét jelzi. A név nagybetűvel kezdődik, és a többi karaktere kisbetű. A feltételeket ki nem elégítő név helyett azonnal másikat kell kérni a probléma kijelzése után! A neveket rendezze névsorba, s listázza ki őket lapokra bontva! A feladat a következőképp fokozható:

- Ha a névben az angol ábécébelieken kívül az ékezetes kis és nagybetűk is megengedettek.
- Ha a neveket közlő listán előre–hátra lehet lapozni.
- Ha egy nevet csak egyszer lehet megadni, azaz a második bevitelt elutasítja hibaként a szoftver.
- Ha a programot „–v” parancssori paraméterrel indítják, akkor a rendezés visszafelé halad a névsoron.
- Ha a neveknek dinamikusan foglal helyet, kezdőcímeiket mutatótömbben helyezi el, és a rendezésnél a mutatótömb elemeket cserélgeti, s nem a név karakterláncokat a szoftver.

9.10 Függvény (kód) mutatók

A mutatók függvények ún. belépési pontjának címét is tartalmazhatják, s ilyenkor függvény vagy kódmutatókról beszélünk.

Ha van egy

```
int fv(double, int);
```

prototípusú függvényünk, akkor erre mutató mutatót

```
int (*pfv)(double, int);
```

módon deklarálhatunk. A **pfv** azonosító ezek után olyan változót deklarál, melyben egy **double**, s egy **int** paramétert fogadó és **int**-tel visszatérő függvények címeit tarthatjuk. A **pfv** tehát változó kódmutató. Kódmutató konstans is létezik azonban, s ez a függvénynév (a példában az **fv**).

💣 Vigyázzunk a deklarációban a függvénymutató körüli kerek zárójel pár el nem hagyhatóságára, mert az

```
int *pfv(double, int);
```

olyan **pfv** azonosítójú függvényt deklarál, mely egy **double**, s egy **int** paramétert fogad, és **int** típusú objektumra mutató mutatóval tér vissza. A probléma az, hogy a mutatóképző operátor (*) prioritása alacsonyabb a függvényképzőénél (()).

Hogyan lehet értéket adni a függvénymutatónak?

Természetesen a szokásos módokon, azaz hozzárendeléssel:

```
pfv = fv;
```

, ill. a deklarációban inicializátor alkalmazásával:

```
int fv(double, int);
int (*pfv)(double, int) = fv;
```

💣 Vigyázzunk nagyon a típussal, mert az most „bonyolódott”! Csak olyan függvény címe tehető be a **pfv**-be, mely a függvénymutatóval egyező típusú, azaz **int**-et ad vissza, egy **double** és egy **int** paramétert fogad ebben a sorrendben. A

```
void (*mfv)();
```

szerint az **mfv** meg nem határozott számú és típusú paramétert fogadó olyan függvényre mutató mutató, melynek nincs visszatérési értéke.

✍ Vegyük észre, hogy a kérdéses függvények definíciója előtt függvénymutatók inicializálására is használható a megadott függvény prototípus!

Hogyan hívhatjuk meg azt a függvényt, melynek címét a kódmutató tartalmazza?

Alkalmaznunk kell a mutatókra vonatkozó ökölszabályunkat, ami azt mondja ki, hogy ahol állhat *azonosító* a kifejezésben, ott állhat (**mutatóazonosító*) is. Vegyük elő újra az előző példát! Ha

```
int a = fv(0.65, 8);
```

az **fv** függvény hívása, és valamilyen módon lezajlott a **pfv** = **fv** hozzárendelés is, akkor az

```
a = (*pfv)(0.65, 8);
```

ugyanaz a függvényhívás.

Itt a **pfv**-re alkalmaztuk az indirekció operátort (*), de mivel ennek prioritása alacsonyabb a függvényhívás operátorénál (()), ezért a ***pfv**-t külön zárójelbe kellett tenni!

A kódmutatóval kapcsolatos alapismeretek letárgyalása után feltétlenül ismertetni kell a C fordító függvényekkel kapcsolatos fontos viselkedését: implicit konverzióját!

Ha a kifejezés *típus*sal visszatérő függvény típusú, akkor hacsak nem cím operátor (&) mögött áll, *típus*sal visszatérő függvénymutató típusúvá konvertálja automatikusan és azonnal a fordító.

Ez az implicit konverzió mindenek előtt megvalósul a

utótag-kifejezés(<*kifejezéslista*>)

függvényhívásban, ahol az *utótag-kifejezés*nek kell *típus*sal visszatérő függvénycímmé kiértékelhetőnek lennie. A *típus* a függvényhívás értékének típusa. A dolog praktikusan azt jelenti, hogy a függvény bármilyen függvényre mutató kifejezéssel meghívható.

Milyen műveletek végezhetők a kódmutatókkal?

- Képezhető a címük.
- **sizeof** operátor operandusai lehetnek.
- Végrehajtható rajtuk az indirekció művelete is, mint láttuk.
- Értéket kaphatnak, ahogyan azt az előzőekben ismertettük.
- Meghívhatók velük függvények. Ezt is áttekintettük.
- Átadhatók paraméterként függvényeknek.
- Kódmutatótömbök is létrehozhatók.

- Függvény visszaadott értéke is lehet.
- Explicit típuskonverzióval más típusú függvénymutatókká alakíthatók.

💣 Kódmutatókra azonban nem alkalmazható a mutatóaritmetika az egyenlőségi reláció operátoroktól (== és !=) eltekintve.

Foglalkozzunk a kódmutató paraméterrel!

🕮 A függvényekre érvényes implicit típuskonverzió a függvényparaméterekre is vonatkozik. Ha a paraméter *típus*sal visszatérő függvény, akkor a fordító automatikusan és rögtön *típus* típusú értéket szolgáltató függvényre mutató mutatóvá alakítja át.

✍ A következő, kissé elvonatkoztatott példában kitűnően megszemlélhető a kódmutató paraméter függvény prototípusban, ill. függvény aktuális és formális paramétereként.

```
/* . . . */
long Emel(int);
long Lep(int);
long Letesz(int);
void Munka(int n, long (* fv)(int));
/* . . . */
void main(void){
  int valaszt=1, n;
  /* . . . */
  switch(valaszt){
    case 1: Munka(n, Emel); break;
    case 2: Munka(n, Lep); break;
    case 3: Munka(n, Letesz); }
  /* . . . */ }
void Munka(int n, long (* fv)(int)){
  int i;
  long j;
  for(i=j=0; i<n; ++i) j+=(*fv)(i); }
```

✍ A kódmutató típusa szerint az ilyen függvény egy **int** paramétert fogad, és **long** értéket szolgáltat.

Kódmutatók paraméterként való átadását a **Programbefejezés** fejezetben már megemlített, de ott nem tárgyalt

9.10.1 atexit függvény

leírásával is szemléltetjük!

#include <STDLIB.H>

int **atexit**(void (cdecl * *fv*)(void));

Az **atexit** regisztrálja a paraméter függvénycímet, s normál programbefejezéskor az **exit** meghívja az *fv*-t a szülő programhoz való visszatérés előtt.

✍ Az *fv* függvénymutató paraméterből látszik, hogy a kilépési függvényeknek nincs paraméterük, és nem adnak vissza értéket.

Az **atexit** mindenegyes hívásával más-más kilépési függvényt regisztráltathatunk. A regisztrálás veremszerű, azaz a legutoljára regisztrált függvényt hajtja végre először a rendszer, s aztán így tovább visszafelé. Az **atexit** a heap–et használja a függvények regisztrálásához, s így a regisztrálható kilépési függvények számát csak a heap mérete korlátozza.

Az **atexit** sikeres híváskor zérust ad vissza, és nem zérust csak akkor kapunk tőle, ha már nem tud több függvényt feljegyezni. Például:

```
#include <stdio.h>
#include <stdlib.h>
void cdecl exitfv1(void){
  printf("Exitfv1 végrehajtva!\n"); }
void cdecl exitfv2(void){
  printf("Exitfv2 végrehajtva!\n"); }
int main(void){
  atexit(exitfv1);
  atexit(exitfv2);
  printf("A main befejeződött.\n");
  return 0; }
```

A szabvány kimenet a következő:

```
A main befejeződött.
Exitfv2 végrehajtva!
Exitfv1 végrehajtva!
```

Folytassuk tovább a kódmutatók tárgyalását!

Azt mondottuk, hogy kódmutatók tömbökben is elhelyezhetők. Visszatérve az első **pfv**-s példánkhoz! Az

```
int (*pfvt[])(double, int) = {fv1, fv2, fv3, fv4, fv5};
```

deklarációval létrehoztunk egy **pfvt** azonosítójú, olyan ötelemű tömböt, mely **int**-et visszaadó, egy **double,** és egy **int** paramétert fogadó függvények címeit tartalmazhatja. Feltéve, hogy **fv1**, **fv2**, **fv3**, **fv4** és **fv5** ilyen prototípusú függvények, a **pfvt** tömb elemeit kezdőértékkel is elláttuk ebben a deklarációban.

Hívjuk még meg, mondjuk, a tömb 3. elemét!

```
a = (*pfvt[2])(0.65, 8);
```

Alakítsuk át függvénymutató tömböt használóvá a kódmutató paraméternél ismertetett példát!

Az új megoldásunk **main**–en kívüli része változatlan, a **main** viszont:

```
void main(void){
  int valaszt=1, n;
  long (*fvmt[])(int) = {Emel, Lep, Letesz};
  /* . . . */
  Munka(n, fvmt[valaszt]);
  /* . . . */ }
```

A kódmutató visszatérési értékhez elemezzük ki a következő függvény prototípust!

void (***signal**(int *jel***,** void (***** *kezelo*)(int)))(int**);**

A **signal** első paramétere **int**, a második (**void** (***** *kezelo*)(**int**)) viszont értéket vissza nem adó, egy **int** paramétert fogadó függvénymutató típusú. Kitűnően látszik ezek után, hogy a visszatérési érték **void** (*****)(**int**), azaz értéket nem szolgáltató, egy **int** paramétert fogadó függvénymutató. A visszaadott érték típusa tehát a **signal** második paraméterének típusával egyezik.

📖 A **SIGNAL.H** fejfájlban elhelyezett prototípusú **signal** függvénnyel különben a program végrehajtása során bekövetkező, váratlan eseményeket (megszakítás, kivétel, hiba stb.), ún. jeleket lehet lekezeltetni. Többféle típusú jel létezik. A **void** (*****)(**int**) típusú kezelőfüggvényt a manipulálni kívánt jelre külön meg kell írni. A **signal** rutinnal hozzárendeltetjük a kezelőt (2. paraméter) az első paraméter típusú jelhez, és a **signal** az eddigi kezelő címével tér vissza.

Egy bizonyos típusú függvénymutató explicit típuskonverzióval

(*típusnév*) *előtag-kifejezés*

átalakítható más típusú kódmutatóvá. Ha az e módon átkonvertált mutatóval függvényt hívunk, akkor a hatás a programfejlesztő rendszertől, a hardvertől függ. Viszont, ha visszakonvertáljuk az átalakított mutatót az eredeti típusra, akkor az eredmény azonos lesz a kiindulási függvénymutatóval.

Szedjük csak megint elő az

```
int fv(double, int), a;
int (*pfv)(double, int) = fv;
```

példánkat, és legyen a

```
void (*vpfv)(double);
vpfv=(void (*)(double))pfv;
```

A

```
(*vpfv)(0.65);
```

eredményessége eléggé megkérdőjelezhető, de a

```
pfv=(int (*)(double, int))vpfv;
```

után teljesen rendben lesz a dolog:

```
a=(*pfv)(0.65, 8);
```

☞ Emlékezzünk csak! Explicit típusmódosított kifejezés nem lehet balérték.

Foglalkozzunk csak újra egy kicsit a *típusnev*ekkel!

9.10.2 Típusnév

Explicit típusmódosításban, függvénydeklarátorban a paramétertípus rögzítésekor, **sizeof** operandusaként stb. szükség lehet a típus nevének megadására. Ehhez kell a típusnév, mely szintaktikailag a kérdéses típusú objektum olyan deklarációja, melyből elhagyták az objektum azonosítóját.

típusnév:
  típusspecifikátor-lista<*absztrakt-deklarátor*>
absztrakt-deklarátor:
  mutató
  <*mutató*><*direkt-absztrakt-deklarátor*>
direkt-absztrakt-deklarátor:
  (*absztrakt-deklarátor*)
  <*direkt-absztrakt-deklarátor*>[<*konstans-kifejezés*>]
  <*direkt-absztrakt-deklarátor*>(<*paraméter-típus-lista*>)

Az *absztrak-deklarátor*ban mindig lokalizálható az a hely, ahol az azonosítónak kéne lennie, ha a konstrukció deklaráción belüli deklarátor lenne.

Lássunk néhány konkrét példát!

```
int *
```

int típusú objektumra mutató mutató.

```
int **
```

int típusú objektumra mutató mutatóra mutató mutató.

```
int *[]
```

Nem meghatározott elemszámú, **int** típusú mutatótömb.

```
int *()
```

Ismeretlen paraméterlistájú, **int**–re mutató mutatóval visszatérő függvény.

```
int (*[])(int)
```

int típussal visszatérő, egy **int** paraméteres, meghatározatlan elemszámú függvénymutató tömb.

```
int (*(*())[])(void)
```

Ismeretlen paraméterezésű, **int** típussal visszatérő függvénymutatókból képzett, meghatározatlan méretű tömbre mutató mutatót szolgáltató, paraméterrel nem rendelkező függvény.

A problémán: a sok zárójelen, a nehezen érthetőségen **typedef** alkalmazásával lehet segíteni.

9.11 Típusdefiníció (typedef)

Az elemi típusdefinícióról szó volt már a **TÍPUSOK ÉS KONSTANSOK** szakasz végén. Az ott elmondottakat nem ismételjük meg, viszont annyit újra szeretnénk tisztázni, hogy:

📖 A típusdefiníció nem vezet be új típust, csak más módon megadott típusok szinonimáit állítja elő. A **typedef** név, ami egy *azonosító*, szintaktikailag egyenértékű a típust leíró kulcsszavakkal, vagy típusnévvel.

A típusdefiníció „bonyolításához” először azt említjük meg, hogy a

typedef *típus azonosító*;

szerkezetben az *azonosító* a prioritás sorrendjében lehet:

- *azonosító*(): függvény típust képző, utótag operátor. Például:

  ```
  typedef double dfvdi(double, int);
  dfvdi hatvany;
  ```

  , ahol a **hatvany** egy **double**, s egy **int** paramétert fogadó és **double**-t visszaadó függvény azonosítója.

- *azonosító*[]: tömb típust képző, utótag operátor. Például:

  ```
  typedef double dtomb[20];
  dtomb t;
  ```

  , ahol a **t** 20 **double** elemből álló tömb azonosítója.

- **azonosító*: mutató típust képző, előtag operátor. Például:

```
typedef short *shptr;
shptr sptr;
```

, ahol az **sptr short** típusú objektumra mutató mutató azonosítója.

- Ezek az operátorok egyszerre is előfordulhatnak az *azonosító*-val. Például:

```
typedef int *itb[10];
itb tomb;
```

, ahol a **tomb** 10 elemű **int** objektumokra mutató mutatótömb azonosítója.

☞ Megemlítendő még, hogy az előzőek alkalmazásával „csínján” kell bánni, hisz az így típusdefiniált azonosítóknak éppen a jellege (függvény, tömb, mutató) veszik el.

A típusdefiníció további komplexitása abból fakad, hogy a

typedef *típus azonosító*;

szerkezetbeli *típus* korábbi **typedef** *típus azonosító*;-ban definiált *azonosító* is lehet. Tehát a típusdefinícióval létrehozott típuspecifikátor *típus* lehet egy másik típusdefinícióban.

Nézzünk néhány példát!

```
typedef int *iptr;
typedef char nev[30];
typedef enum {no, ferfi, egyeb} sex;
typedef long double *ldptr;
ldptr ptr2;   /* long double objektumra mutató mutató.*/
ldptr fv2(nev);/*30 elemű karaktertömb paramétert fogadó,
                 long double objektumra mutató mutatóval
                 visszatérő függvény. */
typedef iptr (*ipfvi)(sex);
ipfvi fvp1;   /* int-re mutató mutatót visszaadó, egy
                 sex típusú enum paramétert fogadó
                 függvényre mutató mutató. */
typedef ipfvi ptomb[5];
ptomb tomb;   /* 5 elemű, int-re mutató mutatót
                 szolgáltató, egy sex típusú paramétert
                 fogadó függvényre mutató mutatótömb. */
iptr fugg(ptomb);/*int-re mutató mutatót visszaadó, 5
                 elemű, int-re mutató mutatóval
                 visszatérő, egy sex enum paraméteres
                 függvényre mutató mutatótömböt
                 paraméterként fogadó függvény. */
```

✍ A típusdefinícióval a program típusai parametrizálhatók, azaz a program portábilisabb lesz, hisz az egyetlen **typedef** módosításával a típus megváltoztatható. A komplex típusokra megadott **typedef** nevek ezen kívül javítják a program olvashatóságát is.

💣 Lokális szinten megadott típusdefiníció lokális hatáskörű is. Az általánosan használt típusdefiníciókat globálisan, a feladathoz tartozó fejfájlban szokták előírni.

Egy utolsó kérdés: Mikor ekvivalens két típus?

- Ha a két *típusspecifikátor-lista* egyezik, beleértve azt is, hogy ugyanaz a *típusspecifikátor* többféleképpen is megadható. Például: a **long**, a **long int** és a **signed long int** azonosak.
- Ha az *absztrakt-deklarátor*aik a **typedef** típusok kifejtése, és bármely függvényparaméter azonosító törlése után ekvivalens *típusspecifikátor-listá*kat eredményeznek.

💣 A típusekvivalencia meghatározásánál a tömbméretek és a függvényparaméter típusok is lényegesek.

9.12 Ellenőrzött bemenet

Jegyzetünkben minden feladat megoldásában azt sugalltuk, hogy:

👍 A programnak ellenőriznie kell a bemenetét. Ez a vizsgálat természetesen csak a konkrét adatok ismerete nélkül a lehetetlenségek, és a problémát okozó értékek kiszűrésére szorítkozhat. Például: Ne etessünk két tonnás kutyát! Ne folyósítsunk nyugdíjat 300 éves embernek!

Nem tekinthető, csak legfeljebb műkedvelő, programnak az, ami egy véletlenül elgépelt információ miatt „feldobja a talpát”!

Írjon **int getint**(**int** *) függvényt, mely ellenőrzötten beolvas egy egész számot a billentyűzetről úgy, hogy a nem megengedett karaktereket nem is echózza a karakteres képernyőn (ablakban). A szám előtti fehér karakterek közül az Enter–t soremeléssel, és minden más fehér karakter szóközzel echózandó! Ezután egy opcionális előjelet követően már csak számjegy karakterek következhetnek. Ha az első számjegy zérus, akkor további szám karakterek sem jöhetnek. A függvény legyen portábilis, azaz működjön 16 és 32 bites **int**–re egyaránt! Ez azt jelenti 16 bites esetre, hogy legfeljebb öt számjegyet echózhat, de legyen tekintettel az ábrázolási korlátokra is! Ha az első 4 számjegy 3276–nál nagyobb, akkor több számjegyet már nem fogadhat. Ha az első 4 számjegy pontosan 3276, akkor ötödik számjegyként nem fogadhatja a függvény a 9–et, s a 8–at is

csak negatív egész szám esetén. A szám megadását fehér karakterrel, Ctrl+Z–vel vagy F6–tal kell lezárni. A fehér karaktert a már leírt módosított echó után vissza kell adni a hívónak. Ctrl+Z vagy F6 esetén viszont EOF szolgáltatandó. A beolvasott egész érték konvertálandó és elhelyezendő a paraméter címen! A címen levő érték azonban nem változhat meg, ha nem adtak meg egyetlen számjegy karaktert sem.

A rutin persze rövid programmal ki is próbálandó!

A feladat megoldásához szükségünk van két konzolkezelő függvényre. Az

int **putch**(int *c*);

rutin a *c* karaktert írja ki közvetlenül (pufferezés nélkül) a konzol képernyőre (ablakba) az aktuális pozíciótól, aktuális színben és megjelenési attribútumokkal, s a kurzort eggyel előbbre állítja. Sikeres esetben a visszakapott érték maga a *c* karakter. A sikertelenséget viszont **EOF**-fal jelzi a függvény.

Kivitelkor nincs transzláció, azaz a függvény az LF ('\n') karakterből nem állít elő CR-LF ("\r\n") karakter párt.

☞ Megoldásunkban nem foglalkozunk majd a **putch** hibakezelésével, mert feltételezzük, hogyha az operációs rendszer működik, akkor a konzol megy.

int **getch**(void);

Echó nélkül beolvas egyetlen karaktert a konzolról (billentyűzet), s ezt szolgáltatja a hívónak. A bejövő karakter rögtön rendelkezésre áll, s nincs pufferezés soremelés karakterig. Funkció vagy nyíl billentyű leütésekor a függvényt kétszer kell hívni, mert az első hívás zérussal tér vissza, s a második szolgáltatja az aktuális gomb kódját. A rutinnak nincs hibás visszatérése.

☞ E függvények nem szabványosak, de szinte minden operációs rendszerben rendelkezésre állnak kisebb–nagyobb eltérésekkel a **CONIO.H** fejfájl bekapcsolása után.

```
/* PELDA27.C: Egészek beolvasása és visszaírása úgy, hogy
   az érvénytelen karakterek echója meg sem történik.*/
#include <stdio.h>
#include <conio.h>
#include <limits.h>
#if SHRT_MAX == INT_MAX
  #define HOSSZ 4
#else
```

```
  #define HOSSZ 9
#endif
#define F6 64
#define CTRLZ 26
#define MAX INT_MAX/10
#define HATAR INT_MAX%10+'0'+1
```

A **HOSSZ** makró azt a számjegy mennyiséget rögzíti 16, és 32 bites **int**–re, ameddig még nem kell foglalkozni a megadott szám karakter értékével.

A **MAX** maga az a **HOSSZ** számjegyű érték, amihez még egy jegyet téve elérhető, de túl nem léphető a felső, vagy az alsó ábrázolási korlát. 16 bites **int**–nél ez az érték 3276, amihez pozitív irányban legfeljebb 7, s negatív irányban maximum 8 jöhet.

A **HATAR** az a számjegy karakter, ami még negatív egész esetében előfordulhat **MAX**–ot követően megadható karakterként a **HOSSZ**+1. pozíción. 16 bites **int** számára ez az érték ’8’.

```
int getint(int *pn){/* Egész beolvasása a bemenetről. */
  int c,    /* A beolvasott karakter. */
  sign=1,   /* Előjel: pozitív +1, negatív -1. Alapér-
               telmezés a pozitív, a kezdőérték miatt. */
  elojel=0, /* Volt-e már előjel? */
  hossz=0,  /* A beolvasott számjegy karakterek száma. */
  null=0;   /* A beolvasott szám zérus-e? */
  while(!hossz) switch(c=getch()){
    case ' ':
    case '\t':
    case '\r': if(!elojel)
               if(c!='\r')putch(' ');
               else {putch('\n'); putch('\r'); }
               break;
    case '+':
    case '-': if(!elojel){
               putch(c); sign=(c=='+')?1:-1; ++elojel; }
               break;
    case '0': if(!elojel){
               putch(c); *pn=0; ++hossz; null=1;}
               break;
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
               putch(c); *pn=c-'0'; ++hossz;
               break;
    default:   if(!c)c=getch();
               if(c==CTRLZ||c==F6) return EOF; }
```

A **getint** lényegében két **while** ciklusra bontható, melyek mindegyike egy–egy **switch**. Az első addig tart, míg

- egy számjegy karaktert meg nem adnak, vagy
- Ctrl+Z–vel, ill. F6–tal le nem zárják a bemenetet.

Az első **switch**:

- Végrehajtja a fehér karakterekre előírt echót, de csak akkor, ha előjel karakter még nem volt. Magyarán előjel után nincs már echó a fehér karakterekre.
- Az előjelet echózza a rutin, ha korábban még nem érkezett, és értékét megjegyzi a **sign** változóban. Bejelöli azt is, hogy volt már előjel, hogy még egyet ne tudjanak megadni.
- Az első szám karaktert echózza a függvény, konvertálva kiteszi a paraméter címre, és a **hossz** változóban számlálja is. Előjel után nem enged már meg zérust gépelni, ill. ha megadható volt a nulla, akkor bejelzi bejövetelét a **null** változóba.
- A **default** ágon újabb olvasás követi az előző zérus beérkezését. Az F6 másként vizsgálható sem lenne.

```
  while(1) switch(c=getch()){
    case ' ':
    case '\t':
    case '\r':if(c!='\r')putch(' ');
              else {putch('\n'); putch('\r'); }
              *pn*=sign; return c;
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
              if(!null&&(hossz<HOSSZ||
             (hossz==HOSSZ&&(*pn<MAX||*pn==MAX&&
             (sign==1&&c<HATAR||sign!=1&&c<=HATAR))))){
              putch(c);
              *pn=*pn*10+c-'0';
              ++hossz; }
            break;
    default:  if(!c)c=getch();
              if(c==CTRLZ||c==F6){
              *pn*=sign;
              return EOF;} } }
void main(void){
  int i;
  printf("Egészek beolvasása CTRL+Z-ig.\n");
  while(getint(&i)!=EOF) printf("%20d\n\r", i); }
```

A második **while**

- fehér karakterrel, vagy
- Ctrl+Z–vel, ill. F6–tal

zárul. A paraméter címen levő konvertált értéket előjel–helyessé teszi a rutin, majd visszatér a ki is echózott, fehér karakterrel, vagy az **EOF**–fal.

A számjegy karakter echója, konverziója és számlálása csak akkor történik meg, ha az első szám nem zérus volt és:

- **HOSSZ**–nál kevesebb karaktert adtak meg eddig, vagy
- épp annyit, de a paraméter címre konvertált érték kisebb **MAX**–nál, ill. pont **MAX** és az utolsó számjegy karakter megfelel az előírt, szigorú feltételeknek.

Megoldandó feladat:

Írja úgy át a **getint** függvényt, hogy a visszatörlés (backspace) gombot funkciója szerinti módon kezelni tudja!

Remélhetőleg mindenki el tudja képzelni, hogy tovább bonyolódna a dolog, ha további szerkesztő billentyűk (Delete, Insert, balra nyíl és jobbra nyíl) használatát is megengednénk, vagy – Uram, bocsá' – lebegőpontos érték bekérését végeznénk és nem egészét.

Magyarán: kitűnően látszik, hogy „többe kerülne a leves, mint a hús". Összefoglalva: A bemenet ellenőrzését sem szabad túlzásba vinni, de az ökölszabálynál írott elveket a jó programnak be kell tartania.

10 STRUKTÚRÁK ÉS UNIÓK

A struktúra és az unió aggregátum. Egy vagy több, esetleg különböző típusú változó (elnevezett tag) együttese, melynek önálló azonosítója van. A struktúrát más nyelvben rekordnak nevezik. Azt teszi lehetővé, hogy a valamilyen módon összetartozó változók egész csoportjára egyetlen névvel hivatkozhassunk, azaz hogy a változócsoport kezelése egyszerűbb legyen.

Tulajdonképpen minden struktúrával és unióval új, összetett, felhasználói adattípust hozunk létre.

Az ANSI szabvány megengedi, hogy a struktúrákat át lehessen másolni egymásba, hozzá lehessen rendelni, és átadhatók legyenek függvényeknek, ill. rutinok visszatérési értéke is lehessen struktúra. Képezhető természetesen a struktúra címe (&), mérete (**sizeof**), és benne lehet explicit típusmódosító szerkezetben is, de

- a struktúrák nem hasonlíthatók össze.

A struktúrában felsorolt változókat struktúratagoknak (member) nevezik. Struktúratag kis megszorításokkal - melyre később kitérünk - akármilyen típusú lehet. Lehet alap és származtatott típusú bármilyen sorrendben.

A *deklaráció*beli *típusspecifikátor* egyik alternatívája a

struktúra-vagy-unió-specifikátor:
  struktúra-vagy-unió<*azonosító*>{*struktúratag-deklarációlista*}
  struktúra-vagy-unió azonosító

struktúra-vagy-unió:
  struct
  union

A *struktúratag-deklarációlista* struktúra, ill. uniótag deklarációk sorozata:

struktúratag-deklarációlista:
  struktúratag-deklaráció
  struktúratag-deklarációlista struktúratag-deklaráció

struktúratag-deklaráció:
  típusspecifikátor-lista struktúra-deklarátorlista

típusspecifikátor-lista:
  típusspecifikátor
  típusspecifikátor-lista típusspecifikátor

struktúra-deklarátorlista:
  struktúra-deklarátor
  struktúra-deklarátorlista, struktúra-deklarátor

A *struktúra-deklarátor* többnyire a struktúra, ill. az unió egy tagjának deklarátora. A struktúratag azonban meghatározott számú bitből is állhat, azaz lehet ún. bitmező (bit field) is, mely a nyelvben struktúrákon kívül nem is használható másutt. A mező bitszélességét a kettőspontot követő, egész értékű *konstans-kifejezés* határozza meg.

struktúra-deklarátor:
deklarátor
<*deklarátor*>**:** *konstans-kifejezés*

☞ A bitmezővel és az unióval még ebben a szakaszban foglalkozunk!

10.1 Struktúradeklaráció

Alakja tehát a következő:

<*tárolási-osztály-specifikátor*> **struct** <*struktúracímke*> <{
struktúratag-deklarációlista }> <*azonosítólista*>;

Például:

```
struct datum{
  int ev, ho, nap, evnap;
  long datumssz;              /* Dátumsorszám. */
  char datumlanc[11]; }
d, dptr, dt[10];              /* Azonosítólista. */
```

, ahol:

- A *tárolási-osztály-specifikátor* elhagyásával, megadásával és ennek értelmezésével nem foglalkozunk újra!
- A **datum** *azonosító* ennek a struktúrának a címkéje (*struktúracímke*), mely azt biztosítja, hogy később **struct datum** módon hivatkozni tudjunk a felhasználói típusra. Például:

  ```
  struct datum *sptr =
     (struct datum *)malloc(sizeof(struct datum));
  ```

- Az **ev**, a **ho**, a **nap** és az **evnap** a **struct datum** típusú struktúra **int** típusú tagjainak, a **datumssz** a **long** típusú tagjának és a **datumlanc** a struktúra **char*** típusú tagjának az azonosítói. A tagneveknek csak a struktúrán belül kell egyedieknek lenniük, azaz a tagazonosítók nyugodtan egyezhetnek például más közönséges változók neveivel, vagy a struktúracímkékkel.
- A **d struct datum** típusú változó, a **dptr** és az **sptr struct datum** típusú objektumra mutató mutatók, és a **dt** tíz, **struct datum** típusú elemből álló tömb azonosítója, azaz a **dt** struktúratömb.

- A fordító a struktúrának éppen annyi helyet foglal a memóriában, hogy benne a struktúra minden tagja elférjen. A struktúratagok a deklaráció sorrendjében, folyamatosan növekvő címeken helyezkednek el, s így a struktúradefinícióban később deklarált tag címe mindig nagyobb.

📖 Az *azonosítólista* nélküli struktúradeklarációt, ahol van *struktúra-tag-deklarációlista*, azaz megadottá válik a struktúra szerkezete, szokás struktúradefinícióna k is nevezni, ugyan nincs memóriafoglalása.

- Minden struktúradeklaráció egyedi struktúra típust hoz létre, s így

```
struct A {
  int i, j;
  double d; } a, a1;
struct B {
  int i, j;
  double d; } b;
```

az **a** és az **a1** objektumok **struct A** típusú struktúrák, de az **a** és a **b** objektumok különböző struktúra típusúak annak ellenére is, hogy a két struktúra szerkezete azonos.

- Az *azonosítólista* nélküli, de *struktúracímké*vel és *tag-deklaráció-listá*val ellátott struktúradefiníció nem foglal ugyan helyet a memóriában, de biztosítja azt a lehetőséget, hogy a struktúradeklaráció hatáskörében később ilyen típusú struktúrával *azonosítólistá*t is megadva helyet foglalhassunk változóinknak, mutatóinknak és tömbjeinknek. Például:

```
struct datum{
  int ev, ho, nap, evnap;
  long datumssz;              /* Dátumsorszám. */
  char datumlanc[11]; };
/* . . . */
struct datum d, *dptr=&d, dt[10];
```

🔍 Fedezzük fel, hogy a felhasználó definiálta típusnév **struct datum**. Hasonlításul:

```
double d, *dptr=&d, dt[10];
```

💣 Struktúra, unió, **enum** deklarációt, definíciót záró kapcsos zárójel után kötelező pontosvesszőt tenni, mert ez zárja az *azonosítólistá*t!

```
struct struki{
  int a, b;
  float matrix[20][10];
  char nev[26]; };   /* Itt nem elhagyható a ; a } után! */
```

- A *struktúracímké*nek egyedinek kell lennie a struktúra, unió és **enum** címke névterületen!
- Mint a tömböknél, struktúráknál is megadható nem teljes típusdeklaráció. Például a

  ```
  struct datum;
  ```

  még akkor is létrehozza az aktuális hatáskörben a **struct datum** nem teljes típust, ha ilyen befoglaló, vagy külső hatáskörben is létezne. Könnyen belátható, hogy ez **struct datum** típusú struktúra objektum definíciójára nem használható a struktúra szerkezetének közbenső, ugyanezen hatáskörbeli definiálása nélkül, hisz ismeretlen a memóriaigény. Arra azonban ez is alkalmas, hogy deklarációban, **typedef**–ben használjuk a típusnevet, vagy hogy **struct datum** típusú objektumokra mutató mutatókat hozzunk létre:

  ```
  struct datum *dptr1, *dptr2, *dptrt[20];
  ```

  Az ilyen mutatók akár más struktúra tagjai is lehetnek:

  ```
  struct A;          /* Nem teljes típusdeklaráció. */
  struct B{          /* Itt tag a nem tejes típusra */
    struct A *pa;};  /* mutató mutató. */
  struct A{
    struct B *pb;};  /* Ez most már teljes típus lesz.*/
  ```

Vigyázat! Struktúradefinícióban a struktúra típusa a *struktúra-tagdeklarációlistá*ban csak akkor válik teljessé, ha elérjük a *specifikátor* bezáró kapcsos zárójelét (}).

- A struktúradeklaráció általános alakjából látható volt, hogy belőle a *struktúracímke* is elhagyható. Ha ezt megtesszük, ún. név nélküli, vagy címkézetlen struktúrához jutunk. Világos, hogy ebben az esetben a nem teljes típusdeklarációnak

  ```
  struct;
  ```

  semmi értelme (szintaktikai hiba is) sincs, de az olyan deklarációnak sincs, amiben csak a struktúra szerkezetét adjuk meg:

  ```
  struct {int tag1, tag2; /* ... */};
  ```

  hiszen később nem tudunk a típusra hivatkozni, s ebből következőleg ilyen típusú objektumokat deklarálni. Név nélküli struktúradeklarációban nem hagyható el tehát az *azonosítólista*, azaz:

  ```
  struct {int tag1, tag2; /* ... */} az1, az2[14];
  ```

10.1.1 Típusdefiníció

- Az előbb vázolt probléma típusdefiníció alkalmazásával áthidalható:

```
typedef struct{  int tag1, tag2; /* ... */} CIMKETLEN;
/* Most sincs címke. */
CIMKETLEN y, *y, ytomb[12];
```

- A típusdefiníció a címkézett struktúrával is használható lett volna:

```
typedef struct datum{
   int ev, ho, nap, evnap;
   long datumssz;               /* Dátumsorszám. */
   char datumlanc[11]; } DATUM;
/* . . . */
DATUM d, *dptr, dt[10];
```

Összesítve: A **typedef** címke nélküli struktúrák, uniók és **enum**-ok típusdefiníciójára is alkalmas. Struktúrák esetében használjunk azonban struktúracímkét, vagy **typedef**-es szerkezetet, de a kettőt együtt nem javasoljuk!

- Egy kicsit összetettebb példát véve:

```
typedef char nev[30];
typedef enum{no, ferfi, egyeb} sex;
typedef struct{
   nev csalad, kereszt; /* Két 30 elemű karaktertömb.*/
   sex fino;            /* no vagy ferfi értékű enum.*/
   /* . . . */
   double osztondij; } hallgato;
typedef hallgato evf[100];/* 100 elemű, fenti szerke-
                             zetű struktúratömb. */
evf evf1, evf2, evf3;   /* Három darab,100 elemű, fenti
                           szerkezetű struktúratömb. */
```

10.2 Struktúratag deklarációk

A { }-ben álló *struktúratag-deklarációlista* a deklarátor szintaktikát követve meghatározza a struktúratagok neveit és típusait.

- A struktúratag bármilyen típusú lehet a **void**, a nem teljes, vagy a függvény típustól eltekintve.

A struktúratag deklaráció nem tartalmazhat azonban tárolási osztály specifikátort vagy inicializátort. Struktúratag nem lehet az éppen definíció alatt álló struktúra sem:

```
struct szoszlo{
  static char *szo;                    /* HIBÁS */
  int szlo=0;                          /* HIBÁS */
```

```
struct szoszlo elozo, kovetkezo; };     /* HIBÁS */
```

- Struktúratag lehet azonban nem teljes típusú struktúrára, így akár az éppen deklaráció alatt állóra mutató mutató:

```
struct szoszlo{
  char *szo;
  int szlo;
  struct szoszlo *elozo, *kovetkezo; };    /* OK */
```

- Struktúratag lehet tömb, sőt már definiált szerkezetű struktúra is:

```
struct sor_lanc{
  int sorszam;
  char megjegyzes[32];
  struct sor_lanc *kovetkezo; };
struct kereszthivatkozas{
  char *szo;
  int szlo;
  struct kereszthivatkozas *elozo, *kovetkezo;
  struct sor_lanc elso; };
```

- A struktúrának nem lehet függvény tagja, de függvényre mutató mutató persze lehet tag:

```
struct pelda{
  char *szoveg;
  int (*hasonlit)(const char *, const char *);};
```

- A struktúratag azonosítójának egy struktúrán belül kell egyedinek lennie, vagyis másik struktúrában nyugodtan létezhet ugyanilyen nevű tag.

- A beágyazott struktúra ugyanúgy elérhető, mint a fájl hatáskörben deklarált, azaz a következő példa helyes:

```
struct a{
  int x;
  struct b{
    int y; } v2;
  } v1;
/* . . . */
struct a v3;
struct b v4;
```

- A beágyazott struktúra gyakran névtelen:

```
struct struki{
  struct { int x, y; } pont;
  int tipus; } v;
```

10.3 Struktúrák inicializálása

A struktúrát konstans kifejezésekből álló inicializátorlistával láthatjuk el kezdő értékkel. Az inicializátorlista elemek értékét a struktúratagok a deklarációbeli elhelyezkedés sorrendjében veszik fel:

```
struct struki {
  int i;
  char lanc[25];
  double d; } s = {20, ”Jancsika”, 3.14};
```

Pontosítsunk még néhány dolgot!

- Lokális élettartamú struktúrák esetén az inicializátor inicializátorlista, vagy kompatibilis struktúra típusú egyszerű kifejezés lehet:

  ```
  struct struki s = {20, ”Juliska”, 3.14}, s1 = s;
  ```

- Lokális (**auto**) struktúra persze akár ilyen típusú struktúrát visszaadó függvény hívásával is inicializálható:

  ```
  struct struki fv(int, char *, double);
  struct struki s2=fv(2, „Boszi”, 1.4);
  ```

- Ha a struktúrának struktúra vagy tömb tagja is van, akkor azt egymásba ágyazott { }-kel lehet inicializálni.

  ```
  struct struki {
    int i;
    long darab[3];
    double d; } s = { 20, { 1l, 2l, 3l}, 3.14};
  ```

- Tudjuk, hogy az inicializátorlista elemeinek száma nem haladhatja meg az inicializálandó struktúratagok számát! Ha az inicializátorlista kevesebb elemű, mint az inicializálandó objektumok száma, akkor a maradék struktúratagok a statikus élettartamú implicit kezdőérték adás szabályai szerint tölti fel a fordító, azaz nullázza:

  ```
  struct struki{
    int cipomeret,/* s.cipomeret==42 */
      magassag;   /* s.magassag==180 */
    char nev[26]; /* az s.nev ”Magas Lajos” */
    char cim[40]; /* s.cim üres karakterlánc (””)
                     kezdőértékű. */
    double fizetes;/* s.fizetes==0.0. */
    } s = { 42, 180, ”Magas Lajos”};
  ```

- Névtelen bitmező tag nem inicializálható!

☞ Ha az inicializátorlistában nincs beágyazott inicializátorlista, akkor az ott felsorolt értékek az alaggregátumok, s őket a deklaráció sorrendjében veszik fel az aggregátum elemei. Kapcsos zárójelek ugyanakkor akár

az egyes inicializátorok köré is tehetők, de ha a fordítót nem kívánjuk "becsapni", akkor célszerű őket az aggregátum szerkezetét pontosan követve használni!

```
typedef struct { int n1, n2, n3; } triplet;
triplet nlist1[2][3] = {           /* Helyes megoldás: */
  {{11, 12, 13}, {4, 5, 6}, {7, 18, 9}},/* Első sor. */
  {{1, 2, 3}, {14, 15, 16}, {7, 8, 9}}  /* 2. sor. */ };
triplet nlist2[2][3] = {           /* Hibás megoldás: */
  {11, 12, 13}, {4, 5, 6}, {7, 18, 9},  /* Első sor. */
  {1, 2, 3}, {14, 15, 16}, {7, 8, 9}    /* 2. sor. */ };
```

A **sizeof**-ot struktúrákra alkalmazva mindig teljes méretet kapunk akár a típust adjuk meg operandusként, akár az ilyen típusú objektumot. Például:

```
#include <stdio.h>
struct st{
  char *nev;           /* A mutató mérete bájtban. */
  short kor;           /* + 2 bájt. */
  double magassag; };  /* + 8 bájt. */
struct st St_Tomb[ ] = {
  {"Jancsika", 18, 165.4},     /* St_Tomb[0] */
  {"Juliska", 116, 65.4}};     /* St_Tomb[1] */
int main(void){
  printf("\nSt_Tomb elemeinek száma = %d\n",
         sizeof(St_Tomb)/sizeof(struct st);
  printf("\nSt_Tomb egy elemének mérete = %d\n",
         sizeof(St_Tomb[0]));
  return 0; }
```

10.4 Struktúratagok elérése

A struktúra és az uniótagok eléréséhez ugyanazokat a tagelérés operátorokat alkalmazza a nyelv. A tagelérés operátort szelekciós operátornak, tagszelektornak is szokás nevezni. Prioritásuk magasabb az egyoperandusos műveletekénél, s közvetlenül a () és a [] után következik. Kétfajta tagelérés operátor van:

- az egyik a közvetlen szelekciós operátor (.) és
- a másik a közvetett (->).

A közvetlen tagelérés operátor alakja:

utótag-kifejezés.azonosító

Az *utótag-kifejezés*nek struktúra típusúnak, s az *azonosító*nak e struktúra típus egy tagja nevének kell lennie. A konstrukció típusa az elért tag típusa, értéke az elért tag értéke, s balérték akkor és csak akkor, ha az *utótag-kifejezés* az, és az *azonosító* nem tömb.

A közvetett tagelérés operátor formája:

utótag-kifejezés->azonosító

Az *utótag-kifejezés*nek struktúra típusra mutató mutatónak, s az *azonosító*nak e struktúra típus egy tagja nevének kell lennie. A konstrukció típusa és értéke az elért tag típusa és értéke. Balérték, ha az elért tag nem tömb.

Feltéve, hogy **s struct S** típusú struktúra objektum, és **sptr struct S** típusú struktúrára mutató mutató, akkor ha **t** az **struct S** struktúrában deklarált, *típus* típusú tag, az

```
s.t
```

és az

```
sptr->t
```

kifejezések típusa *típus*, és mindkettő a **struct S** struktúra **t** tagját éri el. A következők pedig szinonimák, ill. azt is mondhatjuk, hogy a −> szelekció operátoros kifejezés a másik rövidítése:

```
sptr->t ≡ (*sptr).t
```

Az **s.t** és az **sptr->t** balértékek, feltéve, hogy **t** nem tömb típusú. Például:

```
struct S{
  int t;
  char lanc[23];
  double d; } s, *sptr = &s, Stomb[20]={
                { 0, ”nulla”, 0.}, { 1, ”egy”, 1.}};
/* . . . */
s.t = 3;
sptr->d = 4.56;
```

- Az **Stomb** 20 elemű, **struct S** struktúrából álló struktúratömb, melynek első (**Stomb**[0]) és második (**Stomb**[1]) elemét kivéve nincs explicit kezdőértéke, azaz **Stomb**[2], . . ., **Stomb**[19] { 0, ””, 0,} értékű. A következő példák a struktúratömb tagelérést szemléltetik:

  ```
  Stomb[3].t=3;
  strcpy(Stomb[3].lanc, ”három”);
  Stomb[3].d=3.3;
  ```

  vagy:

  ```
  sptr=Stomb+5;
  sptr->t=5;                    /* Stomb[5].t */
  strcpy(sptr->lanc, ”öt”);     /* Stomb[5].lanc */
  sptr->d=5.5;                  /* Stomb[5].d */
  ```

- Ha **struct B** struktúrának van **struct A** struktúra típusú tagja, akkor az ilyen **struct A** tagokat csak a tagszelekciós operátorok kétszeri alkalmazásával lehet elérni:

```
struct A {
  int j;
  double x; };
struct B {
  int i;
  char *nev;
  struct A a;
  double d; } s, *sptr = &s;
/* . . . */
s.i = 3;
s.a.j = 2;
sptr->d = 3.14;
(sptr->a).x = 6.28;
```

- A szelekciós operátorok balról jobbra kötnek, tehát a következők teljesen azonosak:

```
(sptr->a).x ≡ sptr->a.x;
(s.a).x ≡ s.a.x;
```

- Említettük már, hogy a tagszelektorok prioritása magasabb az egyoperandusos műveleteknél. Ezért a

```
++sptr->i ≡ ++(sptr->i)
```

, azaz a kifejezés **struct B i** tagját inkrementálja, s nem **sptr**–t. Ha mégis ezt szeretnénk (bár a konkrét példánál ennek semmi értelme sincs), akkor a ++**sptr** kifejezés részt zárójelbe kell tenni, azaz:

```
(++sptr)->i
```

A

```
++(sptr++)->i
```

ugyancsak **s**.**i**–t inkrementálja, de a kifejezés mellékhatásaként **sptr** is megnő eggyel. Ha a zárójeleket elhagyjuk, persze akkor is idejutunk:

```
++sptr++->i ≡ ++(sptr++)->i
```

- Ugyanígy a **nev** címen levő karaktert éri el a

```
*sptr->nev
```

, hisz az indirekciót az előbbiekből következőleg az **sptr**–rel elért **nev** címen hajtja végre a fordító. Vegyük még a

```
*sptr++->nev++
```

kifejezést, aminek ugyanaz az értéke, mint az előző kifejezésé, de mellékhatásként **sptr** és **nev** inkrementálása is megtörténik.

💣 Ha már mindig hozzárendelési példákat hoztunk, akkor itt kell megemlítenünk, hogy struktúrákat csak akkor lehet egymáshoz rendelni, ha a forrás és a cél struktúra azonos típusú.

```
struct A {
  int i, j;
  double d; } a, a1;
struct B {
  int i, j;
  double d; } b;
/* . . . */
a = a1;     /* OK, a hozzárendelés megy tagról-tagra. */
a = b;      /* HIBÁS, mert eltér a két struktúra típusa.*/
a.i = b.i; /* Tagról-tagra persze most is megy a dolog. */
a.j = b.j;
a.d = b.d;
```

Vegyünk valamilyen példát a struktúratömbökre!

Keressük meg a megadott, síkbeli pontok közt a két, egymástól legtávolabbit! A pontok száma csak futás közben dől el (**n**), de nem lehetnek többen egy fordítási időben változtatható értéknél (**N**). Az x koordináta bevitelekor üres sort megadva, a bemenet előbb is befejezhető, de legalább két pont elvárandó! Az input ellenőrzendő, s minden hibás érték helyett azonnal újat kell kérni. Az eredmény közlésekor meg kell jelentetni a két pont indexeit, koordinátáit és persze a távolságot is.

```
/* PELDA28.C: A két, egymástól legtávolabbi pont
   megkeresése. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#define INP 28       /* Az input puffer mérete. */
#define N 128        /* Pontok maximális száma. */
```

A feladatot síkbeli pontot leíró struktúra segítségével fogjuk megoldani:

```
struct Pont{         /* A Pont struktúra. */
  double x, y; };
int getline(char s[],int n){ /* . . . */ }
int lebege(char s[]){ /* . . . */ }
```

✍ A két függvény forrásszövege idemásolandó!

```
int main(void){
  char sor[INP+1];      /* Input puffer. */
```

A struktúratömb definíciója:

```
struct Pont p[N];     /* Struktúratömb. */
int n=0;              /* Pontok száma. */
double max=-1., d;    /* Pillanatnyi maximum és */
int i, j, tavi, tavj;/* segédváltozók. */
printf("A két egyméstól legtávolabbi pont a síkban.\n"
     "Adja meg a pontok koordinátapárjait rendre!\n"
     "Vége: üres sor az X koordináta megadásánál.\n\n");
for(n=0; n<N; ++n){
  printf("A(z) %d pont koordinátái:\n", n+1);
  if(printf("X: "), getline(sor, INP)<=0) break;
  if(lebege(sor)) p[n].x=atof(sor);
  else { --n; continue;}
  while(printf("Y: "), getline(sor, INP),
        !lebege(sor));
  p[n].y=atof(sor); }
if(n<2){
  printf("Legalább két pontot meg kéne adni!\n");
  return(1);}
for(i=0; i<n-1; ++i)
  for(j=i+1; j<n; ++j)
```

☞ Kitűnően látszik, hogyan kell elérni a struktúra tömbelem tagjait!

```
     if((d=sqrt((p[j].x-p[i].x)*(p[j].x-p[i].x)+
        (p[j].y-p[i].y)*(p[j].y-p[i].y))) > max){
       max=d;
       tavi=i;
       tavj=j; }
printf("A maximális távolságú két pont:\n"
     "P[%d]: (%10.1f, %10.1f) és\n"
     "P[%d]: (%10.1f, %10.1f),\n"
     "s a távolság: %15.2f\n",
     tavi, p[tavi].x, p[tavi].y,
     tavj, p[tavj].x, p[tavj].y, max);
return(0); }
```

Megoldandó feladatok:

Ha fokozni kívánja a feladatot, akkor

- Dolgozzon térbeli pontokkal!
- Rendezze a pontokat az origótól való távolságuk csökkenő sorrendjében, és jelentesse meg a pontokat és a távolságot fejléccel ellátva, táblázatosan és lapozhatóan!
- Esetleg oldja meg, hogy ne lehessen kétszer ugyanazt a pontot megadni!

10.5 Struktúrák és függvények

Említettük már, hogy a struktúra másolható, hozzárendelhető, elérhetők a tagjai, képezhető a címe, ill. tömb is előállítható belőle, de függvény is visszaadhat struktúrát vagy erre mutató mutatót. Az **fv1** visszaadott értéke **struct struki** struktúra.

```
struct struki fv1(void);
```

Az **fv2** viszont **struct struki** struktúrára mutató mutatót szolgáltat.

```
struct struki *fv2(void);
```

A függvény paramétere is lehet struktúra e két módon. Az **fv3 struct struki** struktúrát fogad paraméterként.

```
void fv3(struct struki s);
```

Az **fv4** viszont **struct struki** struktúrára mutató mutatót fogad.

```
void fv4(struct struki *sp);
```

💣 A következő példa a „helytelen” gyakorlatot szemlélteti. A függvény paraméterei és visszaadott értéke egyaránt struktúra. Struktúra persze akármekkora is elképzelhető.

```
typedef struct{
  char nev[20];
  int az;
  long oszt; } STUDENT;
STUDENT strurend(STUDENT a, STUDENT b){
  return((a.az < b.az) ? a : b); }
/* . . . */
STUDENT a, b, c;
/* . . . */
c = strurend(a, b);
```

Amíg a **strurend** fut, hat darab **STUDENT** struktúra létezik: **a**, **b**, **c**, aztán **a** és **b** másolata és a függvény visszaadott értéke a veremben. Célszerű tehát nem a struktúrát, hanem arra mutató mutatót átadni a függvénynek, ill. vissza is kapni tőle, ha lehet, azaz:

```
STUDENT *strurnd(STUDENT *a, STUDENT *b){
  return((a->az < b->az) ? a : b); }
/* . . . */
STUDENT *z;
/* . . . */
z = strurnd(&a, &b);
```

💣 Prototípus hatásköre ellenére a benne megadott **struct** hatásköre globális, azaz figyelmeztető üzenet nélkül nem hívhatjuk meg a következő függvényt:

```
void fv(struct S *);
```

☞ A probléma elhárításához deklarálni vagy definiálni kell a struktúrát prototípus előírása előtt:

```
struct S;
/* . . . */
void fv(struct S *);
```

Készítsünk struktúrát és kezelő függvénycsaládot dátumok manipulálására!

A **datum** struktúrában nyilvántartjuk a dátum évét (**ev**), hónapját (**ho**), napját (**nap**), a Krisztus születése óta a dátumig eltelt napok számát: az ún. dátumsorszámot (**datumssz**), a dátum karakterlánc alakját (**datumlanc**) és azt, hogy a dátum az év hányadik napja (**evnap**). Az egészet úgy képzeljük el, hogy

- vagy megadják a dátumot év, hó és nap alakban, és a **DatumEHN** függvénnyel meghatározzuk a struktúra összes többi adatát (a **main**-ben **d1** objektum így kap értéket),
- vagy karakterlánc alakú dátumból a **DatumKAR**–ral állítjuk elő a **datum** struktúra tagjainak értékeit (a főprogramban **d2** e módon jut értékhez).
- Mindkét **struct datum** objektumnak értéket adó rutin végül dátumellenőrzést végez a **Datume** függvénnyel, s ezt a logikai értéket szolgáltatja.
- A **NapNev** visszaadja a dátum héten belüli napjának nevét. Pontosabban a név karakterláncának címét.
- A további rutinok műveleteket végeznek a dátum struktúrákkal. A **DatumKul** megállapítja két dátum különbségét, s szolgáltatja ezt a napszámot. A **DatumMegEgy** inkrementálja, és visszaadja a dátum objektumot. A **DatumMegint** pozitív, egész értéket ad hozzá.
- A dátumfüggvények mind a **MINDATUM** és **MAXDATUM** közötti tartományban dolgoznak.
- A **main** ki is próbálja az összes dátumfüggvényt.

☞ Figyeljük meg, hogy mindegyik függvény **struct datum** objektumra mutató paraméterként kapja meg a manipulált dátum struktúrá(ka)t! A **DatumMegEgy** és a **DatumMegint** visszatérési értéke **struct datum** struktúra.

✍ Fedezzük fel, hogy a globális **MAXDATUM**, a hónapi napszámokat tartalmazó **honap** tömb, és a **Datume** függvény **static** tárolási osztálya miatt lokális a **DATUM.C** modulra! Nincs is prototípus a **Datume**–re a **DATUM.H** fejfájlban. Az értéküket nem változtató, de csak a rutin blokkjából elérendő **oszto** változó, és a napnév karakterláncokra mutatókból álló **hetnev** mutatótömb statikus élettartamúak, de hatáskörük lokális.

✍ Vegyük még észre a **DATUM.H**–ban, hogy a **_DATUMH** makró egyetlen **struct datum** definíciót tesz lehetővé akkor is, ha a fordítási egységben többször kapcsolnák be a fejfájlt.

```
/* DATUM.H: Dátumok kezelése. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#if !defined(_DATUMH)
#define _DATUMH
struct datum{
  int ev, ho, nap, evnap;
  long datumssz;              /* Dátumsorszám. */
  char datumlanc[11]; };
#endif
const char *NapNev(struct datum *);
int DatumEHN(int, int, int, struct datum *);
int DatumKAR(const char *, struct datum *);
long DatumKul(struct datum *, struct datum *);
struct datum DatumMegEgy(struct datum *);
struct datum DatumMegint(struct datum *, int);
```

```
/* DATUM.C: Dátumok kezelése. */
#include "DATUM.H"
#define MINDATUM 366
static const long MAXDATUM=9999*365l + 9999/4 -
                           9999/100+9999/400;
static int honap[]={0, 31, 28, 31, 30, 31, 30,
                    31, 31, 30, 31, 30, 31};
const char *NapNev(struct datum *pd){
  static char *hetnev[]={"vasárnap", "hétfő", "kedd",
              "szerda", "csütörtök","péntek","szombat"};
  return hetnev[(pd->datumssz)%7l]; }
```

A dátumsorszámból 7–tel képzett modulus alapján állapítja meg a héten belüli napindexet a **NapNev** rutin.

```
static int Datume(struct datum *pd){
  int i;
  honap[2]=28+(!(pd->ev%4)&&pd->ev%100||(!pd->ev%400));
  if(pd->ev<1 || pd->ev>9999 || pd->ho<1 || pd->ho>12 ||
     pd->nap<1 || pd->nap>honap[pd->ho]){
```

```
    pd->evnap=0; pd->datumssz=0l; return 0;}
  else {
    sprintf(pd->datumlanc, "%04d.%02d.%02d",
         pd->ev, pd->ho, pd->nap);
    pd->evnap=pd->nap;
    for(i=1; i<pd->ho; ++i)pd->evnap+=honap[i];
    pd->datumssz=(pd->ev-1)*365l+pd->evnap+
      pd->ev/4-pd->ev/100+pd->ev/400;
    return 1; } }
```

A **Datume** ugyanúgy a dátumot ellenőrzi, s logikai választ ad a „formálisan jó–e a dátum?” kérdésre, mint a korábbi **datume** függvények. Nem karakterláncból dolgozik azonban, hanem a struktúra **ev**, **ho**, **nap** tagjaiból, melyeket a **DatumEHN**, ill. a **DatumKAR** készítettek oda. Ha hibás a dátum, akkor nullázza a rutin az **evnap** és a **datumssz** tagokat.

Ha jó a dátum, akkor a **Datume** képezi a **datumlanc**–ba a dátum karakterlánc alakját, s meghatározza az **evnap** és a **datumssz** értékét. Az **evnap** a dátum napszámáról indul, s a rutin hozzáadogatja a megelőző hónapok maximális napszámait. A dátumsorszám megállapításához a szökőév vizsgálatához használatos kifejezést alkalmazza a függvény.

📖 Az **STDIO.H** bekapcsolásával rendelkezésre álló **sprintf** ugyanúgy működik, mint **printf** társa, de nem a szabvány kimenetre, hanem az első paramétereként kapott karaktertömbbe dolgozik.

✍ A formátumspecifikációkban a mezőszélesség előtt álló 0 hatására a jobbra igazított számok balról nem szóköz, hanem ’0’ feltöltést kapnak. Magyarán a 932.2.3. dátumból 0900.02.03 karakterlánc lesz.

Szóltunk már róla, hogy a **Datume** nem hívható más forrásmodulból. A **DATUM.C**–ben is csak a **DatumEHN** és a **DatumKAR** idézi meg utolsó lépéseként.

```
int DatumEHN(int e, int h, int n, struct datum *pd){
  pd->ev=e;
  pd->ho=h;
  pd->nap=n;
  if(e>=0&&h>=0&&n>=0&&e<10000&&h<100&&n<100)
    sprintf(pd->datumlanc, "%04d.%02d.%02d", e, h, n);
  else *pd->datumlanc=0;
  return Datume(pd);}
int DatumKAR(const char *lanc, struct datum *pd){
  pd->ho=pd->nap=0;
  strncpy(pd->datumlanc, lanc, 10);
  pd->datumlanc[10] = 0;
  pd->ev=atoi(lanc);
  while(isdigit(*lanc))++lanc;
  if(*lanc!=0){
```

```
    ++lanc;
    pd->ho=atoi(lanc);
    while(isdigit(*lanc))++lanc;
    if(*lanc){
      ++lanc;
      pd->nap=atoi(lanc);}}
  return Datume(pd); }
```

A **DatumEHN** a kapott, **int** típusú év, hó, nap segítségével tölti fel az utolsó paraméterként elért dátum struktúrát.

☞ Az **sprintf** hívás előtti vizsgálatra azért van szükség, hogy a rutin ne tudja túlírni a 11 elemű karaktertömb, **datumlanc** tagot a memóriában valamilyen egészen „zöldség” év, hó, nap paraméter miatt. Ilyenkor üres lesz a **datumlanc**.

A **DatumKAR** átmásolja a karakterlánc alakban kapott dátum első 10 karakterét a **datumlanc**–ba. Nullázza a **ho**napot és **nap**ot. Látszik, hogy a függvény nem köti meg olyan szigorúan sem az év, sem a hónap és nap jegyszámát, mint a korábbi **datume**, ill. elválasztó karakterként csak valamilyen nem numerikust vár el. A karakterlánc egésszé konvertált elejét évnek, az első elválasztó karakter utáni részt hónapnak, s a második elválasztó karakter mögöttieket napnak tekinti a rutin, hacsak időközben vége nem lesz a karakterláncnak.

Végül mindkét függvény meghívja a **Datume**–t, s ennek visszatérési értékét szolgáltatja.

```
long DatumKul(struct datum *pd1, struct datum *pd2){
  if(pd1->datumssz>pd2->datumssz)
    return pd1->datumssz-pd2->datumssz;
  else return pd2->datumssz-pd1->datumssz; }
```

A **DatumKul** képzi a két paraméter dátum struktúra dátumsorszám tagjainak különbsége abszolút értékét.

```
struct datum DatumMegEgy(struct datum *pd){
  int e=pd->ev, h=pd->ho, n=pd->nap;
  struct datum d;
  honap[2]=28+(e%4==0 && e%100 || e%400==0);
  if(++n>honap[h]){
    n=1;
    ++h;
    if(h>12){
      h=1;
      ++e; }}
  if(!DatumEHN(e, h, n, &d)) d=*pd;
  return d; }
```

A paramétere dátumot inkrementáló **DatumMegEgy** munkaváltozókba rakja az évet, a hónapot és a napot. Meghatározza az év szerinti február pontos napszámát. Növeli eggyel a napot. Ha ez túlmenne a hónap szerinti maximális napszámon, akkor 1 lesz, és a hónapszám növelése jön. Ha ez 13 lenne, akkor 1 lesz, és az évszám növelése következik.

A megállapított, új év, hó, nap alapján a **DatumEHN** feltölti a lokális, **d** dátum objektumot. Ha az új dátum érvénytelen volt, akkor a változatlanságot jelzendő a rutin hozzárendeli **d**–hez a paraméter címen levő, eredeti dátumot.

☞ A hozzárendelés a paraméter címen levő (ezért kell elé az indirekció) **struct datum** minden tagját egy az egyben átmásolja a balérték, **d**, lokális dátum objektum tagjaiba rendre.

A visszatérés során a **DatumMegEgy** létrehoz a veremben egy ideiglenes dátum objektumot, melybe tagról–tagra bemásolja a **d** lokális dátum változót. Visszatérés után aztán a **main** hozzárendeli az ideiglenes dátum objektumot a **main**–ben lokális **d**–hez.

```
struct datum DatumMegint(struct datum *pd, int np){
  int e=pd->ev, h=pd->ho, n=pd->nap;
  long dpd = pd->datumssz + np + 365;
  /* A tiszta jó konstans 365.24223 lenne kézi
     számítás szerint!!!! */
  static double oszto=365.24225;
  struct datum d=*pd;
  if(np <= 0) return d;
  if(dpd > MAXDATUM){
    e=9999;
    h=12;
    n=31; }
  else {
    e= (int)dpd/oszto;
    n=dpd-e*365l-e/4+e/100-e/400;
    honap[2]=28+(e%4==0 && e%100 || e%400==0);
    for(h=1; n > honap[h]; ++h)n-=honap[h]; }
  if(!DatumEHN(e, h, n, &d)) d=*pd;
  return d; }
```

Csak a kezdetét és a végét tekintve a pozitív napszámot a dátumhoz adó **DatumMegint** a **DatumMegEgy**–gyel megegyezően dolgozik.

☞ Látszik, hogy negatív, hozzáadandó napszámot, vagy a művelet végén érvénytelen dátumot kapva, az eredeti dátum objektumot szolgáltatja a rutin változatlanul.

A dátumsorszámot megnöveli a napszámmal és még 365–tel. Ha így meghaladná a 9999.12.31–et, akkor ezt adná vissza. Ha nem, akkor az új

dátumsorszámot elosztja a tapasztalati alapon a [**MINDATUM**, **MAX-DATUM**] tartományban érvényes évenkénti átlagos napszámmal, s ez lesz az új évszám. Visszaszámolja belőle az új év pontos napszámát, s a két érték különbségéből hónap és napszámot képez.

```
/* PELDA29.C: A dátumok kezelésének kipróbálása. */
#include "DATUM.H"
void main(void) {
  long kul;
  struct datum d1, d2, d;
  printf("Dátum műveletek:\n\n");
  DatumEHN(2003, 12, 31, &d1);
  DatumKAR("2003-2-13", &d2);
  printf("A(z) %s. és a(z) %s. különbsége %ld nap!\n",
         d1.datumlanc, d2.datumlanc,
         (kul=DatumKul(&d1, &d2)));
  d=DatumMegEgy(&d1);
  printf("A(z) %s. + 1 a(z) %s.\n", d1.datumlanc,
         d.datumlanc);
  d=DatumMegint(&d2, (int)kul);
  printf("A(z) %s. + %ld a(z) %s.\n", d2.datumlanc,
         kul, d.datumlanc);
  printf("A(z) %s. %s.\n", d2.datumlanc, NapNev(&d2)); }
```

Megoldandó feladatok:

Bővítse a **DATUM.H** és **DATUM.C** fájlokat a következő funkciókat ellátó függvényekkel! Persze próbálja is ki őket!

- A hónapnév karakterlánc előállítása a hónapszám alapján.
- Olyan karakterlánc alakú dátum létrehozása, melyben a hónap megnevezése szerepel a hónap száma helyett.
- A **DatumMegint** olyan átírása, hogy a napszám paraméter negatív is lehessen.

Készítsen ugyanilyen szellemben struktúrát és kezelő függvénycsaládot az időre is!

10.6 Önhivatkozó struktúrák és dinamikus adatszerkezetek

Tudjuk, hogy a struktúrának nem lehet

- **void,**
- nem teljes és
- függvény

típusú tagja, de nem lehet tag

- az éppen definíció alatt álló struktúra

sem. Lehet viszont tag nem teljes típusú struktúrára, így akár a definíció alatt állóra, mutató mutató.

Azt a struktúrát, melynek legalább egy önmagára mutató tagja van, önhivatkozó struktúrának nevezik.

A dinamikus adatszerkezeteket [3]: listákat, fákat stb. leíró adatkonstrukciók a C–ben önhivatkozó struktúrák. Nézzünk néhányat!

Egyirányú listához például a következő struktúra lenne használható:

```
struct List1{
  ADAT adat;
  struct List1 *kov; };
```

, ahol az **ADAT** típusú **adat** tagon valamilyen, a lista egy elemében tárolandó adatokat leíró struktúrát kell érteni. A **kov** az egyirányú lista következő **struct List1** típusú elemére mutat, ill. a lista végét **NULL** mutató jelzi. A lista kezelhetőségéhez ezen túl már csak egy horgonypontra, és esetleg egy **seged** mutatóra

```
struct List1 *KezdoPont = NULL, *seged;
```

van szükség a listát manipuláló programban.

Tegyük fel, hogy ismertek az **ADAT**ok, s vegyük fel a lista első elemét!

```
if(!(KezdoPont=seged=(struct List1*)malloc(
                          sizeof(struct List1)))){
  printf("Elfogyott a memória!\n");
  exit(1); }
else{
  /* seged->adat vegye fel az ADATok értékét! */
  seged->kov=NULL; }
```

A lista következő eleme:

```
if(!(seged->kov=(struct List1*)malloc(
                          sizeof(struct List1)))){
  printf("Elfogyott a memória!\n");
  exit(1); }
else{
  /* seged->adat vegye fel az ADATok értékét! */
  seged->kov=NULL; }
/* . . . */
```

Elég! Írjunk **Beszur1** függvényt, mely paraméterként megkapja a beszúrandó **ADAT**okat, s annak a listaelemnek a címét, mely utánra az új listaelem kell, hogy kerüljön! Ha még nincs is lista, akkor ezen a pozíción

kapjon **NULL** mutatót a rutin! A visszatérési érték legyen a most létesített listaelem címe, ill. **NULL** mutató, ha elfogyott a memória!

```
struct List1 *BeSzur1(ADAT a, struct List1 *elozo){
  struct List1 *p;
  if(p=(struct List1 *)malloc(sizeof(struct List1))){
    p->adat=a;
    if(elozo){
      p->kov=elozo->kov;
      elozo->kov=p; }
    else p->kov=NULL; }
  return p; }
```

Ekkor a lista létrehozása a következő:

```
KezdoPont=seged=BeSzur1(adatok, NULL);
while(/* Vannak következő adatok? */&&seged!=NULL)
  seged=BeSzur1(adatok, seged);
if(!seged){
  printf("Elfogyott a memória!\n");
  exit(1); }
```

A létrehozott egyirányú lista felhasználás után a következő kóddal semmisíthető meg:

```
while(KezdoPont){
  seged=KezdoPont->kov;
  free(KezdoPont);
  KezdoPont=seged; }
```

Az egyirányú listában lehet új elemet bárhová beszúrni (**BeSzur1**), bárhonnét törölni, de a listát – ahogyan a megsemmisítő kód is mutatja – csak előrehaladva lehet elérni, visszafelé lépkedve nem. Az oda–visszahaladáshoz kétirányú lista kell:

```
struct List2{
  ADAT adat;
  struct List2 *kov, *elo; };
struct List2 *KezdoPont = NULL, *seged;
```

A visszalépegetés lehetőségét a megelőző listaelemre mutató, **elo** mutatótag biztosítja. A lista végét mindkét irányban **NULL** mutató jelzi. A beszúrás:

```
struct List2 *BeSzur2(ADAT a, struct List2 *elozo){
  struct List2 *p;
  if(p=(struct List2 *)malloc(sizeof(struct List2))){
    p->adat=a;
    if(elozo){
      p->elo=elozo;
      p->kov=elozo->kov;
      elozo->kov=p;
```

```
      if(p->kov) p->kov->elo=p; }
    else p->elo=p->kov=NULL; }
  return p; }
```

A **BeSzur2** csak egyet nem tud: a létező első elem elé beszúrni. Ezen így segíthetünk:

```
seged=BeSzur2(adatok, NULL);
seged->kov=KezdoPont;
KezdoPont=seged;
```

A törlés:

```
struct List2 *Torol2(struct List2 *ezt){
  struct List2 *p=NULL;
  if(ezt){
    p=ezt->kov;
    if(ezt->elo) ezt->elo->kov=ezt->kov;
    if(ezt->kov) ezt->kov->elo=ezt->elo;
    free(ezt); }
  return p; }
```

A **Torol2** függvénynek is csak a létező, legelső elem törlésekor kell segíteni, hisz változik a

```
KezdoPont=Torol2(KezdoPont);
```

Felhasználás után a kétirányú lista is ugyanúgy semmisíthető meg, mint az egyirányú.

A fák közül válasszuk ki a bináris keresőfát! Ennek pontjaiban legfeljebb kettő az elágazások száma, és a pontokban helyet foglaló struktúrák **ADAT** része alapján a fa, mondjuk, növekvőleg rendezett. Létezik tehát egy

```
int Hasonlit(ADAT a1, ADAT a2);
```

rutin, mely zérust szolgáltat, ha **a1**==**a2**, pozitív értéket, ha **a1**>**a2**, ill. negatívat egyébként. A bináris keresőfában a mindenkori aktuális pont bal ágán levő pontok közül egy sem nagyobb, s a jobb ágán helyet foglalók közül viszont egyik sem kisebb az aktuális pontnál. Az adatstruktúra:

```
struct BinFa{
  ADAT adat;
  struct BinFa *bal, *jobb; };
struct BinFa *KezdoPont = NULL;
```

A beszúrást végző rekurzív függvény a következő:

```
struct BinFa *BeSzurBF(ADAT a, struct BinFa *p){
  int fel;
  if(!p){                      /* Új pont készül. */
    if(p=(struct BinFa *)malloc(sizeof(struct BinFa))){
```

```
      p->adat=a;
      p->bal=p->jobb=NULL; }
    else printf("Elfogyott a memória!\n"); }
  else if((fel=Hasonlit(a, p->adat))==0)
    /* Volt már ilyen ADAT, s itt ez a kód van. */
  else if(fel<0)   /* A bal oldali részfába való. */
    p->bal=BeSzurBF(a, p->bal);
  else             /* A jobb oldali részfába való. */
    p->jobb=BeSzurBF(a, p->jobb);
  return(p); }
```

A következő rekurzív függvény növekvőleg rendezett sorrendben végez el minden ponton valamilyen tevékenységet:

```
void Tevekeny(struct BinFa *p){
  if(p){
  Tevekeny(p->bal);
  /* Itt van a tevékenység kódja. */
  Tevekeny(p->jobb); } }
```

Keressük meg egy szövegfájlban a benne előforduló szavakat! Állapítsuk meg ezen kívül, hogy a szavak a szövegfájl mely sorszámú soraiban fordulnak elő! Ha ugyanaz a szó egy sorban többször is megtalálható, a sor sorszámát ekkor is csak egyszer kell közölni. Végül közlendők a szavak, és az előfordulási sor sorszámok névsorban!

A megoldásban a szavak tárolásához bináris keresőfát használunk, s a szóhoz tartozó előfordulási sor sorszámokat minden ponthoz tartozóan egyirányú listában tartjuk nyilván.

☞ A **PELDA30** programot a parancssorból

```
PELDA30 < PELDA30.C > EREDMENY.TXT
```

módon indíthatjuk, s a készült lista az **EREDMENY.TXT** fájlban tanulmányozható.

```
/* PELDA30.C: Kiírja a szövegben előforduló szavak listáját
   és megadja azt is, hogy a szavak milyen sorszámú sorok-
   ban fordultak elő! */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXSOR 256     /* A beolvasott sor max. hossza. */
/* A sorok sorszámainak egyirányú listája: */
struct List1{
  int sorsz;
  struct List1 *kov; };
/* A bináris kereső fa: */
struct BinFa{             /* Alapcsomópont. */
```

```
  char *szo;              /* A szóra mutat. */
  struct List1 *sorok;    /* A sorok sorszámai. */
  struct BinFa *bal;      /* A bal oldali ág. */
  struct BinFa *jobb;};   /* A jobb oldali ág. */
/* Függvény prototípusok: */
struct BinFa *BeSzurBF(char *, int, struct BinFa *);
void SorMeg(struct BinFa *, int);
void Kiir(struct BinFa *);
int main(void){
  static char szoelv[]=" \t\n\r\f\"\'\\\a\?"
                       ":;,.()[]{}*/%+-&|^~!<>=#";
  struct BinFa *KezdoPont=NULL;
  char sor[MAXSOR], *szo;
  int sorszam=1;
  printf("Szavak keresztreferenciája szövegben:\n");
  while(sor[MAXSOR-1]=2, fgets(sor, MAXSOR, stdin)){
    if(!sor[MAXSOR-1]&&sor[MAXSOR-2]!='\n')
      printf("Lehet kis elsorszámozás!\n");
    szo=strtok(sor, szoelv);
    while(szo){
      KezdoPont=BeSzurBF(szo, sorszam, KezdoPont);
      szo=strtok(NULL, szoelv); }
    ++sorszam; }
  Kiir(KezdoPont);
  return 0; }
```

A **fgets** függvény a **getline**–hoz nagyon hasonlóan dolgozik. Karakterláncot olvas be a szabvány bemenetről (**stdin**), melyet az első paraméter címtől kezdve letárol a memóriában. Az olvasás leáll, ha a függvény a második paraméterénél eggyel kevesebb (**MAXSOR** – 1), vagy ’\n’ karaktert olvasott. A **getline**–tól eltérően azonban a rutin a ’\n’–t is elhelyezi a láncban, és a lánc végéhez még egy ’\0’–t is hozzáilleszt. Sikeres esetben az **fgets** az első paraméter mutatóval tér vissza. Fájlvégen, vagy hiba esetén viszont **NULL**–t kapunk tőle.

Ha a **sor** tömb utolsó pozícióján van a lánczáró zérus, de előtte nincs ott a soremelés karakter, akkor az aktuálisan olvasott sor nem fért el egy menetben a bemeneti pufferben, s ebből következőleg elsorszámozás történik.

Az **strtok** leírása megtalálható a **MUTATÓK** szakasz **Karakterlánc kezelő függvények** fejezetében!

```
struct BinFa *BeSzurBF(char *a,int sorszam,struct BinFa *p)
{ int fel;
  if(!p){                 /* Új szó érkezett */
    p=(struct BinFa *)malloc(sizeof(struct BinFa));
    if(p&&(p->szo=(char *)malloc(strlen(a)+1))){
      strcpy(p->szo, a);
```

```
      if(p->sorok=(struct List1 *)malloc(
                                    sizeof(struct List1))){
        p->sorok->sorsz=sorszam;
        p->sorok->kov=NULL; }
      p->bal=p->jobb=NULL; }
    if(!p||!p->szo||!p->sorok){
      printf("Elfogyott a memória!\n");
      exit(1); } }
  else if((fel=strcmp(a, p->szo))==0)
    SorMeg(p, sorszam);
  else if(fel<0)
    p->bal=BeSzurBF(a, sorszam, p->bal);
  else p->jobb=BeSzurBF(a, sorszam, p->jobb);
  return(p); }
/* Sorszám hozzáadása az egyirányú lista végéhez: */
void SorMeg(struct BinFa *p, int sorszam){
  struct List1 *seged=p->sorok;
  while(seged->kov!=NULL && seged->sorsz!=sorszam)
    seged=seged->kov;
  if(seged->sorsz!=sorszam){
    if(seged->kov=(struct List1 *)malloc(
                                    sizeof(struct List1))){
      seged->kov->sorsz=sorszam;
      seged->kov->kov=NULL; }
    else{
      printf("Elfogyott a memória!\n");
      exit(1); } } }
/* A fa kiírása: */
void Kiir(struct BinFa *p){
  struct List1 *seged;
  int i;
  if(p){
    Kiir(p->bal);
    printf("%s:\n", p->szo);
    for(seged=p->sorok, i=0; seged; seged=seged->kov,
                            ++i)
      printf("%7d|", seged->sorsz);
    if(i%10!=9) printf("\n");
    Kiir(p->jobb); } }
```

Megoldandó feladatok:

Fejlessze tovább a **PELDA30.C**–ben megoldott feladatot a következőképp, s persze próbálja is ki!

- –C parancssori paraméterrel indítva a program ne gyűjtse a szabványos C kulcsszavak előfordulásait.
- Ha a szabvány kimenet (**stdout**) nem fájl, akkor bontsa lapokra a listát a szoftver.

- –N parancssori paraméterrel startolva jelentesse meg a program megsorszámozva, de egyébként változatlanul a bemenetet.

10.7 Struktúra tárillesztése

A fordító a struktúratagokat deklarációjuk sorrendjében növekvő memória címeken helyezi el. Minden adatobjektum rendelkezik tárillesztési igénnyel is. A fordító olyan eltolással helyezi el az adatobjektumot, hogy az

eltolás % tárillesztési-igény == 0

zérus legyen. Struktúrák esetén ez a szabály a tagok elhelyezésére vonatkozik. Ha példának vesszük a

```
struct struki {
  int i;
  char lanc[3];
  double d; } s;
```

struktúrát, akkor tudjuk, hogy az **s** objektumot növekvő memória címeken úgy helyezi el a fordító, hogy

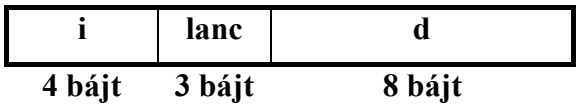
1. négy bájtot (32 bites esetben) foglal az **int** tagnak,
2. aztán a 3 bájtos karakterlánc következik, és
3. végül 8 bájtot rezervál a **double** taghoz.

Bizonyos fordító opciók, vagy valamilyen #**pragma** direktíva segítségével vezérelhetjük a struktúra adatok memóriabeli illeszkedését. Ez azt jelenti, hogy az adatokat 1–gyel, 2–vel, 4–gyel stb. maradék nélkül osztható címeken: bájthatáron, szóhatáron, dupla szóhatáron stb. kell elhelyezni. Felkérjük az olvasót, hogy nézzen utána a dolognak a programfejlesztő rendszere segítségében! Bárhogyan is, eme beállítások hatására a fordító minden struktúratagot, az elsőt követően, olyan határon tárol, mely megfelel a tag tárillesztési igényének.

A bájthatárra igazítás azt jelenti, hogy

- a struktúra objektum elhelyezése bármilyen címen kezdődhet, és
- a struktúratagok ugyancsak bármilyen címen elhelyezhetők típusuktól függetlenül.

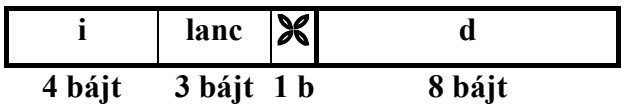
A példa **s** objektum összesen 15 bájtot foglal el ilyenkor, és a memória térkép a következő:



A szóhatárra igazítás azt jelenti, hogy

- a struktúra objektum kezdőcíme páros kell, hogy legyen, és
- a struktúratagok - a **char** típustól eltekintve - ugyancsak páros címeken helyezkednek el.

A példa **s** objektum így összesen 16 bájtot foglal el. Egy bájt elveszik, és a memória térkép a következő:



A dupla szóhatárra igazítás azt jelenti, hogy

- a struktúra objektum elhelyezése néggyel maradék nélkül osztható címen (dupla szóhatáron) történik meg,
- a **char** típusú struktúratagok bájthatáron kezdődnek,
- a **short** típusú tagok szóhatáron indulnak és
- a többi típusú tag dupla szóhatáron (néggyel maradék nélkül osztható címen) kezdődik.

Dupla szóhatárra igazítva az **s** objektum megegyezik az előzővel. A határra igazítási „játék” folytatható értelemszerűen tovább.

✍ Persze a struktúrát nem ilyen „után” definiálva igazítástól függetlenül elérhetjük, hogy egyetlen bájt elvesztése se következzék be:

```
struct struki {
  double d;
  int i;
  char lanc[3]; } s;
```

10.8 UNIÓK

Az unió típus a struktúrából származik, de a tagok között zérus a címeltolás. Az unió azt biztosítja, hogy ugyanazon a memória területen több, különféle típusú adatot tárolhassunk. Az

```
union unio{
  int i;
  double d;
  char t[5]; } u, *pu = &u, tu[23];
```

definícióban az **u union unio** típusú objektum, a **pu** ilyen típusú objektumra mutató mutató, és a **tu** egy 23 ilyen típusú elemből álló tömb azonosítója. Az **u** objektum - például - egyazon memória területen biztosítja az **i** nevű **int**, a **d** azonosítójú **double** és a **t** nevű karaktertömb típusú tagjainak elhelyezését, azaz:

```
&u ≡ &u.i ≡ &u.d ≡ ...
```

Ennek szellemében aztán igaz, hogy az unió objektumra mutató mutató annak egyben minden tagjára is mutat.

```
(pu=&u) ≡ &u.i ≡ &u.d ≡ ...
```

✍ Természetesen a dolog csak a mutatók értékére igaz, mert az **&u** (**union unio ***), az **&u.i** (**int ***) és az **&u.d** (**double ***), azaz típusban eltérnek. Ha azonban uniót megcímző mutatót explicit típusmódosítással tagjára irányuló mutatóvá alakítjuk, akkor az eredmény mutató magára a tagra mutat:

```
u.d=3.14;
printf("*(&u.d) = %f\n", *(double *)pu);
```

Az unió helyfoglalása a tárban akkora, hogy benne a legnagyobb bájtigényű tagja is elfér, azaz:

```
sizeof(union unio) ≡ sizeof(u) ≡ 8.
```

Tehát 4 bájt felhasználatlan, ha **int** adatot tartunk benne, ill. 3 bájt elérhetetlen, ha karaktertömböt rakunk bele. Az unió egy időben csak egyetlen tagját tartalmazhatja.

Láttuk már, hogy az uniótagokat ugyanazokkal a tagszelektor operátorokkal érhetjük el, mint a struktúratagokat:

```
u.d = 3.15;
printf("u.d=%f\n", u.d);  /* OK: u.d=3.15 jelenik meg. */
printf("u.i=%d\n", u.i);  /* Furcsa eredmény születik. */
printf("u.t[0]=%c\n", u.t[0]);/* Valami csak megjelenik,
                          vagy sem. */
printf("u.t=%s\n", u.t);  /* „Csoda" karakterlánc látszik.
                          Ki tudja, hol van a lánc vége!*/
strcpy(pu->t, "Hohó");
printf("u.t=%s\n", pu->t);/* OK: a „Hohó" látszik. */
printf("u.i=%d\n", pu->i);/* Furcsa eredmény születik. */
printf("u.d=%f\n", pu->d);/* Nagyon szorítsunk, hogy ne
                          legyen lebegőpontos túl vagy
                          alulcsordulás! */
```

Valahonnan tehát célszerű tudni - például úgy, hogy nyilvántartjuk - milyen típusú adat is található pillanatnyilag az unió objektumban, és azt szabad csak elérni.

Ha egy unió többféle, de azonos kezdő szerkezetű struktúrával indul, és az unió tartalma e struktúrák egyike, akkor lehetőség van az unió közös kezdeti részére hivatkozni. Például:

```
union{
  struct{ int tipus;} t;
  struct{ int tipus; int iadat;} ti;
  struct{ int tipus; double dadat;} td;
  /* . . . */ } u;
/* . . . */
u.td.tipus = DOUBLE;
u.td.dadat = 3.14;
/* . . . */
if(u.t.tipus == DOUBLE) printf("%f\n", u.td.dadat);
else if(u.t.tipus == INT) printf("%d\n", u.ti.iadat);
else /* . . . */
```

Uniókkal pontosan ugyanazok a műveletek végezhetők, mint a struktúrákkal. Hozzárendelhetők, másolhatók, hozzáférhetünk a tagjaikhoz, képezhető a címük, átadhatók függvényeknek és rutinok visszatérési értékei is lehetnek.

10.8.1 Uniódeklarációk

Az általános deklarációs szabály azonos a struktúráéval. Az eltérések a következők:

- Az uniók tartalmazhatnak bitmezőket. Mindegyik bitmező azonban az unió kezdetétől indul, s így közülük csak egy lehet aktív.

A következő fejezetben tárgyalt bitmezők gépfüggő ábrázolására itt is fel szeretnénk hívni külön a figyelmet!

- Az unió tagja nem lehet **void**, nem teljes, vagy függvény típusú. Nem lehet a definíció alatt álló unió egy példánya, de ilyenre mutató mutató persze lehet.
- Uniók esetében a deklarációban csak az elsőnek deklarált tagnak adható explicit kezdőérték. Például:

```
union unika{
  int i;
  double d;
  char t[6]; } u = { 24 };
```

Csak az **u.i** kaphatott, és kapott is 24 kezdőértéket. Ha kicsit bonyolultabb esetet nézünk:

```
union{
char x[2][3];
int i, j ,k; } y = {{{'1'}, {'4'}}};
```

Az **y** unió változó inicializálásakor aggregátum inicializátort használunk, mert az unió első tagja kétdimenziós tömb. Az '1' inicializátor a tömb első sorához tartozik, így az **y.x**[0][0] felveszi az '1' értéket, s a sor további elemei tiszta zérusok lesznek az implicit kezdőérték adás szabályai szerint. A '4' a második sor első elemének inicializátora, azaz **y.x**[1][0] = '4', **y.x**[1][1] = 0 és **y.x**[1][2] = 0.

- Lokális élettartamú uniók esetén az inicializátor kompatibilis unió típusú egyszerű kifejezés is lehet:

```
union unika{
  int i;
  double d;
  char t[6]; } u = { 24 }, u1 = u;
```

- Az uniódeklarációban is elhagyható az uniócímke. Az uniók előfordulhatnak struktúrákban, tömbökben, és tömbök, ill. struktúrák is lehetnek tagok uniókban:

```
#define MERET 20
struct {
  char *nev;
  int adat;
  int u_tipus;  /* Az unióban aktuálisan tárolt */
  union{        /* típus nyilvántartásához. */
    int i;
    float f;
    char *mutato; } u;
  } tomb[MERET];
```

Ilyenkor a **tomb i**-edik eleme **i** uniótagjához való hozzáférés alakja:

```
tomb[i].u.i
```

és a **mutato** tag mutatta első karakter elérésének formája:

```
*tomb[i].u.mutato
```

10.9 Bitmezők (bit fields)

Bitmezők csak struktúra vagy unió tagjaként definiálhatók, de a struktúrában és az unióban akár keverten is előfordulhatnak bitmező és nem bit-

mező tagok. A bitmező struktúratag deklarációs szintaktikája kicsit eltér a normál tagokétól:

típusspecifikátor <deklarátor> : konstans-kifejezés;

, ahol a *típusspecifikátor* csak

- **signed int**,
- **unsigned int** vagy
- **int**

lehet az ANSI C szabvány szerint. Az **int** tulajdonképpen **signed int.** A *deklarátor* a bitmező azonosítója, mely el is maradhat. Ilyenkor a névtelen bitmező specifikálta bitekre nem tudunk hivatkozni, s a bitek futásidejű tartalma előre megjósolhatatlan. A *konstans-kifejezés* csak egészértékű lehet. Zérus és **sizeof**(**int**)*8 közöttinek kell lennie, s a bitmező szélességét határozza meg.

💣 Bitmező csak struktúra vagy unió tagjaként deklarálható. Nem képezhető azonban bitmezők tömbje. Függvénynek sem lehet visszaadott értéke a bitmező. Nem megengedett a bitmezőre mutató mutató és tilos hivatkozni a bitmező tag címére, azaz nem alkalmazható rá a cím (&) operátor sem.

A bitmezők az **int** területen (dupla szóban, vagy szóban) deklarációjuk sorrendjében az alacsonyabb helyiértékű bitpozícióktól a magasabbak felé haladva foglalják el helyüket.

✍ Az **int** pontos mérete, bitmezővel való feltöltésének szabályai és sorrendje a programfejlesztő rendszertől függ. Célszerű tehát a segítségben utánanézni a dolognak. Maradjunk meg azonban az előző bekezdésben említett szabálynál, és a könnyebb szemléltethetőség végett még azt is tételezzük fel, hogy az **int** 16 bites! Ilyenkor például a:

```
struct bitmezo{
  int i: 2;
  unsigned j: 5;
  int : 4, k: 1;
  unsigned m: 4; } b, *pb = &b;
```

által elfoglalt szó bittérképe a következő:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
m				**k**					**j**					**i**	

Ha az **m** bitmező tag szélessége 4-nél nagyobb lett volna, akkor új szót kezdett volna a fordító, s az előző szó felső négy bitje kihasználatlan ma-

radt volna. Általánosságban: a (dupla)szón túllógó bitmező új (dupla)szót kezd, s az előző (dupla)szóban a felső bitek kihasználatlanok maradnak.

📖 Ha a deklarációban valamely (névtelen) bitmezőnél zérus szélességet adunk meg, akkor mesterségesen kényszerítjük ki ezt a következő (dupla) szóhatárra állást.

💣 A bitmezőnek elég szélesnek kell lennie ahhoz, hogy a rögzített bitminta elférjen benne! Például a következő tagdeklarációk illegálisak:

```
int alfa : 17;
unsigned beta : 32
```

A bitmezők ugyanazokkal a tagszelektor operátorokkal (. és ->) érhetők el, mint a nem bitmező tagok:

`b.i` vagy `pb->k`

A bitmezők kis **signed** vagy **unsigned** egész értékekként viselkednek (rögtön átesnek az egész–előléptetésen), azaz kifejezésekben ott fordulhatnak elő, ahol egyébként aritmetikai (egész) értékek lehetnek. **signed** esetben a legmagasabb helyiértékű bit (MSB - most significant bit) előjelbitként viselkedik, azaz az **int i** : 2 lehetséges értékei például:

```
00: 0, 01: +1, 10: -2, 11: -1
```

Az **unsigned m** : 4 lehetséges értékei:

```
0000: 0, 0001: 1, . . ., 1111: 15
```

Általánosságban:

```
unsigned x : szélesség;/* 0 <= x <= 2^szélesség-1 */
signed   y : szélesség;/* -2^szélesség-1<= y <=+2^szélesség-1-1 */
```

💣 A nyelvben nincs sem egész alul, sem túlcsordulás. Ha így a bitmezőnek ábrázolási határain kívüli értéket adunk, akkor abból is lesz „valami”. Méghozzá az érték annyi alsó bitje, mint amilyen széles a bitmező. Például:

```
b.i = 6;    /* 110 → 10, azaz -2 lesz az értéke! */
```

✍ A bitmezők ábrázolása gépfüggő, mint már mondottuk, azaz portábilis programokban kerüljük el használatukat!

Vegyük elő ismét a **Bit szintű operátorok** fejezetben tárgyalt dátum és időtárolási problémát! Hogyan tudnánk ugyanazt a feladatot bitmezőkkel megoldani?

Dátum:	Bitpozíció:
év – 1980	9 - 15
hónap	5 - 8
nap	0 - 4

Idő:	Bitpozíció:
óra	11 – 15
perc	5 – 10
két másodperc	0 – 4

A dátum és az idő adatot egy-egy szóban, azaz C nyelvi fogalmakkal egy-egy **unsigned short int**-ben tartjuk. A két szó bitfelosztása az ábrán látható!

A bitmezős megoldás például a következő is lehetne:

```
struct datum{
  unsigned short nap: 5, ho: 4, ev: 7; }
  d = { 8, 3, 1996-1980 };
struct ido{
  unsigned short mp2: 5, perc: 6, ora: 5; }
  i = { 2, 59, 11 };
/* . . . */
int ev=1996, ho=3, nap=8, ora=11, perc=59, mp=4;
/* Részeiből a dátum és az idő előállítása: */
d.ev = ev -1980;
d.ho = ho;
d.nap = nap;
i.ora = ora;
i.perc = perc;
i.mp2 = mp >> 1;
/* Ugyanez visszafelé: */
ev = d.ev +1980;
ho = d.ho;
nap = d.nap;
ora = i.ora;
perc = i.perc;
mp = i.mp2 << 1;
```

10.10 Balérték – jobbérték

Most már tökéletesen pontosíthatjuk a balérték kifejezést a C–ben, mely:

- Egész, lebegőpontos, mutató, struktúra vagy unió típusú azonosító.
- Indexes kifejezés, mely nem tömbbé (hanem elemmé) értékelhető ki.
- Tagszelektoros kifejezés (->, .).
- Nem tömbre hivatkozó, indirekciós kifejezés.
- Balérték kifejezés zárójelben.

💣 A **const** objektum nem módosítható balérték, hisz csak a deklarációban kaphat kezdőértéket.

A jobbérték (rvalue) olyan kiértékelhető kifejezés, melynek értékét balérték veheti fel. Például:

```
a = c + d;     /* OK */
c + d = a;     /* HIBÁS */
```

A balérték (lvalue) olyan kifejezés, mely eléri az objektumot (a hozzá allokált memória területet). Triviális például egy változó azonosítója. Lehet azonban ***P** alakú is, ahol a **P** kifejezést nem **NULL** mutatóra értékeli ki a fordító. Onnét is származtatható a két fogalom, hogy a balérték állhat a hozzárendelés operátor bal oldalán, s a jobbérték pedig a jobb oldalán.

Beszélhetünk módosítható balértékről is! Módosítható balérték nem lehet tömb típusú (a tömbazonosító praktikusan cím konstans), nem teljes típusú, vagy **const** típusmódosítóval ellátott objektum. Módosítható balérték például a konstans objektumra mutató mutató maga, miközben a mutatott konstans objektum nem változtatható. Például

```
int tomb[20];
```

esetén balértékek:

```
tomb[3] = 3; *(tomb+4) = 4;
```

A következő deklarációban viszont a **kar** nem balérték, hisz konstansságára való tekintettel értéket egyedül a definíciójában kaphat:

```
const char kar = 'k';
```

Azonban ha van egy

```
char *pozicio(int index);
```

függvény, akkor balérték lehet a következő is:

```
*pozicio(5) = 'z';
```

10.11 Névterületek

A névterület az a „hatáskör”, melyen belül egy azonosítónak egyedinek kell lennie, azaz más–más névterületen konfliktus nélkül használható ugyanaz az azonosító, s a fordító meg tudja különböztetni őket. A névterületeknek a következő fajtái vannak:

- Utasítás címke névterület: Az utasítás címkéknek abban a függvényben kell egyedinek lenniük, amelyben definiálták őket.
- Struktúra, unió és **enum** címke névterület: A struktúra, az unió és az **enum** címkék ugyanazon a névterületen osztoznak. Deklarálásuk

blokkjában kell egyedinek bizonyulniuk. Ha minden függvény testén kívül adják meg őket, akkor viszont fájl hatáskörben kell egyedinek lenniük.

- Struktúra és uniótagok (member) névterülete: A tagneveknek abban a struktúrában vagy unióban kell egyedinek lenniük, amelyben deklarálták őket. Különböző struktúrákban és uniókban előfordulhatnak ugyanazon tagazonosítók akár más típussal, s eltolással. Összesítve: mindenegyes struktúra és unió külön névterülettel rendelkezik.
- Normál azonosítók névterülete: Idetartozik minden más név, ami nem fért be az előző három névterületbe, azaz a változó, a függvény (beleértve a formális paramétereket, s a lokális változókat) és az enumerátorazonosítók. Abban a hatáskörben kell egyedinek bizonyulniuk, ahol definiálják őket. Például a fájl hatáskörű azonosítóknak ugyanebben a hatáskörben kell egyedinek lenniük.
- Típusdefiníció (**typedef**) nevek: Nem használhatók azonosítóként ugyanabban a hatáskörben. Magyarán a típusdefiníciós nevek a normál azonosítók névterületén vannak, de nem futásidejű azonosítók! Tehát, ha a helyzetből eldönthető, akkor lehet a típusdefiníciós név, és például egy lokális hatáskörű változó azonosítója egyforma is:

```
typedef char FT;
int fv(int lo){
  int FT;   /* Ez az FT egy int típusú lokális változó
               azonosítója. */
  /* . . . */ }
```

Nézzünk néhány példát!

```
struct s{
  int s; /* OK: a struktúratag újabb névterületen
            helyezkedik el. */
  float s;/*HIBÁS: így már két azonos tagnév lenne egy
            struktúrán belül. */
  } s;   /* OK: a normál változók névterülete különbözik
            minden eddig használttól. */
union s{ /* HIBA: az s struktúracímke is ezen a
            névterületen van. */
  int s; /* OK: hisz új tag névterület kezdődött. */
  float f;/*OK: más azonosítójú tag. */
  } f;   /* OK: hisz ez az f az normál változók
            névterületén található. */
struct t{
  int s; /* OK: hiszen megint újabb tag névterület
            kezdődött. */
```

```
  /* . . . */
  } s;   /* HIBA: s azonosító most már minden
            névterületen van. */
goto s;  /* OK: az utasítás címke és a struktúracímke
            más-más névterületen vannak. */
/* . . . */
s: ;     /* Utasítás címke. */
```

11 MAGAS SZINTŰ BEMENET, KIMENET

A magas szintű bemeneten és kimeneten olyan folyam, áram (stream) jellegű fájl, ill. eszköz (nyomtató, billentyűzet, képernyő stb.) kezelést értünk, ami a felhasználó szempontjából nézve szinte nincs tekintettel a mögöttes hardverre, s így a lehető legflexibilisebb kimenetet, bemenetet biztosítja.

A valóságban a folyamot egy **FILE** típusú struktúrára mutató mutatóval manipuláljuk. Ezt a struktúrát, a folyamkezelő függvények prototípusait stb. az **STDIO.H** fejfájlban definiálták. A struktúra például legyen a következő!

```
typedef struct{
  short level;          /* Puffer telítettségi szint. */
  unsigned short flags; /* Fájl állapotjelzők. */
  char fd;              /* Fájl leíró. */
  unsigned char hold;   /* ungetc kar., ha nincs puffer. */
  int bsize;            /* A puffer mérete. */
  unsigned char *buffer;/* A puffer címe. */
  unsigned char *curp;  /* Aktuális pozíció a pufferben. */
  /* . . . */
  } FILE;
```

A programunkban

```
FILE *fp;
```

deklarációs utasítással **FILE** típusú struktúrára mutató mutatót kell deklarálni, mely értéket a folyamot megnyitó **fopen**, **freopen** függvényektől kap. Tehát használat előtt a folyamot meg kell nyitni. Megnyitása a folyamot egy fájlhoz, vagy egy eszközhöz kapcsolja. Jelezni kell azt is ilyenkor, hogy a folyamot csak olvasásra, vagy írásra, vagy mind kettőre kívánjuk használni stb. Ezután elvégezhetjük a kívánt bemenetet, kimenetet a folyamon, majd legvégül le kell zárni.

11.1 Folyamok megnyitása

FILE ***fopen**(const char **fajlazonosito*, const char **mod*);

A függvény megnyitja a *fajlazonosito*val megnevezett fájlt, és folyamot kapcsol hozzá. Visszaadja a fájlinformációt tartalmazó **FILE** struktúrára mutató mutatót, mely a rákövetkező műveletekben azonosítani fogja a folyamot, ill. **NULL** mutatót kapunk tőle, ha a megnyitási kísérlet sikertelen volt.

🕮 A *fajlazonosito* természetesen tartalmazhat (esetleg meghajtó nevet) utat is, de a maximális összhossza **FILENAME_MAX** karakter lehet.

A második paraméter *mod* karakterlánc meghatározza a későbbi adatátvitel irányát, helyét és a folyam típusát. Nézzük a lehetőségeket!

r	Megnyitás csak olvasásra.
w	Létrehozás írásra. A már létező, ilyen azonosítójú fájl tartalma megsemmisül.
a	Hozzáfűzés: megnyitás írásra a fájl végén, vagy létrehozás írásra, ha a fájl eddig nem létezett.
r+	Egy létező fájl megnyitása felújításra (írásra és olvasásra).
w+	Új fájl létrehozása felújításra. A létező fájl tartalma elvész.
a+	Megnyitás hozzáfűzésre: a fájl végén felújításra, vagy új fájl létrehozása felújításra, ha a fájl eddig nem létezett.

A folyam típusa szöveges (text), vagy bináris lehet. A szöveges folyam a bemenetet és a kimenetet sorokból állóknak képzeli el. A sorok végét egy '\n' (LF) karakter jelzi. Lemezre történő kimenet esetén a folyam a sorlezáró '\n' karaktert "\r\n" karakter párral (CR-LF) helyettesíti. Megfordítva: lemezes bemenetnél a CR-LF karakter párból ismét LF karakter lesz. Ezt a manipulációt transzlációnak nevezzük. Bemenet esetén a folyam a 0X1A értékű karaktert fájlvégnek tekinti. Összegezve: a szöveges folyam bizonyos, kitüntetett karaktereket speciálisan kezel, míg a bináris folyam ilyent egyetlen karakterrel sem tesz.

Elismerjük természetesen, hogy nincs transzláció mindenegyes operációs rendszerben.

A *mod* karakterláncban expliciten megadhatjuk a folyam típusát. A szövegest a 't', a binárist a 'b' jelöli. A folyamtípus karakter a karakterláncban az első betű után bárhol elhelyezhető, azaz megengedettek az

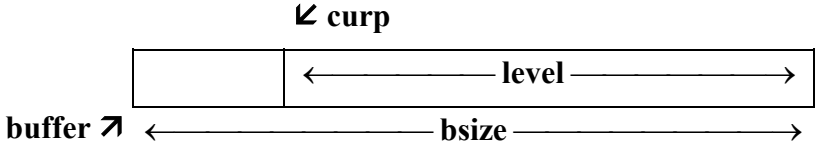
`rt+, r+t` stb.

Nem kötelező azonban a folyamtípust a *mod* karakterláncban expliciten megadni. Ha elhagyjuk, alapértelmezés a szöveges.

Ha a folyamot felújításra (update) nyitották meg, akkor megengedett mind a bemenet, mind a kimenet. A kimenetet azonban **fflush**, vagy pozícionáló (**fseek**, **rewind** stb.) függvény hívása nélkül nem követheti közvetlenül bemenet. A fordított adatirányváltás is csak fájlvégen, vagy e függvények hívásának közbeiktatásával valósítható meg.

11.2 Folyamok pufferezése

A fájlokhoz kapcsolt folyamok szokásosan pufferezettek, s a puffer lefoglalása megnyitáskor automatikusan megtörténik **malloc** hívással. Ez is megengedi azonban az „egy karakteres szintű" bemenetet, kimenetet (**getc**, **putc**), ami nagyon gyors. A pufferrel kapcsolatos információkat a **FILE** struktúra tagjai írják le:



, ahol **buffer** a puffer kezdőcíme és **bsize** a mérete. A **curp** a pufferbeli aktuális pozícióra mutat, s **level** pedig számlálja, hogy még hány karakter van hátra a pufferben. A teljes pufferezettség azt jelenti, hogy kiírás automatikusan csak akkor történik, ha a puffer teljesen feltelt, ill. olvasás csak akkor következik be, ha a puffer teljesen kiürült. Egy karakter írása, vagy olvasása a **curp** pozícióról, ill. pozícióra történik, s a művelet mellékhatásaként a **curp** eggyel nő, s a **level** eggyel csökken.

A pufferezetlenség azt jelenti, hogy a bájtok átvitele azonnal megtörténik a fájlba (fájlból), vagy az eszközre (eszközről).

A mai operációs rendszerek legtöbbje a kisebb fájlokat megnyitásuk után valamilyen rendszer területen (cash) tartja, s a pufferek is csak a memóriabeli fájllal vannak kapcsolatban. Célszerű tehát, a programfejlesztő rendszer segítségében utánanézni, hogy az azonnali fájlba írás, vagy olvasás pontosan hogyan valósítható meg, ha igazán szükség van rá.

A **setbuf** és a **setvbuf** függvényhívásokkal kijelölhetünk saját puffert, módosíthatjuk a használatos puffer méretét, vagy pufferezetlenné tehetjük a bemenetet és a kimenetet.

void **setbuf**(FILE **stream*, char **puff*);

A függvény az automatikusan allokált (**malloc**) puffer helyett a *puff* puffert használtatja a *stream* folyammal adatátvitel esetén. Ha a *puff* paraméter **NULL** mutató, akkor a folyam pufferezetlen lesz, máskülönben a folyam teljesen pufferezett. A puffer különben **BUFSIZ** méretű.

A szabvány bemenet (**stdin**) sorpufferezett és a szabvány kimenet (**stdout**) pufferezetlen, ha nincsenek az operációs rendszerben átirányítva, mert ekkor mindkettő teljesen pufferezett. A sorpufferezettség azt jelenti, hogy ha a puffer üres, a következő bemeneti művelet megkísérli a teljes

puffer feltöltését. Kimenet esetén mindig kiürül a puffer, ha teljesen feltelik, ill. amikor '\n' karaktert írunk bele.

💣 Előre megjósolhatatlan hiba következik be, ha a **setbuf** függvényt nem közvetlenül a folyam megnyitása után hívják meg. Legális lehet még a pufferezetlen folyamra vonatkozó **setbuf** hívás, bárhol is következik be.

💣 Vigyázzunk a puffer **auto** tárolási osztályú deklarációjával, mert akkor csak abból a függvényből lesz elérhető, ahol deklaráltuk! Még „szarvasabb" a hiba, ha kilépünk a folyam lezárása nélkül abból a függvényből, melyre nézve a pufferünk lokális volt.

int **setvbuf**(FILE **stream*, char **puff*, int *tipus*, size_t *meret*);

A függvény ugyanazt teszi, mint a **setbuf**. Látható azonban, hogy expliciten megadható a puffer *tipus*a és *meret*e. A **size_t** típusból következőleg nagy méretű puffer is előírható. A pufferezetlenség ezzel a függvénnyel a *tipus* paraméter megfelelő megadásával érhető el, ugyanis ha az aktuális *puff* paramétert **NULL** mutatónak választjuk, akkor a rutin **malloc**–kal foglal memóriát a puffernek.

A *tipus* paraméter lehetséges értékei a következők:

- **_IOFBF**: A fájl teljesen pufferezett. Ha kiürül, a következő bemeneti művelet megkísérli teljesen feltölteni a puffert. Kimenet esetén fájlba írás automatikusan csak akkor történik, ha a puffer teljesen feltelt.
- **_IOLBF**: A fájl sorpufferezett.
- **_IONBF**: A fájl pufferezetlen. A *puff* és a *meret* paraméter figyelmen kívül marad. Minden bemeneti és kimeneti művelet közvetlen adatátvitelt jelent a fájlba.

A **setvbuf** zérust ad vissza sikeres esetben, és nem zérust kapunk, ha a megadott *tipus*, vagy a *meret* paraméter érvénytelen, vagy nincs elég memória a puffer allokálásához.

Nézzünk egy példát!

```
#include <stdio.h>
char puff[BUFSIZ];
void main(void) {
FILE *input, *output;
if((input=fopen("file.in","r"))!=NULL){
  if(output=fopen("file.out","w")){
    if(setvbuf(input,puff,_IOFBF,BUFSIZ))
      printf("Sikertelen a saját input puffer "
             "allokálása!\n");
```

```
      /* A bemeneti folyam saját puffert használva,
         minimális lemezhez fordulással műveletre kész. */
      if(setvbuf(output,NULL,_IOLBF,128))
        printf("Az output puffer allokálása "
               "sikertelen!\n");
      else{
        /* A kimeneti folyam sorpufferezetten, malloc
           hívással allokált pufferrel műveletre kész. */
        /* Itt intézhető a fájlkimenet és bemenet! */ }
      /* Fájlok lezárása. */
      fclose(output); }
    else printf("Az output fájl megnyithatatlan!\n");
    fclose(input); }
  else printf("Az input fájl megnyitása sikertelen!\n"); }
```

Eddig csak a pufferek automatikus ürítéséről beszéltünk. Lehetséges azonban a pufferek kézi ürítése is. Sőt, adatátviteli irányváltás előtt a kimeneti puffert ki is kell üríteni. Lássuk a függvényt!

int **fflush**(FILE **stream*);

Kimenetre nyitott folyam esetén a rutin kiírja a puffer tartalmát a kapcsolt fájlba.

Bemeneti folyamnál a függvény eredménye nem definiálható, de többnyire törli a puffer tartalmát.

Mindkét esetben nyitva marad a folyam.

Pufferezetlen folyamnál e függvény hívásának nincs hatása.

Sikeres esetben zérust kapunk vissza. Hibás esetben a szolgáltatott érték **EOF**.

Az **fflush**(**NULL**) üríti az összes kimeneti folyamot.

11.3 Pozícionálás a folyamokban

A folyamokat rendszerint szekvenciális fájlok olvasására, írására használják. A magas szintű bemenet, kimenet a fájlt bájtfolyamnak tekinti, mely a fájl elejétől (0 pozíció) indul és a fájl végéig tart. A fájl utolsó pozíciója a fájlméret - 1. Az adatátvitel mindig az aktuális fájlpozíciótól kezdődik, megtörténte után a fájlpozíció a fájlban következő, át nem vitt bájtra mozdul. A fájlpozíciót fájlmutatónak is szokás nevezni.

Eszközhöz kapcsolt folyam mindig csak szekvenciálisan (zérustól induló, monoton növekvő fájlpozícióval) érhető el. Lemezes fájlhoz kapcsolt folyam bájtjai azonban direkt (random) módon is olvashatók és írhatók.

Lemezes fájlok esetén a fájlmutató adatátvitel előtti beállítását az

int **fseek**(FILE **stream*, long *offset*, int *ahonnet*);

függvénnyel végezhetjük el, mely a *stream* folyam fájlmutatóját *offset* bájttal az *ahonnet* paraméterrel adott fájlpozíción túlra állítja be. Szöveges folyamokra az *offset* zérus lehet, vagy egy az **ftell** függvény által visszaadott érték.

Az *ahonnet* paraméter a következő értékeket veheti fel:

- **SEEK_SET**: A fájl kezdetétől.
- **SEEK_CUR**: Az aktuális fájlpozíciótól.
- **SEEK_END**: A fájl végétől.

A függvény elvet minden a bemenetre **ungetc**–vel visszarakott karaktert.

☞ Az **ungetc**–ről a következő fejezetben lesz szó!

Felújításra megnyitott fájl esetén az **fseek** után mind bemenet, mind kimenet következhet.

A függvény törli a fájlvég jelzőt.

☞ Lásd a fájl állapotjelzői közt még ebben a fejezetben!

A függvény zérust ad vissza, ha a fájlpozícionálás sikeres volt, ill. nem zérust kapunk hiba esetén.

Írjunk fájlméretet megállapító függvényt!

```
#include <stdio.h>
long fajlmeret(FILE *stream) {
  long aktpoz, hossz;
  aktpoz=ftell(stream);
  fseek(stream, 0L, SEEK_END);
  hossz=ftell(stream);
  fseek(stream, aktpoz, SEEK_SET);
  return(hossz); }
```

☞ A **fajlmeret** elteszi a pillanatnyi pozíciót az **aktpoz** változóba, hogy a fájl végére állítás után helyre tudja hozni a fájlmutatót. A lekérdezett fájlvég pozíció éppen a fájlméret.

Nézzük a további fájlpozícióval foglalkozó függvényeket!

long int **ftell**(FILE **stream*);

A rutin visszaadja a *stream* folyam aktuális fájlpozícióját sikeres esetben, máskülönben -1L-t kapunk tőle. A

void **rewind**(FILE **stream*);

a *stream* folyam fájlmutatóját a fájl elejére állítja.

Felújításra megnyitott fájl esetén a **rewind** után mind bemenet, mind kimenet következhet.

A függvény törli a fájlvég és a hibajelző biteket.

A **FILE** struktúra **flags** szava bitjei (állapotjelzői) a következő jelentésűek lehetnek!

```
#define  _F_RDWR   0x0003 /* olvasás és írásjelző */
#define  _F_READ   0x0001 /* csak olvasható fájl */
#define  _F_WRIT   0x0002 /* csak írható fájl */
#define  _F_BUF    0x0004 /* malloc pufferelt */
#define  _F_LBUF   0x0008 /* sorpufferelt fájl */
#define  _F_ERR    0x0010 /* hibajelző */
#define  _F_EOF    0x0020 /* fájlvég jelző */
#define  _F_BIN    0x0040 /* bináris fájl jelző */
/* . . . */
```

A megadott *stream* folyam aktuális fájlpozícióját helyezi el az

int **fgetpos**(FILE **stream*, fpos_t **pos*);

a *pos* paraméterrel adott címen. Ez a érték felhasználható az **fsetpos**–ban. A visszaadott érték zérus hibátlan, és nem zérus sikertelen esetben. Az

int **fsetpos**(FILE **stream*, const fpos_t **pos*);

a *stream* folyam fájlmutatóját állítja be a *pos* paraméterrel mutatott értékre.

Felújításra megnyitott fájl esetén az **fsetpos** után mind bemenet, mind kimenet következhet.

A függvény törli a fájlvég jelző bitet, és elvet minden, e fájlra vonatkozó **ungetc** karaktert.

A visszakapott érték egyezik az **fgetpos**–nál írottakkal.

☞ Vegyük észre, hogy az **fseek** és az **ftell long** értékekkel dolgozik. A maximális fájlméret így 2GB lehet. Az **fpos_t** adattípus e korlát áttörését biztosítja, hisz mögötte akár 64 bites egész is lehet.

11.4 Bemeneti műveletek

int **fgetc**(FILE **stream*);

A folyam következő **unsigned char** karakterét adja vissza előjel kiterjesztés nélkül **int**–té konvertáltan, s eggyel előbbre állítja a fájlpozíciót. Sikertelen esetben, ill. fájl végén **EOF**–ot kapunk. A

int **getc**(FILE **stream*);

makró, mint ahogyan ez a lehetséges definíciójából is látszik, ugyanezt teszi:

```
#define getc(f) \
((--((f)->level)>=0) ? (unsigned char)(*(f)->curp++) :\
                       _fgetc(f))
```

int **ungetc**(int *c*, FILE **stream*);

A függvény visszateszi a *stream* bemeneti folyamba a *c* paraméter **unsigned char** típusúvá konvertált értékét úgy, hogy a következő olvasással ez legyen az első elérhető karakter. A szabályos működés csak egyetlen karakter visszahelyezése esetén garantált, de a visszatett karakter nem lehet az **EOF**.

✍ Két egymást követő **ungetc** hívás hatására már csak a másodiknak visszatett karakter érhető el, mondjuk, a következő **getc**–vel, azaz az első elveszik. Gondoljuk csak meg, hogyha nincs puffer, akkor a visszatételhez a **FILE** struktúra egyetlen **hold** tagja áll rendelkezésre!

Az **fflush**, az **fseek**, az **fsetpos**, vagy a **rewind** törli a bemenetre visszatett karaktert.

Sikeres híváskor az **ungetc** a visszatett karaktert adja vissza. Hiba esetén viszont **EOF**–ot kapunk tőle. Az

char ***fgets**(char **s*, int *n*, FILE **stream*);

karakterláncot hoz be a *stream* folyamból, melyet az *s* címtől kezdve helyez el a memóriában. Az átvitel leáll, ha a függvény *n* - 1 karaktert, vagy '\n'–t olvasott. A rutin a '\n' karaktert is kiteszi a láncba, és a végéhez még záró '\0'–t is illeszt.

Sikeres esetben az **fgets** az *s* karakterláncra mutató mutatóval tér vissza. Fájlvégen, vagy hiba esetén viszont **NULL**–t szolgáltat.

✍ Vegyük észre, hogy a jegyzet eleje óta használt **getline** függvény csak annyiban tér el az **fgets**–től, hogy:

- A beolvasott karakterlánc méretét adja vissza.
- A szabvány bemenetről (**stdin**) olvas, s nem más folyamból, így eggyel kevesebb a paramétere.
- *n* karaktert hoz be legfeljebb, vagy '\n'–ig, de magát a soremelés karaktert nem teszi be az eredmény karakterláncba.

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

A függvény *n* * *size* bájtot olvas a *stream* folyamból, melyet a *ptr* paraméterrel mutatott címen helyez el. Visszaadott értéke nem a beolvasott bájtok száma, hanem a

```
beolvasott bájtok száma / size
```

sikeres esetben. Hiba, vagy fájlvég esetén ez persze nem egyezik *n*–nel.

Az eddig ismertetett bemeneti függvények nem konvertálták a beolvasott karakter(lánco)t. Az

int **fscanf**(FILE **stream*, const char **format*<, *cim*, ...>);

viszont a *stream* folyamból karakterenként olvasva egy sor bemeneti mezőt vizsgál. Aztán minden mezőt a *format* karakterláncnak megfelelően konvertál, és letárol rendre a paraméter *cim*eken. A *format* karakterláncban ugyanannyi konverziót okozó formátumspecifikációnak kell lennie, mint ahány bemeneti mező van.

✍ A jelölésben a <> az elhagyhatóságot, a ... a megelőző paraméter tetszőleges számú ismételhetőségét jelenti. A bemeneti mező definíciója, a formázás és a konvertálás részletei a **scanf** függvény leírásában találhatók meg!

Az **fscanf** a sikeresen vizsgált, konvertált és letárolt bemeneti mezők számával tér vissza. Ha a függvény az olvasást a fájl végén kísérelné meg, vagy valamilyen hiba történne, akkor **EOF**-ot kapunk tőle vissza. A rutin zérussal is visszatérhet, ami azt jelenti, hogy egyetlen vizsgált mezőt sem tárolt le.

11.5 Kimeneti műveletek

int **fputc**(int *c*, FILE **stream*);

A függvény a *c* **unsigned char** típusúvá konvertált értékét írja ki a *stream* folyamba. Sikeres esetben a *c* karaktert kapjuk vissza tőle, hiba bekövetkeztekor viszont **EOF**–ot. A

int **putc**(int *c*, FILE **stream*);

makró, mint ahogyan ez a lehetséges definíciójából is látszik, ugyanezt teszi:

```
#define putc(c,f) \
((++((f)->level)<0) ? (unsigned char)(*(f)->curp++)=(c)) :\
                      _fputc((c),f))
```

int **fputs**(const char **s*, FILE **stream*);

A függvény az *s* karakterláncot kiírja a *stream* folyamba. Nem fűz hozzá '\n' karaktert, és a lezáró '\0' karakter sem kerül át.

Sikeres esetben nem negatív értékkel tér vissza. Hiba esetén viszont **EOF**–ot kapunk tőle. Az

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

a *ptr* címmel mutatott memória területről *n* * *size* bájtot ír ki a *stream* folyamba. Visszaadott értéke nem a kiírt bájtok száma, hanem a

```
kiírt bájtok száma / size
```

sikeres esetben. Hiba bekövetkeztekor ez nem egyezik *n*–nel.

Az eddigi kimeneti függvények nem végeztek konverziót. Az

int **fprintf**(FILE **stream*, const char **format*<, *parameter*, ...>);

fogad egy sor *parameter*t, melyeket a *format* karakterláncnak megfelelően formáz (konvertál), és kivisz a *stream* folyamba. A *format* karakterláncban ugyanannyi konverziót okozó formátumspecifikációnak kell lennie, mint ahány *parameter* van.

A jelölésben a <> az elhagyhatóságot, a ... a megelőző paraméter tetszőleges számú ismételhetőségét jelenti. A formázás és a konvertálás részletei a **printf** függvény leírásában találhatók meg!

Az **fprintf** a folyamba kivitt karakterek számával tér vissza sikeres esetben, ill. **EOF**-ot kapunk tőle hiba bekövetkeztekor.

11.6 Folyamok lezárása

int **fclose**(FILE **stream*);

A rutin lezárja a *stream* folyamot. Ez előtt azonban üríti a folyamhoz tartozó puffert, s a pufferhez automatikusan allokált memóriát fel is szabadítja.

Ez utóbbi nem vonatkozik a **setbuf**, vagy a **setvbuf** függvényekkel hozzárendelt pufferekre. Ezek „ügyei" csak a felhasználóra tartoznak.

Sikeres esetben az **fclose** zérussal tér vissza. Hiba esetén viszont **EOF**–ot kapunk tőle.

11.7 Hibakezelés

Tudjuk, hogy a szabvány könyvtári függvények – így a magas szintű bemenetet, kimenetet kezelők is – a hibát, a kivételes esetet úgy jelzik, hogy valamilyen speciális értéket (**EOF**, **NULL** mutató, **HUGE_VAL** stb.) adnak vissza, és az **errno** globális hibaváltozóba beállítják a hiba

kódját. A hibakódok az **ERRNO.H** fejfájlban definiált, egész, nem zérus-értékű szimbolikus állandók.

Programunk indulásakor a szabvány bemeneten (**stdin**) és kimeneten (**stdout**) túl a hibakimenet (**stderr**) is rendelkezésre áll, s a hibaüzeneteket ez utóbbin célszerű megjelentetni. Az **stderr** a képernyő (karakteres ablak) alapértelmezés szerint, s nem is irányítható át fájlba a legtöbb operációs rendszerben, mint ahogyan a bemenettel (<) és a kimenettel (>) ez megtehető volt a programot futtató parancssorban.

Mit jelentessünk meg hibaüzenetként az **stderr**–en?

Természetesen bármilyen szöveget kiírathatunk, de a hibakódokhoz programfejlsztő rendszertől függően hibaüzenet karakterláncok is tartoznak, s ezek is megjelentethetők. A *hibakod*hoz tartozó hibaüzenet karakterlánc kezdőcímét szolgáltatja a szabványos

#include <**STRING.H**>

char ***strerror**(int *hibakod*);

függvény, s az üzenet meg is jelentethető

```
fprintf(stderr, "Hiba: %s\n", strerror(hibakod));
```

módon. A

void **perror**(const char **s*);

kiírja az **stderr**–re azt a hibaüzenetet, melyet a legutóbbi hibát okozó, könyvtári függvény hívása idézett elő. Először megjelenteti az *s* karakterláncot a rutin, aztán kettőspontot (:) tesz, majd az **errno** aktuális értékének megfelelő üzenet karakterláncot írja ki lezáró '\n'-nel.

✍ Tehát például a **perror**("Hiba: ") megfelel a

```
fprintf(stderr, "Hiba: %s\n", strerror(errno));
```

függvényhívásnak.

💣 Vigyázat! Az **errno** értékét csak közvetlenül a hibát okozó rutin hívása után szabad felhasználni, mert a következő könyvtári függvény megidézése felülírhatja e globális változó értékét. Ha a hibakóddal mégis később kívánnánk foglalkozni, akkor tegyük el az **errno** értékét egy segédváltozóba!

✍ Folyamokkal kapcsolatban a **perror** *s* paramétere a fájlazonosító szokott lenni.

Meg kell tárgyalnunk még három, csak a folyamok hibakezelésével foglalkozó függvényt! A

void **clearerr**(FILE ***stream**);

nullázza a *stream* folyam fájlvég és hibajelzőjét. Ha a folyam hibajelző bitje egyszer bebillent, akkor minden a folyamon végzett művelet hibával tér vissza mindaddig, míg a hibajelzőt e függvénnyel, vagy a **rewind**–dal nem törlik.

A fájlvég jelző bitet egyébként minden bemeneti művelet nullázza. Az

int **feof**(FILE ***stream**);

többnyire makró, mely a következő lehetséges

```
#define feof(f) ((f)->flags & _F_EOF)
```

definíciója miatt, visszaadja a fájlvég jelző bit állapotát, azaz választ ad a fájlvég van-e kérdésre.

Az egyszer bebillent fájlvég jelző bit a következő, e folyamra vonatkozó bemeneti, pozícionáló műveletig, vagy **clearerr**–ig 1 marad. Az

int **ferror**(FILE ***stream**);

makró ebben a szellemben

```
#define ferror(f) ((f)->flags & _F_ERR)
```

a hiba jelző bit állapotát adja vissza.

Az egyszer bebillent hiba jelző bitet csak a **clearerr** és a **rewind** függvények törlik.

Ha a kérdéses folyammal kapcsolatban a bebillent hiba jelző bit törléséről nem gondoskodunk, akkor minden e folyamra meghívott további függvény hibát jelezve fog visszatérni.

Írjuk meg az **fputc** segítségével az **fputs** függvényt!

```
int fputs(const char *s, FILE *stream){
  int c;
  while(c=*s++)
    if(c!=fputc(c, stream)) break;
  return ferror(stream) ? EOF : 1; }
```

Készítsünk szoftvert komplett hibakezeléssel, mely az első parancssori paramétere fájlt átmásolja a második paramétere azonosítójú fájlba! Ha a programot nem elég parancssori paraméterrel indítják, akkor ismertesse használatát! A másolási folyamat előrehaladásáról tájékoztasson feltétlenül!

```
/* PELDA31.C: Első paraméter fájl másolása a másodikba. */
#include <stdio.h>
#include <string.h>/* strerror miatt! */
```

```
#include <errno.h> /* Az errno végett! */
#define KENT 10    /* Hányanként jelenjen meg a számláló.*/
#define SZELES 10  /* Mezőszélesség a számló közléséhez. */
int main(int argc, char *argv[]){
  FILE *be, *ki;   /* A be és kimeneti fájlok. */
  long szlo=0l;    /* A számláló. */
  int c;           /* A köv. karakter és segédváltozó. */
  printf("Az első paraméter fájl másolása a másodikba:\n");
  if(argc<3){
    fprintf(stderr, "Programindítás:\n"
             "PELDA30 forrásfájl célfájl\n");
    return 1; }
  if(!(be=fopen(argv[1],"rb"))){
    perror(argv[1]);
    return 1; }
  if(!(ki=fopen(argv[2],"wb"))){
    perror(argv[2]);
    fclose(be);
    return 1; }
  printf("%s --> %s:\n%*ld",argv[1],argv[2], SZELES, szlo);
```

A formátumspecifikációbeli *** SZELES** mezőszélességet eredményez.

```
  while((c=fgetc(be))!=EOF){/* Olvasás fájlvégig, vagy
                               hibáig. */
    if(c==fputc(c,ki)){     /* Kiírás rendben. */
      if(!(++szlo%KENT)){
        for(c=0; c<SZELES; ++c) fputc('\b', stdout);
        printf("%*ld", SZELES, szlo); } }
    else{                   /* Kiírásnál hiba van. */
      perror(argv[2]);
      clearerr(ki);
      if(!fclose(ki))       /* A félkész fájl törlése. */
        remove(argv[2]);
      else perror(argv[2]);
      fclose(be);
      return 1; } }
  /* Az olvasás EOF értékkel fejeződött be. */
  c=errno;                  /* Hibakód mentése. */
  fclose(ki);
  /* A végső méret kiírása: */
  for(c=0; c<SZELES; ++c) fputc('\b', stdout);
  printf("%*ld\n", SZELES, szlo);
  if(ferror(be)){           /* Hiba volt. */
    fprintf(stderr, "%s: %s\n", argv[1], strerror(c));
    clearerr(be);
    fclose(be);
    remove(argv[2]);
    return 1; }
  fclose(be);               /* Minden rendben ment. */
```

```
return 0; }
```

☞ Az **stdout**–ra irányuló műveletek hibakezelésével azért nem foglalkoztunk, mert ahol az sem működik, ott az operációs rendszer sem megy.

Megoldandó feladatok:

Fokozzuk kicsit a **PELDA31.C**–ben megvalósított feladatot! A fájlba írás normál esetben akkor nem megy, ha betelik a lemez. Ezen próbáljunk meg úgy segíteni, hogy a forrásfájl megnyitása után állapítsuk meg a méretét! A célfájlnak is foglaljunk helyet (**fseek**) ugyanekkora méretben, majd felújítva írjuk ki rá a forrás tartalmát!

Készítsen szoftvert, mely eldönti az indító parancssorban megadott azonosítójú fájl típusát, azaz hogy szöveges, vagy bináris! Ha parancssori paraméter nélkül futtatják a programot, akkor ismertesse a használatát!

Írjon szoftvert, mely az indító parancssorban megadott szövegfájlokat egyesíti a megadás sorrendjében a parancssorban utolsóként előírt azonosítójú szövegfájlba! Ha parancssori paraméter nélkül indítják a programot, akkor ismertesse a képernyőn, hogyan kell használni! Ha csak egy fájlazonosító van a parancssorban, akkor a szabvány bemenet másolandó bele. A fájlok egyesítése során a folyamat előrehaladásáról tájékoztatni kell a képernyőn! A szabvány bemenet másolása esetén végül közlendő még az eredményfájl mérete!

11.8 Előre definiált folyamok

Egy időben legfeljebb **FOPEN_MAX**, vagy **OPEN_MAX** folyam (fájl) lehet megnyitva. Ennek megfelelő a globális **FILE** struktúratömb

```
extern FILE _streams[];
```

mérete is, melyből ráadásul még az első három bizonyosan foglalt is:

```
#define  stdin  (&_streams[0])
#define  stdout (&_streams[1])
#define  stderr (&_streams[2])
```

☞ A globális **FILE** struktúratömb neve persze lehet ettől eltérő is.

Ezek az előre definiált folyamok, melyek programunk futásának megkezdésekor már megnyitva rendelkezésre állnak.

Név	B/K	Típus	Folyam	Alapér-telmezés
stdin	bemenet	szöveges	szabványos bemenet	**CON:**
stdout	kimenet	szöveges	szabványos kimenet	**CON:**
stderr	kimenet	szöveges	szabvány hibakimenet	**CON:**

Az **stdin** és az **stdout** átirányítható a programot indító parancssorban szövegfájlba.

```
program < bemenet.txt > kimenet.txt
```

Ha nincsenek átirányítva, akkor az **stdin** sorpufferezett, s az **stdout** pedig pufferezetlen. Ilyen az **stderr** is, tehát pufferezetlen. A legtöbb operációs rendszerben cső (pipe) is használható. Például:

```
program1 | program2 | program3
```

program1 a rendszerben beállított szabvány bemenettel rendelkezik. Szabvány kimenete szabvány bemenete lesz *program2*–nek, aminek szabvány kimenete *program3* szabvány bemenete. Végül *program3* szabvány kimenete az, amit a rendszerben beállítottak.

Mindhárom előre definiált folyam átirányítható a programban is, azaz ha nem felelne meg az alapértelmezés szerint a folyamhoz kapcsolt eszköz, akkor ezt kicserélhetjük az

FILE ***freopen**(const char **fajlazonosito*, const char ***mod*,

FILE ***stream*);

függvénnyel a *fajlazonosito*jú fájlra. A rutin első két paraméterének értelmezése és visszaadott értéke egyezik az **fopen**–évek. A harmadik viszont az előre definiált folyam: **stdin**, **stdout** vagy **stderr**.

✍ Az **freopen** persze nem csak előre definiált folyamokra használható, hanem bármilyen mással is, de ez a legjellemzőbb alkalmazása.

Készítsünk programot, mely a szabvány bemenetről érkező karaktereket a parancssori paraméterként megadott szövegfájlba másolja! Ha indításkor nem adnak meg parancssori paramétert, akkor csak echózza a szoftver a bementet a kimeneten!

A feladatot az **stdout** átirányításával oldjuk meg.

```
/* PELDA32.C: Bemenet másolása fájlba stdout-ként. */
#include <stdio.h>
#include <stdlib.h>  /* A system rutin miatt! */
#define PUFF 257     /* A bemeneti puffer mérete. */
```

```
int main(int argc, char *argv[]){
  char puff[PUFF];   /* Bemeneti puffer. */
  if(system(NULL)) system("CLS");
  printf("A szabvány bemenet fájlba másolása "
         "Ctrl+Z-ig:\n");
  if(argc<2) printf("A program indítható így is:\n"
                    "PELDA32 szövegfájl\n\n");
  else if(!freopen(argv[1],"wt", stdout)){
       perror(argv[1]);
       return 1; }
  while(fgets(puff, PUFF, stdin)){
    if(fputs(puff, stdout)<0){
      perror(argv[1]);
      clearerr(stdout);
      if(!fclose(stdout)) remove(argv[1]);
      else perror(argv[1]);
      return 1; } }
  return 0; }
```

📖 Az **STDLIB.H** bekapcsolásával rendelkezésre álló

int **system**(const char **parancs*);

rutin *parancs* paraméterét átadja végrehajtásra az operációs rendszernek (a parancsértelmezőnek), azaz végrehajtatja a rendszerrel a *parancs*ot. A függvény visszatérési értéke a programfejlesztő rendszertől függ, de többnyire a parancsértelmező által szolgáltatott érték az.

Ha a *parancs* **NULL**, akkor a rutin a parancsértelmező létezéséről számol be, azaz ha van, nem zérussal tér vissza, és zérust szolgáltat, ha nincs.

11.8.1 Bemenet az stdin-ről

int **getchar**(void);

A függvény makró, azaz:

```
#define getchar() getc(stdin)
```

A

char ***gets**(char *s);

az **fgets**–hez hasonlóan karaktereket olvas az **stdin**–ről, melyeket rendre elhelyez a paraméter *s* karaktertömbben. A visszaadott értéke is egyezik az **fgets**–évek, azaz normál esetben *s*–t szolgáltatja, s fájlvég vagy hiba bekövetkeztekor **NULL**–t. Az **stdin**–ről való olvasás azonban az első ’\n’ karakterig tart. Magát az LF karaktert nem viszi át az *s* tömbbe, hanem helyette a karakterláncot záró ’\0’–t ír oda.

A konverziót is végző

int **scanf**(const char **format*<, *cim*, ...>);

függvény az **fscanf–**hoz hasonlóan – de az **stdin** folyamból – olvasva egy sor bemeneti mezőt vizsgál. Aztán minden mezőt a *format* karakterláncnak megfelelően formáz (konvertál), és letárol rendre a paraméter *cím*eken.

A jelölésben a <> az elhagyhatóságot, a ... a megelőző paraméter tetszőleges számú ismételhetőségét jelenti. A bemeneti mező definíciójára rögtön kitérünk!

A **scanf** a sikeresen vizsgált, konvertált és letárolt bemeneti mezők számával tér vissza. A vizsgált vagy akár konvertált, de le nem tárolt mezők ebbe a számba nem értendők bele. Ha a függvény az olvasást a fájl végén kísérelné meg, vagy valamilyen hiba következne be, akkor **EOF–**ot kapunk tőle vissza. A függvény zérussal is visszatérhet, ami azt jelenti, hogy egyetlen vizsgált mezőt sem tárolt le.

A *format* karakterláncban ugyanannyi formátumspecifikációnak kell lennie, mint ahány bemeneti mező van, és ahány *cim* paramétert megadtak a hívásban. Ha a formátumspecifikációk többen vannak, mint a *cim*ek, akkor ez előre megjósolhatatlan hibához vezet. Ha a *cim* paraméterek száma több mint a formátumspecifikációké, akkor a felesleges *cim*eket egyszerűen elhagyja a **scanf**.

A *format* karakterlánc három féle objektumból áll:

- fehér karakterekből,
- nem fehér karakterekből és
- formátumspecifikációkból.

Ha fehér karakter következik a *format* karakterláncban, akkor a **scanf** olvassa, de nem tárolja a bemenetről érkező fehér karaktereket egészen a következő nem fehér karakterig.

Nem fehér karakter minden más a '%' kivételével. Ha a *format* karakterláncban ilyen karakter következik, akkor a **scanf** olvas a bemenetről, de nem tárol, hanem elvárja, hogy a beolvasott karakter egyezzen a *format* karakterláncban levővel.

A formátumspecifikációk vezérlik a **scanf** függvényt az olvasásban, a bemeneti mezők konverziójában és a konverzió típusában. A konvertált értéket aztán a rutin elhelyezi a soron következő paraméterrel adott *cim*–en. A formátumspecifikáció általános alakja:

% <*> <*szélesség*> <h|l|L> ***típuskarakter***

, ahol a <> az elhagyhatóságot és a | a vagylagosságot jelöli. Nézzük a részleteket!

- Minden formátumspecifikáció % karakterrel indul, és típuskarakterrel végződik. Az általános alakban elhagyhatónak jelölt részek csak az ott megadott sorrendben kerülhetnek a % és a típuskarakter közé.
- A * elnyomja a következő bemeneti mező hozzárendelését. A **scanf** a "**%***típuskarakter*" hatására olvassa, ellenőrzi és konvertálja a vonatkozó bemeneti mezőt, de nem helyezi el a kapott értéket az idetartozó *cim* paraméteren. Tehát a bemeneti mező tartalmának ilyenkor is meg kell felelnie a konverziós típuskarakternek.
- A *szélesség* maximális mezőszélességet határoz meg, azaz a **scanf** legfeljebb ennyi karaktert olvashat, de olvashat ennél kevesebbet is, ha fehér, vagy konvertálhatatlan karakter következik a bemeneten.
- A h, az l és az L a *cim* paraméter alapértelmezés szerinti típusát módosítja. A h **short int**. Az l **long int**, ha a típuskarakter egész konverziót specifikál, ill. **double**, ha a típuskarakter lebegőpontos átalakítást ír elő. Az L pedig a **long double** módosítója.

A következő táblázatban felsoroljuk az aritmetikai konverziót okozó típuskaraktereket:

Típuskarakter	Az elvárt bemenet	A paraméter típusa
d	decimális egész	**int** *
i	decimális, oktális vagy hexadecimális egész	**int** *
o	oktális egész (vezető 0 nélkül is annak minősül a szám)	**int** *
u	előjel nélküli decimális egész	**unsigned int** *
x	hexadecimális egész (vezető 0x vagy 0X nélkül is az a szám)	**int** *
e, E	lebegőpontos valós	**float** *
f	lebegőpontos valós	**float** *
g, G	lebegőpontos valós	**float** *

- A %d, a %i, a %o, a %x, a %D, a %I, a%O, a %X, a %c és a %n konverziók esetén **unsigned char**–ra, **unsigned int**–re, vagy **unsig-**

ned long–ra mutató mutatók is használhatók azoknál az átalakításoknál, ahol a **char**–ra, az **int**–re, vagy a **long**–ra mutató mutató megengedett.

- A %e, a %E, a %f, a %g és a %G lebegőpontos konverziók esetén a bemeneti mezőben levő valós számnak ki kell elégítenie a következő formát:

  <+|-> *ddddddddd* <.> *dddd* <E|e> <+|-> *ddd*

  ahol *d* decimális, oktális, vagy hexadecimális számjegyet, a <> elhagyhatóságot és a | vagylagosságot jelöl.

A mutató konverzió típuskarakterei:

Típuskarakter	Az elvárt bemenet	A paraméter típusa
n	Nincs.	**int** *. A %n-ig sikeresen olvasott karakterek számát tárolja ebben az **int**-ben a **scanf**.
p	Megvalósítástól függő formában, de általában hexadecimálisan.	**void ***

A karakteres konverzió típuskarakterei:

Típuskarakter	Az elvárt bemenet	A paraméter típusa
c	karakter	Mutató **char**–ra, ill. mutató **char** tömbre, ha mezőszélességet is megadtak. Pl.: %7c.
%	% karakter	Nincs konverzió. Magát a % karaktert tárolja.
s	karakterlánc	Mutató **char** tömbre.
[*keresőkészlet*]	karakterlánc	Mutató **char** tömbre.
[^*keresőkészlet*]	karakterlánc	Mutató **char** tömbre.

- A %c hatására a **scanf** a következő karaktert (akár fehér, akár nem) olvassa a bemenetről. Ha a fehér karaktereket át kívánjuk lépni, használjuk a %1s formátumspecifikációt!

- A %*szélességc* specifikációhoz tartozó *cim* paraméternek legalább *szélesség* elemű karaktertömbre kell mutatnia.
- A %s specifikációhoz tartozó *cim* paraméternek legalább akkora karaktertömbre kell mutatnia, melyben a vonatkozó bemeneti mező minden karaktere, és a karakterláncot lezáró '\0' is elfér.
- A %[*keresőkészlet*] és a %[^*keresőkészlet*] alakú specifikáció teljes mértékben helyettesíti az s típuskaraktert. A vonatkozó *cim* paraméternek karaktertömbre kell ekkor is mutatnia. A szögletes zárójelben levő karaktereket keresőkészletnek nevezzük.
- %[*keresőkészlet*] esetében a **scanf** addig olvassa a bemenetet, míg a bejövő karakterek egyeznek a keresőkészlet valamelyik karakterével. A karaktereket kiteszi rendre a rutin '\0'–val lezártan a paraméter karaktertömbbe. Például a %[abc]-vel az 'a', a 'b' és a 'c' karakterek valamelyikét kerestetjük a bemeneti mezőben. A %[]xyz] viszont a ']', az 'x', a 'y' és a 'z' után kutat.
- %[^*keresőkészlet*] a **scanf** bármilyen olyan karaktert keres, ami nincs benn a keresőkészletben. Például a %[^]abc] hatására addig tart a bemenet olvasása, míg róla ']', 'a', 'b' vagy 'c' nem érkezik.

📖 Néhány programfejlesztő rendszer esetén a keresőkészletben tartomány is megadható, azaz például a %[0123456789]-et a %[0-9] teljes mértékben helyettesíti. A tartomány kezdő karaktere kódjának azonban kisebbnek kell lenni a tartomány vég karaktere kódjánál. Nézzünk néhány példát!

- %[-+*/]: A négy aritmetikai operátort keresi.
- %[0-9A-Za-z]: Alfanumerikus karaktert keres.
- %[+0-9-A-Z]: A '+', a '-', a szám és a nagybetű karaktereket keresi.
- %[z-a]: A 'z', a '-' és az 'a' karaktereket keresi.

Tisztázzuk végre a bemeneti mező fogalmát!

- Minden karakter a következő fehér karakterig, de a fehér karakter maga már nem tartozik bele.
- Minden karakter az első olyan karakterig, mely az aktuális típuskarakter szerint nem konvertálható.
- Minden karakter, míg a megadott mezőszélesség ki nem merül.

- Keresőkészlet esetén addig tart a bemeneti mező, míg a keresőkészlet feltételeinek meg nem felelő karakter nem érkezik a bemenetről.

📖 A bemeneti mező második alternatívája miatt, nem javasoljuk a **scanf** függvény széleskörű használatát programokban. Helyette olvassuk be a bemeneti karakterláncot, végezzük el rajta az összes formai ellenőrzést! Ha aztán minden rendben volt, a konverzió megvalósítható egy menetben az

int **sscanf**(const char **puffer*, const char **format*<, *cim*, ...>);

függvénnyel, mely ugyanazt teszi, mint a **scanf**, de bemeneti mezőit nem az **stdin**–ről, hanem az első paraméterként kapott karakterláncból veszi.

11.8.2 Kimenet az stdout-ra

int **putchar**(int c);

A függvény makró, azaz:

```
#define putchar(c) putc((c), stdout)
```

A

int **puts**(const char **s*);

függvény a '\0' lezárású *s* karakterláncot az **stdout** folyamba írja a '\0' nélkül, mely helyett viszont kitesz még egy '\n' karaktert.

Sikeres esetben nem negatív értékkel tér vissza. Hiba bekövetkeztekor viszont **EOF**-ot kapunk tőle.

A konverziót is végző

int **printf**(const char **format*<, *parameter*, ...>);

rutin fogad egy sor *parameter*t, melyek mindegyikéhez hozzárendel egy, a *format* karakterláncban lévő formátumspecifikációt, és az ezek szerint formázott (konvertált) adatokat kiviszi az **stdout** folyamba.

✍ A jelölésben a <> az elhagyhatóságot, a ... a megelőző paraméter tetszőleges számú ismételhetőségét jelenti.

💣 A *format* karakterláncban ugyanannyi formátumspecifikációnak kell lennie, mint ahány *parameter* van. Ha kevesebb a paraméter, mint a formátumspecifikáció, akkor ez előre megjósolhatatlan hibához vezet. Ha több a paraméter, mint a formátumspecifikáció, akkor a felesleges paramétereket egyszerűen elhagyja a **printf**.

A rutin a folyamba kivitt bájtok számával tér vissza sikeres esetben, ill. **EOF**–ot kapunk tőle hiba bekövetkeztekor.

A *format* karakterlánc kétféle objektumot tartalmaz:

- sima karaktereket és
- formátumspecifikációkat.

A sima karaktereket változatlanul kiviszi az **stdout**-ra a **printf**. A formátumspecifikációhoz veszi a következő *parameter* értékét, konvertálja, és csak ezután teszi ki az **stdout**-ra.

A formátumspecifikáció általános alakja a következő:

% <*jelzők*> <*szélesség*> <.*pontosság*> <h|l|L> ***típuskarakter***

- Minden formátumspecifikáció % karakterrel kezdődik, és típuskarakterrel végződik.
- Ha a '%' karaktert szeretnénk az **stdout**–ra vinni, akkor meg kell duplázni (%%).
- Az általános alakban elhagyhatónak jelölt részek csak az ott megadott sorrendben kerülhetnek a % és a típuskarakter közé.

A következőkben leírjuk a típuskarakterek értelmezését arra az esetre, ha a formátumspecifikációban a % jelet csak a típuskarakter követi. Nézzük előbb az aritmetikai konverziót okozó típuskaraktereket:

Típuskarakter	Elvárt paraméter	A kimenet formája
d	**int**	Előjeles decimális egész.
i	**int**	Előjeles decimális egész.
o	**int**	Előjel nélküli oktális egész vezető 0 nélkül.
u	**int**	Előjel nélküli decimális egész.
x	**int**	Előjel nélküli hexadecimális egész (a, b, c, d, e, f-fel), de vezető 0x nélkül.
X	**int**	Előjel nélküli hexadecimális egész (A, B, C, D, E, F-fel), de vezető 0X nélkül.
f	**double**	<->dddd.dddd alakú előjeles érték.
e	**double**	<->d.ddd...e<+|->ddd alakú előjeles érték.
E	**double**	<->d.ddd...E<+|->ddd alakú előjeles érték.
g	**double**	Az adott értéktől és a pontosságtól függően e, vagy f alakban előjeles érték.
G	**double**	Ugyanaz, mint a g forma, de az e alak használata esetén az exponens részben E van.

e vagy E típuskarakter esetén a vonatkozó paraméter értékét a **printf**

<->*d.ddd...*e<+|->*ddd*

alakúra konvertálja, ahol:

- Egy decimális számjegy (*d*) mindig megelőzi a tizedes pontot.
- A tizedes pont utáni számjegyek számát a pontosság határozza meg.
- A kitevő rész mindig legalább két számjegyet tartalmaz.

f típuskarakternél a vonatkozó paraméter értékét a **printf**

<->*ddd.ddd...*

alakúra konvertálja, s a tizedes pont után kiírt számjegyek számát itt is a pontosság határozza meg.

g vagy G típuskarakter esetén a **printf** a vonatkozó paraméter értékét e, E, vagy f alakra konvertálja

- Olyan pontossággal, melyet a szignifikáns számjegyek száma meghatároz.

- A követő zérusokat levágja az eredményről, és a tizedes pont is csak akkor jelenik meg, ha szükséges, azaz van még utána értékes tört számjegy.
- A g e, vagy f formájú, a G pedig E, vagy f alakú konverziót okoz.
- Az e, ill. az E formát akkor használja a **printf**, ha a konverzió eredményében a kitevő nagyobb a pontosságnál, vagy kisebb –4–nél.

A karakteres konverzió típuskarakterei:

Típuskarakter	Elvárt paraméter	A kimenet formája
%	nincs	Nincs konverzió. Maga a % karakter jelenik meg.
c	**int**	Egyetlen karakter.
s	**char ***	A karakterlánc karakterei megjelennek a záró '\0'–t kivéve. Ha megadtak pontosságot, akkor legfeljebb annyi karaktert ír ki a **printf**.

A mutató konverzió típuskarakterei:

Típuska-rakter	Elvárt paraméter	A kimenet formája
n	**int ***	A paraméter által mutatott **int**-ben letárolja az eddig kiírt karakterek számát. Nincs különben semmilyen konverzió.
p	**void ***	A paramétert mutatóként jelenteti meg. A kijelzés formátuma programfejlesztő rendszer függő, de általában hexadecimális.

Lássuk a *jelzők*et!

-	Az eredmény balra igazított, és jobbról szóközzel párnázott. Ha a - jelzőt nem adják meg, akkor az eredmény jobbra igazított, és balról szóközökkel, vagy zérusokkal párnázott.
+	Előjeles konverzió eredménye mindig plusz, vagy mínusz előjellel kezdődik. Ha a + jelzővel együtt szóköz jelzőt is megadnak, akkor a + jelző van érvényben.
szóköz	Ha az érték nem negatív, a kimenet egy szóközzel kezdődik a plusz előjel helyett. A negatív érték ilyenkor is mínusz előjelet kap.
#	Azt határozza meg, hogy a paramétert alternatív formát használva kell konvertálni.

Az alternatív formák a típuskaraktertől függnek:

Típ.kar.	A # hatása a paraméterre
c,s,d,i,u	Nincs hatás.
e,E,f	Az eredményben mindenképpen lesz tizedes pont még akkor is, ha azt egyetlen számjegy sem követi. Normálisan ilyenkor nem jelenik meg a tizedes pont.
g,G	Ugyanaz, mint e és E, de az eredményből a követő zérusokat nem vágja le a **printf**.
o	0-t ír a konvertált, nem zérus paraméter érték elé. Ez az oktális szám megjelentetése.
x, X	0x, 0X előzi meg a konvertált, nem zérus paraméter értéket.

A *szélesség* a kimeneti érték minimális mezőszélességét határozza meg, azaz a megjelenő eredmény legalább ilyen szélességű. A *szélesség*et két módon adhatjuk meg:

- vagy expliciten beírjuk a formátumspecifikációba,
- vagy a *szélesség* helyére * karaktert teszünk. Ilyenkor a **printf** hívásban a következő *parameter* csak **int** típusú lehet, s ennek az értéke definiálja a kimeneti érték mezőszélességét.

Bármilyen *szélesség*et is írunk elő, a **printf** a konverzió eredményét nem csonkítja! A lehetséges szélesség specifikációk:

Szélesség	Hatása a kimenetre
n	A **printf** legalább n karaktert jelentet meg. Ha a kimeneti érték n karakternél kevesebb, akkor szóközzel n karakteresre párnázza (jobbról, ha a – jelzőt megadták, máskülönben balról).
0n	Legalább n karakter jelenik meg ekkor is. Ha a kimeneti érték n–nél kevesebb karakterből áll, akkor balról zérus feltöltés következik.
*	A paraméter lista szolgáltatja a szélesség specifikációt, de ennek a paraméter listában meg kell előznie azt a paramétert, amire az egész formátumspecifikáció vonatkozik.

A *pontosság* specifikáció mindig ponttal (.) kezdődik. A szélességhez hasonlóan ez is megadható közvetlenül, vagy közvetve (*) a paraméter listában. Utóbbi esetben egy **int** típusú paraméternek meg kell előznie azt a paramétert a **printf** hívásban, amire az egész formátumspecifikáció vonatkozik.

Megemlítjük, hogy a *szélesség*et és a *pontosság*ot is megadhatjuk egyszerre közvetetten. Ilyenkor a formátumspecifikációban *.* van. A **printf** hívás paraméter listájában két **int** típusú paraméter előzi meg (az első a szélesség, a második a pontosság) azt a paramétert, amire az egész formátumspecifikáció vonatkozik. Lássunk egy példát!

```
printf("%*.*f", 6, 2, 6.2);
```

A 6.2–et f típuskarakterrel kívánjuk konvertáltatni úgy, hogy a mezőszélesség 6 és a pontosság 2 legyen.

A pontosság specifikációk a következők:

Pontosság	Hatása a kimenetre
.*	Lásd előbbre!
nincs megadva	Érvénybe lépnek a típuskaraktertől függő alapértelmezés szerinti értékek. Ezek: • 1 : d, i, o, u, x, X esetén, • 6 : e, E, f típuskaraktereknél, • minden szignifikáns számjegy g és G–nél, • s típuskarakternél a teljes karakterlánc megy a kimenetre és • a c típuskarakterre nincs hatással.
.0	• Az e, E, f típuskaraktereknél nem jelenik meg a tizedes pont. • A d, i, o, u, x, X esetén pedig az alapértelmezés szerinti pontosság lép érvénybe (1). Ha ilyenkor a kiírandó paraméter értéke ráadásul zérus is, akkor csak egyetlen szóköz jelenik meg.
.n	A **printf** legfeljebb n karaktert, vagy decimális helyiértéket jelentet meg. Ha a kimenet n-nél több karakterből áll, akkor csonkul, vagy kerekíti a rutin a vonatkozó típuskaraktertől függően: • d, i, o, u, x és X esetén legalább n számjegy jelenik meg. Ha a kimenet n–nél kevesebb jegyből áll, akkor balról zérus feltöltés történik. Ha a kimeneti n–nél többjegyű, akkor sem csonkul. • e, E, f–nél a **printf** n számjegyet jelentet meg a tizedes ponttól jobbra. Ha szükséges, a legalacsonyabb helyiértéken kerekítés lesz. • g, G esetén legfeljebb n szignifikáns jegy jelenik meg. • A c típuskarakterre nincs hatása. • Az s típuskarakternél legfeljebb n karakter jelenik meg, azaz a hosszabb karakterlánc csonkul.

Legvégül nézzük még a h, l és L méretmódosító karaktereket! A méretmódosítók annak a paraméternek a hosszát módosítják, melyre az egész formátumspecifikáció vonatkozik.

- A d, i, o, u, x és X típuskarakterekkel kapcsolatban csak a h és az l méretmódosítók megengedettek. Jelentésük: h esetén a vonatkozó paramétert a **printf** tekintse **short int**–nek, **l** esetén pedig **long int**–nek.
- Az e, E, f, g, és G típuskarakterekkel kapcsolatban csak az l és az L méretmódosítók megengedettek. Jelentésük: l esetén a vonatkozó paramétert a **printf** tekintse **double**–nek, L–nél pedig **long double**–nek.

Jelentessük meg a 2003. március 2. dátumot ÉÉÉÉ–HH–NN alakban!

```
printf("%04d-%02d-%02d", 2003, 3, 2);
printf("%.4d-%.2d-%.2d", 2003, 3, 2);
```

Mindkét hívás 2003–03–02–t szolgáltat.

Szemléltessük a 0, a #, a + és a – jelzők hatását d, o, x, e és f típuskarakterek esetén!

```
/* PELDA33.C: A printf jelzőinek szemléltetése néhány
   típuskarakterre. */
#include <stdio.h>
#include <string.h>
#define E 555
#define V 5.5
int main(void){
  int i,j,k,m;
  char prefix[7], format[100], jelzok[]=" 0# +   -",
       *tk[]={"6d", "6o", "8x", "10.2e", "10.2f"};
  #define NJ (sizeof(jelzok)-2)*2
  #define NTK sizeof(tk)/sizeof(tk[0])
  printf("prefix    6d      6o      8x"
         "        10.2e       10.2f\n"
         "------+-------+-------+-----"
         "----+-----------+-----------+\n");
  for(i=NJ-1; i>=0; --i){
    strcpy(prefix, "%");
    for(j=k=1; k<NJ; k<<=1)
      if(i&k) prefix[j++]=jelzok[k];
    prefix[j]=0;
```

Az **i** 15–ről indul, s zérusig csökken egyesével, azaz eközben leírja az összes lehetséges négybites bitkombinációt. A **k** felvett értékei 1, 2, 4 és 8, s a **jelzok** tömb épp ezen indexű elemeiben találhatók meg a jelző karakterek.

```
    strcpy(format, "%5s |");
    for(m=0; m<NTK; ++m){
```

```
      strcat(format, prefix);
      strcat(format, tk[m]);
      strcat(format, " |"); }
    strcat(format, "\n");
    printf(format, prefix, E, E, E, V, V); }
  return(0); }
```

A program futtatásakor megjelenő kimenet:

```
prefix    6d      6o      8x        10.2e       10.2f
------+-------+------+---------+-----------+-----------+
%0#+- |+555   |01053  |0x22b    |+5.50e+000 |+5.50      |
 %#+- |+555   |01053  |0x22b    |+5.50e+000 |+5.50      |
 %0+- |+555   |1053   |22b      |+5.50e+000 |+5.50      |
  %+- |+555   |1053   |22b      |+5.50e+000 |+5.50      |
 %0#- |555    |01053  |0x22b    |5.50e+000  |5.50       |
  %#- |555    |01053  |0x22b    |5.50e+000  |5.50       |
  %0- |555    |1053   |22b      |5.50e+000  |5.50       |
   %- |555    |1053   |22b      |5.50e+000  |5.50       |
 %0#+ |+00555 |001053 |0x00022b |+5.50e+000 |+000005.50 |
  %#+ |  +555 | 01053 |   0x22b |+5.50e+000 |     +5.50 |
  %0+ |+00555 |001053 |0000022b |+5.50e+000 |+000005.50 |
   %+ |  +555 |  1053 |     22b |+5.50e+000 |     +5.50 |
  %0# |000555 |001053 |0x00022b |05.50e+000 |0000005.50 |
   %# |   555 | 01053 |   0x22b | 5.50e+000 |      5.50 |
   %0 |000555 |001053 |0000022b |05.50e+000 |0000005.50 |
    % |   555 |  1053 |     22b | 5.50e+000 |      5.50 |
```

int **sprintf**(char **format*, const char ***format*<, *parameter*, ...>);

A függvény ugyanazt teszi, mint a **printf**, de a kimenetét nem az **stdout**–ra készíti, hanem az első paraméterként megkapott karaktertömbbe '\0'–lal lezárva.

☞ A **vfprint**, a **vprintf** és a **vsprintf** rutinokat már megemlítettük a **Változó paraméterlista** fejezetben!

Megoldandó feladatok:

Készítsen **char * kozepre**(**char** **mit*, **int** *szeles*) függvényt, mely a saját helyén középre igazítja a *mit* karakterláncot *szeles* szélességben, és visszaadja az eredmény lánc kezdőcímét! A középre igazítást csak *szeles*–nél rövidebb láncok esetében kell elvégezni. A kétoldali párnázó karakter indulásként lehet szóköz, de lehessen ezt fordítási időben változtatni!

Írjon szoftvert, mely igazított táblázatot hoz létre az alábbi tartalmú **TABLA** fájl

```
Szöveg Forint Egész
Papadopulosz 111222.3 1456
Sodik_sor 2.2 345
```

szabvány bemenetkénti átirányításával. Az eredmény táblázat:

```
+------------------------+--------------------+------------+
| Szöveg                 |             Forint |   Egész    |
+------------------------+--------------------+------------+
| Papadopulosz           |       111222.30Ft |    1456    |
+------------------------+--------------------+------------+
| Sodik_sor              |            2.20Ft |    345     |
+------------------------+--------------------+------------+
```

, ahol az első oszlop balra-, a második jobbra-, s a harmadik középre igazított. A tábla egy sorának szerkezete:

```
| MEZO1| MEZO2Ft | MEZO3 |
```

, ahol **MEZO1**, **MEZO2** és **MEZO3** bruttó adatszélességek a mutatott módon.

Fokozhatja még a feladatot úgy, hogy az adatokat lehessen billentyűzetről is megadni!

11.9 Egyéb függvények

Csak lezárt fájlokkal foglalkozik a következő két függvény. A

int **remove**(const char **fajlnev*);

törli az akár komplett úttal megadott azonosítójú fájlt. Sikeres esetben zérust, máskülönben -1-et szolgáltat a rutin. A

int **rename**(const char **reginev*, const char **ujnev*);

a *reginev* azonosítójú fájlt átnevezi *ujnev*–re. Ha az *ujnev* meghajtónevet is tartalmaz, akkor az nem térhet el attól, ahol a *reginev* azonosítójú fájl elhelyezkedik. Ha viszont az *ujnev* a fájl eredeti helyétől eltérő utat tartalmaz, akkor az átnevezésen túl megtörténik a fájl átmozgatása is.

💣 A függvény egyik paramétere sem lehet globális fájlazonosító!

Sikeres esetben zérust szolgáltat a rutin. A problémát a -1 visszaadott érték jelzi.

FILE ***tmpfile**(void);

A rutin ”wb+” móddal ideiglenes fájlt hoz létre, melyet lezárásakor, vagy normális programbefejeződéskor automatikusan töröl a rendszer. A visszaadott érték az ideiglenes fájl **FILE** struktúrájára mutat, ill. **NULL** jön létrehozási probléma esetén.

char ***tmpnam**(char *s*[L_tmpnam]);

tmpnam(**NULL**) módon hívva olyan fájlazonosítót kapunk, mely egyetlen létező fájl nevével sem egyezik. A szolgáltatott mutató belső, statikus karaktertömböt címez, ahol a fájlazonosító karakterlánc található.

✍ A fájlazonosító karakterlánc nem marad ott örökké, mert a következő **tmpnam** hívás felülírja.

Nem **NULL** mutatóval hívva a rutin kimásolja a fájlazonosítót az *s* karaktertömbbe, s ezzel is tér vissza. Az *s* tömb legalább **L_tmpnam** méretű kell, legyen. Többszöri hívással legalább **TMP_MAX** darab, különböző fájlazonosító generálása garantált.

💣 Vigyázat! A függvény fájlazonosítókat generál és nem fájlokat!

12 IRODALOMJEGYZÉK

[1] Kiss J. – Raffai M. – Szijártó M. – Szörényi M.: *A számítástechnika alapjai*

NOVADAT Bt., Győr, 2001

[2] Marton L. – Pukler A. – Pusztai P.: *Bevezetés a programozásba*

NOVADAT Bt., Győr, 1993

[3] Marton László: *Bevezetés a Pascal nyelvű programozásba*

NOVADAT Bt., Győr, 1998

[4] B. W. Kernighan – D. M. Ritchie: *A C programozási nyelv*

Műszaki Könyvkiadó, Budapest, 1985

[5] B. W. Kernighan – D. M. Ritchie: *A C programozási nyelv, az ANSI szerint szabványosított változat*

Műszaki Könyvkiadó, Budapest, 1996

[6] Benkő Tiborné – Benkő László – Tóth Bertalan: *Programozzunk C nyelven*

ComputerBooks, Budapest, 1999

[7] Benkő Tiborné – Urbán Zoltán: *IBM PC programozása TURBO C nyelven 2.0*

BME Mérnöktovábbképző Intézet, Budapest, 1989

13 TARTALOMJEGYZÉK