

TARTALOM

| | |
|--|-----------|
| Előszó | 11 |
| | |
| 1. Bevezetés | 13 |
| | |
| 2. Memóriaváltozók | 23 |
| 2.1. Változónév (azonosító) | 24 |
| 2.2. Változócím | 24 |
| 2.3. Változótípusok | 24 |
| 2.4. A változó értéke | 25 |
| 2.5. Változódefiniálás | 26 |
| 2.6. A scanf, printf, fscanf és fprintf függvények | 27 |
| 2.7. Kitűzött feladatok | 36 |
| | |
| 3. Utasítások | 38 |
| 3.1. Kifejezés-utasítás | 38 |
| 3.2. Összetett utasítás | 38 |
| 3.3. Döntési utasítások | 39 |
| 3.3.1. Az if utasítás | 39 |
| 3.3.1.1. Logikai operátorok | 42 |
| 3.3.2. A switch utasítás | 43 |
| 3.3.3. Kitűzött feladatok | 45 |
| 3.4. Ciklusutasítások | 47 |
| 3.4.1. A while ciklus (elől tesztelős ciklus) | 48 |
| 3.4.2. A do while ciklus (hátról tesztelős ciklus) | 50 |
| 3.4.3. A for ciklus (ismert lépésszámú ciklus) | 52 |
| 3.5. Ugró utasítások | 55 |
| 3.5.1. A break utasítás | 55 |
| 3.5.2. A continue utasítás | 56 |
| 3.5.3. A goto utasítás | 57 |

| | |
|--|-----------|
| 3.5.4. A return utasítás | 57 |
| 3.6. Megoldott feladatok | 58 |
| 3.7. Elemi algoritmusok összefoglalása | 62 |
| 3.7.1. Összeg- és szorzatszámítás | 62 |
| 3.7.2. Adott tulajdonságú elemek megszámlálása | 63 |
| 3.7.3. Átlagszámítás | 63 |
| 3.7.4. Adott tulajdonságú elem létezésének ellenőrzése | 64 |
| 3.7.5. Adott tulajdonságú elem megkeresése | 64 |
| 3.7.6. Legnagyobb és legkisebb elem értékének meghatározása | 65 |
| 3.7.7. Legnagyobb és legkisebb elem pozíciójának meghatározása | 66 |
| 3.7.8. Legnagyobb közös osztó meghatározása | 66 |
| 3.7.9. Legkisebb közös többszörös meghatározása | 67 |
| 3.7.10. Egy szám tükrözése | 67 |
| 3.8. Kitűzött feladatok | 68 |
| 4. Kifejezések | 74 |
| 4.1. A kifejezések jobb (Rvalue) és bal (Lvalue) értéke | 76 |
| 4.2. Konstansok | 77 |
| 4.2.1. Egész típusú konstansok | 77 |
| 4.2.1.1. Az egész konstansok típusai | 77 |
| 4.2.2. Valós típusú konstansok | 78 |
| 4.2.3. Karakter típusú konstansok | 78 |
| 4.2.4. Karakterlánc típusú konstansok | 80 |
| 4.2.5. Szimbolikus konstansok | 80 |
| 4.3. Operátorok | 81 |
| 4.3.1. Aritmetikai operátorok | 81 |
| 4.3.2. Összehasonlítási operátorok | 81 |
| 4.3.3. Logikai operátorok | 82 |
| 4.3.4. Bitenkénti operátorok | 83 |
| 4.3.5. Értékadás-operátor | 85 |

| | |
|---|-----------|
| TARTALOM | 7 |
| 4.3.6. Összetett operátorok | 85 |
| 4.3.7. Vesszőoperátor | 85 |
| 4.3.8. Feltételes operátor | 86 |
| 4.3.9. A sizeof operátor | 87 |
| 4.3.10. A Cast operátor (explicit típuskonverzió) | 87 |
| 4.4. Megoldott feladatok | 88 |
| 4.5. Kitűzött feladatok | 91 |
| 5. Pointerek (mutatók) | 92 |
| 5.1. Műveletek pointerekkel | 95 |
| 5.1.1. Értékadás | 95 |
| 5.1.2. Összehasonlítás | 95 |
| 5.1.3. <pointer> + / - <egész> | 96 |
| 5.1.4. Pointerek különbsége | 97 |
| 6. Tömbök | 99 |
| 6.1. Egydimenziós tömbök | 99 |
| 6.1.1. Inicializálás definiáláskor | 100 |
| 6.1.2. Kapcsolat a tömbök és pointerek között | 101 |
| 6.1.2.1. Buborékos rendezés | 102 |
| 6.1.2.2. Minimum-kiválasztásos rendezés | 104 |
| 6.1.2.3. Szétválogatás | 105 |
| 6.1.2.4. Összefésülés | 107 |
| 6.1.2.5. Kitűzött feladatok | 108 |
| 6.2. Karakterláncok (stringek) | 110 |
| 6.2.1. Karakterláncok beolvasása/kiírása | 112 |
| 6.2.1.1. A scanf/printf függvények | 112 |
| 6.2.1.2. A gets/puts függvények | 112 |
| 6.2.1.3. A fgets/fputs függvények | 113 |
| 6.2.2. Karakterlánc-kezelő függvények | 113 |
| 6.2.2.1. Az strlen függvény | 114 |
| 6.2.2.2. Az strcpy függvény | 114 |

| | |
|--|------------|
| 6.2.2.3. Az <code>strcat</code> függvény | 115 |
| 6.2.2.4. Az <code>strcmp</code> függvény | 115 |
| 6.2.2.5. Az <code>strstr</code> függvény | 115 |
| 6.2.2.6. Az <code>strchr</code> függvény | 115 |
| 6.2.2.7. Az <code>sscanf/sprintf</code> függvények | 116 |
| 6.2.2.8. Megoldott feladatok | 117 |
| 6.2.2.9. Kitűzött feladatok | 121 |
| 6.3. Kétdimenziós tömbök | 123 |
| 6.3.1. Kétdimenziós tömbök inicializálása definiálásukkor | 124 |
| 6.3.2. Szimmetriák és szabályosságok egy $n \times n$ méretű mátrixban | 127 |
| 6.4. Többdimenziós tömbök | 128 |
| 6.5. Megoldott feladatok | 128 |
| 6.6. Kitűzött feladatok | 131 |
| 7. Dinamikus helyfoglalás | 136 |
| 8. A felhasználó által definiált típusok | 142 |
| 8.1. A <code>typedef</code> módosító jelző | 142 |
| 8.2. A <code>struct</code> típus | 142 |
| 8.2.1. A <code>struct</code> -pointerek | 147 |
| 8.2.2. Bitmezők | 148 |
| 8.3. A <code>union</code> típus | 150 |
| 8.4. Kitűzött feladatok | 153 |
| 8.5. A felsorolás típus (<code>enum</code>) | 155 |
| 9. Függvények | 158 |
| 9.1. Cím szerinti paraméterátadás | 162 |
| 9.2. Paraméterátadás a <code>STACK</code> -en keresztül | 166 |
| 9.3. Globális és lokális változók | 168 |
| 9.4. Programtervezés kicsiben | 171 |
| 9.5. Függvénypointerek | 177 |
| 9.6. Függvény paraméterként való átadása függvénynek | 178 |

| | |
|--|------------|
| TARTALOM | 9 |
| 9.7. Változó paraméterszámú függvények | 179 |
| 9.8. Parancssor-argumentumok (a main paraméterei) | 183 |
| 9.9. Rekurzív függvények | 185 |
| 9.10. Kitűzött feladatok | 188 |
| 10. Makrók | 194 |
| 10.1. Kitűzött feladatok | 198 |
| 11. Állománykezelés (Input/Output műveletek) | 199 |
| 11.1. Állománykezelő függvények | 201 |
| 11.2. A scanf/fscanf, illetve printf/fprintf függvények | 208 |
| 11.2.1. A scanf függvény | 208 |
| 11.2.1.1. A %n formázókarakter | 209 |
| 11.2.1.2. A scanset használata | 209 |
| 11.2.2. A printf függvény | 212 |
| 11.3. Megoldott feladatok | 215 |
| 11.4. Alacsony szintű állománykezelés | 221 |
| 12. Átfogó kép a C nyelvről | 224 |
| 12.1. Strukturált programozás | 224 |
| 12.1.1. Feltételes fordítás | 225 |
| 12.2. Moduláris programozás | 228 |
| 12.2.1. Modulok, blokkok | 229 |
| 12.3. Definíció és deklaráció | 230 |
| 12.4. A változók osztályozása | 231 |
| 12.5. A C nyelv adattípusai felülnézetből | 236 |
| 12.6. Egy összetett feladat | 237 |
| A. Függelék. Rekurzió egyszerűen és érdekesen | 245 |
| B. Függelék. Az adatok belső ábrázolása a számítógép memóriájában | 261 |

| | |
|--------------|----------|
| 10 | TARTALOM |
| Szakirodalom | 271 |

ELŐSZÓ

A könyvet, amelyet kezében tart az olvasó, tankönyvnek szántam. Gondolom, eredményesen fogják használni mind az egyetemi hallgatók, mind a középiskolás diákok. Sőt remélem, tanárkollégáim is találnak benne ötleteket, amelyeket használni tudnak majd bizonyos témák oktatásakor.

A könyv megírásakor többre törekedtem, mint egyszerűen bemutatni a C nyelvet (az ANSI standardnak megfelelően), a célom az volt, hogy programozni tanítsak. Az első fejezettel igyekszem átfogó képet nyújtani a programozásról mint számítógépes feladatmegoldásról. A következő fejezetek fokozatosan vezetik be az olvasót a programozás világába és többé-kevésbé egymásra épülnek. Igyekeztem nem hivatkozni olyasmire, ami még nem került bemutatásra. Például már a második fejezetben – a billentyűzetről való olvasással és a monitorra való írással párhuzamosan – tárgyalom dióhéjban a szöveges állománykezelést. Az utolsó fejezet célja olyan helyzetbe juttatni az olvasót, hogy mintegy felülnézetből lásson bizonyos témaköröket.

Azért, hogy a könyv ne tűnjön száraznak, sok fogalmat megoldott feladatokon keresztül vezetek be. A fogalmak elmélyítését ugyancsak megoldott feladatok által valósítom meg. A legtöbb feladat esetében igyekszem először rávezetni az olvasót a megoldásra, és csak azután közlöm a forráskódot. A C programokat gyakran "magyarázat" illetve "tanulmányok" nevű részek követik, amelyek általában implementációs szempontokra hívja fel a figyelmet. Minden fejezet végén kitűzött feladatok találhatók.

Tizenhat éves oktatói tapasztalatomat igyekeztem beépíteni a könyvbe. Arra törekedtem, hogy gazdagon legyen illusztrálva szemléltető ábrákkal. Egyes fejezetek (például a rekurzió) oktatásához sajátos módszereket javasolok. Diákjaim eredményei azt mutatják, hogy ezen módszerek hatékonyan alkalmazhatók. Több kollégám is kipróbálta őket, szép eredménnyel.

Bízom benne hogy, a kedves olvasó úgy találja majd, hogy gyakorlatias, érdekes, hasznos és eredeti tankönyvet tart a kezében.

Köszönet a lektornak, a műszaki szerkesztőknek és a kiadónak.

A szerző

—

—

|

1. FEJEZET

BEVEZETÉS

A számítógépes feladatmegoldás *kommunikációt* feltételez. Ehhez min-
denekelőtt a következőkre van szükség:

1. Egy nyelvre, amely által közölni tudom a számítógéppel, mit sz-
eretnék, hogy elvégezzen.
2. A gép „gondolkodásmódjához” igazodni, azaz gépi szinten elmag-
yarázni, hogy mi a teendője.

Milyen nyelvet „ért” a számítógép? A gépi kódot. Ennek a gépnyelvnek
az ábécéje a 0 és 1 számjegyekből áll. Elvileg milyen lehetőségek volnának
a kommunikációra?

- A gép megtanul magyarul. Ez kizárt dolog, különben nem gép vol-
na, hanem ember.
- Elsajátítjuk mi a gépi kódnyelvet. Ez természetesen lehetséges
– az első számítógépek így voltak programozva –, de rendkívül
„fárasztó”. Gondoljunk bele, nehéz olyan nyelven „beszélni”, ame-
lynek ábécéje csupán két betűből áll.

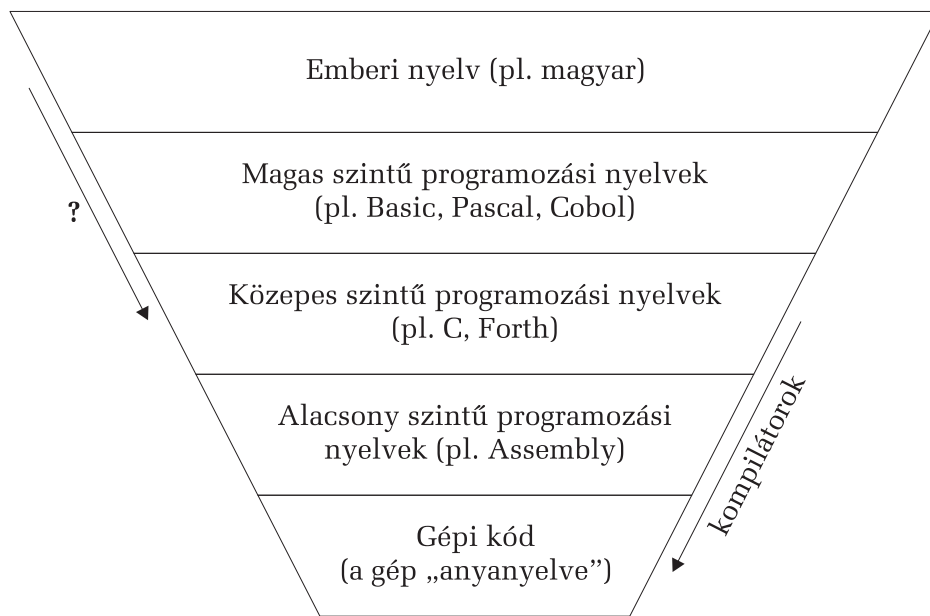
Mivel az ember kényelmes lény, más megoldásokat keresett. Így
született meg a fordító (kompilátor) gondolata: írni egy olyan szoftvert
(programot), amely fordít gépi kódra.

- A legkényelmesebb az lenne, ha létezne fordító magyarról gépi kó-
dra. Egyelőre ilyen fordítót nem írtak.
- A megoldás egy programozási nyelv létrehozása volt. Olyan „prim-
itív” nyelvről van szó, amely közelebb áll az emberhez, mint a gépi
kód, megfogalmazhatók benne a gépnek szánt utasítások, és a jelen-
legi eszközeinkkel képesek vagyunk olyan fordítót írni, amely fordít
az illető nyelvről gépi kódra.

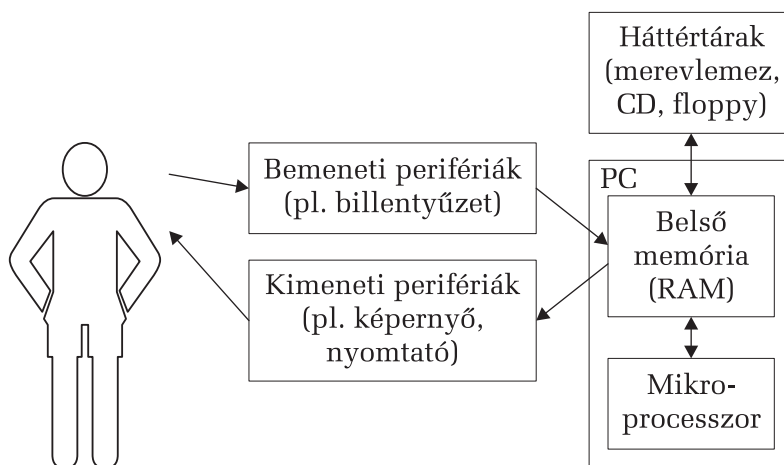
Attól függően, hogy mennyire van közel az illető nyelv az emberi
nyelvhez, és mennyire van távol a gépi nyelvtől, megkülönböztetünk *magas*,
közepes és *alacsony szintű* programozási nyelveket.

A C közepes szintű programozási nyelv.

A mellékelt 1.1. ábra szemléletesen mutatja be mindezt. A kérdőjel azt
a szakadékot szemlélteti, amelyet át kell hidalnia valakinek, ha programozó
szeretne lenni:



1.1. ábra. *Nyelvhierarchia*



1.2. ábra. *Kommunikáció az ember és a számítógép között*

- meg kell tanulnia egy programozási nyelvet,
- meg kell tanulnia *programozni* a számítógépet az illető nyelven.

Ebben a hasznos és egyben érdekesítő törekvésben szeretne segítséget nyújtani a könyv, amelyet az olvasó a kezében tart.

E könyvben a C nyelvnek főleg az ANSI szabvány szerinti változatát mutatjuk be. Miután az 1970-es években Dennis Ritchie és Brian Keringhan létrehozták a C nyelvet, a nyelv különböző változatai kezdtek megjelenni. Szükségessé vált tehát a szabványosítása. Ez lehetővé teszi olyan programok írását, amelyeket bármely modern fordító felismer. Ezt nevezik hordozhatóságnak.

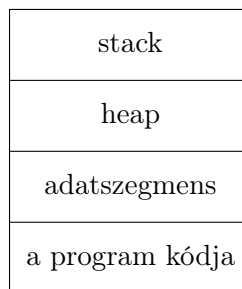
Hogyan történik fizikailag a kapcsolatteremtés a számítógép feladatmegoldó részével? Ezt mutatja be az 1.2. ábra.

A „programozó \rightarrow PC (*personal computer*)” irányú kommunikáció általában a billentyűzeten, a fordított irányú pedig a monitoron vagy a nyomtatón keresztül történik.

A számítógépbe (PC) bejuttatott adatok a belső memóriában kerülnek tárolásra. Neumann Jánostól errefelé mind a feladat adatai, mind a feladatot megoldó program itt tárolódnak el. Ezt nevezzük a „tárolt program elvének”, amely fordulópontot jelentett a számítástechnika történetében.

A program végrehajtása abból áll, hogy a mikroprocesszor egyenként kiolvassa a memóriából a program utasításait és elvégzi őket. A mikroprocesszor az előírt művelethez szükséges adatokat a belső memóriából veszi, és az eredményeket ugyanitt tárolja el. Ezért kétirányú a kommunikáció a mikroprocesszor és a belső memória között.

A lefordított C program négy, logikailag különböző memóriaterületet használ, mindegyiknek meglévén a maga szerepe. Ezek egyike a program kódját (az utasításokat) tartalmazza, a másik három pedig az adatokat, amelyekkel a program dolgozik. Később látni fogjuk, mitől függ, hogy bi-



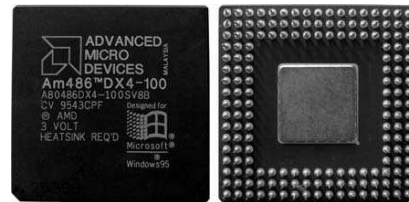
1.3. ábra. Egy C program elvi memóriatérképe



Központi egység, Monitor, Billentyűzet, Egér.



Belső memória (RAM).



Mikroprocesszor.



Alaplap.



Merevlemez.

zonyos adatok melyik memóriaterületen (adatszegment, stack, heap) kerülnek eltárolásra. Függ a fordítótól, a programozási környezettől és a mikroprocesszor típusától, hogyan néz ki pontosan a programunk által használt memóriaterület térképe. Az 1.3. ábra elvileg mutatja be C programunkat a számítógép memóriájában.

Mivel a belső memória tartalma törlődik a számítógép kikapcsolásakor, kialakították a háttértárat (külső memóriák). Ide ki lehet menteni adatokat, majd pedig vissza lehet tölteni őket. A háttértáron az adatok állományokba (*file*) vannak rendezve.

Hogyan kell megoldani egy feladatot számítógéppel?

Hat lépést különíthetünk el. A jobb megértés kedvéért egy jól ismert feladaton keresztül mutatjuk be a lépéseket.

1.1. feladat. Írjunk programot, amely egy másodfokú egyenlet valós együtthatóiból kiindulva (a, b, c) , kiírja a gyökeit (x_1, x_2) .

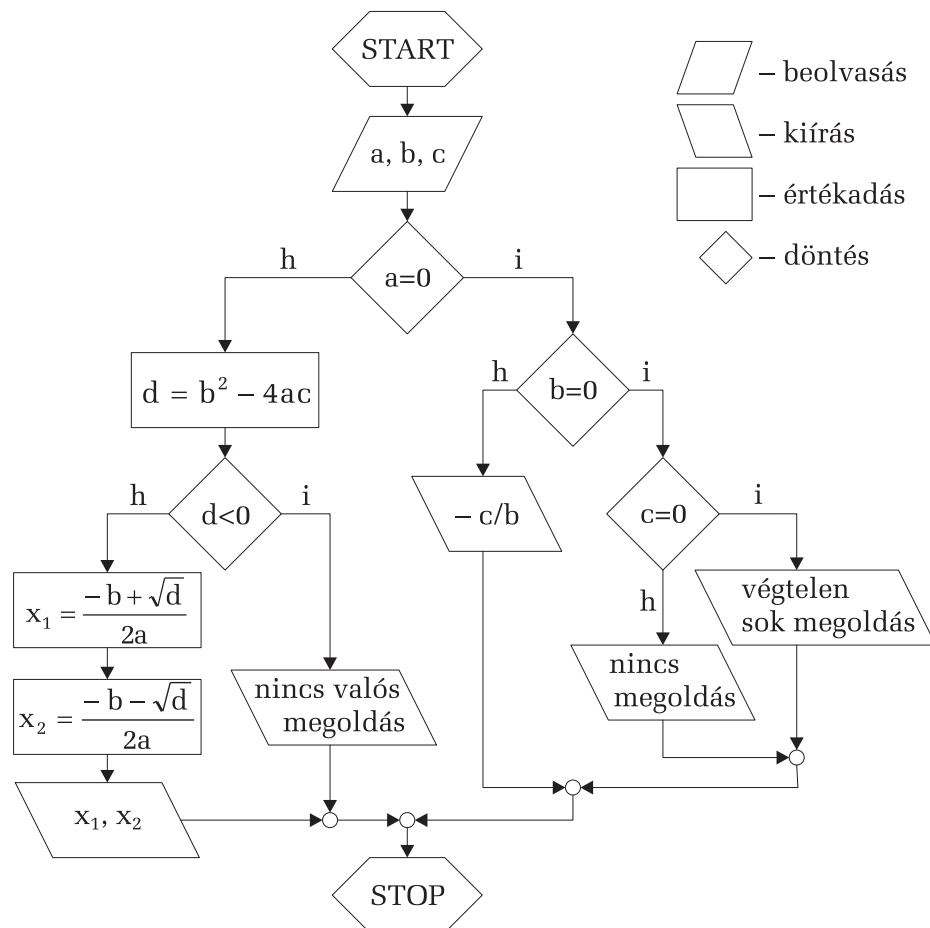
1. lépés. *A feladat elemzése.* A feladat elemzésekor az alábbi kérdésekre keressük a választ:

1. Mi adott a feladatban? Ezek lesznek a *bemenő adatok*, jelen esetben az a, b, c együtthatók.
2. Mit kell kiszámítani vagy elvégezni? Ezeket nevezzük *eredményeknek*. A másodfokú egyenlet megoldásának eredményei az x_1, x_2 gyökök.
3. Melyik az a számítógép szintjén megfogalmazott művelet, amelyet ha végrehajt, lépésről lépésre elvezeti a bemenő adatoktól a megoldáshoz? Ezt nevezzük a feladat megoldási *algoritmusának*. Ennek meghatározása jelenti lényegében a feladat megoldását. Ez a legfontosabb és egyben legnehezebb lépés.

Az algoritmusok két kitüntetett művelete a bemenő adatok *beolvasása*, illetve az eredmények *kiírása*. Honnan történik a bemenő adatok beolvasása, illetve hova írónak ki az eredmények? A beolvasás általában a billentyűzetről vagy egy állományból, a kiírás a képernyőre vagy egy állományba történik.

Egy lehetséges forgatókönyv annak, ahogy a számítógép megoldja a feladatot, a következő:

Beolvassa a billentyűzetről azt, ami adott a feladatban, a megoldási algoritmusnak megfelelően *feldolgozza* ezeket az adatokat, és *kiírja* az eredményeket a képernyőre.



1.4. ábra. A másodfokú egyenlet megoldási algoritmusának logikai sémája

Az algoritmusokat *logikai sémák* vagy *pszeudokódnyelv* segítségével szokás ábrázolni. Az alábbiakban megtekinthető a fenti feladatot megoldó algoritmus logikai sémával és pszeudokódnyelven felvázolva, illetve C nyelvre átírva. Nem fűzünk magyarázatot hozzájuk, hiszen célunk velük egyszerűen annyi, hogy megfoghatóbbá tegyük a tárgyalt fogalmakat. Ennek ellenére csak nyer az olvasó, ha megpróbálja magától megérteni őket.

Algoritmusok tulajdonságai

- Teljesség: Az algoritmusnak minden olyan lépést tartalmaznia kell, ami a feladathoz szükséges.
- Egyértelműség: A feladatot egyértelműen kell megoldani, azaz nem lehetnek kétértelmű részei.
- Végesség: Az algoritmusnak véges sok lépésben be kell fejeződnie, a lépések száma és a végrehajtás ideje egyaránt véges.
- Általános érvényűség: Egy algoritmus a feladatok egész osztályát képes megoldani bármilyen bemenő adatokra.
- Determinizmus: Ez azt jelenti, hogy az algoritmus minden lépése előre ismert, és minden műveletet előre ismert művelet követ.

```

algoritmus másodfokú egyenlet(a, b, c)
Pszudokód
beolvas a, b, c
ha a = 0
    akkor ha b = 0
        akkor ha c = 0
            akkor kiír "Végtelen sok megoldás"
            különben kiír "Nincs megoldás"
        vége ha
    különben kiír - c / b
    vége ha
különben
    d ← b2 - 4ac
    ha d < 0
        akkor kiír "Nincs valós megoldás"
        különben
            x1 ← (-b + √d) / 2a
            x2 ← (-b - √d) / 2a
            kiír x1,x2
        vége ha
    vége ha
vége algoritmus
    
```

```

# include <stdio.h>
C nyelvű program
# include <math.h>
main()
{
    double a, b, c, d, x1, x2;
    scanf("%lf%lf%lf", &a, &b, &c);
    if (a == 0)
        if (b == 0)
            if (c == 0) printf("Vegtelen sok megoldas");
    
```

```

        else printf("Nincs megoldas");
    else printf("%lf", - c / b);
else
{
    d = b * b - 4 * a * c;
    if (d < 0) printf("Nincs valos megoldas");
    else
    {
        x1 = (- b + sqrt(d)) / (2 * a);
        x2 = (- b - sqrt(d)) / (2 * a);
        printf("%lf, %lf", x1, x2);
    }
}
return 0;
}

```

Ha rendelkezésre áll az algoritmus, milyen további lépésekre van szükség ahhoz, hogy a számítógép által végrehajtható programhoz jussunk?

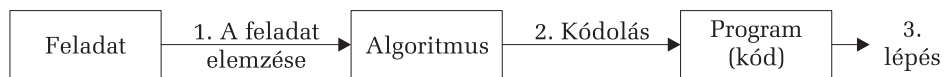
2. lépés. Kódolás. Az algoritmus átírása C nyelvre. Eredménye a program (kód), amely utasítja a számítógépet az algoritmusban előírt műveletek elvégzésére.

3. lépés. Szerkesztés (Editálás). A C program megszerkesztése egy megfelelő (általában a C programozási környezetbe beépített) szövegszerkesztővel. Eredménye a forráskódnak nevezett állomány.

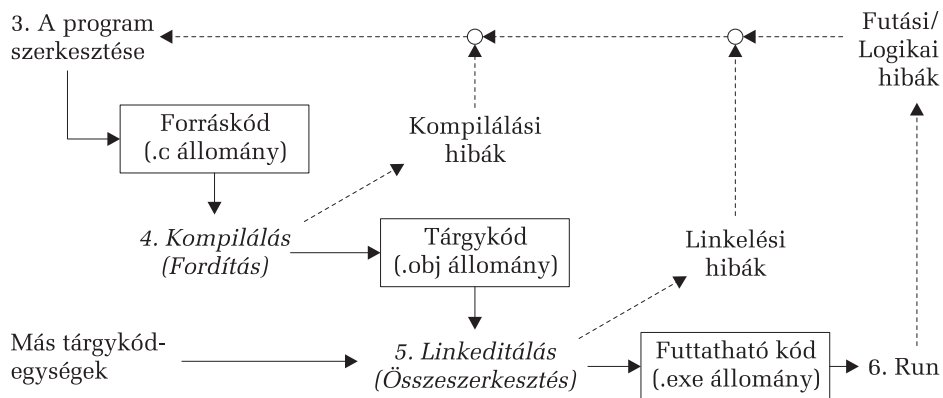
4. lépés. Fordítás (Kompilálás). A forráskód lefordítása gépi kódra. Eredménye a tárgykód.

5. lépés. Összeszerkesztés (Linkeditálás). Eredménye a futtatható kód (a gép képes végrehajtani).

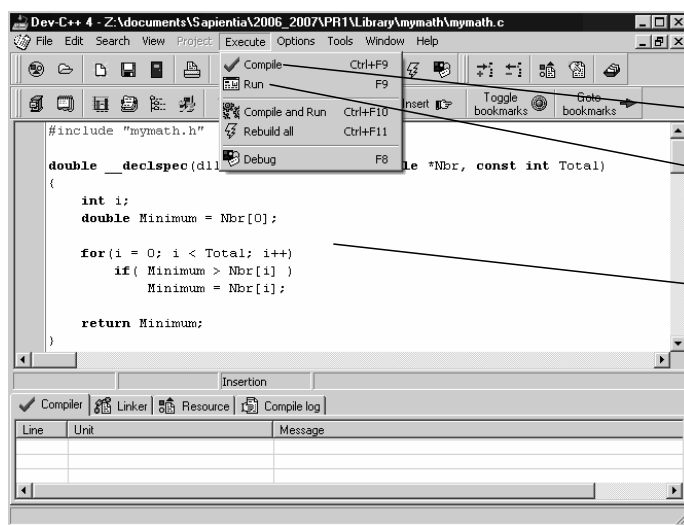
6. lépés. Futtatás (Run). A gép végrehajtja a program utasításait az előírt műveleteket, megoldva ezáltal a kívánt feladatot.



1.5. ábra. 1–2. lépés (fejen/papíron)



1.6. ábra. 3–6. lépés (számítógépen)

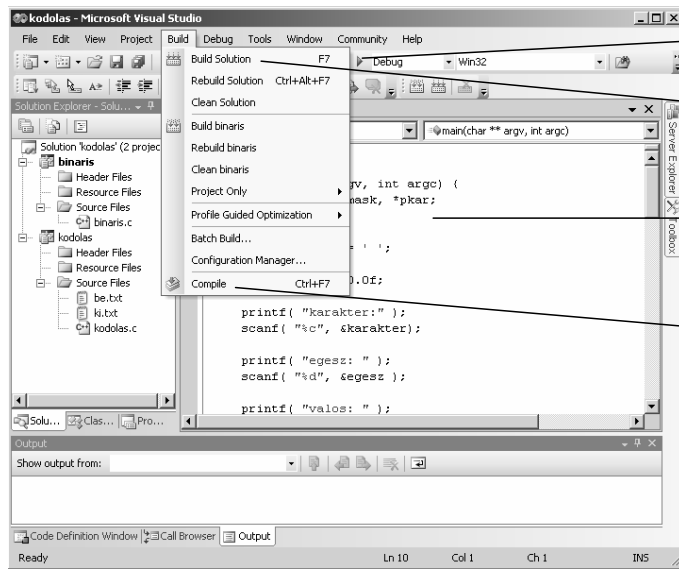


Fordítás

Összeszerkesztés és Futtatás

Kódolás és Editálás

Dev-C programozási környezet.



Futtatási ikon

Fordítás
és
Összeszerkesztés

Kódolás
és
Editálás

Fordítás

A Visual Studio C programozási környezete.

2. FEJEZET

MEMÓRIAVÁLTOZÓK

Hova kerülnek a feladat bemenő adatai a beolvasás nyomán? A memóriába, pontosabban az úgynevezett *memóriaváltozó*kba. Ugyancsak változókból tároljuk a feladat eredményeit és részeredményeit is.

A memóriát úgy kell elképzelni, mint sorszámozott memóriarekeszek egymásutánját. Minden rekeszbe nyolc – bitnek nevezett – bináris számjegy (0 vagy 1) kerül tárolásra. Mivel 8 bit 1 byte-ot alkot, a rekeszek száma megadja a memória kapacitását byte-ban. Egy rekesz címét a sorszáma fejezi ki. A memóriaváltozók számára egymás utáni memóriarekeszek foglalódnak le. Az első rekesz címe számít a változó címének. Egy változó mérete annyi byte, ahány memóriarekesz van lefoglalva számára.

Tekintsük egyelőre a változók következő négy jellemzőjét: név, cím, típus és érték.

| Memória | |
|------------|----------|
| 0 | 00000000 |
| 1 | 00000000 |
| 2 | 00000000 |
| . | 00000000 |
| . | 00000000 |
| . | 00000000 |
| 1000 | 00000000 |
| cím → 1001 | 00100100 |
| 1002 | 00011101 |
| 1003 | 00000000 |
| . | 00000000 |
| . | 00000000 |
| . | 00000000 |

2.1. ábra. *Változó a memóriában*

2.1. Változónév (azonosító)

Egy változónév az angol ábécé kis- és nagybetűit, az aláhúzásjelt, valamint számjegyeket tartalmazhat, de nem kezdődhet számjeggyel. A C nyelv CASE-érzékeny, ami azt jelenti, hogy különbséget tesz kis- és nagybetű között: `mas`, `Mas`, `MAS`, más-más változónevek. Ez a meghatározás érvényes minden azonosítóra a C nyelvben, nemcsak a változónevekre. Ajánlatos „beszédes” azonosítókat választani, vagyis olyanokat, amelyeknek a jelentése sugalmazza, hogy mire, milyen céllal fogjuk használni. Sőt hasznos lehet szokásokat kialakítani. Például az `s` változóban mindig összeget tároljunk, az `i`-ben számlálót...

2.2. Változócím

Az `&` operátor segítségével hivatkozhatunk egy változó címére. Például `&mas` a `mas` nevű változó címét jelenti.

2.3. Változótípusok

A változó típusa határozza meg a méretét, valamint azt, hogy milyen értéktartományból tárolhat értékeket. Később látni fogjuk, hogy a *típus* magába foglalja az illető adattal elvégezhető műveleteket is.

Alaptípusok:

`char`, `int`: egész számok tárolására,
`float`, `double`: valós számok tárolására,
`void`: semmilyen/bármilyen típus (lásd később).

Megjegyzés. Nem minden valós szám tárolható el pontosan a számítógép memóriájában. Ez logikus, hiszen egyes valós számok *vége*len tizedes számjegyet tartalmaznak, amit lehetetlen ábrázolni a ma használatos számítógépeken.

Az alaptípusokból úgynevezett *típusmódosítókkal* további típusok nyerhetők.

Típusmódosítók:

`signed`, `unsigned`, `short`, `long`.

A 2.1. táblázat összefoglalja a leggyakrabban használt egyszerű típusokat, megadva a méretüket bitben, és a hozzájuk kapcsolódó értéktartományt. Amennyiben hiányzik az alaptípus, implicit `int`-nek tekintendő. Például a `short` valójában `short int`-et jelent.

2.1. táblázat. *A C nyelv leggyakrabban használt egyszerű típusai*

| Típus | Méret | Értéktartomány |
|-----------------------------|-------|---|
| <code>char</code> | 8 | −128 .. 127 |
| <code>unsigned char</code> | 8 | 0 .. 255 |
| <code>short</code> | 16 | −32768 .. 32767 |
| <code>unsigned short</code> | 16 | 0 .. 65535 |
| <code>int</code> | 16 | −32768 .. 32767 |
| <code>int</code> | 32 | −2147483648 .. 2147483647 |
| <code>unsigned int</code> | 16 | 0 .. 65535 |
| <code>unsigned int</code> | 32 | 0 .. 4294967295 |
| <code>long</code> | 32 | −2147483648 .. 2147483647 |
| <code>unsigned long</code> | 32 | 0 .. 4294967295 |
| <code>float</code> | 32 | $3.4 \cdot 10^{-38}$.. $3.4 \cdot 10^{38}$ |
| <code>double</code> | 64 | $1.7 \cdot 10^{-308}$.. $1.7 \cdot 10^{308}$ |
| <code>long double</code> | 80 | $3.4 \cdot 10^{-4932}$.. $1.1 \cdot 10^{4932}$ |

A fenti táblázatba foglalt értékek csak általában érvényesek, hiszen a típusméretek függhetnek a fordítótól. Az ANSI standard csak az alábbiakat írja elő:

$$1 = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \\ \text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double})$$

A `sizeof` operátor egy típus méretét adja meg byte-ban. (4.3.9. alfejezet)

2.4. A változó értéke

A változó értéke a benne eltárolt érték. Ezen érték a programfutás alatt változhat (ezért is nevezik változónak). A mindennapi életben általában 10-es számrendszerben dolgozunk, ami azt jelenti, hogy 10 számjegyet (0, 1, 2, ... 9) használunk. Ezzel szemben a számítógép csak 2 számjegyet ismer, a 2-es számrendszer számjegyeit, a 0-t és az 1-et. Mindennek dacára, a mi

kényelmünkért, 10-es számrendszerben értekeznek a felhasználóval. Ez viszont azt jelenti, hogy a billentyűzetről begépett adatokat a számítógépnek először át kell alakítania, és csak azután kerülhet sor a tárolásukra. Azt a bináris alakot, amelyben az adatok eltárolásra kerülnek a memóriában, *belső ábrázolásnak* nevezzük. Anélkül, hogy részleteznénk, megemlítjük, hogy az egész számok belső ábrázolását fixpontos, a valós számokét pedig lebegőpontos (angolul *floating*) ábrázolásnak nevezik (lásd a B. függelék).

2.5. Változódefiniálás

Hogyan közöljük a számítógéppel, hogy hozzon létre a memóriában egy változót? Az illető változó definiálása által. Egy változó definiálásakor meg kell adnunk a változó nevét, típusát (ez határozza meg a méretét), és adhatunk egy kezdőértéket is (ez nem kötelező). Ha egy változónak nem adunk értéket a definiálásakor, akkor bizonyos esetekben (programozási környezettől függően) automatikusan 0-val inicializálódik, máskor viszont úgynevezett „memóriaszemetet” fog tartalmazni.

A változódefiniálás szintaxisa:

`<típus> <változó>[= <kezdőérték>];`

A szögletes zárójelek azt jelzik, hogy a kezdőértékadás nem kötelező.

Példák:

```
int a;
float b1, b2;
long i, j = 1, k;
unsigned char c = 65;
long double x, y = 3.14;
```

Az algoritmusok kapcsán két kitüntetett műveletről beszéltünk: a bemenő adatok *beolvasásáról* és az eredmények *kiírásáról*. Hogyan utasítható a számítógép egy C programból ezen műveletek lebonyolítására? A billentyűzetről való olvasásra, illetve a monitorra való írásra például a `scanf` és `printf` függvények által. Szöveges állomány(ból/ba) való olvasás/írás esetén használhatjuk az `fscanf` és `fprintf` függvényeket.

2.6. A **scanf**, **printf**, **fscanf** és **fprintf** függvények¹

Lássunk mindenekelőtt egy konkrét példát az említett függvények használatára.

2.1. feladat. Írjunk programot, amely beolvas a billentyűzetről/állományból egy egész számot, majd kiírja a képernyőre/állományba.

C nyelven

```
#include <stdio.h>
main()
{
    int a;
    scanf("%d", &a);
    printf("A szám:%d", a);
    return 0;
}
```

C nyelven

```
#include <stdio.h>
main()
{
    FILE * f;
    FILE * g;
    int a;
    f = fopen("be.txt", "r");
    g = fopen("ki.txt", "w");
    fscanf(f, "%d", &a);
    fprintf(g, "A szám:%d", a);
    fclose(f); fclose(g);
    return 0;
}
```

Magyarázat, tanulságok:

– A program „keretéről”,

```
#include <stdio.h>
main()
{
    ...
    return 0;
}
```

¹ Nem áll szándékunkban itt, most a teljesség igényével bemutatni ezeket a függvényeket. Csak annyit szeretnénk elmondani, amennyire szükségünk lesz ahhoz, hogy használni tudjuk a könyv első felében szereplő feladatokban. Az állománykezeléssel foglalkozó fejezetben (11.2. alfejezet) újra visszatérünk rájuk. Ha az olvasó teljes képet szeretne kapni róluk, forduljon a Help-hez vagy a könyvészet azon kötetéhez, amelyek teljes leírást adnak a C nyelvről.

egyelőre csak annyit, hogy

- az `#include <stdio.h>` programsort a `scanf`, `printf`, `fscanf` és `fprintf` függvények használata teszi szükségessé,
- a `main() {...return 0;}` főfüggvény minden C program része. Egy program végrehajtása alapvetően a `main` függvény utasításainak végrehajtását jelenti. Az operációs rendszer, a program futtatásakor, ennek a függvénynek adja át a vezérlést és ettől is kapja vissza (`return 0;`). Természetesen a `main` függvény meghívhat más függvényeket, átadva és visszakapva a vezérlést. Ezek lehetnek könyvtári függvények (jelen programjainkban az `fopen`, `fclose`, `scanf`, `fscanf`, `printf`, `fprintf`) vagy saját függvények (9. fejezet).
 - Minden kapcsolósárójelpár közé zárt programrészt *blokknak* nevezünk. Vegyük észre, hogy a *változók definiálására a blokkok elején* kerül sor. Lesznek példák, amelyekben – helytakarékosági okokból – csak a `main` függvény blokkjának tartalmát fogjuk megírni.
 - Az első program a billentyűzetről olvas és a monitorra ír. A második programban viszont a beolvasás a `be.txt` nevű állományból történik, a kiírás pedig a `ki.txt` nevű állományba.
 - Az állomány(ból/ba) való olvasás/írás esetén szükséges az állományt ezt megelőzően megnyitni (`fopen`), majd a végén bezárni (`fclose`).
 - A bemeneti állományt (`be.txt`) olvasásra (`r` – *read*) nyitjuk meg, a kimeneti (`ki.txt`) pedig írásra (`w` – *write*).
 - A bemeneti állománynak léteznie kell a program futtatását megelőzően, és tartalmaznia kell a bemenő adatokat, a kimeneti állományt viszont a program fogja létrehozni, a megadott név alatt, és ebbe írja az eredményeket.
 - Az `f` és `g`, `FILE *` típusú változókat *állománymutatóknak* nevezük, és ezek által jelezzük az `fscanf`, illetve `fprintf` függvényeknek, hogy melyik állományból olvassanak, illetve melyikbe írjanak. A `FILE *` típus egy *pointer* típus, amelyet később fogunk bemutatni.
 - Vegyük észre, hogy a `scanf`, illetve `fscanf` és a `printf`, illetve `fprintf` függvények csak abban különböznek, hogy az utóbbiakban szerepel az állománymutató.

Az állománymutató definiálásának szintaxisa:

```
FILE * <állomány_mutató>;
```

Az állomány megnyitásának szintaxisa:

```
<állomány_mutató> = fopen("<állomány_név>",  
    "<megnyitás_módja>");
```


Az állomány bezárásának szintaxisa:

```
fclose(<állomány_mutató>);
```

A `scanf`, `printf`, `fscanf` és `fprintf` függvények szintaxisai:

```
scanf("<formázósor>", <változócím_lista>);
fscanf(<állomány_mutató>, "<formázósor>",
      <változócím_lista>);
printf("<formázósor>", <kifejezés_lista>);
fprintf(<állomány_mutató>, "<formázósor>",
      <kifejezés_lista>);
```

Említettük, hogy a beolvasott adatok bizonyos átalakulásokon mennek át, mielőtt eltárolásra kerülnének a megfelelő változókbán. Ha azt az alakot, amelyben begépeljük az adatokat a billentyűzetről, vagy amelyben a bemeneti állományban vannak, *külső ábrázolásnak* nevezzük, akkor beolvasáskor át kell térni külső ábrázolásról *belső ábrázolásra*, kiíráskor pedig fordítva, belsőről külsőre. Mivel az átalakítások típus-függőek, tájékoztatnunk kell függvényeinket arról, hogy tekintettel a változók (`scanf/fscanf` esetében), illetve a kifejezések (`printf/fprintf` esetében) típusára, milyen átalakításra van szükség. Ezt a szerepet a formázósorok % karakterrel bevezetett úgynevezett formázókarakterei töltik be. Például a `scanf` (`fscanf`) beolvassa a billentyűzetről begévelt (az állományban található) adatokat, a megfelelő formázókarakter által megjelölt módon átalakítja őket, majd belső ábrázolásukat eltárolja a változólistában megjelölt változócímekre. A `scanf` (`fscanf`) formázósora gyakran csak formázókaraktereket tartalmaz.

A `printf` (`fprintf`) a formázósorában található nem formázó karaktereket egy az egyben kiírja a képernyőre (állományba), a %-kal bevezetett formázókarakterek helyére pedig beírja a kifejezéslista megfelelő kifejezésének az értékét (miután átalakította a formázókarakterrel megadott módon belső ábrázolásból külső ábrázolásba).

A leggyakrabban használt formázókaraktereket a 2.2. táblázat tartalmazza.

2.2. táblázat.

| Típus | Formázókarakter |
|--------------------|--|
| char | c |
| int | d vagy i (10-es számrendszerben) o (8-as számrendszerben) x, X (16-os számrendszerben) |
| unsigned int | u |
| short int | hd vagy hi |
| unsigned short int | hu |
| long int | ld vagy li |
| unsigned long int | lu |
| float | f |
| double | lf |
| long double | Lf |
| karakterlánc | s |
| pointer | p |

Az alábbi példák bemutatják a formázó sor formázó karakterei, valamint a `scanf` változó cím listája, illetve a `printf` kifejezés listája közötti kapcsolatot:

```
long a;
float b;
double c;
long double d;
scanf("%ld%f%lf%Lf", &a, &b, &c, &d);

printf("a=%ld\nb=%f\nc=%lf\nd=%Lf\n", a, b, c, d);
```

Következzen egy példa az o, x és X formázó karakterek használatára.

A `printf("%d, %o, %x, %X ", 10, 10, 10, 10);` utasítás nyomán a képernyőn a következő jelenik meg: 10, 12, a, A.

Megjegyzés. A tízes számrendszerbeli 10-es nyolcas számrendszerben (oktál) 12, tizenhatos számrendszerben (hexa) pedig a vagy A.

Ha több adatot szeretnénk beolvasni ugyanazzal a `scanf`-fel, akkor ezeket fehér karakterekkel (*white-space*-nek nevezzük, például, a SPACE,

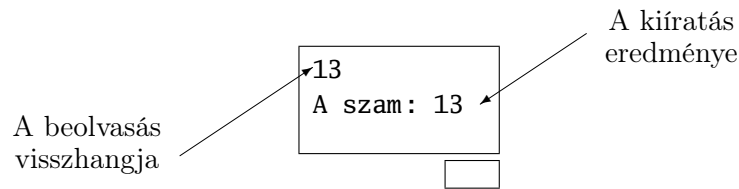
2.6. A SCANF, PRINTF, FSCANF ÉS FPRINTF FÜGGVÉNYEK

31

TAB és ENTER billentyűk leütésekor továbbbítódó karaktereket) elválasztva kell begépelni, és a végén ENTER-t kell leütni. Ugyanezeket az elválasztó karaktereket kell használni a bemeneti állományokban is. Billentyűzetről való beolvasás alatt a számítógép megjeleníti az úgynevezett felhasználói képernyőt (UserScreen), amely visszhangozza a begépelte karaktereket. Ugyanezen a képernyőn jeleníti meg a **printf** is az eredményeket.

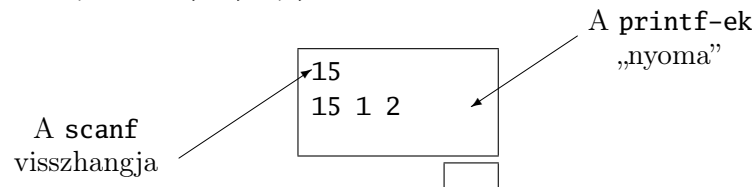
Ha a fenti feladat első megoldásában a billentyűzetről beolvasott érték 13, akkor a képernyőn a következő jelenik meg: **A szam: 13.**

A UserScreen tartalma a program lefutása nyomán:



Megjegyzés. A **scanf**-nek és **fscanf**-nek, illetve **printf**-nek és **fprintf**-nek mint függvényeknek az értéke a sikeresen beolvasott értékek, illetve a kiírt karakterek száma lesz. Figyeljük meg, hogyan fog kinézni a UserScreen az alábbi utasítások végrehajtása nyomán, ha n-be 15-öt olvassunk be:

```
int a, b;
a = scanf("%d", &n);
b = printf("%d ", n);
printf("%d %d", a, b);
```



A további feladatokból még több derül ki a bemutatott függvényekről.

2.2. feladat. Írjunk programot, amely beolvas a billentyűzetről két valós számot, majd kiírja a képernyőre magukat a számokat, illetve az összegüket.

```
#include <stdio.h>
main()
{
    float x1, x2;
    printf("Írd be az első számot:");
    scanf("%f", &x1);
    printf("Írd be a második számot:");
    scanf("%f", &x2);
    printf("A két szám:%5.2f és %5.2f\n", x1, x2);
    printf("A két szám összege:%5.2f", x1+x2);
    return 0;
}
```

Megjegyzés. Biztosan feltűnt, hogy a **printf** formázósorában a % karaktert nem közvetlenül követte az **f** (**float**) formázókarakter. Igen, megadható, hány pozíción kerüljenek kiírásra az értékek, és valós számok esetén az is, hogy ebből hány legyen a tizedesrésznek fenntartva (ennek hiányában implicit 6 tizedes pontossággal történik a valós számok kiírása). Úgy is mondhatnánk, formázható a megjelenítés módja. A fenti példában öt pozíción történik a kiírás, amiből kettő a tizedesrészé. Ez azt jelenti, hogy az egészrésznek ugyancsak két pozíció marad, hiszen egy pozíciót a tizedespont fog elfoglalni. Ha az egészrész több számjegyű, mint ahány pozíción kérjük a kiíratást, akkor figyelmen kívül marad a kért pozíciószám, hogy az információ ne sérüljön. Ellenkező esetben az üresen maradt pozíciók szóköz karakterrel pótolódnak ki. Ha a tizedes számjegyek száma több, mint a számukra fenntartott pozíciószám, akkor a plusz számjegyek „lemaradnak”, és kerekítés történik. Kevesebb tizedes számjegy esetén nullákkal egészítődik ki a tizedesrész.

Példa:

| printf("%5.2f", x) | megjelenítés | | | | | | |
|------------------------|--------------|---|---|---|---|---|---|
| ha x értéke: 13.47 | | | 1 | 3 | . | 4 | 7 |
| ha x értéke: 1352.4712 | 1 | 3 | 5 | 2 | . | 4 | 7 |
| ha x értéke: 3.4777 | | | | 3 | . | 4 | 8 |
| ha x értéke: .5 | | | | 0 | . | 5 | 0 |
| ha x értéke: 0. | | | | 0 | . | 0 | 0 |

2.3. feladat. Írjunk programot, amely beolvas a billentyűzetről két egész számot, majd kiírja a képernyőre az összegüket, különbségüket, szorzatukat, hányadosukat, valamint osztási maradékukat (feltételezzük, hogy a második szám különbözik nullától).

```
#include <stdio.h>
main()
{
    int a, b, osszeg, kulonbseg, szorzat, hanyados, maradek;
    printf("Írd be a két számot:");
    scanf("%d%d", &a, &b);
    osszeg = a + b;
    kulonbseg = a - b;
    szorzat = a * b;
    hanyados = a / b;
    maradek = a % b;
    printf("Összeg: %4d\nKülönbség: %4d\nSzorzat: %7d\n
           Hányados: %4d\nMaradék: %4d", osszeg, kulonbseg, szorzat,
           hanyados, maradek);
    return 0;
}
```

Megjegyzés. Egész számok kiírásakor is megadható a mezőszélesség, azaz, hogy hány pozíción kerüljenek kiírásra. Ugyanaz a megjegyzés érvényes rájuk, mint amit a valós számok egészrészénél tettünk.

Példa:

| printf("%5ld", x) | megjelenítés | | | | |
|------------------------|--------------|---|---|---|-------|
| ha x értéke: 12947 | 1 | 2 | 9 | 4 | 7 |
| ha x értéke: 135294712 | 1 | 3 | 5 | 2 | 9 4 7 |
| ha x értéke: 77 | | | | 7 | 7 |

Megjegyzések a programban használt operátorokkal (műveletjelekkel) kapcsolatban.

1. A +, -, * operátorok a matematikában megszokott módon viselkednek. A / operátor esetében, ha mindkét operandus egész, akkor egész osztás történik, ha viszont valamelyik valós, akkor valós osztás. A % az egész osztás maradékát adja meg, ezért operandusai csak egészek lehetnek.

Példák:

```
7 / 2      ← 3
7.0 / 2.0  ← 3.5
7.0 / 2    ← 3.5
7 / 2.0    ← 3.5
7 % 2      ← 1
```

2. Az „=” jelet értékadás-operátornak nevezzük.

Az értékadás szintaxisa:

```
<változó> = <kifejezés>;
```

Hatására a változó (bal oldal) felveszi a kifejezés (jobb oldal) értékét.

Példák:

```
double x = 1, y = 2, z = 3, w;
w = (x + y) * (z - 1);
```

A w változó értéke 6 lesz.

3. Különböző lehetőségek egy változó értékének eggyel való növelésére, illetve csökkentésére:

```
int a = 7;
a = a + 1;  ⇔  a += 1;  ⇔  a++;  ⇔  ++a;
a = a - 1;  ⇔  a -= 1;  ⇔  a--;  ⇔  --a;
```

4. Mi a különbség az a++, illetve ++a inkrementálások között?

Példa:

```
int a = 1, b, c = 2, d;
d = c++; /* A d értéke 2, a c értéke pedig 3 lesz */
b = ++a; /* Mind a b, mind az a változók értéke 2 lesz */
```

Az első utasítás értelmében a számítógép először használja a változó értékét a kifejezés kiértékelésében, és csak azután növeli, a második esetben pedig fordítva: először növeli és azután használja.

Természetesen ugyanez igaz a -- operátor esetében is.

5. A változó értékének növelése egy adott értékkel:

```
int a = 7, b = 3;
a = a + b;  ⇔  a += b;
a = a - b;  ⇔  a -= b;
```

6. Kombinált operátorok használatának szintaxisa:

```
<változó>=<változó>+<kifejezés>;
⇔ <változó>+= <kifejezés>;
```

Más kombinált aritmetikai operátorok: +=, -=, *=, /=, %=.

7. A \n-nel jelölt karaktert a **printf**, illetve a **fprintf** formázó-sorában újsor (*newline*) karakternek nevezik, ugyanis új sorba lépteti a képernyőn vagy a kimeneti állományban a kurzort.

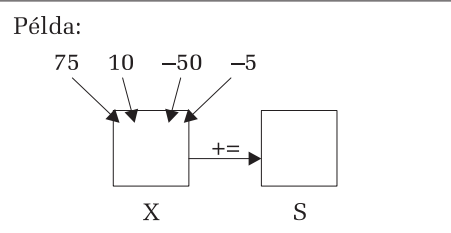
2.4. feladat. Adott 4 egész szám (4 elemű számsorozat) a **szamok.be** szöveges állományban. Számítsuk ki az összegüket, és írjuk ki a **szamok.ki** szöveges állományba.

Megjegyzés. Első látásra úgy tűnhet, hogy szükségünk lesz négy változóra a négy szám eltárolása végett. Az alábbi megoldásból kiderül, hogy elég egy változót használni erre a célra. A mellékelt ábra szemlélteti az ötletet. Ezen megközelítés előnye, hogy a változók száma nem függ a beolvasott számok számától.

2.6. A SCANF, PRINTF, FSCANF ÉS FPRINTF FÜGGVÉNYEK

35

Példa:



| szamok.be | szamok.ki |
|--------------|---------------|
| 75 10 -50 -5 | Az összeg: 30 |

```
#include <stdio.h>
main()
{
    FILE *f1, *f2;
    int x, s = 0;
    f1 = fopen("szamok.be", "r");
    f2 = fopen("szamok.ki", "w");
    fscanf(f1, "%d", &x); s += x;
    fscanf(f1, "%d", &x); s += x;
    fscanf(f1, "%d", &x); s += x;
    fscanf(f1, "%d", &x); s += x;
    fprintf(f2, "Az Összeg:%d", s);
    fclose(f1); fclose(f2);
    return 0;
}
```

Tanulság: Egy számsorozatot gyakran elemenként dolgozunk fel.

2.5. feladat. Írjunk programot, amely beolvas a billentyűzetről egy négyjegyű természetes számot, és kiírja a számjegyei összegét a képernyőre.

Ötlet: Az $n \% 10$ kifejezés az n utolsó számjegyét *adja meg*, az $n /= 10$; utasítás pedig *levágja* (elosztja n -et 10-el) az n utolsó számjegyét. A mellékelt táblázat nyomon követi, hogyan változik az n és a p változók értéke a program végrehajtása alatt.

```
#include <stdio.h>
main()
{
    int n, p = 1;
    printf("Írd be a számot:");
    scanf("%d", &n);
    /*
        4-szer fogom megismételni a következő
        két műveletet:
        p-t megszorozom n utolsó számjegyével,
        majd levágom n utolsó számjegyét.
    */
    p *= n % 10; n /= 10;
    p *= n % 10; n /= 10;
    p *= n % 10; n /= 10;
    p *= n % 10; n /= 10;
    printf("A szorzat:%d", p);
    return 0;
}
```

Példa: ha a beolvasott szám 2541

| n | p |
|------|----|
| 2541 | 1 |
| 254 | 1 |
| 25 | 4 |
| 2 | 20 |
| 0 | 40 |

Tanulságok:

- Egy természetes számot gyakran számjegyenként dolgozunk fel.
- Az n érték 10-el való osztási maradékának előállítása nem jár együtt az n változó értékének megváltozásával. (Sőt n -nek a 10-el való osztási hányadosának az előállítása sem módosítja n -t) Amint megfigyelhettük, n -nek 10-el való elosztását az $n = n/10$; utasítás valósítja meg (ami ekvivalens az $n/=10$; alakkal), amely hatására a számítógép először előállítja az $n/10$ értéket, majd ezzel felülírja az n változó értékét.

Egy C program tartalmazhat magyarázó szövegeket, úgynevezett *kommentárokat* (`/* ... */`). Ezeket a fordító figyelmen kívül hagyja. Céljuk könnyebben érthetővé tenni a forráskódot. Az olyan programok esetében, amelyeket hosszú időn keresztül használunk, elengedhetetlen a megfelelő kommentárok használata.

2.7. Kitűzött feladatok

1. Írjunk programot, amely beolvas két valós számot két változóba, kicseréli a változók tartalmát, majd pedig kiírja a számokat fordított sorrendben. Próbáljuk megoldani a feladatot úgy is, hogy ne használjunk harmadik változót a cseréhez.

2. Írjunk programot, amely beolvas egy ötszámjegyű természetes számot egy egész változóba, előállítja a fordítottját egy másik változóban, majd kiírja a két számot egymás alá a képernyőre.

3. Írjunk programot, amely beolvas négy, háromszámjegyű természetes számot, és kiírja a számjegyeik összegének átlagát.

4. Írjunk programot, amely kiszámítja egy adott sugarú gömb, illetve adott sugarú és magasságú egyenes henger és kúp felszínét és térfogatát. Az eredményeket táblázatos formában jelenítsük meg.

5. Írjunk programot, amely kiszámolja és kiírja egy gépkocsi féktávolságát a sebesség és az útviszonyok függvényében. A feltételezett lassulás

- a) normál úton: $4,4 \text{ m/s}^2$,
- b) vizes úton: $3,4 \text{ m/s}^2$,
- c) vizes, nyálkás úton: $2,4 \text{ m/s}^2$.

A reakcióidő: 1 másodperc.

A gépkocsi kezdősebessége bemeneti adat.

6. Írjunk programot, amely beolvassa egy derékszögű háromszög egyik szögének értékét fokokban, az átfogót cm-ben, és kiírja a háromszög befogóinak hosszát és a háromszög köré írható kör területét és kerületét.

7. Olvassuk be a képernyőről egy piskótatartó méreteit – átmérőjét és magasságát –, valamint a ráteendő krém vastagságát cm-ben. Számoljuk ki, mennyi krémre van szükség a torta bevonásához, ha 5%-os ráhagyással dolgozunk (gyerekek is vannak a családban...)!

8. Adott két pont, A és B a síkban a koordinátáik által. Határozzuk meg:

a) az AB szakasz hosszát

$$|AB| = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2},$$

b) az AB szakasz középpontjának koordinátáit

$$x_M = (x_A + x_B) / 2$$

$$y_M = (y_A + y_B) / 2,$$

c) az AB egyenes irányítányezőjét

$$m_d = \frac{y_A - y_B}{x_A - x_B},$$

d) egy M pont távolságát az AB egyenestől

$$\text{távolság}(M, d) = \frac{|a \cdot x_M + b \cdot y_M + c|}{\sqrt{a^2 + b^2}},$$

ahol a, b, c a d egyenes egyenletének együtthatói ($aX + bY + c = 0$). Továbbá két ponton átmenő egyenes egyenlete: $(X - x_A)(y_A - y_B) = (Y - y_A)(x_A - x_B)$.

e) Adott egy háromszög. Határozzuk meg a háromszög kerületét és területét a csúcpontjainak koordinátái alapján.

$$S_{ABC} = \sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)},$$

ahol a, b, c az oldalhosszak, p pedig a félkerület.

Megjegyzés. A feladatokat oldjuk meg az alábbi esetekben:

| bemenet | kimenet |
|--------------|----------|
| billentyűzet | monitor |
| billentyűzet | állomány |
| állomány | monitor |
| állomány | állomány |

3. FEJEZET

UTASÍTÁSOK

Az utasítások utasítják a számítógépet a feladat megoldási algoritmusában előírt műveletek elvégzésére. A következőkben bemutatjuk a C nyelv utasításkészletét.

3.1. Kifejezés-utasítás

Szintaxisa:

`<kifejezés>;`

A leggyakrabban használt kifejezés-utasítás az *értékadás*. Egy *függvényhívás* is kifejezés-utasítás. Eddigi programjainkban kizárólag ilyen utasítások szerepeltek. Íme hat példa kifejezés-utasításra:

```
printf("Írd be a két számot:");
scanf("%d%d", &a, &b);
osszeg = a + b;
a + b; /* bár szintaktikailag helyes, értelmetlen */
p *= n % 10;
n /= 10;
```

Egy speciális kifejezés-utasítás az *üres utasítás*: `;`

Általában olyan helyen használjuk, ahol a szintaxis utasítást követel, de nekünk nem áll szándékunkban oda utasítást írni (például olyan ciklusokban, amelyeknek hiányzik a *magjuk* – 3.4. alfejezet).

3.2. Összetett utasítás

Ha szeretnénk több egymás után következő utasítást egyetlen összetett utasításként kezelni, akkor össze kell fogunk őket egy *kapcsoszárojpárral* (`{ }`). Összetett utasítások képezésére akkor lehet szükség, amikor a szintaxis egyetlen utasítást kér, de mi több utasítást szeretnénk odaírni. Mivel egy kapcsoszárojpár közé zárt rész blokkot képez, az előző fejezetben

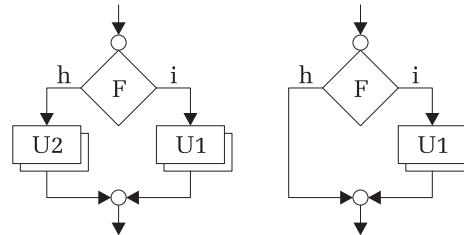
tett megjegyzéssel összhangban, minden összetett utasítás elején definiálhatunk változókat. Ezek viszont lokálisak lesznek az illető blokkra nézve, ami azt jelenti, hogy csakis a blokk keretén belül használhatók.

3.3. Döntési utasítások

3.3.1. Az if utasítás

Akkor használjuk, amikor egy feltétel igaz vagy hamis voltától függően, más-más műveletet kell elvégezni. (Mintha két "forgatókönyvünk" lenne: az egyik arra az esetre, ha a feltétel igaz, a másik pedig arra, ha hamis.) Olyan helyzet is előfordulhat, amikor, amennyiben igaz a feltétel, el kell végezni egy bizonyos műveletet, különben nem. Az alábbi ábrák logikai sémával, pszeudokódban és C nyelven mutatják be a döntésutasítást.

A döntés feltétele gyakran összehasonlítást foglal magába. Az összehasonlítási operátorok a következők: <, <=, >, >=, == (egyenlőségvizsgálat), != (különbözőségvizsgálat).



3.1. ábra. Az *if* utasítás logikai sémája. *F* – feltétel (logikai kifejezés); *U1*, *U2* – utasítások; *i* – igaz; *h* – hamis

Pszeudokód

```

ha <logikai_kifejezés>
  akkor <utasítás>
  [különben <utasítás>]
vége ha
    
```

C nyelven

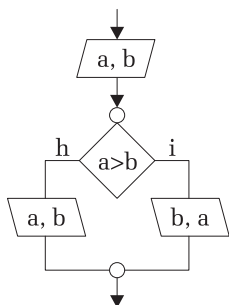
```

if (<logikai_kifejezés>)
  <utasítás>;
[else <utasítás>;]
    
```

Lássuk, hogyan használhatók a döntési utasítások feladatok megoldásában.

3.1. feladat. Írjunk programot, amely beolvassa a billentyűzetről két természetes számot, a -t és b -t, majd kiírja őket növekvő sorrendben.

1. megoldás



Pszéudokód

```

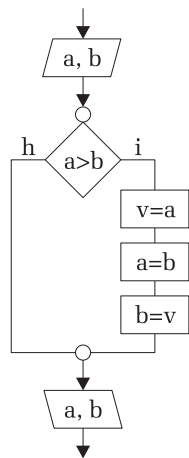
beolvas a, b
ha a > b
    akkor kiír b, a
    különben kiír a, b
vége ha
  
```

C nyelven

```

int a, b;
scanf("%d%d", &a, &b);
if ( a > b ) printf("%d %d", b, a);
else printf("%d %d", a, b);
  
```

2. megoldás



Pszéudokód

```

beolvas a, b
ha a > b
    akkor
        v <-- a
        a <-- b
        b <-- v
    vége ha
    kiír a, b
  
```

C nyelven

```

int a, b, v;
scanf("%d%d", &a, &b);
if ( a > b ) {v = a; a = b; b = v;}
printf("%d %d", a, b);
  
```

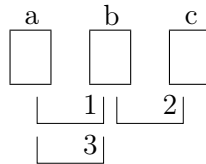
3.3. DÖNTÉSI UTASÍTÁSOK

41

Megjegyzés. Az első megoldásban egyszerűen kiírtuk a két változó értékét a kért sorrendben, míg a második megoldás fel is cseréli a két változó értékét. Így ebben a kért sorrendben állnak rendelkezésre az esetleges további feldolgozások során.

3.2. feladat. Az `input.c` bemeneti állományban adott három valós szám, írjuk ki őket növekvő sorrendben az `output.c` kimeneti állományba.

Ötlet: Összehasonlítgatjuk a változókat kettőnként az ábrán bemutatott stratégia szerint, és ahol szükséges (az összehasonlított változók értékei nincsenek növekvő sorrendben), kicseréljük a tartalmukat.



```
FILE *f, *g;
float a, b, c, v;
f = fopen("input.c", "r");
g = fopen("output.c", "w");
fscanf(f, "%f%f%f", &a, &b, &c);
if ( a > b ) {v = a; a = b; b = v;}
if ( b > c ) {v = b; b = c; c = v;}
if ( a > b ) {v = a; a = b; b = v;}
fprintf(g, "%f %f %f", a, b, c);
fclose(f);
fclose(g);
```

3.3. feladat. Olvassunk be négy egész számot a billentyűzetről. Ellenőrizzük, van-e közöttük páros szám, és írjunk ki egy megfelelő üzenetet a képernyőre.

1. megoldás

```
int a, b, c, d;
scanf("%d%d%d%d", &a, &b, &c, &d);
if (a%2 == 0) printf("Van közöttük páros." );
else if (b%2 == 0) printf("Van közöttük páros.");
    else if (c%2 == 0) printf("Van közöttük páros.");
        else if (d%2 == 0) printf("Van közöttük páros.");
            else printf("Nincs közöttük páros.");
```

Mielőtt adnánk egy elegánsabb megoldást is a fenti feladatra, ismerkedjünk meg a logikai operátorokkal.

3.3.1.1. Logikai operátorok

| | | | |
|-------------------------|-------------------|---------|------------------|
| <i>logikai és:</i> | && | (and) | – két operandusú |
| <i>logikai vagy:</i> | | (or) | – két operandusú |
| <i>logikai tagadás:</i> | ! | (not) | – egy operandusú |

- Az *and* művelet eredménye akkor igaz, ha mindkét operandusa igaz, különben hamis.
- Az *or* művelet eredménye akkor igaz, ha legalább az egyik operandusa igaz, különben hamis.
- A *not* művelet eredménye igaz/hamis, attól függően, hogy operandusa hamis/igaz.

2. megoldás

```
int a, b, c, d;
scanf("%d%d%d", &a, &b, &c, &d);
if (a%2 == 0 || b%2 == 0 || c%2 == 0 || d%2 == 0)
    printf( "Van közöttük páros." );
else printf("Nincs közöttük páros.");
```

- C-ben nincs logikai típus.
- Minden 0-tól különböző értéknek *igaz* a logikai értéke, a 0-nak pedig *hamis*. Ezért kifejezőbb, ha a (<kifejezés> == 0) alak helyett a (!<kifejezés>) alakot használjuk. Hasonlóképpen (<kifejezés> != 0) helyett írjunk egyszerűen(<kifejezés>)-t.

3. megoldás

```
int a, b, c, d;
scanf("%d%d%d", &a, &b, &c, &d);
if (!(a%2) || !(b%2) || !(c%2) || !(d%2))
    printf( "Van közöttük páros." );
else printf("Nincs közöttük páros.");
```

3.3.2. A switch utasítás

Az if utasítás esetében két "forgatókönyvünk" van arra az eshetőségekre, hogy egy logikai feltétel igaz vagy hamis. Néha arra van szükség, hogy a számítógép aszerint válasszon több "forgatókönyv" közül, hogy egy úgynevezett szelektor-kifejezés milyen értéket vesz fel. Ilyen esetben lehet célszerű switch utasítást használni.

Az alábbi feladat rávezet a **switch** utasítás gyakorlati értékére.

3.4. feladat. Írjunk programot, amely beolvas egy, az [1, 5] intervallumhoz tartozó természetes számot, és kiírja a neki megfelelő minősítést: 1 – elégtelen, 2 – elégséges, 3 – közepes, 4 – jó, 5 – jeles.

1. megoldás (egymásba ágyazott if utasításokkal)

```
int jegy;
scanf("%d", &jegy);
if(jegy == 1) printf("Elégtelen");
    else if(jegy == 2) printf("Elégséges");
        else if(jegy == 3) printf("Közepes");
            else if(jegy == 4) printf("Jó");
                else if(jegy == 5) printf("Jeles");
                    else printf("Téves beolvasás");
```

A switch utasítás szintaxisa:

```
switch(< szelektor >)
{
    case <k1>: <U1>; [ break; ]
    case <k2>: <U2>; [ break; ]
    :
    case <kn>: <Un>; [ break; ]
    [default <Un + 1>;]
}
```

Megjegyzések:

- a szelektor bármilyen *egész* típusú kifejezés lehet,
- a **case**-konstansok (k_1, k_2, \dots, k_n) egész *konstansok*,
- a **break** utasítás (opcionális) befejezi a **switch** utasítást (3.5.1. alfejezet).

Lássuk, hogyan működik a **switch** utasítás.

- Kiértékelődik a szelektor-kifejezés.

- A szelektor értéke összehasonlítódik sorra (fentről lefele) a **case** konstansokkal.
- Ha létezik a szelektorról egyenlő értékű **case** konstans (például k_i), a program „átlép” a megfelelő utasításhoz (U_i).
- Végrehajtódik U_i .
- Ha U_i -t **break** követi, befejeződik a **switch** utasítás.
- Ha U_i -t nem követi **break**, folytatódik a **switch** végrehajtása az U_{i+1} , U_{i+2} , ... utasításokkal U_n -ig, vagy egy olyan utasításig, amelyet **break** követ.
- Ha a szelektor egyetlen **case** konstanssal sem egyenlő, akkor az U_{n+1} utasítás hajtódik végre, amennyiben létezik a **default** ág.
- Ahhoz, hogy a **switch** utasítás a Pascal-beli **case**-hez hasonlóan működjön, szükséges minden ág végére a **break** utasítás.

2. megoldás (switch utasítással)

```
int jegy;
scanf("%d", &jegy);
switch(jegy)
{
    case 1 : printf("Elégtelen"); break;
    case 2 : printf("Elégséges"); break;
    case 3 : printf("Közepes"); break;
    case 4 : printf("Jó"); break;
    case 5 : printf("Jeles"); break;
    default : printf("Téves beolvasás");
}
```

Megjegyzés. Ha nem egyetlen kifejezés különböző értékeitől függ az elágazás, vagy a szelektor-kifejezés nem egész típusú, akkor nincs más választás, mint az egymásba ágyazott **if**.

Az alábbi feladat megoldása bemutatja a **switch** használatát **break** nélkül.

3.5. feladat. Írjunk programot, amely beolvas egy, az $[1, 5]$ intervallumhoz tartozó természetes számot, és kiír a számmal egyenlő számú csillag karaktert (*).

```
int szam;
scanf("%d", &szam);
switch(szam)
{
    case 5 : printf("*");
```



```

    case 4 : printf("*");
    case 3 : printf("*");
    case 2 : printf("*");
    case 1 : printf("*");
}

```

3.3.3. Kitűzött feladatok

Megjegyzés. Mivel a valós számok nem mindig tárolhatók pontosan a számítógép memóriájában, ezért a velük végzett műveletek alkalmazással számítási hibák adódhatnak. Ebből kifolyólag két valós szám egyenlőségének vizsgálatakor az $A = B$ alak helyett használjuk $|A-B| < \varepsilon$ alakot, ahol ε a pontosság, amellyel dolgozunk. Például $\varepsilon = 0.0001$ esetén az összehasonlítás három tizedes pontossággal fog történni.

1. Rajzoljuk fel logikai sémákkal az alábbi programrészleteket!

```

a)      if( F1 )
        if( F2 ) U2;
        else U1;
b)      if( F1 )
        {if( F2 ) U2; }
        else U1;

```

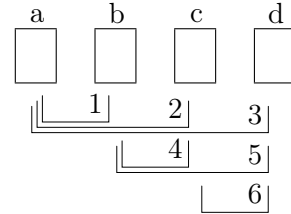
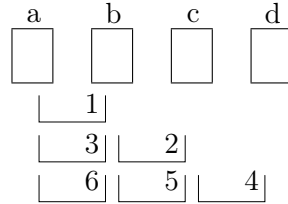
2. Írjunk programot, amely beolvas négy valós számot, és megszámlálja, hány negatív!

3. Írjunk programot, amely beolvas egy négyjegyű természetes számot, és összeadja külön a páros, illetve páratlan számjegyeit!

4. Írjunk programot, amely beolvas négy ötjegyű természetes számot, megszámlálja, melyikben található több 5-ös számjegy, és kiír egy megfelelő üzenetet! (Használjunk `long` változót a számok eltárolására.)

5. Írjunk programot, amely beolvas egy négyjegyű természetes számot, és kiszámítja a prímszámjegyeinek számtani közepét!

6. Írjunk programot, amely beolvas négy valós számot, és rendezi őket csökkenő sorrendbe az alábbi stratégiák szerint:



7. Írjunk programot, amely a koordinátaival megadott P1, P2, P3 pontokról eldönti, hogy egy egyenesen találhatók-e! A kollinearitás feltétele:

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = 0.$$

8. Írjunk programot, amely egy, a csúcspontjainak koordinátaival megadott háromszögről eldönti, hogy egyenlő szárú-e!

9. Olvassuk be egy háromszög 3 oldalát cm-ben (egy oldal leg több 255 cm lehet)! Amennyiben szerkeszthető e három adatból háromszög, számítsuk ki a területét!

10. Adott két szakasz (AB, CD) a végpontjaik koordinátái által. Döntsük el, hogy metszik-e egymást, és ha igen, állapítsuk meg, melyik pontban!

Ötlet: A metszés feltétele, hogy ne legyenek párhuzamosak, és az A, illetve B pontok legyenek a CD egyenes különböző oldalain, valamint a C, illetve D pontok az AB egyenes más-más oldalán. Két pont akkor van egy egyenes különböző oldalain, ha koordinátaikat behelyettesítve az egyenes egyenletébe, ellenkező előjelű értékeket adnak.

11. Olvassunk be egy karaktert! Írjuk ki az ASCII kódját, a következő és az előző karaktert az ASCII táblában, valamint azt, hogy a beolvasott karakter nagybetű-e vagy nem!

12. Olvassunk be egy karaktert, majd írjuk ki, hogy nagybetű, kisbetű, szám, speciális karakter vagy egyéb!

13. Olvassunk be egy dátumot: év, hó, nap. Írjuk ki, hogy ez a dátum az év hányadik napja!

Ötlet: Helyezzük egy break nélküli switch útásításba a hónapokat, ezek fordított sorrendje szerint.

14. Adott két egyenes két-két pontjuk koordinátái által. Határozzuk meg egymáshoz viszonyított helyzetüket (párhuzamosak ($m_1=m_2$), metszők, merőlegesek ($m_1m_2=-1$)), ahol m_1 és m_2 a két egyenes irányításeffektív.

15. Adottak egy kör középpontjának koordinátái és a sugara. Ellenőrizzuk egy pont, egy egyenes, illetve egy másik kör hozzá viszonyított helyzetét!

Ötlet: Kiszámítjuk a kör középpontjának távolságát a ponttól, az egyenestől és a másik kör középpontjától.

16. Ugyanaz a feladat, de a körök három pont által adottak.

Adott középpontú és sugarú kör egyenlete:

$$(X - x_0)^2 + (Y - y_0)^2 = r^2.$$

Három nem kollineáris ponton átmenő kör egyenlete:

$$\begin{vmatrix} X^2 + Y^2 & X & Y & 1 \\ x_1^2 + y_1^2 & x_1 & y_1 & 1 \\ x_2^2 + y_2^2 & x_2 & y_2 & 1 \\ x_3^2 + y_3^2 & x_3 & y_3 & 1 \end{vmatrix} = 0.$$

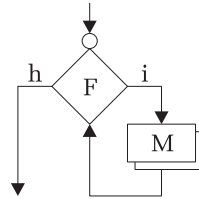
Megjegyzések:

- Csak az ismert utasításokat használjuk!
- Használjunk minimális számú változót!
- Használjuk mind a **scanf**, **printf**, mind az **fscanf**, **fprintf** függvényeket!

3.4. Ciklusutasítások

Eddigi feladatainkban többször is szembe találtuk magunkat azzal a helyzettel, hogy bizonyos utasítást vagy utasításokat többször el kellett végezni. Ahelyett, hogy ilyenkor többször leírnánk (kiadnánk) az illető utasítást (nagy ismétlésszám esetén ez nem is lenne lehetséges), élhetünk a ciklusutasítások nyújtotta lehetőséggel. Egy ciklusutasítás utasítja a számítógépet egy bizonyos utasítás többszöri végrehajtására.

A C nyelvben három ciklusutasítás létezik, amelyeket a következőkben mutatunk be.



3.2. ábra. A *while* ciklus logikai sémája. *F* – feltétel (logikai kifejezés); *M* – ciklusmag; *i* – igaz; *h* – hamis

3.4.1. A *while* ciklus (elől tesztelős ciklus)

Pszudokód

```
amíg <logikai_kifejezés> végezd
    <utasítás>
vége amíg
```

C nyelven

```
while (<feltétel>)
    < utasítás >;
```

Hogyan hajtódik végre? Amíg igaz a feltétel, ismételd a magot!

Részletezve:

- kiértékelődik a feltétel: ha igaz (F_i), végrehajtódik a ciklusmag (M),
- újra kiértékelődik a feltétel: ha igaz (F_i), ismét végrehajtódik a ciklusmag (M),
-
- újra kiértékelődik a feltétel: ha hamis (F_h), befejeződik a ciklusutasítás.

Tömören:

$$F_i \ M \ F_i \ M \ F_i \ M \dots \ F_i \ M \ F_h \quad (3.1)$$

Megjegyzések:

- Ha az F feltétel már kezdetben hamis, akkor az M mag egyszer sem hajtódik végre.
- Ha a mag több utasításból áll, C nyelven össze kell fogni őket egyetlen összetett utasítássá.
- A mag végrehajtásának hatással kell lennie a feltételre, különben a ciklus végtelen ciklus.

Példa végtelen ciklusra:

```
k = 0; i = 0;
while (i <= 10) k++;
```

```
/* attól, hogy k-t növeljük, i sosem lesz 10-nél
   nagyobb */
```

3.6. feladat. Írjunk programot, amely egész számokat olvas be a billentyűzetről nulla végjelig, és kiírja az összegüket a képernyőre.

Az összegszámítás technikája:

1. definiálunk egy változót (legyen s), amelyben az összeget fogjuk képezni, és értékét nullára állítjuk,
2. sorra olvassuk be a számokat (például egy x változóba), és mindeniket – közvetlenül a beolvasás után – hozzáadjuk az s összeghez ($s += x$).

Tehát egyetlen változót használunk, az x -et, és mindenik számot ebbe olvassuk be. Mivel a következő számot ráolvassuk az előzőre, minden beolvasás után a beolvasott értéket úgymond át kell menteni s -be: ($s += x$).

Az alábbi okfejtés logikusan próbál rávezetni arra, miként oldható meg ezen feladat egy **while** ciklus segítségével.

Mit kell csinálnia a számítógépnek, miután létrehozta az s , illetve x változókat? A számítógépnek végre kell hajtania az alábbi műveleteket:

$$I_{s=0} \text{ Be}_x \text{ T}_{x!=0} \text{ S}_x \text{ Be}_x \text{ T}_{x!=0} \text{ S}_x \text{ Be}_x \text{ T}_{x!=0} \text{ S}_x \dots \text{ Be}_x \text{ T}_{x==0} \text{ Ki}_s, \quad (3.2)$$

ahol:

$I_{s=0}$ = inicializáljuk az s változót 0-val,

Be_x = beolvassunk egy számot az x változóba,

$\text{T}_{x!=0}$ = teszteljük (ellenőrizzük) az x értékét, hogy 0-tól különböző-e,

$\text{T}_{x==0}$ = teszteljük (ellenőrizzük) az x értékét, hogy 0-val egyenlő-e,

S_x = hozzáadjuk s -hez x -et: $s += x$,

Ki_s = kiírjuk az s értékét.

Mi lesz a ciklusfeltétel? Mitől függ, hogy kell-e folytatódnia az ismétlésnek vagy sem? Az x tesztelésétől!

Tehát:

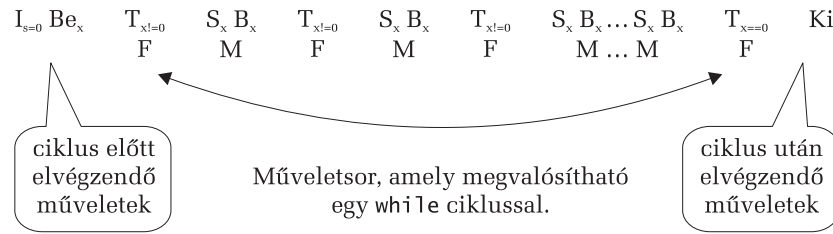
$$F \Leftrightarrow T_x$$

A (3.1) megállapítás alapján a ciklus magvát két tesztelés közötti utasítás/utasítások képezik.

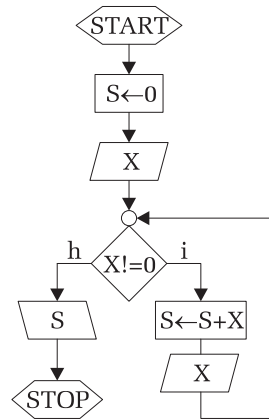
Tehát:

$$M \Leftrightarrow S_x \text{ Be}_x$$

Elvégezve a (3.2)-es műveletsorban a fenti helyettesítéseket, a következőt kapjuk:



Figyelembe véve mindezt, íme a feladat megoldása logikai sémával és C nyelven:

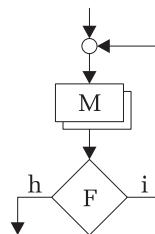


C nyelven

```

int s = 0, x;
scanf("%d", &x);
while(x)
{
    s += x;
    scanf("%d", &x);
}
printf("S=%d", s);
    
```

3.4.2. A do while ciklus (hátul tesztelős ciklus)



3.3. ábra. A do while ciklus logikai sémája. *F* – feltétel (logikai kifejezés); *M* – ciklusmag; *i* – igaz; *h* – hamis

Pszudokód

```
végezd
    <utasítás>
amíg <logikai_kifejezés>
```

C nyelven

```
do
    <utasítás>;
while (<feltétel>);
```

Hogyan hajtódik végre? Ismételd a magot, amíg igaz a feltétel!

Részletezve:

- végrehajtódik a ciklusmag (M),
- kiértékelődik a feltétel: ha igaz (F_i), ismét végrehajtódik a ciklusmag (M),
- újra kiértékelődik a feltétel: ha igaz (F_i), ismét végrehajtódik a ciklusmag (M),
-
- újra kiértékelődik a feltétel: ha hamis (F_h), befejeződik a ciklusutasítás.

Tömören:

$$M \ F_i \ M \ F_i \ M \ F_i \ \dots \ F_i \ M \ F_h \quad (3.3)$$

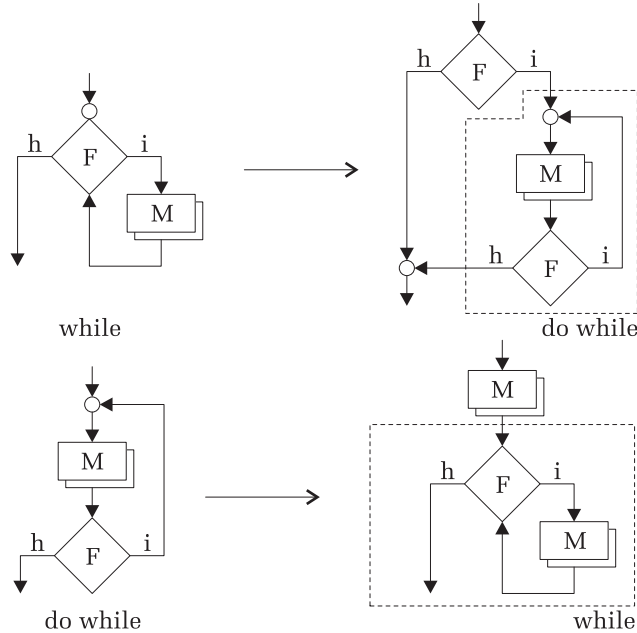
Megjegyzés.

- A mag egyszer mindenképp végrehajtódik!
- Különbözik a Pascal **repeat until** ciklustól, hiszen a ciklusból való kilépés – akárcsak a **while** ciklus esetében – *hamis* ágon történik.

3.7. feladat. Számokat olvasunk be 1-től 5-ig 0 végjelig. Írjuk ki, hogy minden egyes számnak, mint jegynek, milyen minősítés felel meg!

```
int jegy;
do
{
    scanf("%d", &jegy);
    switch(jegy)
    {
        case 1 : printf("Elégtelen"); break;
        case 2 : printf("Elégséges"); break;
        case 3 : printf("Közepes"); break;
        case 4 : printf("Jó"); break;
        case 5 : printf("Jeles"); break;
    }
while (jegy != 0);
}
```

3.8. feladat. Hogyan írható át a while do while alakba és fordítva?



3.4.3. A for ciklus (ismert lépésszámú ciklus)

Kezdjük egy feladattal:

3.9. feladat. Adott n darab természetes szám. Számoljuk meg, hány páros ezek közül!

Adott tulajdonságú elemek megszámlálásának technikája:

1. definiálunk egy k változót (számlálót), amelyet kezdetben nullára állítunk,
2. valahányszor találunk egy megfelelő tulajdonságú elemet, a számláló (változó) értékét növeljük 1-gyel.

Mit kell ismételni?

1. Beolvassunk egy számot a billentyűzetről, például egy x változóba.
2. Ellenőrizzük, hogy x páros-e, ha igen, növeljük a számlálót.

Meddig kell ismételni a fenti műveleteket? n -szer!

Hogyan valósítható meg ez while ciklussal?

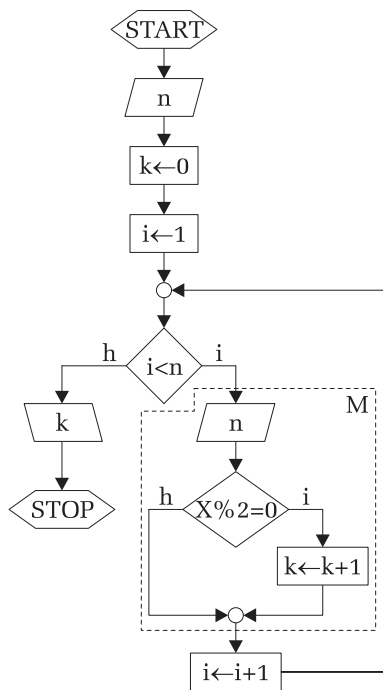
Vesszünk egy i változót, amelyet kezdetben 1-re állítunk, majd a ciklusmag minden egyes végrehajtásakor növeljük az értékét 1-gyel. Ha ciklusfeltételnek a $(i \leq n)$ logikai kifejezést választjuk, akkor a ciklusmag pontosan n -szer lesz megismételve.

3.4. CIKLUSUTASÍTÁSOK

53

C nyelven, while ciklussal

```
int n, k, i, x;
scanf("%d", &n);
k = 0;
i = 1;
while (i <= n)
{
    scanf("%d", &x);
    if (!(x%2)) k++;
    i ++;
}
printf("%d", k);
```



Az ismert lépésszámú ismétlésekre van egy speciális ciklus, a **for** ciklus.

```
i = 1;
while ( i <= n )
{
    <M>
    i++;
}
```

ekvivalens

```
for ( i = 1; i <= n; i ++ )
    <M>
```

Megjegyzés. Az *i* változót a **for** ciklus ciklusváltozójának szokás nevezni.

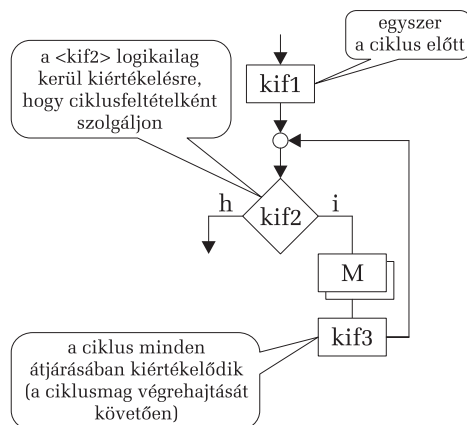
Az előbbi feladat megoldása **for** ciklussal:

```
int n, k, i, x;
scanf("%d", &n);
k = 0;
for ( i = 1; i <= n; i ++ )
{
    scanf("%d", &x);
    if(!(x % 2)) k ++;
}
printf("%d", k);
```

A **for** ciklus általános alakja:

```
for (<kif1>; <kif2>; <kif3>)
    < M >;
```

kif1, *kif2*, *kif3* kifejezések; *M* ciklusmag



3.4. ábra. Általános alakú **for** ciklus logikai sémával

Általában:

- `<kif1>` egy hozzárendelés, amely által a ciklusváltozó kezdőértéket kap.
- `<kif2>` egy összehasonlítás, amely azt vizsgálja, hogy a ciklusváltozó nem haladta-e meg a végső értéket.
- `<kif3>` a ciklusváltozó léptetésére szolgál.

Klasszikus értelemben használt **for** ciklus:

```
for(i = <kezdőérték>; i <= <végsőérték>; i++)
    < M >;
```

Az `< M >` magot $(\text{végsőérték} - \text{kezdőérték} + 1)$ -szer fogja végrehajtani!

Az alábbi ciklusok közül mindenik n -szer hajtja végre a magot:

```
for(i = 1; i <= n; i++) <M>;
for(j = 0; j < n; j++) <M>;
for(k = n; k >= 1; k--) <M>;
for(p = n-1; p >= 0; p--) <M>;
```

Amennyiben hiányoznak a `<kif1>` és `<kif3>` kifejezések, a **for** ciklus úgy működik, mint egy **while** ciklus:

```
while (<F>) <M>;    ekvivalens    for(; <F>;) <M>;
```

Végtelen ciklusok

```
while (1) <M>;
do{<M>}while (1);
for (;;) 
```

3.5. Ugró utasítások

3.5.1. A **break** utasítás

A **break** utasítás lehetővé teszi a kiugrást bármely ciklusból, illetve **switch** utasításból. A program végrehajtása a ciklus/ **switch** utáni utasítással folytatódik.

3.10. feladat. Egész számokat olvasunk be 0 végjelig. Számítsuk ki a pozitívak szorzatát, illetve a negatívak átlagát.

```
int x, k1 = 0, k2 = 0, p = 1;
float s = 0;
for (;;)
{
    scanf ("%d", &x);
```

```

    if (!x) break;
    if ( x < 0 ) { k1 ++; s += x; }
    else { k2 ++; p *= x;}
}
if (k2) printf ("%d ", p);
else printf("Nincs pozitív szám.");
if (k1) printf ("%f", s / k1);
else printf("Nincs negatív szám.");

```

Tanulság: Ha az elvégzendő műveletsor $F_i M F_i M F_i M \dots F_i M F_h$ alakú, akkor **while** ciklust célszerű használni.

Ha $M F_i M F_i M F_i \dots F_i M F_h$ alakú műveletsorra van szükségünk, akkor a **do-while** ciklus a kézenfekvő. Mi a helyzet a következő típusú műveletsor esetén: $M_1 F_i M_2 M_1 F_i M_2 M_1 F_i \dots F_i M_2 M_1 F_h$? Bár megvalósítható **while** vagy **do-while** ciklusokkal is, kényelmesebb az alábbi szerkezet:

```

végtelen-ciklus
    M1
    ha F akkor kiugrás
vége ha
    M2
vége végtelen-ciklus

```

3.5.2. A continue utasítás

A **continue** hatására a **while**, illetve **do while** ciklusok esetén megszakad a ciklusmag végrehajtása, és a ciklusfeltétel kiértékelésével folytatódik. Tehát nem ugrik ki a ciklusból, csupán az aktuális átjárásából.

A **for** ciklus esetén félbehagyja a ciklusmag aktuális átjárását, végrehajtja a **< kif3 >** kifejezés-utasítást, majd a **< kif2 >** logikai kifejezésnek mint feltételnek a kiértékelésével folytatja a ciklust.

3.11. feladat. Adott n természetes szám. Számítsuk ki a 0-tól különböző számok szorzatát.

```

int n, x, i, p = 1, k = 0;
scanf("%d", &n);
for ( i = 1; i <= n; i ++ )
{
    scanf("%d", &x);
    if (!x) continue;
    p *= x; k ++;
}

```

```
}
if (k) printf ("%d ", p);
else printf("Nincs nem nulla szám.");
```

3.5.3. A goto utasítás

Szintaxis:

goto <címke>;

Egy C program (egy adott függvényen belül; lásd 9. Fejezet) bármely utasítása megcímkézhető, és egy **goto** utasítással oda lehet ugrani, hogy az illető utasítással folytatódjon a program végrehajtása.

A címkézés szintaxisa:

<címke>: <utasítás>;

A **goto** utasítás általában kerülendő, mert használata ütközik a strukturált programozás alapelveivel, és áttekinthetetlenné teszi a programot.

Mégis bizonyos esetekben (például többszörösen egymásba ágyazott ciklusokból való kilépésre) előnyös lehet a használata.

3.12. feladat. Keressük meg 100 és 1000 között a legkisebb püthagoraszi számhármast.

```
for(a=100; a<1000; a++)
    for(b=a; b<1000; b++)
        for(c=b+1; c<=1000; c++)
        {
            if (c*c==a*a+b*b) goto cimke;
        }
cimke: printf("a=%d, b=%d, c=%d", a, b, c);
```

Figyelem! A **break** utasítás csak a legbelső ciklusból ugrott volna ki. Természetesen a ciklusváltozó határok optimalizálhatók.

3.5.4. A return utasítás

Szintaxisa:

return [<kifejezés>;

A **return** lehetővé teszi a visszatérést egy függvényből (9. fejezet).

Végül lássunk egy olyan példát, amelyben üres utasítást használunk. Amint említettük, általában olyan ciklusok esetén használjuk, amelyeknek hiányzik a magjuk.

```
int i;
for ( i = 0; i < 10; printf ("%d ", i++));
```

A fenti programrészlet kiírja a számjegyeket 0-tól 9-ig.

3.6. Megoldott feladatok

(A programkódokat körülvevő keretek bal oldalán szemléletesen ábrázoltuk az algoritmusok ciklusvázát.)

1. Írjunk programot, amely beolvas két természetes számot, és kiírja az első szám számjegyeinek az összegét és a második szám számjegyeinek a szorzatát.

A program két egymás utáni ciklust tartalmaz, tehát ciklusváza az alábbi módon ábrázolható:

```
int x, y, s = 0, p = 1;
scanf("%d", &x);
while (x)
{
    s += x % 10 ;
    x /= 10 ;
}
scanf("%d", &y);
while (y)
{
    p *= y % 10 ;
    y /= 10 ;
}
printf("S=%d\nP=%d", s, p);
```

2. Írjunk programot, amely véletlenszerűen generál egész számokat a $[-20, 20]$ intervallumból, amíg eltalálja a nullát, majd írjuk ki a generált prímek összegét.

Megjegyzés. A `rand()` függvény véletlenszerűen generál egy természetes számot a $[0, \text{RAND_MAX})$ intervallumból. A feladat megoldása két egymásba ágyazott ciklust feltételez.

```
int x, s = 0, prim, i;
srand(time(0)); /* inicializálja a
                  véletlenszám generátort */
for( x = rand() % 41 - 20; x; x = rand() % 41 - 20)
{
    if (x < 0) x = -x;
    if (!x || x==1) prim = 0;
    else
    {
        prim = 1;
        for(i = 2; i <= sqrt(x); i++)
            if (x % i == 2) {prim = 0; break;}
        if (prim == 1) s += x;
    }
}
printf("A prímelek összege=%d", s);
```

Magyarázat: Ha a szám negatív, akkor képezzük az abszolút-értékét. Az algoritmusra nézve rendhagyó esetek amikor az x értéke 0 vagy 1 (ezeket nem tekintjük prímeleknek). Egy $x > 1$ egész szám akkor prím, ha nincs osztója a $2, 3, \dots, [\sqrt{x}]$ halmazban.

Tanulság: A véletlenszám-generálás feltételezi a `# include <time.h>`, illetve az `srand(time(0));` programsorokat.

3. Írjunk programot, amely generálja és kiírja az alábbi számsorozat első n tagját:

1, 1, 2, 2, 1, 2, 3, 3, 3, 1, 2, 3, 4, 4, 4, 4, ...

```
int i, j, n, k = 0;
scanf("%d", &n);
for( i = 1; i <= n; i++)
{
    for( j = 1; j <= i; j++)
    {
        printf("%d, ", j);
        k ++; if (k == n) goto vege;
    }
    for( j = 1; j <= i - 1; j++)
    {
        printf("%d, ", i);
        k ++; if (k == n) goto vege;
    }
}
vege;
```

Magyarázat: Csoportosítjuk a számsorozat elemeit: az i -edik csoportban az első i természetes számot követi $(i - 1)$ -szer az i érték.

A külső ciklus csoportról-csoportra lépegető végtelen ciklus.

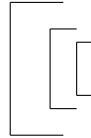
Az első belső ciklus előállítja az $1, 2, \dots, i$ részsorozatot.

A második belső ciklus kiír $(i - 1)$ darab i számot.

A k változó számolja a kiírt értékeket. Ha k eléri n -et a **goto** utasítás segítségével befejezzük a számgenerálást.

4. Írjunk programot, amely beolvas n természetes számot, és addig „csomagolja” mindeniket önmagába, míg egyszámjegyűek lesznek. Írjuk ki az így kapott csomagokat. Csomagoláson azt értjük, hogy a számot helyettesítjük a számjegyeinek összegével.

Ciklusváza:



Top Down módszer: fokozatosan finomítjuk a megoldást.

```
int i, n, x, s;
scanf("%d", &n);
for( i=1; i<=n; i++)
{
    scanf("%d", &x);
    * Magába csomagoljuk x-et,
    amíg egyszámjegyű lesz. *
    printf("A csomag:%d\n",x);
}
```

```
/* Magába csomagoljuk x-et,
míg egyszámjegyű lesz */

while (x > 10)
{
    * Számítsuk ki s-ben x
    számjegyeinek összegét. *
    x = s;
}
```



```
/* Számítsuk ki s-ben x
számjegyeinek összegét. */

s = 0;
while (x > 0)
{
    s += x % 10;
    x /= 10;
}
```

A megoldás egy programban:

```
C nyelven

int i, n, x, s;
scanf("%d", &n);
for( i = 1; i <= n; i++)
{
    scanf("%d", &x);
    while (x > 10)
    {
        s = 0;
        while (x > 0)
        {
            s += x % 10;
            x /= 10;
        }
        x = s;
    }
    printf("A csomag:%d\n",x);
}
```

5. Írjunk programot amely előállítja a 0, 1, 1, 2, 3, 5, 8, 13... úgynevezett Fibonacci számsorozatot. ($f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$, ha $n > 1$).

Ötlet: Használunk három változót:

ue - az utolsó előtti előállított Fibonacci szám. (kezdetben 0)

u - az utolsó előállított Fibonacci szám. (kezdetben 1)

uj - a következő Fibonacci szám. ($uj = u + ue$)

Minden lépésben frissítjük az **u** és **ue** változók értékeit (**ue** = **u**, az utolsóból lesz utolsó előtti; **u** = **uj**, az újonnan előállítottból utolsó lesz)

```
int u, ue, uj, i, n;
```

```
scanf("%d", &n);
ue = 0; printf("0 "); // a nulladik Fibonacci szám
u = 1; printf("1 "); // az első Fibonacci szám
for(i=2; i<=n; i++)
{
    uj = u + ue;
    printf("%d ", uj);
    ue = u;
    u = uj;
}
```

Megjegyzés. A megoldott feladatokból arra lehet következtetni, hogy a feladatok különböző szempontok szerint oszthatók: például a ciklusvázuk alapján, vagy aszerint, hogy honnan származnak az adataik (billejtűzetről vagy állományból, illetve a gép generálja egy szabály szerint vagy véletlenszerűen). Hasznos lehet besorolni a feladatot bizonyos feladatosztályokba, mielőtt hozzálátunk a megoldásához.

3.7. Elemi algoritmusok összefoglalása

Az alábbiakban pszeudokódnnyelven mutatunk be néhány alapalgoritmust. Azon olvasók számára, akik először találkoznak pszeudokód programrészletekkel, szolgáljanak magyarázatul a következők:

| | |
|--------------------------------------|-----------------------------------|
| minden ciklus | – for ciklus |
| amíg ciklus | – while ciklus |
| ha ... akkor ... különben ... | – if ... then ... else ... |
| MOD | – osztási maradék |
| DIV | – osztási hányados |
| [...] | – egész rész |
| <> | – nem egyenlő |

3.7.1. Összeg- és szorzatszámítás

Egy n elemű számsorozat elemeinek összege:

Algoritmus összeg
 összeg = 0
 beolvas n
 minden $i = 1, n$ végezd
 beolvas szám

```
    összeg = összeg + szám
```

```
  vége minden
```

```
  kiír összeg
```

```
vége algoritmus
```

Egy természetes szám számjegyeinek szorzata:

Algoritmus számjegyekszorzata

```
  szorzat = 1
```

```
  beolvas szám
```

```
  amíg szám > 0 végezd
```

```
    szorzat = szorzat * (szám MOD 10)
```

```
    szám = szám DIV 10
```

```
  vége amíg
```

```
  kiír szorzat
```

```
vége algoritmus
```

3.7.2. Adott tulajdonságú elemek megszámlálása

Egy n elemű valós számsorozat egész értékű elemeinek megszámlálása:

Algoritmus egészszámlálás

```
  számláló = 0
```

```
  beolvas n
```

```
  minden i = 1, n végezd
```

```
    beolvas szám
```

```
    ha szám = [szám] akkor
```

```
      számláló = számláló + 1
```

```
    vége ha
```

```
  vége minden
```

```
  kiír számláló
```

```
vége algoritmus
```

3.7.3. Átlagszámítás

Nulla végjelű számsorozat páros elemeinek átlaga:

Algoritmus párosokátlaga

```
  számláló = 0
```

```
  összeg = 0
```

```
  beolvas szám
```

```
  amíg szám <> 0 végezd
```

```
    ha szám MOD 2 = 0 akkor
```

```

        összeg = összeg + szám
        számláló = számláló + 1
    vége ha
    beolvas szám
    vége amíg
    ha számláló > 0 akkor
        kiír összeg/számláló
    különben
        kiír "Nincs páros szám."
    vége algoritmus

```

3.7.4. Adott tulajdonságú elem létezésének ellenőrzése

Ellenőrizzük, hogy van-e teljes négyzet egy n elemű számsorozatban!

Ötlet: A következőképpen ellenőrizzük, hogy teljes négyzet-e egy szám: indulva nullától, addig lépegetünk teljes négyzetről teljes négyzetre, míg vagy eltaláljuk, vagy átlépjük az illető számot.

Algoritmus négyzetellenőrzés

```

    kém = 0
    beolvas n
    minden i = 1,n végezd
        beolvas szám
        j = 0
        amíg j * j < szám végezd
            j = j + 1
        vége amíg
        ha j * j = szám akkor
            kém = 1
        vége ha
    vége minden
    ha kém = 0 akkor
        kiír "Nincs teljes négyzet."
    különben
        kiír "Van teljes négyzet."
    vége algoritmus

```

3.7.5. Adott tulajdonságú elem megkeresése

Nulla végjelű számsorozatban keressünk meg egy adott számot!

```

Algoritmus keresés1
  beolvas keresett
  beolvas szám
  sorszám = 1
  amíg szám <> 0 és szám <> keresett végezd
    beolvas szám
    sorszám = sorszám + 1
  vége amíg
  ha szám <> 0 akkor
    kiír sorszám
  különben
    kiír "A ", keresett, " nem található meg."
  vége ha
vége algoritmus

```

Megjegyzés. Ha többször is előfordul a keresett szám a számsorozatban, akkor az első előfordulás helyét adja meg. Ha valamennyi előfordulás helyét ki szeretnénk íratni, akkor az algoritmus a következő:

```

Algoritmus keresés2
  beolvas keresett
  beolvas szám
  sorszám = 0
  amíg szám <> 0 végezd
    beolvas szám
    sorszám = sorszám + 1
    ha szám = keresett akkor
      kiír sorszám
    vége ha
  vége amíg
  ha sorszám = 0 akkor
    kiír "A ", keresett, " nem található meg."
  vége ha
vége algoritmus

```

3.7.6. Legnagyobb és legkisebb elem értékének meghatározása

Egy n elemű számsorozat maximumának meghatározása.

Ötlet: Kijelölöm maximumnak az első számot, majd valahányszor érkezik egy nagyobb érték, frissítem a max változó tartalmát.

Algoritmus legnagyobbérték

```

beolvas n
beolvas szám
max = szám
minden i = 2,n végezd
    beolvas szám
    ha szám > max akkor
        max = szám
    vége ha
vége minden
kiír max
vége algoritmus
    
```

3.7.7. Legnagyobb és legkisebb elem pozíciójának meghatározása

Egy természetes szám legkisebb számjegyének pozíciója (balról jobbra):

```

Algoritmus LegkisebbSzámjegyHelye
számláló = 0
beolvas szám
min = 10
amíg szám > 0 végezd
    számjegy = szám MOD 10
    ha számjegy < min akkor
        min = számjegy
        minpozíció = számláló
    vége ha
    szám = szám DIV 10
    számláló = számláló + 1
vége amíg
kiír számláló - minpozíció + 1
vége algoritmus
    
```

3.7.8. Legnagyobb közös osztó meghatározása

Addig vonjuk ki a nagyobb számból a kisebbiket, amíg a két szám egyenlő nem lesz. Ez az érték lesz a két szám lnko-ja.

```

Algoritmus LegNagyobbKözösOsztó
    beolvas szám1, szám2
    amíg szám1 <> szám2 végezd
    
```

```

    ha szám1 > szám2 akkor
        szám1 = szám1 - szám2
    különben
        szám2 = szám2 - szám1
    vége ha
    vége amíg
    kiír szám1
vége algoritmus

```

3.7.9. Legkisebb közös többszörös meghatározása

Képezünk két összeget, az összeg1-et szám1 értékekből, az összeg2 pedig szám2 értékekből. Addig adjuk hozzá a kisebb összeghez újra és újra a megfelelő számot, amíg a két összeg egyenlő nem lesz. Ez az érték lesz a két szám lkkt-je. (Szemléltetésül: a legkisebb közös többszörös az a legkisebb érték, amelyen magas épület úgy szám1, mint szám2 méretű téglákból felépíthető; amíg a két épület nem egyforma a kisebbikre teszünk egy megfelelő téglát)

Algoritmus LegKisebbKözösTöbbszörös

```

    beolvas szám1, szám2
    összeg1 = szám1
    összeg2 = szám2
    amíg összeg1 <> összeg2 végezd
        ha összeg1 < összeg2 akkor
            összeg1 = összeg1 + szám1
        különben
            összeg2 = összeg2 + szám2
    vége ha
    vége amíg
    kiír összeg1
vége algoritmus

```

3.7.10. Egy szám tükrözése

Tekintjük a szám számjegyeit fordított sorrendben és felépítünk belőlük egy új számot, a tükörszámot.

Algoritmus Tükrözés

```

    beolvas szám
    tükörszám = 0

```

```

    amíg szám > 0 végezd
        tükörszám = tükörszám * 10 + (szám MOD 10)
        szám = szám DIV 10
    vége amíg
    kiír tükörszám
vége algoritmus

```

3.8. Kitűzött feladatok

1. Generáljuk a következő számsorozatok esetében az első n elemet, és egy másik megoldással a sorozat n -edik elemét:

- 1, 2, 3, 4, 5, ...
- 1, 3, 5, 7, ...
- 0, 2, 4, 6, 8, ...
- 2, 3, 5, 7, 11, 13, 17, 19, ...
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, ...
- 1, 2, 1, 3, 2, 1, 4, 3, 2, 1, ...
- 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...
- 2, 3, 3, 4, 4, 4, 5, 5, 5, 5, ...
- 0, 1, 2, 1, 2, 2, 3, 2, 3, 3, 3, 4, 3, 4, 4, 4, 4, 5, ...
- 1, 1, 2, 2, 1, 2, 3, 3, 3, 1, 2, 3, 4, 4, 4, 4, ...

2. Számítsuk ki a következő kifejezések értékét:

- $E1 = 1 + 2 + 3 + \dots + n$
- $E2 = -1 + 2 - 3 + 4 - 5 + 6 + \dots + (-1)^n \cdot n$
- $E3 = 1 \cdot 2 + 3 \cdot 4 + 5 \cdot 6 + \dots + (2n - 1) \cdot 2n$
- $E4 = 1/(1 + 2) \cdot 2/(2 + 3) \cdot 3/(3 + 4) \cdot \dots \cdot n/[n + (n + 1)]$
- $E5 = -2 + 3 - 5 + 7 - 11 + 13 - 17 + 19 - \dots$ (n -edik tag)
- $E6 = -1/1 + (1 \cdot 2)/(1 + 2) - (1 \cdot 2 \cdot 3)/(1 + 2 + 3) + \dots + (-1)^n (1 \cdot 2 \cdot 3 \cdot \dots \cdot n)/(1 + 2 + 3 + \dots + n)$
- $E7 = 1 + 2 + 3 - 4 + 5 - 6 - 7 + 8 - 9 - 10 - 11 - 12 + 13 - 14 \dots$ (n -szer)
- $E9 = 1/2 + 4/3 + 9/5 + 16/7 + 25/11 + 36/13 + \dots$ (n -edik tag)

3. Egy számsorozat elemein végezzük el az alábbi feladatokat:

Összeg-, átlag-, szorzatszámítás, számoljuk meg a párosok, illetve páratlanok számát, ellenőrizzük, hogy tartalmaz-e teljes négyzetet, állapítsuk meg a minimum értékét, az első minimum pozícióját, az utolsó minimum pozícióját, számítsuk ki a párosok médiáját, a páratlanok maximumát, a prímek számát!

3.8. KITŰZÖTT FELADATOK

69

A számsorozat elemeit állítsuk a program rendelkezésére:

- Véletlenszám-generátorral n darab 100-nál kisebb természetes számot!
- Véletlenszám-generátorral egész számokat a $[-10, 10]$ intervallumból, amíg eltaláljuk a 0-t!
- Állományból olvasva egy n elemű egész számsorozatot!
- Egész számokat olvasva a billentyűzetről 0 végjelig!

4. Olvassunk be egy 0-val végződő, egészekből álló számsorozatot! Írjuk ki a harmadik legnagyobb elemét és annak sorszámát is! Valamely feltétel nem teljesülése esetén (például, ha 3-nál kevesebb elemünk van) a program adjon hibajelzést.

5. Mit valósít meg az alábbi programrészlet? Írd át úgy, hogy ugyanezt valósítsa meg, de `do-while` ciklussal! A feladatot papíron oldjuk meg!

```
int x, k = 0;
scanf("%d", &x);
while(x) k++;
printf("%d", k);
```

6. Melyek a végtelen ciklusok? A feladatot papíron oldjuk meg!

- `i = 10; while(i-);`
- `while(i = 10)i-;`
- `for(x = 1; x = 10; x++);`
- `for(x = 1; x == 10; x++);`

7. Írjunk programot, amely kiszámítja és másodpercenkénti bontásban táblázatosan kiírja a v_0 kezdősebességű, a gyorsulással egyenletesen gyorsuló mozgást végző test által t idő alatt megtett utat! A v_0 , a és t bemenő adatok.

8. Írjunk programot, amely a Pascal-háromszög első m sorát állítja elő (m bemenő adat)! A Pascal-háromszög egyes elemei az úgynevezett *binomiális együtthatók*. Az n -edik sor k -adik eleme:

$$C(n, k) = n! / ((n-k)! \cdot k!) = (n \cdot (n-1) \cdot (n-2) \dots (n-k+1)) / (1 \cdot 2 \cdot 3 \dots k).$$

9. Egy k egész számot *majdnemprím*nek nevezzük, ha nem prím, de ugyanakkor két prím szorzata. Ha k és $k + 1$ is majdnemprímek, akkor *iker majdnemprímek*. Írjunk programot, amely 100-ig megkeresi az iker majdnemprímeket!

10. Írjunk programot, amely kiszámítja és kiírja az olyan püthagoraszai számhármasságokat, melyeknek egyik tagja sem nagyobb 50-nél!

11. Ábrázoljuk a képernyőn az $F(x) = (\sin x)/x$ függvényt a $[-2\pi, 2\pi]$ intervallumon úgy, hogy a függőlegesen futó x tengely mentén a megfelelő koordinátájú pontokba egy „*” karaktert helyezünk! Rajzoljuk ki az x tengelyt is!

12. Olvassuk be egy sakkfigura (vezér, király, bástya, futó, huszár, gyalog) aktuális koordinátáit a sakktáblán! Írjunk programot, amely kiírja a sakkfigura által támadható mezőket!

13. Írjunk programot, amely for ciklussal számítja ki az m^n értékét, ahol az m egy valós szám, az n egész típusú!

14. Írjuk ki azokat az 500-nál kisebb, legalább kétjegyű páros számokat, amelyekben a tízesek helyén páratlan szám áll!

15. Adott egy kör a síkban középpontjának és sugarának koordinátái által. Írjunk programot, amely megszámlálja, hogy hány darab egész koordinátával jellemezhető koordinátapont esik a körön belülre!

16. Olvassunk be tanulmányi átlagokat, és határozzuk meg a megfelelő minősítéseket: kitűnő $[9,50-10]$, jeles $[9-9,50)$, jó $[8-9)$, közepes $[7-8)$, elégséges $[4,50-7)$, elégtelen $[1-4,50)$.

Megjegyzés. A 17–20 feladatokban használjuk ki a `scanf` és `printf` függvények formázó karakterei nyújtotta lehetőségeket.

17. Olvassunk be decimális számokat nulla végjelig, és alakítsuk oktálissá (8-as számrendszer) őket!

18. Olvassunk be decimális számokat nulla végjelig, és alakítsuk hexadecimálissá őket!

19. Olvassunk be hexadecimális számokat nulla végjelig, és alakítsuk decimálissá őket!

20. Olvassunk be oktális számokat nulla végjelig, és alakítsuk decimálissá őket!

21. Olvassunk be bináris számokat nulla végjelig, és alakítsuk decimálissá őket!

22. Olvassunk be 1000-nél kisebb decimális számokat nulla végjelig, és alakítsuk binárisra őket!

23. Olvassunk be természetes számokat a billentyűzetről, és számoljuk ki az átlagukat! A számsor végét két egymás utáni 0 jelezze!

24. Írjunk programot, amely beolvas 20 valós számot, és megállapítja a leghosszabb szigorúan növekvő összefüggő részsorozat összegét!

25. Írjunk programot, amely beolvas n egész számot, és kiírja a szomszédos számok különbségét!

26. Adott egy természetes szám. Határozzuk meg a szám számjegyeinek szorzatát, összegét, átlagát, a legkisebbet, a legnagyobbat, valamint a páros, a páratlan és a prímszámjegyek számát.

27. Oldjuk meg a 42. feladatot n darab egész szám esetében!

28. Oldjuk meg a 42. feladatot n darab véletlenszerűen generált egész számra a (200, 3000) intervallumból!

29. Adott egy természetes szám, képezzük a fordított számot!

30. Hány olyan háromjegyű egész szám van, amely prím, és amelynek a fordítottja is prím?

31. Adott egy természetes szám, képezzük azt a számot, amelyet úgy kapunk, hogy felcseréljük az első és utolsó számjegyeit.

32. Hány olyan háromjegyű egész szám van, amely prím és a belőle az előbbi módon képzett szám is prím?

33. Adott egy n elemű számsorozat. Számoljuk meg, hogy a 2-es számrendszerbeli alakjaikban számonként és összesen hány 1-es és hány 0 van!

34. Olvassunk be pozitív egész számokat hexadecimális alakban 0 végjelig, és állapítsuk meg a számsorozat rendezettségét (egyenlő elemekből áll-e, növekvő, csökkenő vagy rendezetlen)! A növekvő, illetve a csökkenő rendezettségénél az egyenlő számokat is megengedjük. Ha a sorozatról menet közben kiderül, hogy rendezetlen, fejezzük be a bevitelt!

35. Írjuk ki az 1 m^3 -nél kisebb térfogatú, 10 cm -enként növekvő sugarú gömbök térfogatát!

36. Írjuk ki az angol ábécé összes nagybetűjét növekvő, majd csökkenő sorrendben!

37. Olvassunk be egy dátumot: év, hó, nap! Írjuk ki, hogy ez a dátum az év hányadik napja!

38. Rajzoljunk adott méretű X-eket a képernyőre! A méreteket a rajzolás előtt olvassuk be, és amikor 0 méretet olvastunk, a program befejeződik.

39. Tegyük a képernyő közepére egy jelet, majd fel, le, balra, jobbra nyilak segítségével mozgassuk a képernyőn! A jelet a képernyőről ne engedjük kimenni, s az kezdetben húzzon maga után vonalat! Az <insert> gomb hatására, ha eddig volt vonalhúzás, ne legyen, ha nem volt, legyen, <enter>-re a program fejeződjön be.

40. Írjunk programot, amely 0 végjelig olvas be számokat! A bevitt számot csak akkor fogadjuk el, ha az előző számtól való eltérés (a két szám különbsége abszolút értékben) annak 20%-ánál nem nagyobb.

41. Generáljunk véletlenszám-generátorral háromjegyű számokat, amíg olyat találunk el, amelynek számjegyei csökkenő sorrendben vannak! Írjuk ki sorszámozva a generált számokat!

42. Generáljunk véletlenszám-generátorral számokat az `int` tartományból, amíg olyat találunk el, amelynek számjegyei tető alakot írnak le (egy pontig növekednek, majd csökkennek)! Írjuk ki sorszámozva a generált számokat!

43. Adott egy n csúcspontú sokszög a csúcsai (az óra járásának sorrendjében) koordinátái által. Írjuk ki a sokszög területét!

$$S_{\text{poligon}} = \pm \frac{1}{2} \cdot \sum_{i=1}^n \begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix},$$

ahol jelölés szerint a képzeletbeli $n+1$ -edik pont koordinátái: (x_1, x_2) .

44. Adott n esetén határozzuk meg az n -nél nagyobb legkisebb prím értékét.

45. Határozzuk meg az $n1$ és $n2$ természetes számok közé eső iker-prímeket. (p és q iker-prímek, ha prímek, és $p - q = 2$, amennyiben $p > q$)

46. Számítsuk ki egy személy korát napokban kifejezve. (Adott a születési dátuma és az aktuális dátum)

47. Adott n és m természetes számok. Írjuk ki az alábbi számsorozat n egymásutáni elemét, az m -edik elemmel kezdődően.

1, 2, 3, 2, 5, 2, 3, 7, 2, 4, 3, 2, 5, 11, 2, 3, 4, 6, 13, ...

(Minden összetett számot helyettesítünk a saját osztóival)

48. Adott n és m természetes számok. Írjuk ki az alábbi számsorozat n egymásutáni elemét, az m -edik elemmel kezdődően.

1, 2, 3, 4, 2, 5, 6, 2, 3, 7, 8, 2, 9, 3, 10, 2, 5, 11, ...

(Minden összetett szám után beszúrjuk a prím osztóival)

49. Adott n és m természetes számok. Írjuk ki azt az $n \times n$ méretű mátrixot, amelynek elemei (soronkénti bejárás szerint) azonosak az alábbi számsorozat $n2$ egymásutáni elemével, az m -edik elemmel kezdődően.

1, 2, 2, 3, 2, 3, 2, 3, 3, 3, 2, 3, 2, 3, 3, 3, 2, ...

(Minden prím számot 2-essel és minden összetett számot 3-assal helyettesítettünk)

50. Adott n és m természetes számok. Írjuk ki azt az $n \times n$ méretű mátrixot, amelynek elemei (soronkénti bejárás szerint) azonosak az alábbi számsorozat első $n2$ egymásutáni elemével.

1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 0, 1, 1, 1, 2, 1, 3, 1, 4, ...

(Minden számot helyettesítettünk a számjegyeivel)

51. Adott n és m természetes számok. Írjuk ki azt az $n \times n$ méretű mátrixot, amelynek elemei (soronkénti bejárás szerint) azonosak az alábbi számsorozat $n2$ egymásutáni elemével, az m -edik elemmel kezdődően.

1, 2, 3, 4, 2, 5, 6, 2, 3, 7, 8, 2, 4, 9, 3, 10, 2, 5, 11, ...

(Minden összetett szám után beszúrtuk a saját osztóit)

52. Adott n természetes szám p_1, p_2, \dots, p_n számrendszerekbenli ábrázolása. Határozzuk meg a legnagyobb számot, illetve a számok összegét q alapú számrendszerben.

4. FEJEZET

KIFEJEZÉSEK

Egy kifejezés *operátorokat* és *operandusokat* tartalmaz. Egy operátor azt írja elő, hogy milyen műveletet kell elvégezni a megfelelő operandusokon. A műveletek végrehajtásának a sorrendjét az operátorok *prioritása* (elsőbbsége, precedenciája) adja meg. Azonos prioritású operátorok esetén általában balról jobbra hajtódnak végre a műveletek, de léteznek olyan operátorok is, amelyek esetében jobbról balra. Az implicit sorrendet felül lehet bírálni – akárcsak a matematikában – zárójelezéssel (csak kerek zárójelek használhatók).

A 4.1. táblázat a C nyelv operátorait tartalmazza prioritásuk szerint csökkenő sorrendben. Egyesekkel már találkoztunk, másokkal ebben a fejezetben ismerkedünk meg, a többivel pedig későbbi fejezetekben.

4.1. táblázat. *Prioritástáblázat*

| Operátorok | Sorrend azonos prioritás esetén |
|-----------------------------------|---------------------------------|
| () [] -> . | balról jobbra |
| ! ~ ++ -- - (típus) * & 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 |

Alapelvnek számít, hogy a két operandusú operátorok operandusainak azonos típusúaknak kell lenniük ahhoz, hogy a művelet végrehajtsódjon. Ez a típus lesz az eredmény típusa is. És ha nem azonosak? Implicit típuskonverzióra kerül sor valamelyiknél, vagy éppenséggel mindkettőnél, az alábbi szabályok szerint:

1. *Aritmetikai* konverzió

- a) Első lépés: minden rövid típus (**char**, **short**) **int** típusúvá konvertálódik.
- b) Második lépés: a kisebb értéktartományú típus igazodik a nagyobb értéktartományúhoz.

**int => unsigned => long => unsigned long => float =>
double => long double**

Figyelem! A **signed char** **int** típusúvá konvertálásakor előjel-kiterjesztés történik.

Például, ha egy **char** változó (amely általában **signed**) értéke **-2** (**11111110**), akkor miután típusa **int** típusúvá konvertálódott, a belső ábrázolása (**1111111111111110**), az értéke pedig ugyancsak **-2** lesz (feltételeztük, hogy az **int** típus 2 byte-on van tárolva).

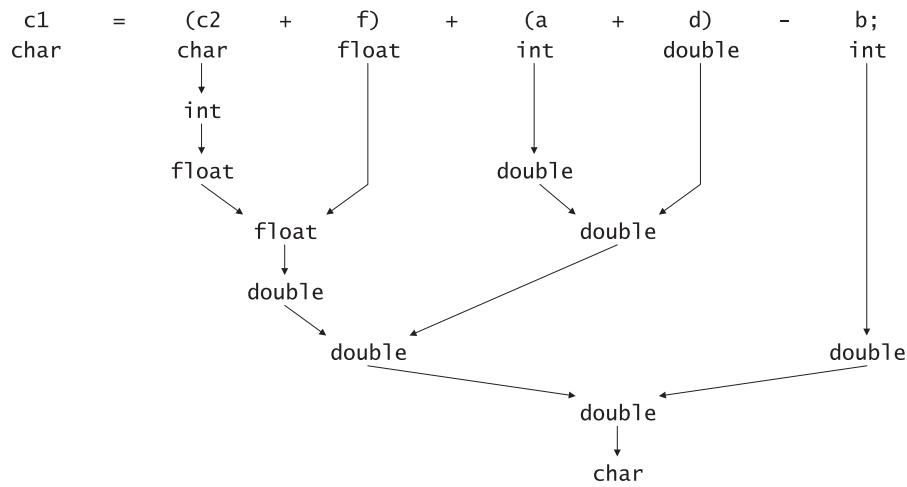
2. *Értékadás* esetén mindig a jobb oldal igazodik a bal oldalhoz.

Figyelem! Ha a jobb oldal nagyobb értéktartományú típusú, akkor az értékadás információvesztéssel járhat.

Példa:

```
int a = 1, b = 2;
char c1, c2 = 15;
float f = 1;
double d = 3.14;
```

Mi lesz az alábbi kifejezés típusa, hát a **c1** változó értéke?



Tehát a kifejezés típusa `char` lesz, a `c1` értéke pedig 64.

4.1. A kifejezések jobb (Rvalue) és bal (Lvalue) értéke

Egy $\langle V \rangle = \langle kifejezés \rangle;$ hozzárendelés esetén a $\langle V \rangle$ változónak a címével dolgozunk, a $\langle kifejezés \rangle$ -nek viszont az értékével. A $\langle V \rangle$ változó címére eltároljuk a kifejezés értékét. Tehát egy hozzárendelés bal oldalán csakis olyan kifejezés jelenhet meg, amely egy konkrét memóriacímen létező objektumra vonatkozik. Az ilyen kifejezésre azt mondjuk, hogy van *bal értéke*. Ilyen kifejezések a változók. Azon kifejezésekről, amelyek egy hozzárendelésnek csakis a jobb oldalán szerepelhetnek, azt mondjuk, hogy csak *jobb értékük* van. A változóknak például van bal értékük is (a címük) és jobb értékük is (az értékük).

Példa:

```
int a = 1, b = 2, c = 3;
a = a + b; /* Helyes, mert az a változónak van bal értéke */
a + b = a; /* Helytelen, mert az a + b kifejezésnek nincs bal
           értéke */
c = a; /* Helyes, mert a c változónak van bal értéke */
```

Az `a + b` kifejezésnek azért nincs bal értéke, mert nem olyan objektum, ahova el lehetne tárolni az értékadás jobb oldalán lévő `a` változó értékét. Tehát nem szerepelhet egy értékadás bal oldalán, a jobb oldalán viszont igen.

A kifejezésekben szereplő operandusok alapvetően változók, konstansok (állandók) és függvények lehetnek. A változókról már beszéltünk, a függvényekről később lesz szó, következzenek hát a konstansok.

4.2. Konstansok

Ahogy erre a nevük is utal, a konstansok értéke – a változókkal ellentétben – nem változtatható meg a program futása közben.

4.2.1. Egész típusú konstansok

Az egész típusú konstansok megjelenhetnek programjainkban 10-es (decimális), 8-as (oktál) vagy 16-os (hexadecimális) számrendszerben.

Példák:

0, 7, 19, 25432 (10-es számrendszerben)

0, 013, 0257 (8-as számrendszerben)

0x1a, 0x234, 0xAB2F (16-os számrendszerben)

Mivel 65 nyolcas (oktál) számrendszerben 101, tizenhatos (hexa) számrendszerben pedig 41, az alábbi egész konstansok azonosak:

65 \iff 0101 \iff 0X41

Tehát azzal jelezzük, hogy egy konstans nyolcas, illetve tizenhatos számrendszerben van, hogy 0-t, illetve 0x-et vagy 0X-et írunk eléje.

Megjegyzés. Nincs negatív egész konstans a C-ben. A negatív számok a *–* (*mínusz*) operátornak előjelként való alkalmazásával képezhetők.

Például az `int x = -1;` definícióban a mínuszjel operátor, az `1`-es pedig egész konstans.

4.2.1.1. Az egész konstansok típusai

Minden egész konstans, amely belefér az `int` tartományba, `int` típusú. Amelyek nem férnek bele az `int` tartományba, de beleférnek az `unsigned int` tartományba, azok `unsigned int` típusúak. Amelyek nem férnek bele az `unsigned int` tartományba, de beleférnek a `long int` tartományba, azok `long int` típusúak. Az ennél is nagyobbak nyilván `unsigned long int` típusúak.

A fenti szabály szerinti implicit típus felülbíráható explicit módon, úgynevezett *típuszuffixekkel*. A szuffixek jelentése:

u vagy U – unsigned int
 l vagy L – long int
 ul vagy UL – unsigned long int

Példa:

1 – int típusú 1-es
 1U – unsigned int típusú 1-es
 1L – long int típusú 1-es
 1UL – unsigned long int típusú 1-es

4.2.2. Valós típusú konstansok

Minden valós konstans implicit **double** típusú, de ez felülbírálható explicit módon **F** (float), illetve **L** (long) szuffixekkel.

Példák:

3.14, -17.65, 1., 1.0, 0.1, .1, -0.025e-1, -12.6E2 (double konstansok)

-1.F (float konstans)

0.01E+5L (long double konstans)

Megjegyzés. A 0.01E+5L értéke $0.01 * 10^5 = 1000.0L$.

4.2.3. Karakter típusú konstansok

Különböző módokon hivatkozhatunk a karakter típusú konstansokra is.

1. A megjeleníthető karakterekre úgy hivatkozunk, hogy aposztrófok közé írjuk: 'A', 'a', '0', '9', '!', '+', ' '
2. Mind a 256 karakterre lehet hivatkozni az ASCII kódja¹ által a következőképpen: '\ooo', vagy '\xhh', ahol ooo és hh a karakter ASCII kódja nyolcas, illetve tizenhatos számrendszerben.
3. Néhány fontosabb nem megjeleníthető karakterre úgynevezett *escape szekvenciák* által is hivatkozhatunk:

¹ A karakterkészlet karakterei sorszámozva vannak 0-tól 255-ig (ASCII karaktertábla). Ezt a táblázatbeli sorszámot nevezzük a karakterek ASCII kódjának. Hasznos megjegyezni néhány gyakran használt karakter kódját.

Például:

'0'..'9' – 48..57

'A'..'Z' – 65..90

'a'..'z' – 97..122

| | | |
|-------------------|----------|----------------------|
| <code>'\a'</code> | BEL | csengő |
| <code>'\b'</code> | BS | visszalépés |
| <code>'\f'</code> | FF | lapdobás |
| <code>'\n'</code> | LF | új sor (soremelés) |
| <code>'\r'</code> | CR | kocsi vissza |
| <code>'\t'</code> | HT / TAB | vízszintes tabulátor |
| <code>'\v'</code> | VT | függőleges tabulátor |
| <code>'\\'</code> | \ | backslash |
| <code>'\"'</code> | ' | aposztróf |
| <code>'\"'</code> | " | idézőjel |
| <code>'\?'</code> | ? | kérdőjel |

Megjegyzések:

- Az ENTER leütésekor gyakran két karakter továbbbítódik a memóriába, a CR és az LF. Ezek visszhangja a képernyőn a kurzor új sorba ugrása.
- A karakter típusú konstansok a C-ben egész típusúak. Például az 'A' karakter belső ábrázolása (amennyiben az `int` típus 2 bájtban van tárolva): `0000000001000001`. Tehát `sizeof('A') = sizeof(int)`.

A karakterek beolvasására a billentyűzetről három sajátos függvény létezik.

Példa:

```
char c1, c2, c3;
c1 = getchar(); /* pufferelt beolvasás képernyővisszhanggal;
ENTER-t vár */
c2 = getche(); /* direkt a memóriába olvas képernyővisszhang-
gal; nem vár ENTER-t */
c3 = getch(); /* direkt a memóriába olvas képernyővisszhang
nélkül; nem vár ENTER-t */
```

Csak a `getchar()` ANSI standard.

A `getche()` és `getch()` használata igényli az `#include <conio.h>` programsort.

A pufferelt, illetve direkt olvasás részletes leírása végett lásd a 11.1. és 11.2. alfejezeteket.

Karakterolvasás állományból.

Példa:

```
char c;
```

```
c = fgetc(fp); /* fp – állománymutató */
```

Karakterírás képernyőre/állományba.

Példa:

```
char c = 'A';
putchar(c);
fputc(c, fp); /* fp – állománymutató */
```

Karakterolvasás/írás nyilván lebonyolítható a **scanf/printf**, illetve a **fscanf/fprintf** függvények segítségével is, használva a **%c** formázókaraktert (részletek végett lásd a 11.2. alfejezetet).

4.2.4. Karakterlánc típusú konstansok

A karakterlánc típusú konstansok idézőjelek között jelennek meg programjainkban. Például a **scanf** formázósoraként vagy az **fopen** függvény paramétereiként karakterlánc típusú konstansokat használtunk. Részletesebben a karakterláncokkal foglalkozó fejezetben beszélünk róluk.

Példa:

```
"Alma", "He\'s strong", "két\nsor", "a + b", "%d%d"
```

Megjegyzések:

- "a" nem azonos 'a'-val, ugyanis az első egyelemű karakterlánc, a második pedig karakter.
- A hosszú karakterlánc típusú konstansok több sorba tördelhetők a '\'

```
    "két sorba tört hosszú\
    karakterlánc konstans"
```

- Az egymás után elhelyezkedő karakterlánckonstansokat egyesíti a fordító:

```
"a kettő" "egy lesz"
```

4.2.5. Szimbolikus konstansok

A szimbolikus konstansok használata áttekinthetőbbé teszi a programot, és megkönnyíti az utólagos változtatást. A **const** módosító jelzővel (típusminősítővel), a **#define** direktívával, illetve az **enum** típus segítségével hozhatók létre. Az **enum** típus az összetett típusok keretén belül, a **#define** direktíva pedig a makrókkal foglalkozó fejezetben kerül részletes bemutatásra.

A **const** típusminősítő egy változódefiniálást konstansdefiniálássá alakít. Az így létrejött konstans azonban nem igazi konstans, hanem

4.3. OPERÁTOROK

81

„írásvédetté” tett változó. Helyfoglalás történik számára a memóriában, akár a változók esetében, sőt pointeren keresztül (tehát indirekt módon) még is változtatható az értéke (5. fejezet).

Az alábbi példa bemutatja a **const** és a **#define** használatának módját.

Példa:

```
const float PI = 3.14;
#define PI 3.14
```

Megjegyzés. A szimbolikus konstansok nevét nagybetűkkel szokás írni.

Néhány előre definiált szimbolikus konstans: **RAND_MAX**, **CHAR_MAX**, **CHAR_MIN**, **INT_MAX**, **INT_MIN**, **UINT_MAX**, **LONG_MAX**, **LONG_MIN**, **ULONG_MAX**

Használatuk igényli az **#include <limits.h>** programsort.

Értékeik végett lásd a Help-et.

4.3. Operátorok

Az operátorok lehetnek egy, két vagy három operandusúak.

4.3.1. Aritmetikai operátorok

+ összeadás
 - kivonás
 * szorzás
 / osztás
 % osztási maradék

A +, illetve - operátorok előjelként is szerepelhetnek mint egyoperandusú operátorok.

4.3.2. Összehasonlítási operátorok

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

Mivel a C-ben nincs logikai típus, ezért ha egy összehasonlítás eredménye igaz, akkor az összehasonlításnak mint kifejezésnek az értéke egy **int** típusú 1-es (igaz), ellenkező esetben **int** típusú 0-s (hamis).

Például az `1 == 0` kifejezésnek az értéke `0`, mert hamis. Viszont a `13 > 10` kifejezésnek az értéke `1`, ugyanis igaz.

4.3.3. Logikai operátorok

- `!` tagadás (1 operandusú)
- `&&` logikai és (2 operandusú)
- `||` logikai vagy (2 operandusú)

Az alábbi példák esetében vegyük figyelembe a C nyelv azon jellegzetességét, hogy minden nem nulla érték logikai értéke igaz, a nulla logikai értéke pedig hamis:

| Kifejezés | Logikai értéke |
|---------------------------------------|----------------|
| <code>!5</code> | <code>0</code> |
| <code>!!5</code> | <code>1</code> |
| <code>!0</code> | <code>1</code> |
| <code>5 && 6</code> | <code>1</code> |
| <code>0 && 13</code> | <code>0</code> |
| <code>0 13</code> | <code>1</code> |
| <code>0 0</code> | <code>0</code> |
| <code>2 3</code> | <code>1</code> |
| <code>!(1 == 1)</code> | <code>0</code> |
| <code>7 == 8 8 == 7</code> | <code>0</code> |
| <code>7 == 7 && 8 == 8</code> | <code>1</code> |

De Morgan-képletek:

$$!(a \ \&\& \ b) \Leftrightarrow !a \ || \ !b,$$

illetve

$$!(a \ || \ b) \Leftrightarrow !a \ \&\& \ !b.$$

Megjegyzés. Figyelem! Egy logikai kifejezés kiértékelése csak addig folytatódik, amíg az eredmény egyértelművé válik. Ebből probléma adódhat, ha nem vagyunk figyelmesek.

Például az `(a && b++)` kifejezés kiértékelésekor, ha az `a` értéke `0`, akkor a kifejezés értéke automatikusan `0` lesz, és nem kerül sor a `b` növelésére.

4.3.4. Bitenkénti operátorok (csak egész típusú operátorokra alkalmazhatók)

Az operandus/operandusok belső ábrázolásának bitjein kerül végrehajtásra az operandus által előírt művelet.

<<, >> bitenkénti balra/jobbra léptetés (2 operandusú)
 & bitenkénti *és* (2 operandusú)
 | bitenkénti *vagy* (2 operandusú)
 ^ bitenkénti *kizáró vagy* (2 operandusú)
 ~ bitenkénti *tagadás* (1 operandusú)

A *kizáró vagy* akkor igaz, ha az operandusok (a megfelelő bitek) különböző logikai értékűek.

A következő táblázat példákon keresztül, szemléletesen mutatja be, miként működnek a bitenkénti operátorok.

Példák:

```
unsigned char a=1, b=3;
```

| Kifejezés | Belső ábrázolás | Érték |
|-----------|-----------------|-------|
| a | 00000001 | 1 |
| b | 00000011 | 3 |
| a << 3 | 00001000 | 8 |
| a >> 1 | 00000000 | 0 |
| a & b | 00000001 | 1 |
| a b | 00000011 | 3 |
| a ^ b | 00000010 | 2 |
| ~a | 11111110 | 254 |

Megjegyzések:

- Balra léptetéskor a legnagyobb helyértékű bit (a legbaloldalabbi) „kiesik”, jobbról pedig 0-s bit „jön be”.
- Jobbra léptetéskor a legkisebb helyértékű bit (a legjobboldalabbi) „esik ki”, balról pedig előjel nélküli (**unsigned**) típus esetén 0, előjeles (**signed**) típus esetén az előjel bit „jön be”.

```
char a=-1;
```

```
unsigned char b = 255;
```

| Kifejezés | Belső ábrázolás | Érték |
|---------------------|-----------------|-------|
| a | 11111111 | -1 |
| b | 11111111 | 255 |
| a >> 1 | 11111111 | -1 |
| b >> 1 | 01111111 | 127 |
| a << 1 | 11111110 | -2 |
| b << 1 | 11111110 | 254 |

Megjegyzések:

– A balra/jobbra léptetés 2-vel való szorzást/osztást jelent. Mivel a bitműveleteket gyorsabban hajtja végre a számítógép, ezért 2 hatványaival való szorzáskor vagy osztáskor hatékonyabb a bitművelet használata.

Például:

$a * 2 \Leftrightarrow a \ll 1$
 $a / 2 \Leftrightarrow a \gg 1$

– Ha **b** egy 0 vagy 1 értékű bit, akkor:

$b \mid 0 = b$
 $b \mid 1 = 1$
 $b \& 0 = 0$
 $b \& 1 = b$
 $b \wedge 0 = b$
 $b \wedge 1 = \sim b$

A maszkolás technikája

(A példákban az **x** változó **long** típusúnak tekintendő.)

Hogyan tudjuk lekérdezni egy egész változó **p**-edik helyértékű bitjét?

`if (x & (1UL<<p)) ...`

Hogyan tudjuk kiíratni egy egész változó **p**-edik helyértékű bitjét?

`printf("%d", !!(x & (1UL<<p)));`

Hogyan tudjuk ellenőrizni, hogy egy egész változó **p**-edik és **q**-adik helyértékű bitjei azonosak-e?

`if (!(x & (1UL<<p)) == !(x & (1UL<<q))) ...`

Hogyan tudjuk 0-ra állítani egy egész változó **p**-edik helyértékű bitjét?

`x &= ~(1UL<<p);`

Hogyan tudjuk 1-re állítani egy egész változó **p**-edik helyértékű bitjét?

`x |= 1UL<<p;`

Hogyan tudjuk tagadni egy egész változó p -edik helyértékű bitjét?

```
x ^= 1UL<<p;
```

4.3.5. Értékadás-operátor

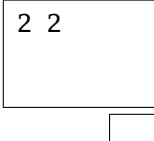
Az értékadás szintaxisa:

```
<változó> = <kifejezés>;
```

Mivel a C-ben az „=” operátor, ezért az értékadás maga is kifejezés, amelynek az értéke egyenlő a változóhoz rendelt értékkel.

Példa:

```
int a;
printf("%d ", a = 2);
printf("%d", a);
```



A többszörös értékadás szintaxisa:

```
<v1> = <v2> = <v3> = ... = <vn> = <kifejezés>;
```

Példa (négy változó egy utasításból való nullával történő inicializálása):

```
int a, b, c, d;
a = b = c = d = 0;
```

Megjegyzés. Többszörös értékadás esetén az „=” operátorok jobbról balra sorrendre aktiválódnak.

Mindenik változó felveszi az értékadás jobb oldalán levő *kifejezés* értékét.

4.3.6. Összetett operátorok

+=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=

Általánosan:

```
<v> = <v> <op> <kifejezés> ⇔ <v> <op>= <kifejezés>
```

Példa:

```
a = a | b; ⇔ a |= b; /* a-t bitenként „vagyolja” b-vel */
a = a >> 4; ⇔ a >>= 4; /* a-t jobbra lépteti 4 bittel */
```

4.3.7. Vesszőoperátor

Szintaxisa:

```
<kif1>, <kif2>, <kif3>, ..., <kifn>
```

A $\langle \text{kif}_i \rangle$ kifejezések balról jobbra értékelődnek ki, az egésznek mint kifejezésnek pedig az értéke az utolsó kifejezés $\langle \text{kif}_n \rangle$ értéke lesz.

A vesszőoperátor használatának csak akkor van értelme, ha a $\langle \text{kif}_i \rangle$, $i = 1, \dots, n-1$ kifejezéseknek önmagukban is van értelmük (például értékadások). Gyakran olyan helyen használjuk, ahol a szintaxis egy kifejezést kér, de mi többet szeretnénk odaírni.

Példák:

```
int a, b, c, d;
d=(a=1,b=2,c=a+b,c*c);
printf("%d %d %d %d", a, b, c, d);
```

1 2 3 9

```
int i, j;
for(i=1, j=100; i<j; i++, j--) ...
```

4.3.8. Feltételes operátor

Szintaxisa:

$\langle \text{kif}_1 \rangle ? \langle \text{kif}_2 \rangle : \langle \text{kif}_3 \rangle ;$

Kiértékelődik $\langle \text{kif}_1 \rangle$. Ha a logikai értéke 1 (igaz), akkor kiértékelődik $\langle \text{kif}_2 \rangle$, és ez lesz az egésznek mint kifejezésnek az értéke, és $\langle \text{kif}_3 \rangle$ figyelmen kívül marad. Ha viszont $\langle \text{kif}_1 \rangle$ logikai értéke 0 (hamis), akkor a $\langle \text{kif}_3 \rangle$ értékelődik ki, és ennek az értéke lesz az egésznek mint kifejezésnek az értéke, és $\langle \text{kif}_2 \rangle$ marad figyelmen kívül.

Példák:

```
x = x >= 0 ? x : -x; /* az x változó abszolút értékét képezi */
max = a > b ? a : b; /* a max változóba a és b közül a nagyobbik kerül */
```

4.1. feladat. Legyenek a és b egy elsőfokú egyenlet ($ax + b = 0$) valós együtthatói. Mit valósít meg a következő utasítás?

```
a? printf("x=%f", -b/a):
b? printf("nincs megoldás"): printf("végtelen sok
megoldás");
```

4.3.9. A sizeof operátor

Megadja byte-ban a méretét egy adott kifejezés típusának, illetve egy adott típusnak.

Szintaxisa:

sizeof(<kifejezés>) vagy **sizeof(<típus>)**

Megjegyzések:

- A kifejezés nem kerül kiértékelésre, csupán a típusa kerül meghatározásra, amelyből egyértelműen adódik a mérete, tehát nincsenek mellékhatások.
- A típusméret, amint már említettük, függ a fordítótól. Ezért, ha hordozható programot szeretnénk írni, használjuk a **sizeof** operátort.

Példák:

```
float a = 1; double b;
```

| Kifejezés | Értéke |
|-----------------------|--|
| sizeof(a) | 4 |
| sizeof(double) | 8 |
| sizeof(a + b) | 8 |
| sizeof(++a) | 4 /* az a értéke 1 marad, mert a ++a kifejezés nem értékelődött ki */ |

4.3.10. A Cast operátor (explicit típuskonverzió)

Vannak esetek, amikor azt szeretnénk, hogy egy kifejezés kiértékelésekor egy adott operandus ne az implicit típusa szerint legyen figyelembe véve, hanem egy más típus szerint. Ilyenkor explicit típuskonverzióval rákényszerítjük a kívánt típust.

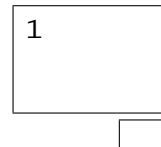
Szintaxisa:

(<típus>)<operandus>

Példák:

- Explicit típuskonverzióval meghatározható egy valós szám egész része.

```
float x = 1.9;
printf("%d", (int)x);
```



- Explicit típuskonverzió segítségével megvalósítható két egész változó valós osztása.

```
int a = 3, b = 6;
printf("%f", (float)a / b);
```

0.500000

- Még egy példa:

```
float x = 1.9;
printf("%d", (int)(++x + 3));
```

5

4.4. Megoldott feladatok

1. Írjunk programot, amely beolvas egész számokat nulla végjelig, és kiírja az összegüket, a számtani közepüket és a párosak, illetve páratlanok összegeinek abszolút értékei közül a nagyobbbat!

```
int x, s, s1, s2, k;
s = s1 = s2 = k = 0;
while(scanf("%d", &x), x)
{
    s += x; k++;
    x % 2 ? s2 += x : s1 += x;
}
printf("Az összeg: %d", s);
printf("A számtani közép: %f", (float)s/k);
printf("A nagyobb összeg modulusza: %d",
(s1=(s1>=0?s1:-s1)) >= (s2=(s2>=0?s2:-s2))?s1:s2);
```

Magyarázat: Azzal, hogy a beolvasást (a `scanf` függvény által) beépítettük a `while` ciklus feltételébe, elkerültük, hogy az első számot a ciklus előtt, a többieket pedig a ciklus-magban olvassuk be. A ciklus-feltételben vessző operátort használtunk. A `while` ciklus addig ismétel amíg az `x` változóba beolvasott szám logikai értéke igaz, azaz különbözik nullától.

2. Írjunk programot, amely karaktereket olvas be * végjelig, és kiírja, hány '0'-s karakter volt köztük! Mi változik, ha a karakter-beolvasásra rendre a `getchar()`, `getche()`, `getch()` függvényeket használjuk?

```
int k = 0;
char c;
while((c = getchar()) != '*')
```

4.4. MEGOLDOTT FELADATOK

89

```
if(c == '0') k++;
printf("Az 0-k száma: %d", k);
```

Magyarázat: Figyelembe vettük, hogy a getchar függvény visszatéríti a beolvasott értéket. A ciklus-feltételben kihasználtuk, hogy az értékadásnak, mint kifejezésnek, az értéke azonos a hozzárendelt értékkel.

3. Írjunk programot, amely beolvas egy egész számot egy **long** változóba, és kiírja a belső ábrázolását!

```
long x;
int i;
unsigned long maszk = 1UL << sizeof(long)*8-1;
scanf("%ld", &x);
printf("A szám belső ábrázolása:\n:");
for(i = 1; i <= sizeof(long)*8; i++)
{
    if(x & maszk) putchar('1');
    else putchar('0');
    maszk >>= 1;
}
```

Magyarázat: Ha az **(x & maszk)** logikai értéke igaz, azaz különbözik nullától, akkor az **x** szám belső ábrázolásában, a maszk 1-es bitjének irányában, 1-es bit van, különben 0. (a maszk belső ábrázolása egyetlen 1-es bitet tartalmaz, balról számítva az **i**-edik pozícióban)

4. Írjunk programot, amely egész számokat olvas be egy **long** típusú változóba 0 végjelig, és megszámlálja a pozitívakat, negatívakat, párosokat, páratlanokat, valamint azokat, amelyek belső ábrázolásában több 1-es bit volt, mint 0-s bit!

```
long x;
int pozitiv=0, negativ=0, paros=0, paratlan=0, k=0, k1, k2, i;
while(scanf("%ld", &x), x)
{
    if(x&1L) paratlan++;
    else paros++;
    if(x&1L<<sizeof(long)*8-1) negativ ++;
    else pozitiv ++;
    k1 = k2 = 0;
    for(i = 1; i <= sizeof(long)*8; i++)
    {
        if(x & (1L << sizeof(long)*8 - i)) k1++;
        else k2++;
    }
    if(k1 > k2)k++;
}
```

```
}
printf("%d\n%d\n%d\n%d\n",
       pozitiv,negativ,paros,paratlan,k);
```

Magyarázat:

- Egy szám aszerint páros/páratlan, hogy belső ábrázolásának legkisebb helyértékű bitje 0 vagy 1.
- Egy előjeles egész szám aszerint pozitív/negatív, hogy belső ábrázolásának legnagyobb helyértékű bitje 0 vagy 1.
- Az `(1L « sizeof(long)*8-i)` értéket maszk-ként használtuk, hiszen a `(sizeof(long)*8)` biten való belső ábrázolása egyetlen 1-es bitet tartalmaz, balról számítva az *i*-edik pozícióban.

5. Írjunk programot, amely *n* darab természetes számot **unsigned short** típusú változóba olvas, és előállítja azokat a számokat, amelyeket úgy nyerünk a beolvasott számokból, hogy a belső ábrázolásuk alsó byte-ját kicseréljük a felsővel! Írjuk ki az eredeti számot, az előállított számot, valamint a belső ábrázolásait!

```
int j, n;
unsigned short x, y, maszk;
scanf("%d", &n);
for(j = 1; j <= n; j++)
{
    scanf("%hu", &x);
    y = (x << sizeof(short)*8/2) |
        (x >> sizeof(short)*8/2);
    printf("%hu :", x);
    maszk = 1 << sizeof(short)*8-1;
    while(maszk)
    {
        if(x & maszk) putchar('1');
        else putchar('0');
        maszk >>= 1;
    }
    putchar('\n');
    printf("%hu :", y);
    maszk = 1 << sizeof(short)*8-1;
    while(maszk)
    {
        if(y & maszk) putchar('1');
        else putchar('0');
        maszk >>= 1;
    }
    putchar('\n');
}
```

Ötlet: Ha n belső ábrázolását eltoljuk jobbra (`sizeof(unsigned)*8/2`) pozícióval, akkor a felső byte-ja átkerül az "alsó byte-pozícióba", miközben helyébe nulla bitek "jönnek be" (n előjel nélküli egész). Hasonlóképpen, ha n belső ábrázolását eltoljuk balra (`sizeof(unsigned)*8/2`) pozícióval, akkor az alsó byte-ja átkerül a "felső byte-pozícióba" miközben helyébe nulla bitek "jönnek be". Ha "bitenként vagy-oljuk" az így kapott két értéket, akkor a kért eredményhez jutunk.

4.5. Kitűzött feladatok

1. Adott egy egész szám, amelyet egy **long** típusú változóba tárolunk el. Forgassuk körkörösén (ami kiesik egyik felől, az jön be a másik felén) jobbra/balra a belső ábrázolása bitjeit, és írjuk ki az így kapott számokat belső ábrázolásukkal együtt!

2. Ugyanaz a feladat, de a forgatást úgy valósítjuk meg, hogy a legkisebb helyértékű és legnagyobb helyértékű bitek fixen maradjanak (a többbit forgatjuk maguk között).

3. Adott egy egész szám, amelyet egy **long** típusú változóba tárolunk el. Állítsuk elő azt a számot, amelyet úgy kapunk, hogy a belső ábrázolása szomszédos bitjeit (0.-1., 2.-3., ...) kicseréljük egymás között!

4. Adott egy x természetes szám az **unsigned long** tartományból, valamint az a, b, c, d 0..31 közti természetes számok. x belső ábrázolásának az a -edik bitjét állítsuk 0-ra, a b -edik helyértékű bitjét 1-re, a c -edik bitjét tagadjuk, majd léptessük d pozícióval balra. Írjuk ki az eredeti számot és belső ábrázolását, majd az újonnan nyert számot és belső ábrázolását!

5. Adott n egész szám az **int** tartományból. Csomagoljuk össze négyenként a belső ábrázolásuk bitjeit egy-egy hexadecimális számjegybe, és írjuk ki az így kapott 16-os számrendszerbeli számot!

5. FEJEZET

POINTEREK (MUTATÓK)

Beszéltünk már arról, hogy minden változónak van címe, és pedig annak a memóriarekesznek a címe, amellyel kezdődően hely van foglalva számára a memóriában.

Eltárolható-e a memóriában valamely változó címe? Igen, címtípusú változóknak. Ezeket *pointereknek* vagy *mutatóknak* nevezzük.

A pointerdefiniálás szintaxisa:

```
<típus> * <pointer>;
```

Példák:

```
int *pi;
float *pf;
long double *pld;
```

pi egy **int** típusú változó *címét* tartalmazhatja, ezért azt mondjuk, hogy **int**-pointer (típusa **int***). **pf**-ben, illetve **pld**-ben valamely **float**, illetve **long double** változó *címe* tárolható el.

Hogyan tudunk hivatkozni egy változó címére? Az **&** operátor segítségével!

Szintaxisa:

```
& <változó>
```

Példa:

```
int a , *p;
p = &a;
```

Azt mondjuk, hogy a **p** pointer az **a** változóra mutat.

Hogyan lehet egy pointeren keresztül hivatkozni arra a változóra, amelyre mutat? A ***** operátor segítségével!

Szintaxisa:

```
*<pointer>
```


5. POINTEREK (MUTATÓK)

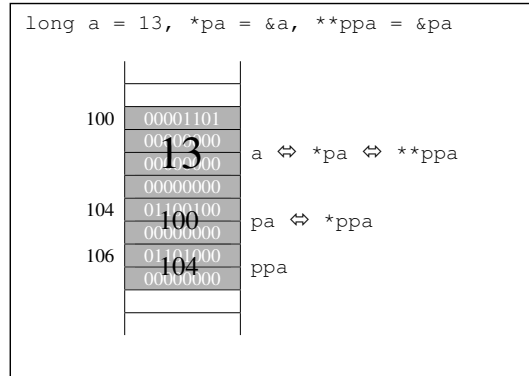
93

Példa:

```
float x = 3.14, *px = &x;
printf(" %f\n%f ", *px, x);
```

3.140000
3.140000

Tehát: $*px \Leftrightarrow x$.



A különféle címtípusok között nincs implicit típuskonverzió, de alkalmazható a CAST operátor (explicit típuskonverzió).

Példa:

```
long x = 2003, *p1; float *p2;
p1 = &x; /* helyes */
p2 = &x; /* helytelen, mert p2 float* típusú, azaz float-
        pointer, &x viszont long* típusú, lévén long-cím. */
p2 = (float *)&x; /* helyes */
```

Mi jelenik meg a képernyőn a `printf ("%f", *p2);` kiírás nyomán? Az a valós szám, amelynek `float`-ként való belső ábrázolása azonos a 2003-as egész `long`-kénti belső ábrázolásával.

Léteznek olyan pointerok, amelyek bármely típusú változó címét eltárolhatják. Ezek a `void`-pointerok, típusuk `void*`.

Példa:

```
int a = 1; double b = 2; char c = 'A';
void * p;
p = &a; printf("%d\n", *(int*)p);
p = &b; printf("%lf\n", *(double*)p);
p = &c; printf("%c", *(char*)p);
```

1
2.000000
A

Tehát, ha egy **void**-pointeren keresztül szeretnénk hivatkozni arra a változóra, amelynek címét tartalmazza, először a **CAST** operátor segítségével a **void** pointert a változó típusának megfelelő pointerre kell konvertálnunk, és csak azután alkalmazhatjuk a ***** operátort.

A fenti példából az is kiderül, hogy ha egy **void**-pointerhez rendelünk bármilyen típusú pointert, akkor nincs szükség explicit típuskonverzióra. A C nyelvben fordítva is igaz (bármilyen típusú pointerhez hozzárendelhető egy **void**-pointer), de C++-ban már nem. Ezért ezt az utóbbi esetet kerüljük, vagyis használjunk explicit típuskonverziót, ha hordozható programot szeretnénk írni.

A pointerek és a const módosító jelző

const <típus> * <pointer> – nem változtatható meg a pointeren keresztül a változó, ahova a pointer éppen mutat
<típus> * const <pointer> – nem változtatható meg a pointer

Példa:

```
int a = 1, b = 2;
const int c = 3;
int * pc = &c;
const int * p; /* p, int típusú konstansra mutató pointer változó */
int * const q = &a; /* q, int-pointer-konstans, amely az a címét tartalmazza */
p = &a; /* helyes, mert p változó */
a++; /* helyes, mert a változó */
(*p)++; /* helytelen, mert *p konstans (p-n keresztül a konstansnak látszik) */
p++; /* helyes, mert p változó */
(*q)++; /* helyes, mert *q változó */
q++; /* helytelen, mert q konstans */
c++; /* helytelen, mert c konstans */
(*pc)++; /* helyes, mert *pc változó; ezért mondtuk, hogy c nem igazi konstans */
```

5.1. Műveletek pointerekkel

5.1.1. Értékadás

Az azonos típusú pointerek egymáshoz rendelhetők. Amint már láttuk, különböző típusú pointerek esetén szükséges az explicit típuskonverzió, kivéve, ha void-pointerhez rendelünk pointert.

Példa:

```
char c, * pc = &c; long double ld, * pld = &ld;
void * p;
pc = pld; /* helytelen */
pc = (char *)pld; /* helyes */
p = pld; /* helyes */
pld = p; /* helyes, de kerülendő */
pld = (long double *)p; /* így ajánlatos */
```

5.1.2. Összehasonlítás

Az azonos típusú pointerek összehasonlíthatók. Mivel a pointerek a bennük tárolt memóriacímek alapján vannak összehasonlítva, ezért, ha két pointer ugyanarra a változóra mutat, akkor egyenlő.

Példa:

```
unsigned short u1 = 1, u2 = 2, *p1 = &u1, *q1 = &u1,
*p2 = &u2;
```

| | | | |
|-------------------|-------------------------------|---|----|
| | | | |
| 1A00 _H | 00000001 | 1 | u1 |
| | 00000000 | | |
| 1A02 _H | 00000010 | 2 | u2 |
| | 00000000 | | |
| 1A04 _H | 0 _H 0 _H | | p1 |
| | 1 _H A _H | | |
| 1A06 _H | 0 _H 0 _H | | q1 |
| | 1 _H A _H | | |
| 1A08 _H | 0 _H 2 _H | | q2 |
| | 1 _H A _H | | |
| | | | |

A fenti példában p1 és q1 egyenlő, p2 viszont nagyobb náluk, hiszen u2 nagyobb memóriacímre kerül, mint u1.

5.1.3. <pointer> + / - <egész>

Legyen a következő általános definiálás:

<típus> v, *p = &v; (p a v változóra mutató <típus>* típusú pointer)

Ha n egész szám, akkor $p + n$, illetve $p - n$ p-vel azonos típusú pointerek lesznek, amelyek a $p + n * \text{sizeof}(\text{típus})$, illetve $p - n * \text{sizeof}(\text{típus})$ címen levő <típus> típusú adatokra mutatnak.

Úgy is mondhatnánk, hogy minden pointernek van egy lépésmérete, amely azonos azzal a „típusmérettel”, amely megfelel annak a típusnak, amilyen típusú változóra mutat. Ha egy pointerhez hozzáadunk egy egész számot, akkor az összeg az illető számmal megegyező számú „lépéssel” fog „arrébb” mutatni a memóriában.

Szemléletesebben:

```
char c, *p1= &c;
float *p2 = (float *)&c;
```

| | | | | | |
|--------|----|---------|-----|--|---|
| | p2 | p1 | 100 | | c |
| | | p1 + 1 | 101 | | |
| | | p1 + 2 | 102 | | |
| | | p1 + 3 | 103 | | |
| p2 + 1 | | p1 + 4 | 104 | | |
| | | p1 + 5 | 105 | | |
| | | p1 + 6 | 106 | | |
| | | p1 + 7 | 107 | | |
| p2 + 2 | | p1 + 8 | 108 | | |
| | | p1 + 9 | 109 | | |
| | | p1 + 10 | 110 | | |
| | | p1 + 11 | 111 | | |
| p2 + 3 | | p1 + 12 | 112 | | |
| | | p1 + 13 | 113 | | |

Példák:

```
int a = 1, b = 2, c = 3, *p = &a;
printf("%d\n%d\n%d", *p, *(p-1), *(p-2));
```

1
2
3

| | |
|---------|---|
| (p-2)-> | c |
| (p-1)-> | b |
| (p)-> | a |

Az **a**, **b**, **c** cellák a megfelelő változókat szemléltetik a memóriában. Mindenikük mérete `sizeof(int)` memóriarekesz.

A függvényekkel foglalkozó fejezet fogja tisztázni, hogy miért kerülnek a változók csökkenő memóriacímekre (amennyiben lokális változóként, valamely függvényen belül, például a **main**-en belül definiáljuk őket).

Mi a hatása a következő **for** ciklusnak?

```
float x, *p;
for(p = &x; p < &x + 10; p++)
    printf("%f\n", *p);
```

A **p** pointert ráállítja az **x** változóra, majd 4 byte-onként (**float** méret) lépteti a memóriában (10-szer). Minden lépésben kiírja az illető címen található **float** értéket, tehát azt, ahova éppen mutat.

Figyelem! A pointerok nyújtotta ezen szabadsággal felelősségteljesen kell élnünk, mert ha visszaélünk vele, könnyen kiakadhat a programunk.

Ha **p** pointer, **n** pedig egész szám, akkor helyesek az alábbi műveletek:

```
p = p + n; p = p - n;
p += n; p -= n;
p ++; ++ p;
p --; -- p;
```

5.1.4. Pointerok különbsége

Két azonos típusú pointer kivonható egymásból, különbségük az az egész szám lesz, amennyivel az első pointer el van léptetve a másodikhoz képest.

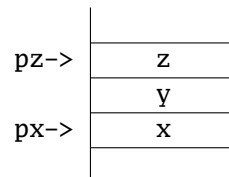
Ez a művelet az 5.1.3. alpontban bemutatott műveletnek a következménye:

```
ha p1 = p2 + n, akkor p1 - p2 = n
ha p1 = p2 - n, akkor p1 - p2 = -n
```

Például:

```
double x, y, z, px = &x, pz = &z;
```

A `pz-px` pointerkülönbség értéke `-2` lesz, mivel `pz` (mint `double-pointer`) `2 double`-val „mutat előre” („van visszaléptetve”) a memóriában `px`-hez képest.



Az `x`, `y`, `z` cellák a megfelelő változókat szemléltetik a memóriában. Mindenikük mérete `sizeof(double)` memóriarekesz.

5.1. feladat. Adott egy valós szám, amelyet `long double` változóban tárolunk el. Írjuk ki a belső ábrázolását.

```
long double x; int j;
unsigned char maszk;
char *p = (char*)&x; p += sizeof(long double)-1;
/* Definiálunk egy char-pointert, amit ráállítunk
   x legnagyobb helyértékű byte-jára */
scanf("%Lf", &x);
for( j = 1; j <= sizeof(long double); j++)
{
    maszk = 1 << sizeof(char)*8-1;
    while(maszk)
    {
        if(*p & maszk) putchar('1');
        else putchar('0');
        maszk >>= 1;
    }
    p--;
}
```

Magyarázat: Miközben a külső ciklus byte-ról byte-ra lépteti a `p` char-pointert, a belső ciklus kiírja a `p` által megcímzett byte belső ábrázolását. Figyelembe vettük, hogy a memóriában az adat ábrázolás byte-fordítottan történik.

6. FEJEZET

TÖMBÖK

6.1. Egydimenziós tömbök

Az előző fejezetekben több olyan algoritmust is bemutattunk, amelyek egy számsorozat elemeivel dolgoztak (például kiszámították az elemek összegét, megszámozták a párosokat, meghatározták a maximumot stb.). Ezen algoritmusok egyik közös vonása az volt, hogy a számsorozat elemeit ugyanabba – az elemek típusának megfelelő típusú – memóriaváltozóba olvasták be. Ez azt jelentette, hogy a memóriában egy adott pillanatban a számsorozat egyetlen eleme volt eltárolva. Amikor beolvastuk a következő elemet a változóba, az előző elem értéke elveszett. Ez azért nem jelentett gondot, mert az algoritmusok olyan természetűek voltak, hogy a számsorozat elemein, rögtön a beolvasásuk után, elvégezték az összes műveletet, amelyet a feladat megoldása megkövetelt.

Vannak azonban olyan feladatok, amelyek megoldási algoritmusai megkövetelik, hogy egy számsorozat elemei egyidőben bent legyenek a memóriában (ilyen feladatok például a rendezések).

Mi a megoldás? A *tömbtípus*.

Definiálásának szintaxisa:

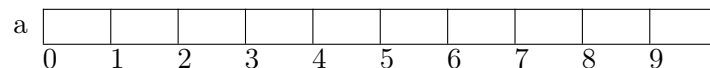
```
<elemtípus> <név>[<elemszám>];
```

Példák:

```
# define N 100
int a[10];
float b[20];
char c[N];
```

Megjegyzés. Az elemszám egész típusú *konstans* kell legyen.

Az első definíció nyomán létrejött tömbváltozót így lehetne ábrázolni:



A fenti tömbben egy maximum 10 elemű egész számsorozat elemei tárolhatók el.

A tömb elemei azonos típusúak. Például az **a**, **b**, **c** tömbök elemei rendre **int**, **float**, **char** típusúak.

Azért, hogy hivatkozni tudjunk a tömb egyes elemeire, ezek indexelve vannak **0**-tól kezdve **<elemszám>-1**-ig.

Szintaxis:

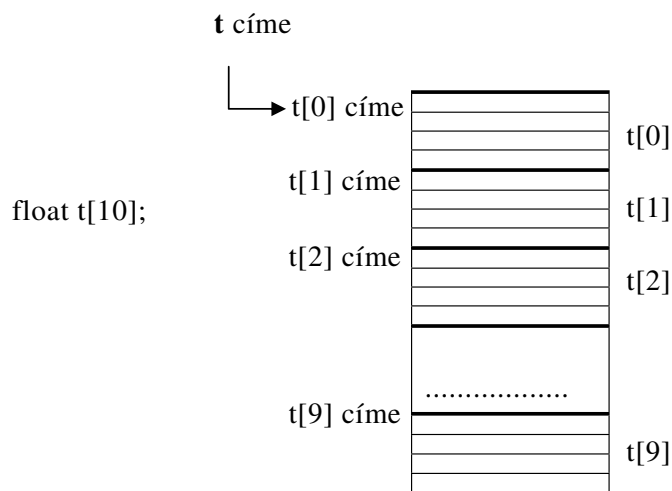
<tömbnév> [<index>]

Példák:

a[0], b[5], a[99], a[i], b[2*i+1]

Megjegyzés. Az index egész típusú *kifejezés* lehet.

Egy tömb elemei számára egymás utáni memóriaterület kerül lefoglalásra.



A tömb számára lefoglalt memóriaterület mérete: **<elemszám> * sizeof (<elemtípus>)**. Például a **t** tömb számára lefoglalt tárhely mérete: **10*4 byte = 40 byte**. Egy tömb címe a **0**-adik elemének a címe.

6.1.1. Inicializálás definiáláskor

Példák:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float f[100] = {3.14, -12.45}; /* A többi elem 0-val
    inicializálódik. */
int b[50] = {0}; /* Minden eleme 0-val inicializálódik;
    csak 0-val lehet ily módon inicializálni. */
```



```
double d[] = {1, 5.7, -3.45, 12, 0.66}; /* A tömb
    automatikusan 5 eleműre definiálódik. */
int a[5] = {1, 2, 3, 4, 5, 6, 7}; /* Helytelen, túl
    sok a kezdőérték. */
```

6.1.2. Kapcsolat a tömbök és pointerek között

Egy tömb neve úgy tekinthető, mint a 0-adik elemének a címe; pontosabban, mint egy pointer *konstans*, amely a tömb 0-dik elemére mutat. Például legyen az alábbi tömbdefiniálás:

```
float a[100];
```

a-nak, mint pointernek a típusa `float*`, az értéke pedig `&a[0]`.

Továbbá:

```
a      ⇔ &a[0]  - a 0-dik elem címe
a + i  ⇔ &a[i]  - az i-edik elem címe
*a     ⇔ a[0]   - a 0-dik elem
*(a + i) ⇔ a[i] - az i-edik elem
```

Egy másik bizonyítékát a tömbök és pointerek rokonságának a következő példa illusztrálja:

```
int a[10], *p = a;
a[i] ⇔ p[i] ⇔ *(a+i) ⇔ *(p+i) - az i-edik elem
&a[i] ⇔ &p[i] ⇔ a + i ⇔ p + i - az i-edik elem címe
Tehát a pointerek indexelhetők: p[i] = *(p+i);
```

Az alapvető különbség *a* és *p* között az, hogy *a* konstans, *p* pedig változó: amíg *(p++)*-t követően *p* az *a[1]* elemre fog mutatni, addig az *a++* inkrementálás hibás.

Ha *p* `long`-pointer (`long *p;`), akkor úgy is felfogható, mint egy *p* címen kezdődő, `long` elemtípusú tömb „neve”, és *p[i]* mint ennek a tömbnek az *i*-edik eleme. Ha *p* nem egy – a memóriában létező – tömbre mutat, akkor *p[i]* a *p+i* címen lévő 4 byte-os területet jelenti, ahol „ki tudja, mi van”.

6.1. feladat. Adott egy *n* elemű egész számokat tartalmazó sorozat. Olvassuk be egy egydimenziós tömbbe, majd pedig írjuk ki a tömb elemeinek értékét a képernyőre.

```
int a[100], n, i;
scanf("%d", &n);
for(i = 0; i < n; i++)
    scanf("%d", &a[i]);
for(i = 0; i < n; i++)
    printf("%d\n", a[i]);
```

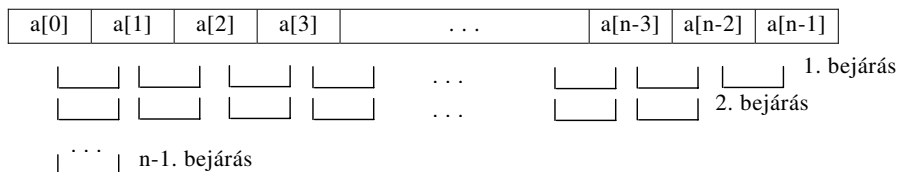
```
int a[100], n, i, *p;
scanf("%d", &n);
for(p = a; p - a < n; p++)
    scanf("%d", p);
for(p = a; p - a < n; p++)
    printf("%d\n", *p);
```

6.2. feladat. Adott egy n elemű egész számokat tartalmazó sorozat. Rendezzük az elemeit növekvő sorrendbe.

A számsorozat elemeit egy a tömb $a[0]$, $a[1]$, ..., $a[n-1]$ elemeiben tároljuk el.

6.1.2.1. Buborékos rendezés

A buborékos rendezés stratégiáját – első megközelítésben – az alábbi ábra mutatja be.



A számsorozatot $(n-1)$ -szer fogjuk bejárni. Pontosabban: az i -edik bejárásban a számsorozat $a[0..n-i]$ szakaszát. Hogyan? Például az első bejárásban összehasonlítjuk $a[0]$ -t $a[1]$ -gyel, majd $a[1]$ -et $a[2]$ -vel, ... végül $a[n-2]$ -t $a[n-1]$ -gyel. Általánosan: $a[j]$ -t $a[j+1]$ -gyel, ahol $j=0, 1, \dots, n-2$. Valahányszor úgy találjuk, hogy $a[j] > a[j+1]$, felcseréljük őket. Az eredmény: a legnagyobb elem kivándorol a tömb végére (az $a[n-1]$ tömbbelembe). A következő bejárásokban hasonlóképpen járunk el, és további elemek vándorolnak a helyükre, ahhoz hasonlóan, ahogy a levegőbuborékok szoktak feljönni a víz felszínére. Mivel minden bejárással helyére kerül legalább egy elem, a bejárt elemek szakasza – minden bejárás után – rövidíthető legalább eggyel.

Hány összehasonlítást végez a fenti algoritmus? $n * (n - 1) / 2$.

És hány értékadást? Ez attól függ, hány cserére van szükség. A legrosszabb esetben, ha a számsorozat eredetileg csökkenő sorrendben

6.1. EGYDIMENZIÓS TÖMBÖK

103

van, minden összehasonlítást csere követ, ami viszont három értékadást feltételez. Ez összesen $3 * n * (n - 1) / 2$ értékadást jelent.

Ha alapműveletnek az értékadást és az összehasonlítást választjuk, akkor a buborékos rendezés legrosszabb esetben $4 * n * (n - 1) / 2$ alapműveletet feltételez. Mivel ez a szám a bemenet méretétől (n) négyzetesen függ, azt mondjuk, hogy ezen algoritmus bonyolultsága (komplexitása) $O(n^2)$.

Nem nehéz átlátni, hogy jelenlegi állapotában, legjobb esetben is – ha a számsorozat már eredetileg növekvő sorrendben van – $O(n^2)$ az algoritmus bonyolultsága (bár nem kerül sor cserére, az összehasonlítások száma ugyanannyi marad).

Hogyan lehet javítani az algoritmus bonyolultságán? Vegyük észre, hogy a fenti algoritmus „vak”, abban az értelemben, hogy nem veszi észre, ha a számsorozat már rendezve van. Hogyan lehetne „kinyitni a szemét”? Figyeljük, hogy történik-e csere egy bejárás alatt. Ha nem, akkor a számsorozat rendezett. Sőt a tömb egyszeri bejárása alkalmával szerencsés esetben több elem is a helyére kerülhet. Ha megjegyezzük az utolsó csere helyét, a következő lépésben elég csak addig menni.

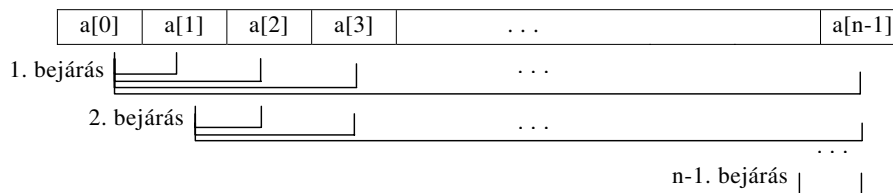
A javított algoritmus legrosszabb esetben továbbra is $O(n^2)$ bonyolultságú, viszont legjobb esetben csak $O(n)$, hiszen az első bejárás után „észreveszi”, hogy a számsorozat rendezett.

```
int a[100], n, u, i, j, v, ucs;
scanf("%d", &n);
for(i = 0; i < n; i++)
    scanf("%d", &a[i]);
u = n - 1; /* az aktuális bejárás vége */
do
{
    ucs = 0; /* az utolsó csere helye */
    for(j = 0; j < u; j++)
        if(a[j] > a[j + 1])
        {
            v = a[j];
            a[j] = a[j + 1];
            a[j + 1] = v;
            ucs = j;
        }
    u = ucs; /* A következő bejárás az előző
              utolsó cseréjének helyéig megy. */
}
while(ucs); /* Akkor van vége, ha nem volt csere
            egy adott bejárásban. */
```

```
for(i = 0; i < n; i++)
    printf("%d\n", a[i]);
```

6.1.2.2. Minimum-kiválasztásos rendezés

A minimum-kiválasztásos rendezés bármilyen számsorozat esetén kötelezően végig kell hogy szaladjon $(n-1)$ -szer a számsorozat bizonyos szakaszain. Stratégiája abban áll, hogy az i -edik bejárásban az $a[i..n-1]$ ($i=0..n-2$) szakasz legkisebb elemét előrehozza az i -edik pozícióba. Ezt úgy valósítja meg, hogy $a[i]$ -t sorra összehasonlítja az $a[j]$ ($j=i+1..n-1$) elemekkel, és valahányszor kisebbet talál, felcseréli őket. Ezt mutatja be szemléletesen az alábbi ábra:



Az alábbi programrészlet a minimum-kiválasztásos rendezés implementációja:

```
...
for(i=0; i<n-1; i++)
    for(j=i+1; j<n; j++)
        if(a[i]>a[j]) {v=a[i]; a[i]=a[j]; a[j]=v;}
...
```

Bár megírni egyszerűbb, mint a buborékos rendezést, kevésbé hatékony, ugyanis képtelen észrevenni azt az esetet, amikor a számsorozat rendezetté vált, vagy már eredetileg rendezett. Ez a magyarázata annak, hogy bonyolultsága $O(n^2)$ minden esetben.

Megjegyzés. Léteznek hatékonyabb rendezési algoritmusok is, amelyeknek bonyolultsága $O(n \log_2 n)$ a legrosszabb esetben is, de ezek implementálása meghaladja ezen könyv célkitűzéseit. Ilyenek például: a *gyors rendezés* (*quicksort*), az *összefésülő rendezés* (*mergesort*), a *kupacrendezés* (*heapsort*).

Az alábbiakban két olyan algoritmus kerül bemutatásra, amelyeknek bonyolultsága lineáris ($O(n)$). Ezek képezik a fenti megjegyzésben említett első két rendezés magvait.

Megjegyzés: Az algoritmus-bonyolultság fogalma megfoghatóbbá válik, ha le tudjuk mérni, hogy egyes programrészletek végrehajtása mennyi időt igényel a számítógép részéről. Az alábbi programban lemérjük, hogy mennyi időbe telik egy üres utasítás száz milliószor való ismétlése egy while ciklus révén. A **kezdet** változóban megjegyezzük a vizsgált programrészlet végrehajtásának kezdeti időpillanatát (processzor-időegységben kifejezve), a **veg** változóban pedig a befejezés időpillanatát. A **CLOCKS_PER_SEC** (programozási környezetfüggő) elődefiniált konstanst arra használjuk, hogy a leért időintervallum hosszát másodpercben kifejezve írjuk ki. A **clock_t** típus, illetve a **clock** függvény használata igényli a **time.h** header átlomány beszurását.

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

main()
{
    long i = 100000000L;
    clock_t kezdet, veg;
    double idointervallum;
    printf("Ennyi időbe telik %ld -szer csinálni ""semmit"": ", i );
    kezdet = clock();
    while(i--);
    veg = clock();
    idointervallum = (double)(veg - kezdet) / CLOCKS_PER_SEC;
    printf( "%2.11f másodperc\n", idointervallum);
    return 0;
}
```

6.1.2.3. Szétválogatás

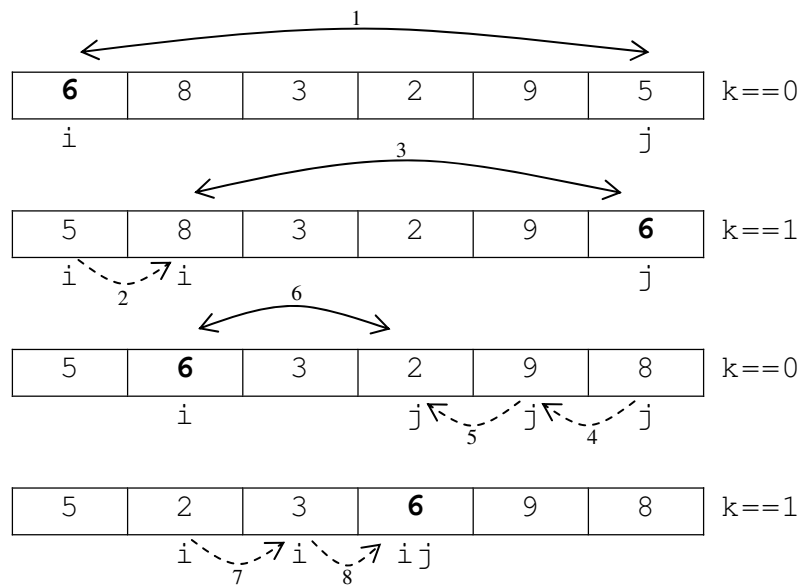
Egy **n** elemű számsorozat elemeit, amelyek az **a** tömbben vannak tárolva, válogassuk szét „az első elem szerint” (az első elemnél kisebb elemek előtte, a nagyobbak mögötte legyenek).

Példa:

a számsorozat: **6, 8, 3, 2, 9, 5**

szétválogatva: **5, 2, 3, 6, 9, 8**

Ötlet: Egy **i**, illetve egy **j** változót a számsorozat elejére, illetve végére állítunk. Kezdetben az elem, amely szerint a szétválogatás történik (továbbiakban kulcselem), nyilván az **i**-edik pozícióban van. Miközben **i** áll, jövünk előre **j**-vel (magunk mögött hagyva a kulcselemnél nagyobb elemeket), míg találunk egy **a[j]** kisebb, mint **a[i]** elemet. Felcserélve



6.1. ábra. Szétválogatás. A folytonos nyíl az $a[i]$ és $a[j]$ elemek cseréjét, a szaggatott nyíl pedig az i és j változók léptetését ábrázolják. Sorszámoztuk a műveleteket.

egymás közt $a[i]$ -t és $a[j]$ -t, a kulcselem hátrakerül a j -edik pozícióba, a nála kisebb elem pedig előrekerül az i -edikbe. Ekkor „üzemmódváltás” történik. Miközben j áll, i -vel megyünk hátrafele (magunk mögött hagyva a kulcselemnél kisebb elemeket), míg találunk egy $a[i]$ nagyobb, mint $a[j]$ elemet. Ismét felcserélve egymás közt $a[i]$ -t és $a[j]$ -t, a kulcselem visszakerül az i -edik pozícióba, a nála nagyobb elem pedig hátra a j -edikbe. Most újra j -vel jövünk előre (1. üzemmód), majd ismét i -vel megyünk hátra (2. üzemmód), mindaddig, amíg i és j találkozik. Figyeljük meg, hogy a kulcselem mindig az álló index irányában van, és az előre lépegető j a kulcselemnél nagyobb, a hátrafele mozgó i pedig a kulcselemnél kisebb elemeket hagyja maga mögött. A kulcselem a helyét, amelyet a rendezett számsorozatban foglalna el, ott találja meg, ahol találkoznak az i és j indexek. Lássuk, miként működik az algoritmus a fenti példán (6.1. ábra).

```
i=0; j=n-1;
k=0; /* k logikai értékétől függően vagy
      i, vagy j lépeget */
while (i<j)
if(a[i]<=a[j])
{
```

6.1. EGYDIMENZIÓS TÖMBÖK

107

```

    k ? i++ : j--;
}
else
{
    v=a[i]; a[i]=a[j]; a[j]=v;
    k = !k; /*üzemmódváltás */
}
for(i=0; i<n; i++)
    printf("%d ",a[i]);

```

A folytonos nyilak az $a[i]$ és $a[j]$ elemek cseréjét, a szaggatottak pedig az i és j változók léptetését ábrázolják. Sorszámoztuk a műveleteket.

Megjegyzés: A kulcselem egy csillapított rezgésre emlékeztető pályát követve találja meg a végső helyét.

6.1.2.4. Összefésülés

Adott két növekvő számsorozat, fésüljük össze őket egyetlen növekvő számsorozatba (a számsorozatok elemeinek száma n és m , és az a , illetve b tömbökben vannak tárolva).

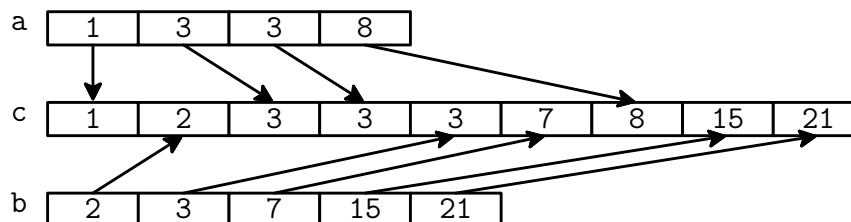
Példa:

első számsorozat (a): 1, 3, 3, 8

második számsorozat (b): 2, 3, 7, 15, 21

összefésült számsorozat (c): 1, 2, 3, 3, 3, 7, 8, 15, 21

Ötlet: Az i változó az első számsorozat, a j pedig a második elemein fog lépegetni. Az $a[i]$ és $b[j]$ elemek közül mindig a kisebbiket tesszük a c tömbbe, majd lépünk egyet abban a számsorozatban, amelyből „elfogyasztottuk” azt az elemet, amelyet a c tömbbe helyeztünk.



```

i=j=k=0;
while (i<n && j<m)
/* Amíg nem érünk valamelyik számsorozat végére,
   mikor a-ból, mikor b-ből másolunk elemet c-be
   (mindig azt, amelyik a kisebb). */

```

```

if(a[i]<=b[j])
    c[k++]=a[i++];
else
    c[k++]=b[j++];
/* Miután kifutottunk valamelyik számsorozatból,
   a másik számsorozat megmaradt elemeit egyszerűen
   bemásoljuk a c tömb végére. */
while (i<n) c[k++]=a[i++];
while (j<m) c[k++]=b[j++];

for(i=0; i<k; i++)
    printf("%d ",c[i]);
    
```

Tanulság: A `(c[k++]=a[i++];)` utasítás tömörített változata a `(c[k]=a[i]; k++; i++;)` utasítássornak.

6.1.2.5. Kitűzött feladatok

1. Adott egy n elemű valós számokat tartalmazó sorozat. Ellenőrizzük, szimmetrikus-e, ha nem, írjuk ki a tükörképét.

2. Mit ír ki még az alábbi programrészlet? Fogalmazd meg általánosan! A feladatot papíron oldjuk meg!

```

int i, a[10]={5, 3, 8, 2, 9, 0, 7, 1, 6, 4};
for(i=0; i<10; i++)
{
    if(a[0]>a[i]){v=a[0]; a[0]=a[i];a[i]=v;}
    if(a[9]<a[i]){v=a[9]; a[9]=a[i];a[i]=v;}
}
printf(„%d, %d”, a[0], a[9]);
    
```

3. *Bináris keresés:* Adott egy n elemű, szigorúan növekvő egész számokat tartalmazó sorozat, valamint egy x egész szám. Keressük meg x -et a számsorozatban az alábbi algoritmus szerint. Ha megtalálható, írjuk ki a sorszámát, ha nem, akkor egy megfelelő üzenetet.

- Összehasonlíttuk x -et a számsorozat középső elemével.
- Ha egyenlők, akkor megtaláltuk és kiíratjuk a sorszámát.
- Ha nem egyenlők, akkor folytatjuk a keresést (ugyanilyen módon) vagy a felső, vagy az alsó számsorozatszakaszban.
- Ha nulla hosszúságú szakaszhoz jutunk, ez azt jelenti, hogy x nem található meg a sorozatban.

4. Karaktereket olvasunk be * végjelig. Készítsünk karakter-előfordulási statisztikát.

5. Adott egy természetes szám, írjuk ki a kettes számrendszerbeli alakját.

6. Írjunk programot, amely megállapítja, hogy egy 50 adatból álló mérési sorozat adatai közül hány tér el 10%-nál kevésbé az átlagértéktől.

7. Írjunk programot, amely egy n hosszúságú, egészeket tartalmazó adatsorban megkeresi egy adott érték utolsó előfordulási helyét (sorszámát). Írja ki a program azt is, hogy ez az adott érték hányadik előfordulása.

8. Töltsünk fel egy 20 elemű egész számokat tartalmazó tömböt számokkal. Írassuk ki ezek közül sorrendben a három legkisebb értékűt, és azt is, hogy hányadik helyen helyezkednek el!

9. Adott n valós szám. Alakítsuk halmazzá őket (vagyis töröljük azokat az elemeket, amelyek egy bizonyos értéket másodszor, harmadszor stb. tartalmaznak).

10. Írjuk ki a $(1, 2, \dots, n)$ halmaz összes részhalmazát.

11. Adott két halmaz. Számítsuk ki a két halmaz egyesített halmazát, metszetét, különbségét.

12. Adott két halmaz. Ellenőrizzük, hogy részhalmaz-e valamelyik valamelyiknek.

13. Adott n halmaz. Számítsuk ki az n halmaz egyesített halmazát, metszetét, különbségét.

14. Adott két halmaz. Számítsuk ki a Descartes-szorzatukat.

15. Írjuk ki egy adott halmaz összes részhalmazát!

16. Adott egy n elemű számsorozat. Rendezzük növekvő, illetve csökkenő sorrendbe

a) buborékos rendezéssel,

b) minimum-kiválasztásos rendezéssel,

c) maximum-kiválasztásos rendezéssel.

d) Megszámoljuk, hány elem kisebb egy adott elemnél, és így megtaláljuk végleges helyét a rendezett sorozatban.

17. Adott egy n elemű számsorozat. Rendezzük maguk között külön a páros, illetve külön a páratlan pozíciókban lévőket.

18. Adott egy n elemű számsorozat. Rendezzük maguk között külön a pozitívakat, illetve külön a negatívakat.

19. Adott egy n elemű számsorozat és egy k szám (n legyen k többszöröse). Rendezzük úgy a számsorozat első k elemét, hogy növekvő rész-sorozatot alkossanak, a második k darab elemét úgy, hogy csökkenő rész-sorozatot alkossanak, a harmadik k darabot megint növekvő sorrendbe stb.

20. Adott n darab szigorúan növekvő számsorozat. Fésüljük össze őket egyetlen szigorúan növekvő számsorozattá.

21. Adott egy n elemű számsorozat, minden két eleme közé szűrjük be az átlagukat

- segéd tömbbel,
- segéd tömb nélkül.

22. Adott egy n elemű számsorozat. Töröljük ki belőle a prímeket

- segéd tömbbel,
- segéd tömb nélkül.

23. Adott egy n elemű számsorozat. Cseréljük fel egymás között az első minimumot és az utolsó maximumot.

6.2. Karakterláncok (stringek)

A karakterláncok C-ben karaktertömbökben kerülnek eltárolásra (nincs külön string típus, mint például a Pascal nyelvben).

Az alábbi példák sokat elárulnak erről az érdekes típusról:

- `char s0[10];`
- `char s1[10]="alma";`
- `char s2[]="dió";`
- `char s3[10]={'a','l','m','a','\0'};`
- `char s4[]={ 'a','l','m','a','\0' };`
- `char *s5;`
- `char *s6="körte";`
- `char s7[]; /* hibás */`
- `char s8[3]="alma"; /* hibás */`
- `char* s9[]={ "alma", "körte", "dió", "banán" };`

- Helyfoglalás történik egy `s0` nevű 10 elemű karaktertömb számára.
- Helyfoglalás történik egy `s1` nevű 10 elemű karaktertömb számára, amelyben eltárolódik az `"alma"` karakterlánc. Hogyan?

S1

| | | | | | | | | | |
|-----|-----|-----|-----|------|---|---|---|---|---|
| 'a' | 'l' | 'm' | 'a' | '\0' | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Megjegyzések:

- A karakterlánc karakterei – pontosabban az ASCII kódjai – a karaktertömb 0-dik elemével kezdődően kerülnek eltárolásra, és követi őket a

111

- Miként történik a karaktertömbök karakterlánckonstanssal való inicializálása? A C fordító a forráskódban megjelenő összes karakterlánc-onstansról (beleértve a `printf()` és `scanf()` formázósorait is) készít egy *string literálnak* nevezett táblázatot. Például a fenti definíciók közül a helyeseknek (2., 3., 7.) megfelelő táblázatrészlet memóriakiépe a következőképpen képzelhető el:

A 2-es definíció alkalmával a valóságban egy másolat készül a string literáció táblázatában lévő **"alma"** karakterlánkonstansról az **s1** tömbben.

4., 5. Klasszikus tömbinicializálással történik az **s3** és **s4** karaktertömbökben az **"alma"** karakterlánc eltárolása.

7. Csak egy `s6` karakterpointer változó számára történik helyfoglalás, amely megkapja a `"körte"` karakterlánckonstans string literációk táblázatbeli címét. Úgy is mondhatnánk, hogy `s6`-ot ráállítjuk a `"körte"` karakterlánckonstansra. Tehát nem történik karakterlánc-másolás. Mivel `"körte"` karakterlánc*konstans*, ezért amíg `s6` rámutat, hibás lenne a következő inkrementálás: `(*s6)++`

8. Azért helytelen, mert a szögletes zárójelek ([]) azt jelzik, hogy s7-nek tömbnek kell lennie, de sem közvetlen, sem közvetett módon nincs közölve, hogy hány elemű legyen.

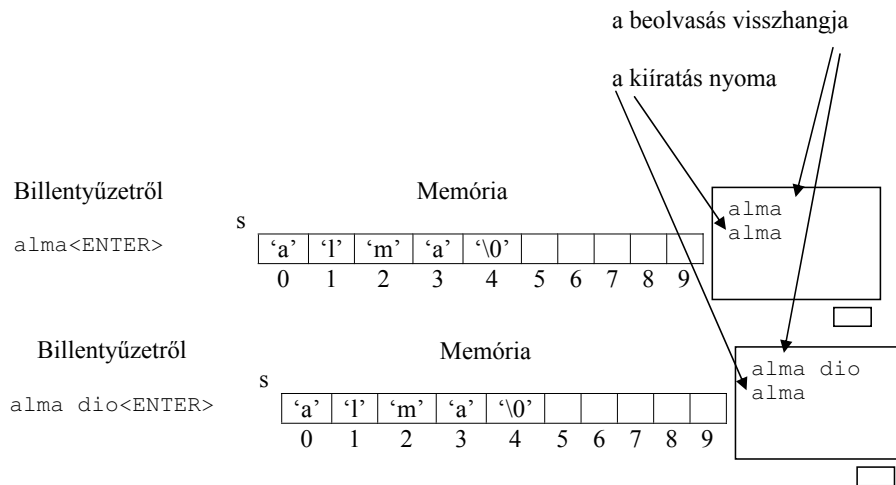
9. Az "alma" karakterlánc eltárolásához legalább egy 5 elemű karaktertömbre lenne szükség.

6.2.1. Karakterláncok beolvasása/kiírása

6.2.1.1. A scanf/printf függvények

Példa:

```
char s[10];
scanf("%s",s);
printf("%s",s);
```



Megjegyzések:

- Figyelem! A `scanf` csak `whitespace`-ig olvas. Tehát általában nem olvasható be `scanf`-fel több szavas karakterlánc.
- Az `fscanf` és `fprintf` hasonlóképpen működnek.

6.2.1.2. A gets/puts függvények

Prototípusa:

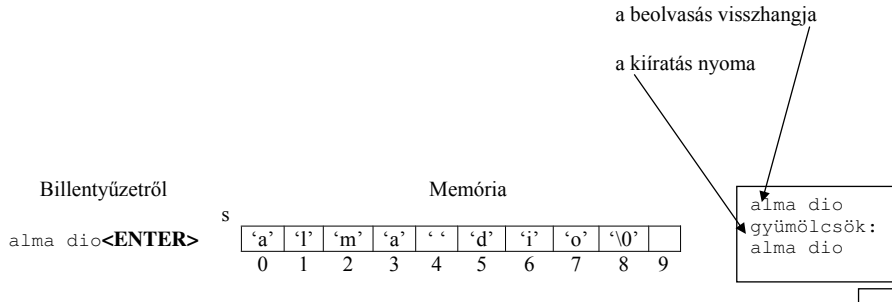
```
char *gets(char *s);
int puts(const char *s);
```

A `gets` `<ENTER>`-ig olvas. Az újsor karakter nem kerül eltárolásra, helyette a `'\0'` karakter kerül a karaktertömbbe.

Példa:

```
char s[10];
gets(s);
puts("gyümölcsök:");
```

```
puts(s);
```



6.2.1.3. A fgets/fputs függvények

Prototípusa:

```
char *fgets(char *s, int h, FILE* fp);
```

```
int fputs(const char *s, FILE* fp);
```

Megjegyzések:

- Az **fgets** sorvégig olvas, és – a **gets**-től eltérően – az újsor karakter is eltárolásra kerül a karaktertömbbe.
- A **h** paraméterrel azt adjuk meg, hogy maximum hány karakter kerüljön beolvasásra. Általában az **s** tömb méretét írjuk ide.
- Az **fputs** hasonlóan működik, mint a **puts**.
- Figyelem! Ha felváltva olvasunk **scanf**-fel és **gets**-szel, illetve **fscanf**-fel és **fgets**-szel, akkor a következő problémával találkozhatjuk szemben magunkat:
 - a **scanf/fscanf** nem olvassa be a sorvégjelt, ezért, ha **gets/fgets** követi, akkor ez ezt a sorvégjelt fogja beolvasni.
 - Egyik módja annak, hogy ezt elkerüljük, hogy a **scanf/fscanf**-fel való olvasás után, egy karakterolvasó függvénnyel (**getchar/fgetc**) kiolvassuk az ottmaradt sorvégjelt.

6.2.2. Karakterlánc-kezelő függvények¹

Megjegyzés. A karakterlánc-kezelő függvények prototípusait a **string.h** header állomány tartalmazza, ezért használatuk szükségessé teszi az **#include<string.h>** programsort.

¹ A karakterláncokat kezelő függvények részletesebb leírása végett, illetve egyéb függvények tanulmányozása érdekében lásd a Help-et.

6.2.2.1. Az `strlen` függvény

Prototípusa:

```
int strlen(const char *s);
```

Az `strlen` függvény visszatéríti az `s` címen kezdődő karakterlanc hosszát.

Példa:

```
char s[10]="alma";
printf("%d\n%d",strlen(s),strlen("dió"));
```

| |
|---|
| 4 |
| 3 |

6.2.2.2. Az `strcpy` függvény

Prototípusa:

```
char *strcpy(char *s1, const char *s2);
```

A `strcpy` függvény az `s2` címen kezdődő karakterláncot átmásolja az `s1` címtől kezdődően. A másolás karakterenként történik, míg átmásolásra kerül a `'\0'` is.

Példa:

```
char s[10], *p, ss[10];
s="alma"; /* helytelen */
p="alma"; /* a p pointert ráállítjuk az "alma"
           karakterlánc konstansra */
strcpy(s,"alma"); /* az s tömbbe bemásolódik az
                  "alma" karakterlanc */
strcpy(ss,s); /* az ss tömbbe átmásolódik az s
              tömbben eltárolt karakterlanc */
strcpy("alma",s); /* helytelen */
```

Megjegyzések:

- Az első két programsorban nem történik karakterlanc-másolás, csupán pointer-hozzárendelés. Az első értékátadás azért hibás, mert `s` mint tömbnév egy pointer *konstans*.
- Az utolsó programsor azért helytelen, mert az `strcpy` első paramétere `char*` pointer kell legyen, az `"alma"` címe viszont `const char*` pointer. Míg a `(char*) -> (const char*)` konverzió helyes, a fordított irányú nem az.

Példa:

```
char s[10];
strcpy(s,"almafa");
strcpy(s+2,s+4); /* törli s-ből a "ma"
                részkarakterláncot */
printf("%s",s);
```



6.2.2.3. Az `strcat` függvény

Prototípusa:

```
char *strcat(char *s1, const char *s2);
```

Az `strcat` függvény az `s2` címen kezdődő karakterláncot az `s1` címen kezdődő karakterlánc után fűzi. Az `s1` karaktertömbnek nyilván elég nagy-
nak kell lennie ahhoz, hogy elférjen benne az `s2` karakterlánc is.

6.2.2.4. Az `strcmp` függvény

Prototípusa:

```
int strcmp(const char *s1, const char *s2);
```

Az `strcmp` függvény összehasonlítja karakterenként az `s1`, illetve `s2` címeken kezdődő karakterláncokat, és visszatérít negatívát, nullát vagy pozitívát attól függően, hogy lexikografikus sorrendben `s1` van előbbre, `s1` azonos `s2`-vel vagy `s2` van előbbre.

6.2.2.5. Az `strstr` függvény

Prototípusa:

```
char *strstr(const char *s1,const char *s2);
```

Az `s2` címen kezdődő karakterláncot megkeresi az `s1` címen kezdődő karakterláncban. Ha megtalálja, visszatéríti annak az `s1`-beli karakternek a címét, amelytől kezdődően `s2` először megtalálható `s1`-ben. Sikertelen keresés esetén a `NULL` pointert téríti vissza.

6.2.2.6. Az `strchr` függvény

Prototípusa:

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

Az **strchr** függvény megkeresi a **c** karaktert az **s1** címen kezdődő karakterláncban. Ha megtalálja, visszatéríti annak az **s1**-beli karakternek a címét, amely a **c** első előfordulási helye. Sikertelen keresés esetén a **NULL** pointert téríti vissza.

6.2.2.7. Az **sscanf**/**sprintf** függvények

Hasonló módon működnek, mint az **fscanf**/**fprintf** függvények, azzal a különbséggel, hogy az **sscanf** nem állományból olvas, hanem karakterláncból, és az **sprintf** nem állományba ír, hanem karakterláncba.

Gyakran arra használjuk őket, hogy számot karakterláncná alakítsunk és fordítva. Ezt mutatják be az alábbi példák is.

6.3. feladat. Adott egy természetes szám az **int** tartományból. Számoljuk meg, hány 5-ös számjegyet tartalmaz a négyzete.

```
int n, k=0, h, i;
unsigned long nn;
char sn[11];
scanf("%d", &n);
nn=n*n;
sprintf(sn,"%lu", nn);
h=strlen(sn);
for(i=0;i<h;i++)
    if(sn[i]=='5')k++;
printf("k=%d", k);
```

Magyarázat: A **scanf** függvény a billentyűzetről karakterláncként beütött számot **int** típusú egész számmá alakítja, hogy négyzetre emelhető legyen. Mivel "könnyebb" egy karakterláncban megszámolni adott tulajdonságú karaktereket, mint egy egész számot számjegyenként feldolgozni, ezért az **sprintf** függvényt úgy használtuk, mint "**int**→**karakterlánc** konvertert".

6.4. feladat. Egy számot úgy kódolunk, hogy négyzetre emeljük, majd minden számjegyet helyettesítjük az angol ábécé annyiadik nagybetűjével, amely a számjegy értékének felel meg. Írjuk meg a dekódoló programot. A szám négyzete belefér az **unsigned long** tartományba.

```
int n, i;
unsigned long x;
char kod[11];
gets(kod);
```



```
n=strlen(kod);
for(i=0;i<n;i++)
    kod[i]='A'-'0';
sscanf(kod,"%lu",&x);
x=(unsigned long)sqrt((double)x);
printf("A szam=%lu", x);
```

Magyarázat: Ha a kód bármelyik elemének ASCII kódjából kivonjuk az ('A'-'0') értéket, akkor megkapjuk az illető karakternek megfelelő számjegyet, mint karakternek, az ASCII kódját. Kihasználtuk, hogy az sscanf függvénnyel egy karakterlánc számmá alakítható.

6.2.2.8. Megoldott feladatok

Megjegyzés. A feladatokban „mondat”-on olyan egysoros karaktersort értünk, amelyben a „szavak” szóközzel vannak elválasztva.

1. Írjunk programot, amely beolvas a billentyűzetről egy többszavas mondatot, és készít egy karakter-előfordulási statisztikát.

Ötlet: Létrehozunk egy 256 elemű `int` típusú tömböt (legyen a neve `st`), amelynek minden eleme számlálóként fog működni. Az `st[i]` elem az `i` ASCII kódú karaktereket fogja számolni.

```
int n, i, st[256]={0};
char s[100];
gets(s);
n=strlen(s);
for(i=0;i<n;i++)
    st[s[i]]++;
for(i=0;i<256;i++)
    if(st[i])
        printf("%c: %d",i,st[i]);
```

2. Írjunk programot, amely beolvas a billentyűzetről egy többszavas mondatot, és kiírja a palindrom (előlről olvasva ugyanaz, mint hátulról) szavakat egymás alá.

Ötlet: Beolvassuk a mondatot `gets`-szel, majd kiolvassuk belőle a szavakat `sscanf`-fel.

```
int n, i, j;
char s[100],szo[30];
gets(s);
strcat(s," ");
while(sscanf(s,"%s",szo) != EOF)
{
```

```
n=strlen(szo);
for(i=0,j=n-1;;i++,j--)
    if(szo[i]!=szo[j]) break;
    else if(i>=j){puts(szo);break;}
strcpy(s,s+n+1); /* törlöm a kiolvasott szót */
}
```

Üres karakterláncból való olvasáskor az **sscanf** függvény **-1**-et, azaz az (EOF) konstanst téríti vissza. Ha a karakterlánc nem üres, akkor a sikeresen kiolvasott „szavak” („szó”-n itt egy olyan részkarakterláncot értünk, amely nem tartalmaz whitespace karaktert) száma lesz a függvény visszatérő értéke.

3. Írjunk programot, amely beolvas a billentyűzetről egy többszavas mondatot, és kiírja a **szavak.ki** állományba a szavakat egymás alá ábécé-sorrendben.

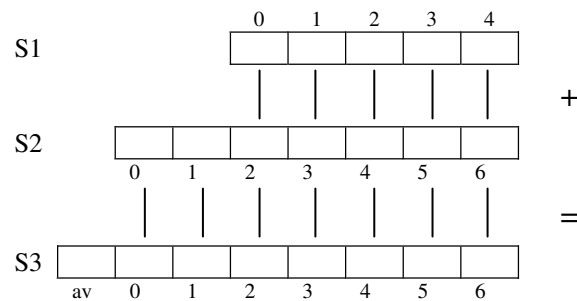
Ötlet: A mondatot „feldaraboljuk” úgy, hogy minden szó külön karakterlánc legyen. Minden szóra ráállítunk egy **char**-pointert. Rendezzük a pointereket aszerint, hogy milyen szóra mutatnak. Ezen módszer előnye abban áll, hogy a rendezés tárgyát képező entitások nem kerülnek „mozgatásra”. Különösen hasznos ez, ha az entitások mérete nagy.

```
FILE*f;
int n, i, j, k;
char s[100],*p[20],*v;
f=fopen("szavak.ki","w");
gets(s);
n=strlen(s);
p[0]=s;k=1;
/* a mondat "feldarabolása" */
for(i=1;i<n;i++)
    if(s[i-1]==' ')
        {p[k++]=s+i;s[i-1]='\0';}
/* a pointerek rendezése */
for(i=1;i<k-1;i++)
    for(j=i+1;j<k;j++)
        if(strcmp(p[i],p[j])>0)
            {v=p[i];p[i]=p[j];p[j]=v;}
/*a szavak kiírása ábécésorrendben
az állományba */
for(i=1;i<k;i++)
    {fputs(p[i],f);fputc('\n',f);}
fclose(f);
```

4. Írjunk programot, amely beolvas a billentyűzetről két, legtöbb 50 számjegyű természetes számot, és kiírja az összegüket.

```
int n, n1, n2, i1, i2, i3, av;
char s1[51], s2[51], s3[51];
gets(s1); gets(s2);
n1=strlen(s1);
n2=strlen(s2);
n=n1>n2?n1:n2;
av=0; /* átvitel */
for(i1=n1-1,i2=n2-1,i3=n-1;i1>=0&& i2>=0;i1--,i2--,i3--)
{
    s3[i3]=((s1[i1]-'0')+(s2[i2]-'0')+av)%10+'0';
    av=((s1[i1]-'0')+(s2[i2]-'0')+av)/10;
}
for(;i1>=0;i1--,i3--)
{
    s3[i3]=((s1[i1]-'0')+av)%10+'0';
    av=((s1[i1]-'0')+av)/10;
}
for(;i2>=0;i2--,i3--)
{
    s3[i3]=((s2[i2]-'0')+av)%10+'0';
    av=((s2[i2]-'0')+av)/10;
}
s3[n]='\0'; // az s3 karakterlánc végére betesszük a karakterlánc-végjelet
if(av==1) putchar('1'); /* ha van átvitel kiírjuk */
puts(s3);
```

Magyarázat: Az `s1` és `s2` karaktertömbökbe beolvasott számok összegét az `s3` karaktertömbben alakítjuk ki. Az összeadás elvégzéséhez a karakterláncként tárolt összeadandók számjegyeit számmá kell alakítsuk, majd az eredmény-számjegyeket vissza kell alakítsuk karakterekké (hogy eltárolhatók legyenek az `s3` karaktertömbben). Az első ciklus a rövidebb szám hosszával megegyező közös szakaszon végzi el a számjegy-összeadásokat. (lásd a mellékelt ábrát). A következő két ciklus folytatja az összeadást a hosszabb szám fennmaradt számjegyeivel. (A második ciklusba akkor lépünk be, ha `s1` tárolja a hosszabbik számot, a harmadik ciklusba pedig akkor, ha `s2`).



5. Írjunk programot, amely beolvas a billentyűzetről n mondatot, és megszámolja, hányban jelenik meg a 'C' karakter.

```
int n,i,k=0;
char s[100];
scanf("%d",&n);
getchar(); /* kiolvassuk a scanf otthagyta sorvégjelt */
for(i=0 ;i<n ;i++)
{
    gets(s);
    if(strchr(s,'C'))k++;
}
printf("k=%d",k);
```

6. Írjunk programot, amely beolvas a `mondat.be` állományból n mondatot, és kiírja a képernyőre annak a mondatnak a sorszámát, amelyben legelőbbre fordul elő a "C nyelv" kifejezés.

```
FILE*f;
int n,i,k,min_i,min;
char s[100];
f=fopen("mondat.be", "r");
fscanf(f,"%d",&n);
fgetc(f); /* kiolvassuk az fscanf otthagyta sorvégjelt */
min=100;
for(i=1 ;i<=n ;i++)
{
    fgets(s,sizeof(s),f);
    k=strstr(s, "C nyelv")-s; /* k-ba a "C nyelv" s-beli
        pozíciója kerül */
    if(k>0 && k<min){min=k;min_i=i;}
    /* gondolunk arra az eshetőségre is, hogy a "C nyelv"
        karakterlánc nem szerepel a kiolvasott mondatban */
}
fclose(f);
```

6.2. KARAKTERLÁNCOK (STRINGEK)

121

```
printf("A sorszám: %d",min_i);
```

7. Írjunk programot, amely beolvas a **mondatok.be** állományból egy szöveget és kiírja a leghosszabb szót.

Ötlet: Kihasználjuk, hogy az **fscanf** szavanként olvas, valamint azt, hogy az állományvégjel olvasásakor **-1(EOF)**-et térít vissza (különben a sikeresen kiolvasott „szavak” számát).

```
FILE*f;
int n,max=0;
char s[30],s0[30];
f=fopen("mondat.be", "r");
while(fscanf(f,"%s",s) != EOF)
{
    n=strlen(s);
    if(n>max){max=n;strcpy(s0,s);}
}
fclose(f);
puts(s0);
```

6.2.2.9. Kitűzött feladatok

24. Írjunk programot, amely beolvas egy személyi számot (CNP), és kiírja az illető személy nemét és születési dátumát.

25. Melyek helyesek? A feladatot papíron oldjuk meg!

1. char s1[];
2. char s2;
3. char *s3;
4. char s4="A";
5. char s5='A';
6. char *s6="A";
7. char s7[]="A";
8. char s8[1]="A";

26. Írjunk programot, amely egy legtöbb 20 elemű karaktersorozatról eldönti, hogy palindrom-e. Palindromnak olyan szöveget nevezünk, amely balról jobbra és jobbról balra olvasva ugyanaz, a sorközöktől és írásjelektől eltekintve. Például: Géza, kék az ég.

27. Írjunk programot, amely a képernyőről olvas be számokat. A bevitt számot csak akkor fogadjuk el, ha az ténylegesen szám, egész és az előző számtól való eltérés 20%-nál nem nagyobb.

28. Adott két azonos hosszúságú, különböző karaktereket tartalmazó karakterlánc (bármely karakter egyszer fordulhat elő), amelyek egymás anagrammái. Ellenőrizzük az adatok helyességét, majd sokszorozzuk meg az első karakterlánc karaktereit annyiszor, ahányadik pozícióban található az illető karakter a második karakterláncban.

Például: "szilva", "vaszil" => "sssszzzzziiiiilllllvvaa"

29. Egy bemeneti karakterlánc szavakat tartalmaz szóközzel elválasztva. Töröljük a legrövidebb szavakat.

30. Adott két, azonos hosszúságú karakterlánc. Az első betűket, a második számjegyeket tartalmaz. Ellenőrizzük az adatok helyességét, majd az első minden betűjét sokszorozzuk annyiszor, amennyi a második karakterláncból sorszám szerint megfelelő számjegy értéke.

Pl. "abcd" , "3124" => "aaabccddddd"

31. Egy bemeneti karakterlánc bináris számokat tartalmaz szóközzel elválasztva. Készítsünk egy másik karakterláncot, amelyben megőrizzük (szóközzel elválasztva) azon bináris számok tízes számrendszerbeli alakját, amelyeknek bináris alakja páros számú 1-est tartalmaz.

Például: "10010 1101 10001 111 1000 1100" => "18 17 12"

32. Egy bemeneti karakterláncban helyettesítsük egy adott karakter többszöri előfordulásait egy másik, adott, karakterláncsal.

33. Olvassunk be a billentyűzetről szavakat, és építsünk fel egy karakterláncot, amely betűrendi sorrendben tartalmazza a szavakat.

34. Adott egy szöveg. Ha a szöveg tartalmaz számjegyeket, változtassuk meg úgy, hogy minden számjegyet annyiszor sokszorozzunk meg, mint amennyi a számjegy értéke.

35. Adott egy szöveg. Ha a szöveg tartalmaz számjegyeket, helyettesítsünk a szövegben minden számjegyet annyi (*) karakterrel, mint amennyi a számjegy értéke.

36. Adott egy mondat. Írjuk át „madárnyelvre” (minden <magánhangzó>-t helyettesítsünk a <magánhangzó>p<magánhangzó> karakterláncsal).

Például: "Jó reggelt" => "Jópó repeggepelt".

37. Adott egy karakterlánc, amely szavakat tartalmaz szóközzel elválasztva. Tükrözzük külön mindenik szót, és írjuk ki az így nyert karakterláncot.

6.3. Kétdimenziós tömbök

Egy kétdimenziós tömb úgy képzelhető el, mint egy táblázat, amelynek sorai és oszlopai vannak. Mind a sorok, mind az oszlopok indexelve vannak, hogy lehessen hivatkozni rájuk.

A definiálás szintaxisa:

```
<elem_típus> <név>[<sorok_száma>][<oszlopok_száma>;
```

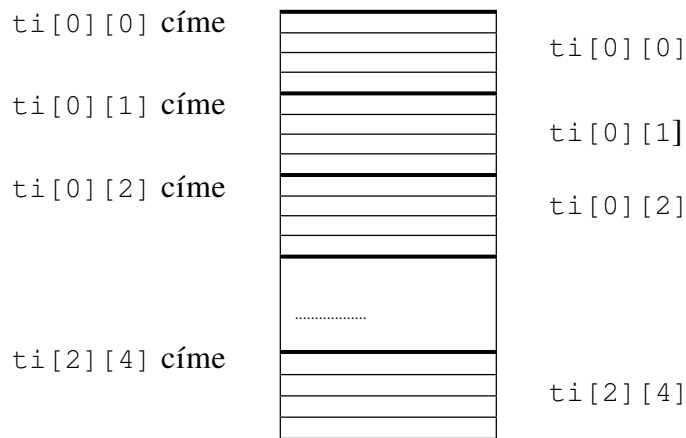
Példa:

```
char tc[20][20];
int ti[3][5];
long double tld[10][10];
```

A `ti` tömböt például így lehetne ábrázolni:

| ti | 0 | 1 | 2 | 3 | 4 |
|----|---|---|---|---|---|
| 0 | | | | | |
| 1 | | | | | |
| 2 | | | | | |

A 6.2. ábra a `ti` tömb memóriaképét szemlélteti (sorfolytonos ábrázolás). (Feltételeztük, hogy az `int` típusú adatok 4 byte-on vannak tárolva.)



6.2. ábra.

A kétdimenziós tömbök számára lefoglalt tárhely méretét a következő képlettel lehet kiszámítani:

`<sorok_száma> * <oszlopok_száma> * sizeof(<elem_típus>)`

Például a fent definiált tömbök mérete sorra:

400 byte, 15 * sizeof(int) byte, 1000 byte.

A kétdimenziós tömbök elemeire való hivatkozás szintaxisa:

`<név>[<sor_index>][<oszlop_index>]`

A `ti` tömb besatírozott elemeinek azonosítói:

`ti[0][0], ti[1][2], ti[2][4]`

6.3.1. Kétdimenziós tömbök inicializálása definiálásukkor

Példák:

```
int a[2][3]={1, 3, -5, 0, 77, -13};
int b[][3]={1, 3, -5, 0, 77, -13, 9, 1, 34}; /* b
3x3-as méretű lesz */
int c[100][100]={0}; /* minden elem 0-val inicial-
izálódik */
```

A kétdimenziós tömbökbe leggyakrabban mátrixok elemeit tároljuk.

6.5. feladat. Adott egy n soros, illetve m oszlopos egész számokat tartalmazó mátrix. Írjunk programot, amely beolvassa a mátrix elemeit a billentyűzetről egy kétdimenziós tömbbe, majd kiírja az elemeket a képernyőre mátrix alakban.

```
int a[50][50], n, m, i, j;
scanf("%d%d", &n, &m);
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        scanf("%d", &a[i][j]);
for(i = 0; i < n; i++)
{
    for(j = 0; j < m; j++)
        printf("%d ", a[i][j]);
    printf("\n");
}
```

A C nyelvben a kétdimenziós tömbök úgy tekinthetők, mint olyan egydimenziós tömbök, amelyek elemei ugyancsak egydimenziós tömbök.

Például az `int a[7][5];` tömbdefiniálás nyomán `a` egy olyan 7 elemű egydimenziós tömb, amelynek minden eleme 5 elemű egydimenziós tömb.

6.3. KÉTDIMENZIÓS TÖMBÖK

125

Milyen típusú egy kétdimenziós tömb neve mint pointer?

Mivel úgy fogható fel, mint egy olyan egydimenziós tömbnek a neve, amelynek 0-dik eleme a 0-dik sora, ezért például a fennebb definiált **a** egy „5 elemű **int**-tömb-pointer konstans”.

Hogyan lehet „leírni” egy ilyen összetett típust?

Általános szabály:

Ha szeretnéd „leírva látni” valamely típust, definiálj egy ilyen típusú változót, majd töröld a definiálásból a változó nevét.

Példák:

```
int v;           /* v int-változó,
                  típusa: int */
int *p;          /* p int-pointer,
                  típusa: int* */
int t[10];       /* t 10 elemű int-tömb,
                  típusa: int[10] */
int *pt[10];     /* pt 10 elemű int-pointer-tömb,
                  típusa: int*[10] */
int (*tp)[10];   /* tp 10 elemű int-tömb-pointer,
                  típusa: int(*)[10] */
int m[5][3];     /* m 5x3-as kétdimenziós int-tömb,
                  típusa: int[5][3] */
int *pm[5][3];   /* pm 5x3-as kétdimenziós int-pointer-tömb,
                  típusa: int*[5][3] */
int (*pm)[5][3]; /* mp 5x3-as kétdimenziós int-tömb-pointer,
                  típusa: int(*)[5][3] */
```

Tehát az **int a[7][5]** kétdimenziós tömb neve **int (*)[5]** típusú pointerkonstans.

Legyen a következő tömbdefiniálás:

```
float b[3][4];
```

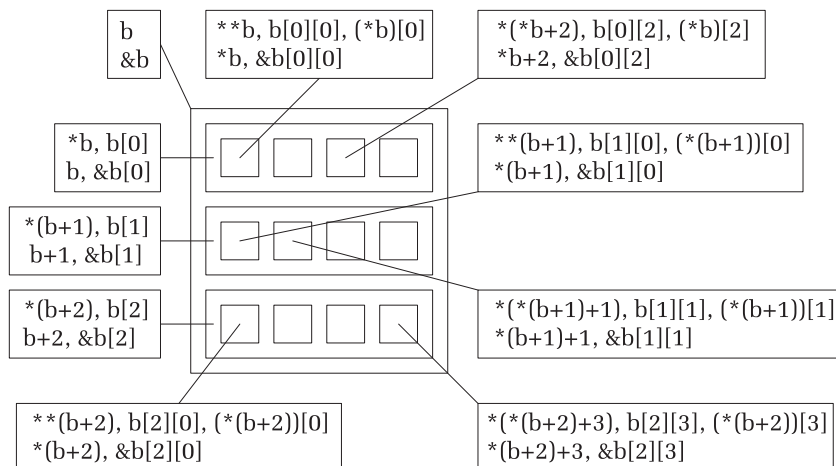
A 6.3. ábra bemutatja, hogyan lehet hivatkozni a nyíllal megjelölt egységekre (felső sor), illetve címekre (alsó sor):

Szabály:

$a[i][j] \Leftrightarrow *(a[i] + j) \Leftrightarrow *((a + i) + j)$

Példa: Az alábbi egymásba ágyazott **for** ciklusok beolvasnak egy $n \times m$ méretű mátrixot az **x** kétdimenziós tömbbe.

```
float x [10][10], (*sp)[10], *ep;
int n, m;
scanf("%d%d", &n, &m);
for(sp = x; sp - x < n; sp++)
```



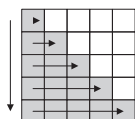
6.3. ábra.

```
for(ep = *sp; ep - *sp < m; ep++)
    scanf("%f", ep);
```

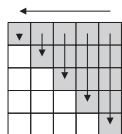
Megjegyzések:

- **sp** (sor-pointer) az **x** tömb egy sorára mutat (10 elemű **float**-tömb-pointer), ezért növelve (**sp++**) sorról sorra lépeget.
- **ep** (elem-pointer) az **x** tömb egy elemére mutat (**float**-pointer), ezért növelve (**ep++**) elemről elemre lépeget.

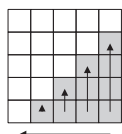
Hogyan írhatók ki a besatírozott területek elemei a nyilakkal megjelölt sorrendben? Feltételezzük, hogy az ábrák $n \times n$ méretű területei egy **a** nevű kétdimenziós tömbnek, és a bal felső sarok indexei **(0,0)**.



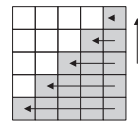
```
for(i=0; i<n; i++)
    for(j=0; j<=i; j++)
        printf("%d ", a[i][j]);
```



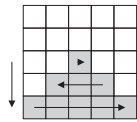
```
for(j=n-1; j>=0; j--)
    for(i=0; i<=j; i++)
        printf("%d ", a[i][j]);
```



```
for(j=n-1; j>0; j--)
    for(i=n-1; i>n-1-j; i--)
        printf("%d ", a[i][j]);
```



```
for(i=n-1; i>=0; i--)
    for(j=n-1; j>=n-1-i; j--)
        printf("%d ", a[i][j]);
```

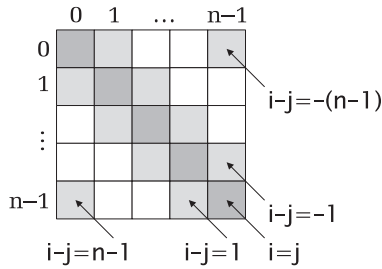


```
for(i=n/2; i<=n-1; i++)
    for(i%2?j=i:j=n-1-i;
        j>=n-1-i && j<=i;
        i%2?j--:j++)
        printf("%d ", a[i][j]);
```

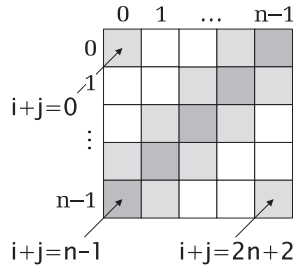
Megjegyzés. Az utolsó példában, páros n esetén, a vízszintes nyilak irányításával fordított irányítású bejárás történik.

6.3.2. Szimmetriák és szabályosságok egy $n \times n$ méretű mátrixban

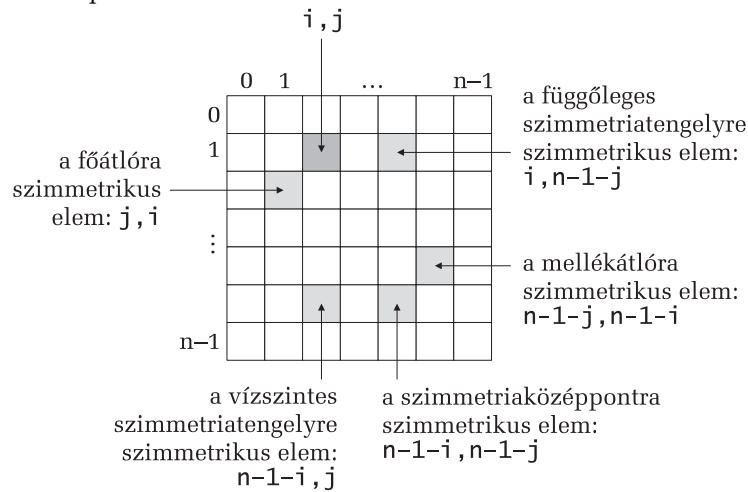
A főátlóval párhuzamos átlók mentén a sorindex (i) és oszlopindex (j) különbsége állandó:



A mellékátlóval párhuzamos átlók mentén a sorindex (i) és oszlopindex (j) összege állandó:



Szimmetriapontok:



6.4. Többdimenziós tömbök

Az egy-, illetve kétdimenziós tömbökhöz hasonlóan definiálhatunk három-, négy-, általánosan n -dimenziós tömböket is.

A tömbdefiniálás általános szintaxisa:

```
<elem_típus> <név>[<méret_1>][<méret_2>]...[<méret_n>];
```

$méret_i$ az i -edik dimenzió irányába az elemszám

A tömb valamely elemére való hivatkozás szintaxisa:

```
<név>[<index_1>][<index_2>]...[<index_n>]
```

6.5. Megoldott feladatok

8. Adott egy $n \times m$ méretű mátrix egy `matrix.be` nevű szöveges állományban. Az első sor tartalmazza az n és m értékeit, a következők pedig a mátrix egy-egy sorát (egy sor elemei szóközzel vannak elválasztva). Írjuk ki a képernyőre a mátrix *nyeregpont*jainak indexeit. Egy pont akkor tekinthető nyeregpontnak, ha sorának minimuma, oszlopának pedig maximuma.

Ötlet: Összeállítunk egy `min`, illetve egy `max` nevű egydimenziós tömböt. A `min[i]` elemben eltároljuk a mátrix i -edik sora minimumának os-

zlopindexét, a `max[j]`-ben pedig a `j`-edik oszlop maximumának sorindexét. Az `i`-edik sor minimuma akkor nyeregpont, ha `i=max[min[i]]`.

```
FILE*f;
int a[50][50], min[50], max[50], n, m, i, j, volt;
f=fopen("matrix.be", "r");
fscanf(f, "%d%d", &n, &m);
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        fscanf(f, "%d", &a[i][j]);
fclose(f);
/* soronkénti bejárás */
for(i = 0; i < n; i++)
{
    min[i]=0;
    for(j = 1; j < m; j++)
        if(a[i][j]<a[i][min[i]])min[i]=j;
}
/* oszloponkénti bejárás */
for(j = 0; j < m; j++)
{
    max[j]=0;
    for(i = 1; i < n; i++)
        if(a[i][j]>a[max[j]][j])max[j]=i;
}
volt=0;
for(i = 0; i < n; i++)
    if(i==max[min[i]])
    {
        printf("(%d,%d)=%d\n",i,min[i],a[i][min[i]]);
        volt=1;
    }
if(!volt) printf("Nincs nyeregpont!");
```

9. Adott `n` mondat egy bementi állományban (`mondat.be`). Az első sor tartalmazza az `n` értékét, a következők pedig egy-egy mondatot. Írjuk ki a mondatokat a `mondat.ki` kimeneti állományba, ábécésorrendben.

Ötlet: A mondatokat egy kétdimenziós karaktertömb egy-egy sorában tároljuk el.

```
FILE*f,*g;
char a[50][50], v[50];
int n, i, j;
f=fopen("mondat.be", "r");
g=fopen("mondat.ki", "w");
fscanf(f, "%d", &n);
fgetc(f);/* kiolvassa az n értéke utáni újsor karaktert */
```

```

for(i = 0; i < n; i ++)  

    fgets(a[i], sizeof(a[i]), f);  

fclose(f);  

/* rendezés soronként */  

for(i = 0; i < n-1; i ++)  

{  

    for(j = i+1; j < n; j ++)  

        if(strcmp(a[i],a[j])>0)  

        {  

            strcpy(v,a[i]);  

            strcpy(a[i],a[j]);  

            strcpy(a[j],v);  

        }  

}  

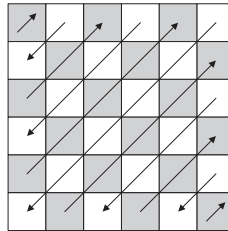
for(i = 0; i < n; i ++)  

    fputs(a[i], g);  

fclose(g);

```

10. Adott egy négyzetes mátrix a szokásos módon a **be.txt** bemeneti állományban. Írjuk ki a képernyőre az elemeit az ábra szerinti bejárési sorrendnek megfelelően.



```

FILE*f; int a[50][50],n,i,j,k;  

f=fopen("be.txt", "r");  

fscanf(f, "%d", &n);  

for(i = 0; i < n; i ++)  

    for(j = 0; j < n; j ++)  

        fscanf(f, "%d", &a[i][j]);  

fclose(f);  

/* átlónkénti bejárás */  

for(k = 0; k <= 2*n-2; k ++)  

    if(k<=n-1) /* mellékátló felett */  

        for((k%2) ? (i=0,j=k) : (j=0,i=k);  

            i>=0 && j>=0;  

            (k%2) ? (i++,j--) : (j++,i--))  

            printf("%d ",a[i][j]);  

    else /* mellékátló alatt */  

        for((k%2) ? (j=n-1,i=k-j) : (i=n-1,j=k-i);  

            i<n && j<n;

```

```
(k%2) ? (i++,j--) : (j++,i--)
printf("%d ",a[i][j]);
```

Magyarázat: A mellékátlóval párhuzamos átlók mentén az $(i+j)$ összeg állandó ezért azonosíthatók ezen érték által. A külső ciklus átlóról átlóra lépeget (a k -adik átló mentén éppen k az $(i+j)$ összeg értéke). A belső ciklus egyetlen átlót jár be:

- A páros sorszámú ($k\%2 == 0$) átlókon felfele ($i--; j++;$), a páratlanokon pedig ($k\%2 == 1$) lefele ($i++; j--;$) haladunk.
- A mellékátló felett ($k \leq n-1$) az első oszlopból ($i=k; j=0;$) indulunk felfele az első sorig ($i \geq 0$), illetve az első sorból ($i=0; j=k;$) indulunk lefele az első oszlopig ($j \geq 0$). Ezért az összevont megállási feltétel a mellékátló felett: $i \geq 0 \ \&\& \ j \geq 0$.
- A mellékátló alatt ($k > n-1$) az $(n-1)$ -edik sorból ($i=n-1; j=k-i;$) indulunk felfele az $(n-1)$ -edik oszlopig ($j < n$), illetve az $(n-1)$ -edik oszlopból ($j=n-1; i=k-j;$) indulunk lefele az $(n-1)$ -edik sorig ($i < n$). Ezért az összevont megállási feltétel a mellékátló felett: $i < n \ \&\& \ j < n$.

6.6. Kitűzött feladatok

(Az alábbi feladatok esetében az adatokat állományból olvassuk be.)

1. Egy $N \times M$ méretű mátrixot a $V[0 \dots N \times M - 1]$ tömbben tárolunk el sorfolytonosan. Tegyük fel, hogy a mátrix i -edik sorának j -edik eleme a tömb k -adik pozíciójába kerül (a sorokat, illetve oszlopokat 0-tól kezdődően számozzuk).

- a) Fejezzük ki k -t, i és j függvényében!
- b) Adjuk meg i -t, illetve j -t, k függvényében!
- c) Ha $N=M$, akkor milyen tömbpozíciókba kerülnek a főátló elemei? (Általánosan fogalmazd meg.)
- d) Ha $N=M$, akkor a mátrix mely elemei kerülnek a tömb azon pozícióiba, amelyek indexeinek N -nel való osztási maradéka maximális? (Általánosan fogalmazd meg.)

2. Írjunk programot, amely beolvas egy 5×5 -ös mátrixot, és kiírja az úgynevezett tükrömátrixát. A tükrömátrix elemeit az eredeti mátrixelemek főátlóra tükrözésével kapjuk. A tükrömátrix az eredeti helyén képződjön.

3. Adott egy $n \times n$ méretű mátrix, amely különböző egész számokat tartalmaz. Ellenőrizzük, hogy bővös négyzet-e. A bővös négyzet minden sorának és oszlopának, valamint átlóinak összege azonos.

4. Töltsünk fel egy $n \times n$ méretű tömböt a következő jelentésű számhármassok alapján:

- sorindex (egész szám az $[1, n]$ intervallumból),
- oszlopindex (egész szám az $[1, n]$ intervallumból),
- érték (valós szám).

A beolvasás 0 végjelig tart.

A program a sorindex és oszlopindex ellenőrzése után az általuk meghatározott helyre írja az értéket. Felülírás nem megengedett! A tömb többi eleme 0 legyen.

5. Egy véletlenszerűen feltöltött $n \times n$ méretű karaktermátrix sorait alkotó karakterekből építsünk karakterláncokat, majd ezeket fűzzük össze betűrendi sorrendbe, szóközzel elválasztva.

6. Egy állományból kiolvasott négyzetes karaktermátrix fő- és mellékátlóin található karakterekből építsünk karakterláncokat. Vizsgáljuk meg, hogy ezek tükörszavak-e, illetve, hogy anagrammák-e.

7. Adott n , m és egy $n \times m$ -es mátrix. Döntsük el, hogy tartalmaz-e egynél több nullát a főátló felett vagy a mellékátló alatt.

8. Olvassunk be egy négyzetes mátrixot, amely valós számokat tartalmaz. Állítsuk elő azt a mátrixot, amelyet úgy nyerünk, hogy minden elemét helyettesítjük a float típusnak megfelelő belső ábrázolás 0-ás és 1-es bitjei számának különbségeivel.

9. Adott egy mátrix. Ellenőrizzük, mely sorai vannak növekvő sorrendben, és mely oszlopai csökkenőben.

10. Adott egy mátrix. Rendezzük a sorait külön-külön növekvő sorrendbe.

11. Adott egy mátrix. Rendezzük az oszlopaikat külön-külön csökkenő sorrendbe.

12. Adott egy mátrix. Rendezzük át az elemeit úgy, hogy sorfolytonos bejárás szerint csökkenő sorrendben legyenek.

13. Adott egy mátrix. Írjuk ki az elemeit csigabejárás, illetve fordított csigabejárás szerint. Oldjuk meg a feladatot egyetlen ciklus-utasítást használva a bejáráshoz.

14. Írjunk programot, amely kiszámítja egy n -ed rendű determináns értékét.

Ötlet: a főátló alatt képezzünk nullákat (Gauss-módszer).

15. Írjunk programot, amely egy iteratív algoritmus révén – használva egy megfelelő c tömböt – kiszámítja

a) a $c[1][n]$ értéket az alábbi képlet alapján:

$$\begin{aligned} c[i][i] &= 0, \text{ bármely } i \in \{1, 2, \dots, n\} \\ c[i][j] &= \min_{i \leq k < j} \{c[i][k] + c[k+1][j] + d[i-1] * \\ &\quad d[k] * d[j]\} \\ &\text{bármely } i, j \in \{1, 2, \dots, n\} \text{ és } i < j; \text{ ahol ismert a} \\ &\quad d[0..n] \text{ tömb tartalma} \end{aligned}$$

b) a $c[n][m]$ értéket az alábbi képlet alapján:

$$\begin{aligned} c[i][0] &= 0, \text{ bármely } i \in \{1, 2, \dots, n\} \\ c[0][j] &= 0, \text{ bármely } j \in \{1, 2, \dots, m\} \\ c[i][j] &= c[i-1][j-1] + 1 \text{ ha } a[i] = b[j] \\ c[i][j] &= \min \{c[i-1][j], c[i][j-1]\} \text{ ha} \\ &\quad a[i] \neq b[j] \text{ bármely } i \in \{1, 2, \dots, n\} \text{ és } j \in \{1, \\ &\quad 2, \dots, m\}; \text{ ahol az } a[1..n] \text{ és } b[1..m] \text{ tömbök} \\ &\quad \text{tartalma ismertnek tekintendő} \end{aligned}$$

c) a $\max_{1 \leq k \leq n} \{c[k]\}$ értéket az alábbi képlet alapján:

$$\begin{aligned} c[n] &= 1 \\ c[i] &= \max_{j=i+1, n} \{1 + c[j] \mid a[i] \leq a[j]\} \\ &\text{bármely } i \in \{1, 2, \dots, n-1\}; \text{ ahol ismert az} \\ &\quad a[1..n] \text{ tömb tartalma} \end{aligned}$$

16. Adott egy négyzetes mátrix. Számítsuk ki a determinánsa értékét!

17. Adott n darab számsorozat. Mindenikben határozzuk meg a leghosszabb egymásutáni elemekből álló szigorúan növekvő részsorozatot, majd fésüljük össze ezeket egyetlen növekvő sorozattá.

18. Egy $n \times m$ méretű mátrix sorai egész elemű halmazokként értelmez-
zük. Határozzuk meg az n halmaz egyesítését, illetve metszetét.

19. Adott N darab $n \times n$ méretű négyzetes mátrix. Határozzuk meg azon mátrixok összegét, amelyek determinánsa nem nulla, illetve azt a mátrixot, amelyiknek determinánsa maximális értékű.

20. Adott n darab halmaz. Jelölje $I(M)$ azon halmazok számát, amelyek valódi részhalmazként tartalmazzák az M halmazt. Határozzuk meg azt az M halmazt, amely esetén maximális az $I(M)$ értékű

21. Adottak egy $n \times m$ méretű ritka-mátrix nem nulla elemei a (sorindex, oszlopindex, érték) hármassok által. Adjunk össze, illetve szoroz-
zunk össze két ily módon megadott mátrixot.

22. Adott egy n elemű természetes számsorozat. Készítsünk számjegy-
elfordulási statisztikát.

23. Adott egy n elemű természetes számsorozat. Írjuk ki az egymás után következő tükör-számpárokat. (Például: 3251, 1523)

24. Adott egy n elemű természetes számsorozat. Írjuk ki az összes maximális hosszú egymásutáni elemekből álló csupa prímet tartalmazó részsorozatot.

25. Adott egy n elemű természetes számsorozat. Írjuk ki az összes maximális hosszú egymásutáni elemekből álló részsorozatot, amelyekben a szomszédos elemek "rokon számok" (ugyanazokat a számjegyeket tartalmazzák; Például: 12545, 5412).

26. Adott egy n elemű természetes számsorozat. Írjuk ki az összes maximális hosszú egymásutáni elemekből álló részsorozatot, amelyekben a szomszédos elemek "idegen számok" (nem tartalmazznak egyetlen közös számjegyet sem; Például: 12545, 7893).

27. Adott egy $n \times n$ méretű négyzetes mátrix (legyen a neve A). Határozzuk meg az A, A^2, A^3, \dots, A^n mátrixokat, illetve ezek összegét.

28. Adott egy n -ed fokú egész együtthatójú polinom vagy a monomai által vagy foksám és együtthatók által. Határozzuk meg a racionális gyökeket és ezek multiplicitását.

29. Legyen az alábbi algoritmus:

```

0.  ALGORITMUS
1.  beolvas n
2.  minden i = 1,n végezd
3.    beolvas v[i]
4.    nrc[i] = 0
5.  vége minden
6.  minden i = 1,n-1 végezd
7.    minden j = i+1,n végezd
8.      ha v[i] > v[j]
9.        akkor nrc[i]++
10.       különben nrc[j]++
11.     vége ha
12.   vége minden
13. vége minden
14. VÉGE ALGORITMUS
    
```

a) Mennyi lesz az $nrc[3]$ értéke, ha az 5,7,3,1,8,10 értékeket olvasunk be?

b) Mennyi lesz az $nrc[3]$ értéke, ha n -nek 5-öt olvasunk be, a v vektorba pedig azonos értékeket?

- c) Ha azonos értékeket olvasunk be a \mathbf{v} vektorba, milyen szabályosságot fognak mutatni az \mathbf{nrc} tömb elemei?
- d) Ha $n=7$, hányszor hajtódik végre a **ha** utasítás?
- e) Fogalmazd meg tömören, mit valósít meg az algoritmus!
- f) Hogyan fognak megváltozni az a) – e) pontokra adott feleletek, ha a **ha** utasítás feltételét $\mathbf{v[i] \geq v[j]}$ változtatjuk?

30. Legyen az alábbi algoritmus:

0. ALGORITMUS

1. beolvas n

2. minden $i = 1, n$ végezd

3. minden $j = 1, n$ végezd

4. $\mathbf{a[i][j] = i + j}$

5. vége minden

6. vége minden

7. minden $i = 1, n$ végezd

8. $\mathbf{a[i][i] = a[i][i] \% a[6][4]}$

9. vége minden

10. VÉGE ALGORITMUS

- a) Az \mathbf{a} tömb hány eleme lesz 3-mal egyenlő értékű az algoritmus végén, ha n -nek 10-et olvasunk be?
- b) Ha n -nek 10-et olvasunk be, mennyi lesz a tömb legnagyobb elemének értéke az algoritmus végén?
- c) Ha n -nek 10-et olvasunk be, hány maximális eleme lesz a tömbnek az algoritmus végén?
- d) Az \mathbf{a} tömb hány eleme lesz 10 az 1–6 programsorok végrehajtása után, ha n -nek 7-et olvasunk be?
- e) Milyen halmazból vesznek értéket a főátló elemei az algoritmus végén, ha $n \geq 6$ értéket olvasunk be?
- f) Mi mondható el a mellékátló elemeiről az algoritmus végén, ha n -nek páros értéket olvasunk be?

7. FEJEZET

DINAMIKUS HELYFOGLALÁS

Feladataink bemenő adatai gyakran számsorozatok vagy mátrixok voltak, amelyeket egy-, illetve kétdimenziós tömbökbe kellett eltárolnunk. Hogyan határoztuk meg ezek méreteit? Mivel a számsorozat elemszáma (**n**), illetve a mátrix mérete (**n** × **m**) csak programfutás közben, az **n**, illetve az **n** és **m** beolvasása után derült ki, ezért a tömböknek olyan méretet írtunk elő, hogy a „legrosszabb esetben” is elég nagyok legyenek. Ha lehetőség lenne programfutás közben létrehozni változókat, sokkal gazdaságosabban ki lehetne használni ilyen esetekben a memóriát. Éppen ezt a szükségletet elégítik ki a *dinamikus változók*. A dinamikus helyfoglalásra a memória Heap-nek nevezett részén kerül sor.

A dinamikus helyfoglalást a **malloc**, **calloc** és a **realloc** függvények végzik, a helyfelszabadítást pedig a **free**.

Szintaxisuk:

```
void* malloc(<méret bájtban>);
```

Lefoglalja a megadott méretű memóriaterületet, és visszatéríti a címét mint **void**-pointert.

```
void* calloc(<elemszám>, <elemméret bájtban>);
```

Lefoglal és lenulláz **<elemszám> * <elemméret>** méretű memóriaterületet, és visszatéríti a címét mint **void**-pointert.

```
void* realloc(<átméretezendő blokk címe>, <új méret bájtban>);
```

Átméretez egy **malloc**-kal vagy **calloc**-kal előzőleg lefoglalt memóriaterületet. Területnövelés esetén először megpróbálja kiterjeszteni a jelenlegi blokkot. Ha ez nem lehetséges, akkor új területet foglal le (a régit felszabadítja), ahova átmásolja a jelenlegi terület tartalmát. Visszatéríti az új méretű blokk címét, ami különbözhet az eredeti címtől.

```
free(<cím>);
```

Felszabadítja a megadott címen lévő memóriaterületet. Fontos, hogy egy **malloc** vagy **calloc** függvény által megadott címet kapjon a **free** függvény.

Megjegyzések:

- A `malloc` és `calloc` között az alapvető különbség az, hogy az első mint egyetlen blokk, a második pedig mint adott számú blokk kapja meg a lefoglalandó terület méretét.
- A dinamikus helyfoglalást kezelő függvények prototípusait a `stdlib.h` header állomány tartalmazza, ezért használatuk szükségessé teszi az `#include<stdlib.h>` programsort.

Példa: Hozzunk létre dinamikusan egy `n` elemű `int` típusú tömböt, majd szabadítsuk fel a lefoglalt területet.

Első megoldás:

```
int n, *p;
scanf("%d", &n);
p=(int*)malloc(n*sizeof(int));
...
free(p);
```

Második megoldás:

```
int n, *p;
scanf("%d", &n);
p=(int*)calloc(n, sizeof(int));
...
free(p);
```

Megjegyzés. Emlékezzünk rá, hogy a `p` pointer használható a tömb nevéként, tehát hivatkozhatunk a tömb elemeire a megszokott módon, indexeléssel. Például: `p[0]`, `p[1]`, `p[i]`, `p[n-1]`

Az alábbi példák azt mutatják be, hogyan történhet a dinamikus helyfoglalás bizonyos esetekben, és hogyan hivatkozhatunk a létrejött dinamikus változóra, illetve az elemeire (tömbök esetén).

Dinamikus `double` változó:

```
double* p;
p=(double*)malloc(sizeof(double));
```

Hivatkozás: `*p`

Dinamikus helyfoglalás konstans elemszámú egydimenziós tömb számára:

```
#define N 100
double* p;
p=(double*)malloc(N*sizeof(double));
```

Hivatkozás a tömb elemére: `p[i]`

Dinamikus helyfoglalás n elemű egydimenziós tömb számára:

```
int n;
double* p;
scanf("%d", &n);
p=(double*)malloc(n*sizeof(double));
```

Hivatkozás a tömb elemére: $p[i]$

Dinamikus helyfoglalás konstans méretű kétdimenziós tömb számára:

```
#define N 100
#define M 100
double(*p)[M];
p=(double(*)[M])malloc(N*M*sizeof(double));
```

Hivatkozás a tömb elemére: $p[i][j]$

Dinamikus helyfoglalás n soros, de konstans oszlopszámú kétdimenziós tömb számára:

```
#define M 100
int n;
double(*p)[M];
scanf("%d", &n);
p=(double(*)[M])malloc(n*M*sizeof(double));
```

Hivatkozás a tömb elemére: $p[i][j]$

Dinamikus helyfoglalás $n \times m$ méretű kétdimenziós tömb számára.

Első megoldás (úgy fogjuk fel, mint egydimenziós tömböt):

```
int n,m;
double* p;
scanf("%d%d", &n, &m);
p=(double*)malloc(n*m*sizeof(double));
```

Hivatkozás a kétdimenziós tömb (i,j) elemére: $p[i * m + j]$

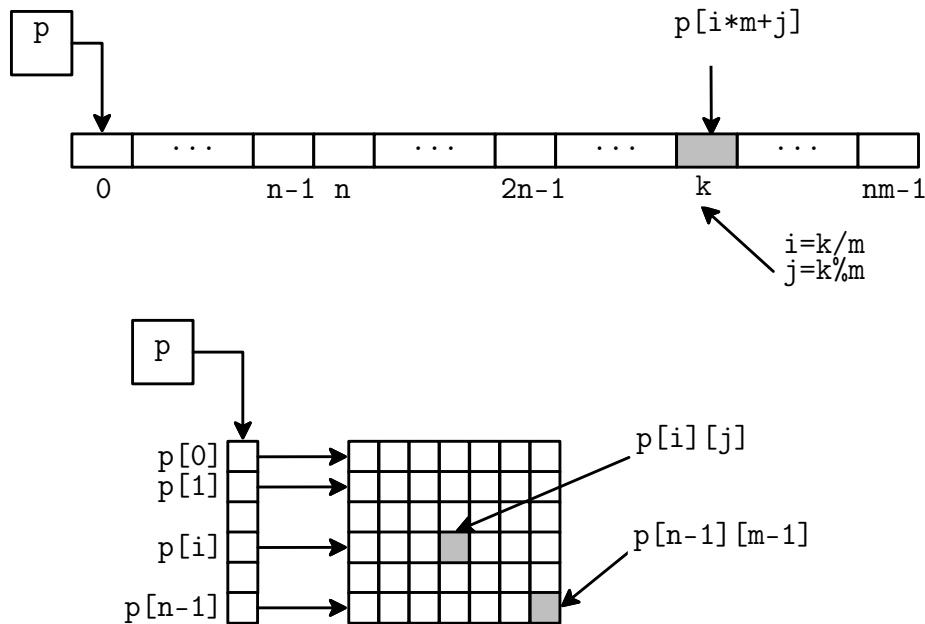
Fordítva, ha k egy elem indexe az egydimenziós tömbben, akkor ennek a kétdimenziós tömbbeli indexei: $i=k/m$, $j=k\%m$

Második megoldás (létrehozunk egy n elemű **double**-pointer tömböt, amelynek minden eleme mutat egy m elemű **double** tömbre):

```
int n,m,i;
double ** p;
scanf("%d%d", &n, &m);
p=(double**)malloc(n*sizeof(double*));
for(i=0;i<n;i++)
    p[i]=(double*)malloc(m*sizeof(double));
```

Hivatkozás a kétdimenziós tömb (i,j) elemére: $p[i][j]$

Az ábrák szemléletesen mutatják be az utolsó példa kétféle megoldását:



Megjegyzések:

- Ahhoz, hogy „igazi” kétdimenziós tömböt tudjunk létrehozni dinamikusan, szükséges az oszlopszám rögzítése (konstansnak kell lennie). Ez azért van így, mert egy „igazi” kétdimenziós tömb neve csakis olyan pointer lehet, amely a tömb 0-dik sorára mutat, tehát a típusa tartalmazza az oszlopszámot.
- Az utolsó példában különböző trükkökhöz folyamodtunk, hogy $n \times m$ méretű kétdimenziós tömböt hozzunk létre. Az első megoldás előnye, hogy nem igényel plusz memóriaterületet, hátránya viszont, hogy nehézkesen lehet hivatkozni a tömb elemeire. A második megoldás esetén a klasszikus módon tudunk hivatkozni az elemekre, de n mutatónyi memóriaterületet pluszban foglaltunk le. A második megoldás másik előnye, hogy a soroknak nem föltétlenül kell egyforma hosszúaknak lenniük.

7.1. feladat. Adott n darab különböző hosszúságú számsorozat a `sorozat.be` szöveges állományban, amelyeknek elemei egész számok. Haladjunk végig a számsorozatokon a következő algoritmus szerint:

- Indulunk az első számsorozat első elemétől.
- Ha pozitív elemen állunk, akkor jobbra lépünk, ha negatívon, akkor balra.
- Ha nullás elemhez jutottunk, vagy kiértünk a számsorozat valamelyik végére, akkor átlépünk a következő számsorozat ugyanazon indexű elemére (amennyiben ez létezik), ahol hasonló módon járunk el.

Írjunk programot, amely eldönti, hogy ki lehet-e jutni a fenti algoritmus szerint a számsorozatokból, és ha igen, hány lépést igényel a kijutás.

Példa:

| | | | | | | |
|---|----|----|----|----|---|---|
| | | | | | | |
| ↓ | -3 | | | | | |
| | 2 | 4 | 7 | 0 | 8 | 3 |
| | 1 | -9 | 5 | 0 | 4 | 4 |
| | -9 | -1 | -1 | -1 | 1 | 1 |
| ↓ | | | | | | |

```
FILE*f;
int *N, **p, n, i, j, k, kem;
f=fopen("sorozat.be", "r");
fscanf(f, "%d", &n); /* n-a sorozatok száma */
N=(int*)malloc(n*sizeof(int));
p=(int**)malloc(n*sizeof(int*));
for(i = 0; i < n; i ++)
{
    fscanf(f, "%d", &N[i]);
    p[i]=(int*)malloc(N[i]*sizeof(int));
    for(j = 0; j < N[i]; j ++)
        fscanf(f, "%d", &p[i][j]);
}
fclose(f);
i=j=k=kem=0;
while(i<n)
{
    if(!j&&N[i][j]<0||j==N[i]-1&&p[i][j]>0||!p[i][j])
        {i++; k++; if(j>=N[i]){kem=1;break;}}
    if(j>0&&p[i][j]<0&&p[i][j-1]>0||
        j<N[i]-1&&p[i][j]>0&&p[i][j+1]<0){kem=1;break;}
    else {p[i][j]>0?j++:j--; k++;}
}
if(kem==1)puts("Nincs megoldás!");
else printf("%d lépés", k);
```


7. DINAMIKUS HELYFOGLALÁS

141

```
for(i = 0; i < n; i++) free(p[i]);  
free(p); free(N);
```

Magyarázat:

- **n** – a számsorozatok száma.
- Az **N** dinamikus tömbben tároltam a számsorozatok hosszát.
- **p** egy **n** elemű dinamikusan létrehozott pointertömb, amelynek elemei azokra a dinamikus tömbökre mutatnak, ahol a számsorozatok eltárolásra kerültek. A **p[i]** című (nevű) tömb **N[i]** elemszámú.
- **k**-ban számoljuk a lépéseket.
- A **kem** változó figyel, nem kerültünk-e olyan helyzetbe, amelyet végtelen ide-oda lépegetés követne.
- **i** – az aktuális számsorozat sorszáma.
- **j** – az aktuális sorban az aktuális elem oszlopszáma.

8. FEJEZET

A FELHASZNÁLÓ ÁLTAL DEFINIÁLT TÍPUSOK

8.1. A **typedef** módosító jelző

A **typedef** nem hoz létre új típust, csupán átnevez vagy nevet ad egy típusnak.

Ha egy változódefiniálás elé odaírjuk a **typedef**-et, akkor azt típusdefiniálássá változtatja.

Szintaxisa:

```
typedef <típus> <azonosító>;
```

Példák:

```
int egész; /* az egész egy int típusú változó */
typedef int egész; /* az egész egy típus, az int típus
    egy újabb neve */
egész x; int y; /* mind x, mind y int típusú változó */
typedef float * FLOATPTR;
FLOATPTR fp; /* fp, float-pointer-változó */
typedef int IVEKTOR[10];
IVEKTOR a; /* a, 10 elemű int egydimenziós tömbvál-
    tozó */
typedef double DMATRIX[10][20];
DMATRIX b; /* b, 10 × 20-as double tömbváltozó */
```

Amint a fenti példákból is kiderül, a **typedef** igazi haszna abban áll, hogy nevet adhatunk általa az összetett típusoknak. Eddig egyetlen összetett típussal ismerkedtünk meg, a tömbtípussal. Következzenek a C nyelv más összetett típusai.

8.2. A **struct** típus

Tegyük fel, hogy egy személy adatait (például, név: Nagy Péter, életkor: 35, magasság: 1.82) szeretnénk eltárolni. Természetesen definiálnánk ebből a célból három változót. Mivel egy személy neve karak-

8.2. A STRUCT TÍPUS

143

terlanc, életkora természetes szám és magassága méterben kifejezve valós szám, ezek a változók lehetnének, például:

```
char nev[30];
unsigned életkor;
float magassag;
```

A fenti megoldásnak az a hátránya, hogy nem tükröződik benne az, hogy a három adat összetartozik, ugyanannak a személynek az adatai.

Három elemű tömb szóba sem jöhet, hiszen egy tömb elemeibe csakis azonos típusú adatok tárolhatók el.

Mi a megoldás? A **struct** típus!

A **struct** típus egy összetett típus, amelynek elemeit mezőknek nevezzük, és ezekbe különböző típusú adatok is eltárolhatók.

Egy **struct** típusú változó definiálásának szintaxisa:

```
struct
{
    <típus_1> <mező_1>;
    <típus_2> <mező_2>;
    ...
    <típus_n> <mező_n>;
} <változó_lista>;
```

Egy **struct** változó valamely mezőjére való hivatkozás szintaxisa:

```
<struct_változó>.<mező>
```

Egy **struct** változó tárhelyigénye a mezői tárhely igényeinek összege.

Egy személy adatainak beolvasása egy **struct** változóba:

```
struct
{
    char nev[30];
    unsigned életkor;
    float magassag;
} sz;
printf("A neve: "); gets(sz.nev);
printf("Az életkora: "); scanf("%d", &sz.életkor);
printf("A magassága: "); scanf("%f", &sz.magassag);
```

Célszerű elnevezni a létrehozott **struct** típusokat, hogy hivatkozni lehessen rájuk. A **struct** típus kapcsos zárójelek közé írt mezőleírása címkével látható el az alábbi módon:

```
struct <címke>
{
    <típus_1> <mező_1>;
```

```
<típus_2> <mező_2>;
```

```
...
```

```
<típus_n> <mező_n>;
```

```
} [<változó_lista>];
```

A teljes **struct** típus nevet kaphat a **typedef** alkalmazása által:

```
typedef struct
```

```
{
```

```
<típus_1> <mező_1>;
```

```
<típus_2> <mező_2>;
```

```
...
```

```
<típus_n> <mező_n>;
```

```
} <típusazonosító>;
```

Példák a címkézés és a **typedef** alkalmazására:

```
struct aru
```

```
{
```

```
    unsigned kod, ar;
```

```
    char nev[25];
```

```
} v1, v2, v3;
```

```
struct aru v4, v5[100];
```

```
typedef struct
```

```
{
```

```
    char marka[20];
```

```
    char tipus[15];
```

```
    char szin[10];
```

```
    unsigned long ar;
```

```
} auto;
```

```
auto a, b[50];
```

Megjegyzés. Mivel **aru** csupán a mezőleírást címkézi, ezért az első **struct** típus azonosítója **struct aru**. A második esetben **auto** a teljes típus azonosítója.

8.1. feladat. Adottak **n** személy adatai (név, telefonszám) egy **szemely.dat** bemeneti állományban. Hozzunk létre egy **nevsor.dat** kimeneti állományt, amely névsor szerint, sorszámozva tartalmazza a személyek adatait.

8.2. A STRUCT TÍPUS

145

A személy.dat struktúrája:

```
<n>
<név_1>
<telefonszám_1>
<név_2>
<telefonszám_2>
...
<név_n>
<telefonszám_n>
```

A nevsor.dat struktúrája:

```
1. <név>: <telefon>
2. <név>: <telefon>
...
n. <név>: <telefon>
```

```
typedef struct
{
    char nev[30], tel[15];
} személy;
személy *v, temp;
int n, i, j, cseresz, vege;
FILE* f = fopen("szemely.dat", "r");
FILE* g = fopen("nevsor.dat", "w");
/* helyfoglalás és beolvasás */
fscanf(f, "%d", &n); fgetc(f);
v = (személy*)malloc(n * sizeof(személy));
for(i = 0; i < n; i++)
{
    fgets(v[i].nev, sizeof(v[i].nev), f);
    fgets(v[i].tel, sizeof(v[i].tel), f);
}
fclose(f);
/* buborékos rendezés */
vege=n;
do
{
    vege--;
    cseresz=0;
    for(i=0;i<vege;i++)
    {
        if (strcmp(v[i].nev, v[i+1].nev) > 0)
        {
            cseresz=1;
            temp=v[i];
```

```

        v[i]=v[i+1];
        v[i+1]=temp;
    }
}
} while (cseresz);
/* kiíratás */
for(i = 0; i < n; i++)
{
    fprintf(g, "%d. %s: %s\n",
        i+1, v[i].nev, v[i].tel);
}
fclose(g);

```

A **struct**-ok egymásba is ágyazhatók. Ezt mutatja be az alábbi feladat.

8.2. feladat. Adott **n** személy neve és címe egy bemeneti állományban, amelynek nevét a billentyűzetről olvassuk be. Írjuk ki a képernyőre a marosvásárhelyiek számát.

A bemeneti állomány struktúrája:

```

<n>
<név_1>
<város_1> <utca_1> <házszám_1>
<név_2>
<város_2> <utca_2> <házszám_2>
...
<név_n>
<város_n> <utca_n> <házszám_n>

```

```

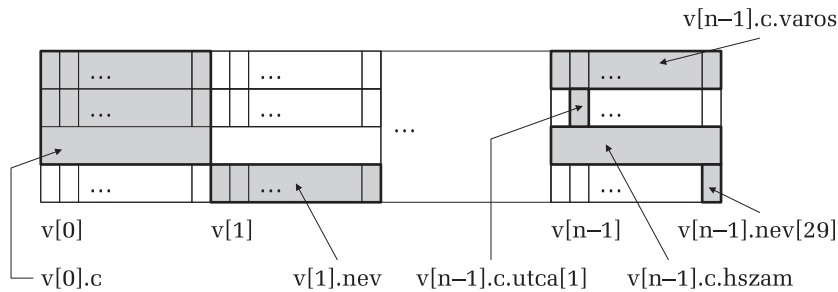
struct cim
{
    char varos[30], utca[25];
    int hszam;
};
typedef struct
{
    struct cim c;
    char nev[30];
} szemely;
szemely *v;
int n, i, k;
FILE* f;
char all_nev[30];
puts("Írd be a bemeneti állomány nevét:");

```

```

gets(all_nev);
f = fopen(all_nev, "r");
fscanf(f, "%d", &n); fgetc(f);
v = (szemely*)malloc(n * sizeof(szemely));
for(i = 0; i < n; i++)
{
    fgets(v[i].nev, sizeof(v[i].nev), f);
    fscanf(f, "%s%s%d", v[i].c.varos, v[i].c.utca,
        &v[i].c.hszam);
    fgetc(f);
}
fclose(f);
k = 0;
for(i = 0; i < n; i++)
    if (!strcmp(v[i].c.varos, "Marosvásárhely")) k++;
printf("A marosvásárhelyiek száma: %d", k);
    
```

Az alábbi ábra szemléletesen mutatja be, miként hivatkozhatunk a **v** **struct**-tömbben eltárolt adatokra:



8.2.1. A struct-pointerek

A következő példa összefoglalja, miként lehet hivatkozni egy **struct** változó címére, hogyan definiálunk **struct**-pointereket, és milyen módokon hivatkozhatunk **struct**-pointer segítségével a struktúrák mezőire.

Az alábbi példában beolvassuk egy osztály tanulóinak számát, valamint az **oszi** nevét egy **struct** változóba (**v1**), készítünk erről egy másolatot egy másik **struct** változóba (**v2**), majd kiírjuk az adatokat a képernyőre.

```

struct osztaly
{
    int letszam;
    
```

```
char oszi[30];
} v1, *p1; /* p1, struct osztaly pointer */
struct osztaly v2, *p2 =&v2; /* p2, struct osztaly
    pointer, amely v2-re mutat */
p1 = &v1; /* p1, v1-re fog mutatni */
puts("Írd be a létszámot:");
scanf("%d", &v1.letszam);
puts("Írd be az oszi nevét:");
gets((*p1).oszi);
(*p2) = v1; /*másolatot készítünk v1-ről a p2 című struktúra-
    ba, azaz v2-be */
printf("A létszám: %d\nAz oszi: %s", p2->letszam,
    p2->oszi);
```

Tehát egy struktúra mezőire, a rá mutató pointer segítségével, az alábbi módokon hivatkozhatunk:

`<pointer> -> <mező>` vagy `(*<pointer>).<mező>`

8.3. feladat. Legyenek az alábbi definiálások:

```
typedef struct {int a,b;} tipus1;
typedef struct {tipus1 c[50];} tipus2;
tipus2 x, *p = &x;
```

Hogyan hivatkozhatunk az `x` struktúra `c` mezőjének 25. eleme `b` mezőjére?

Megoldások:

1. `x.c[25].b`
2. `(*p).c[25].b`
3. `p->c[25].b`
4. `(((*p).c+25)).b`
5. `(p->c+25)->b`

8.2.2. Bitmezők

A C nyelv különleges lehetősége, hogy megadható, hány biten legyenek tárolva egy `struct` változó egyes mezői.

A bitmezőket szükségszerűen `int`, `unsigned` vagy `signed` típusúaknak kell definiálnunk (egyes fordítók csak az `unsigned` típust engedik meg). Az 1 bit hosszú bitmezőknek természetesen `unsigned` típusúaknak kell lenniük, hiszen egy biten nincs lehetőség külön előjelbitet is tárolni.

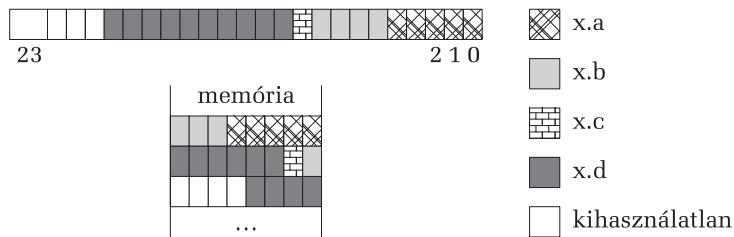
8.2. A STRUCT TÍPUS

149

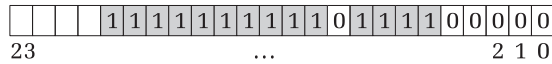
Legyen az alábbi példa:

```
struct címke
{
    unsigned a:5; /* 5 biten tárolt előjel nélküli egész */
    signed b:4; /* 4 biten tárolt előjeles egész */
    unsigned c:1; /* 1 biten tárolt előjel nélküli egész */
    int d:10; /* 10 biten tárolt előjeles egész */
} x;
```

A fent definiált `x` változó tárhelyigénye 3 byte (`sizeof(x)=3`), amelyből a mezői effektíve 20 bitet használnak ki. Ezt szemlélteti az alábbi ábra, amennyiben a bitmezők sorrendje jobbról balra:



Az `x.a = 0`; `x.b = -1`; `x.c = 0`; `x.d = -1`; értékadások után `x` belső ábrázolása:



Általános szintaxis:

```
struct
{
    <típus_1> <mező_1>:<hossz_1>;
    <típus_2> <mező_2>:<hossz_2>;
    ...
    <típus_n> <mező_n>:<hossz_n>;
} <változó_lista>;
```

A bitmezők használatának előnyeit a következőképpen lehetne összefoglalni:

- memóriatakarékosság,
- bit szinten kódolt információk feldolgozása,
- a memóriához bit szinten való hozzáférés.

Bár mindez megvalósítható bitműveletekkel is, a bitmezők alkalmazása növeli a program áttekinthetőségét és hatékonyságát.

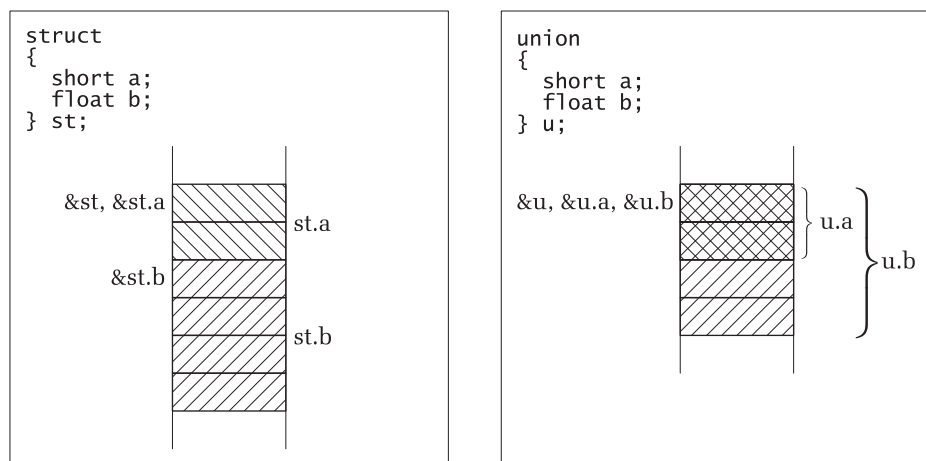
Megjegyzések:

- Egy **struct** változónak vegyesen lehetnek bitmezői és nem bitmezői.
- Nem lehet hivatkozni egy bitmező címére.
- A bitmezők nem rendezhetők tömbökbe.
- A géptől függ, hogy a bitmezők jobbról balra vagy balról jobbra sorrendűek.

A most következő, *union*okkal foglalkozó rész végén található feladat bemutatja, miként élhetünk a bitmezők nyújtotta lehetőségekkel.

8.3. A union típus

A **struct** típus esetén a mezők egymás után kerülnek eltárolásra. A **union** típust pontosan úgy definiáljuk, mint a **struct**-ot, csak hogy mezői egymásra tevődnek. Úgy is mondhatnánk, hogy a mezők együtt használják ugyanazt a memóriaterületet. Az alábbi ábrák jól érzékeltetik a különbséget:



Egy **union** változó tárhelyigényét nyilván a legnagyobb méretű mezője határozza meg. Például az `u` változó mérete 4 byte lesz (`sizeof(float)`).

8.3. A UNION TÍPUS

151

Az első 2 byte-ra, mint **short** típusú mezőre, **u.a**-val hivatkozhatunk. Az **u.b** hivatkozás mind a 4 byte-ra, mind a **float** típusú adatra utal.

A **union** típus segítségével érdekes típuskonverziók valósíthatók meg. Az alábbi feladatok a **union** típus gyakorlati értékét emelik ki.

8.4. feladat. Adott egy természetes szám. Melyik az a valós szám, amelynek **float** típusúkénti belső ábrázolása azonos az illető természetes szám **unsigned long**-kénti belső ábrázolásával?

```
union
{
    unsigned long x;
    float y;
} a;
puts("Írd be a természetes számot!");
scanf("%lu", &a.x);
printf("A megfelelő valós szám: %f", a.y);
```

8.5. feladat. Adott egy valós szám, amit **long double**-ként tárolunk el. Írjuk ki a belső ábrázolása 10 byte-ját hexadecimálisan.

```
typedef union
{
    unsigned char x[10];
    long double y;
} u_type;
u_type a; int i;
puts("Írd be a valós számot:");
scanf("%Lf", &a.y);
puts("A belső ábrázolás hexában:");
for(i = 9; i >= 0; i--)
    printf("%02X", (int)a.x[i]);
/* két karakterpozíción ír ki, 0-val pótolva balról, ha az érték
egy hexaszámjegyű */
```

Az alábbi feladat célja, hogy összefoglalja a struktúrák, bitmezők, illetve unionok használatának egyes előnyeit.

8.6. feladat. Olvassunk be a billentyűzetről **n** ($n \leq 50$) természetes számot az **unsigned short** tartományból, amelyeknek **unsigned short** típusúkénti belső ábrázolásuk személyek születési dátumainak kódjai, a következőképpen:

7 bit – az év utolsó két számjegye

4 bit – a hónap

5 bit – a nap

Ellenőrizzük, van-e a személyek között májusi születésű, hányan születtek a hetvenes években, és született-e valaki január elsején.

```
/* feltételezzük, hogy a bitmezők sorrendje
és az év, hó, nap kódbeli sorrendje talál */
typedef struct
{
    unsigned ev:7;
    unsigned ho:4;
    unsigned nap:5;
} datum;
typedef union
{
    unsigned short kod;
    datum d;
} személy;
struct
{
    unsigned m:1; /* van-e majusi */
    unsigned h:6; /* hányan hetvenesek */
    unsigned j1:1; /* van-e januar elseji */
} v;
szemely p[50];
int n,i;
v.m = v.h = v.j1 = 0;
scanf("%d", &n);
for(i = 0; i < n; i++)
{
    scanf("%hu", &p[i].kod);
    if(p[i].d.ho == 5) v.m = 1;
    if(p[i].d.ev >= 70 && p[i].d.ev < 80) v.h++;
    if(p[i].d.ho == 1 && p[i].d.nap == 1) v.j1 = 1;
}
if(v.m) puts("Van majusi!");
else puts("Nincs majusi!");
printf("%hu személy hetvenes\n", v.h);
if(v.j1) puts("Van januar elseji!");
else puts("Nincs januar elseji!");
```

8.4. Kitűzött feladatok

1. Adott n személy neve, címe (város, utca, házszám) és telefonszáma egy bemeneti állományban. Írjuk át egy kimeneti állományba névsor szerint a budapestiek nevét és telefonszámát (a rendezést egy pointertömb segítségével valósítsuk meg, anélkül hogy effektíve rendeznénk a **struct** tömböt, ahova a személyek adatait beolvastuk).

2. Adott n személy neve, k tantárgy és mindeniknek minden tantárgyból az átlaga. Írjuk ki névsor szerint a személyek nevét és mindeniknek az átlagát csökkenő sorrendben.

3. Milyen értéket ír ki az alábbi programrészlet? Feltételezzük, hogy a **char** 1 bájt, a **long** pedig 4 bájt van tárolva.

```
union elem{char b[4]; long c;}x;
x.c = 0;
x.b[0]^=1;
x.b[1]|=1;
x.b[2]^=1;
x.b[3]|=1;
printf("%ld",x.c);
```

4. Milyen értéket ír ki az alábbi programrészlet? Feltételezzük, hogy a **unsigned char** 1 bájt, a **unsigned long** pedig 4 bájt van tárolva.

```
union elem{unsigned char b[4]; unsigned long c;}x;
x.c = 0;
x.b[0]^=1;
x.b[1]&=1;
x.b[2]&=2;
x.b[3]|=1;
printf("%ld",x.c);
```

5. Legyen az alábbi C programrészlet:

```
typedef struct {float b[77]; int a;} elemke;
elemke w[77], *q = w+1;
```

- Hivatkozz w utolsó elemének, első mezőjének utolsó elemére.
- Állítsd a q címen levő struktúra második mezőjének legkisebb helyértékű bitjét 1-re.
- Hány bájt memória terület kerül lefoglalásra a fenti definíciók nyomán?

6. Legyen az alábbi C programrészlet:

```
typedef struct {int a; double b[100];} elem1;
typedef struct {int c; elem1 d[100];}elem2;
elem2 v[100], *p = v;
```

- Hivatkozz v utolsó elemének, második mezőjének, utolsó elemének első mezőjére.
- Hivatkozz a p címen levő struktúra második mezőjének, utolsó elemének, második mezőjének utolsó elemére.

7. Adott n személy neve, telefonszáma, születési dátuma és helye, valamint lakcíme.

- Írjuk ki a neveket és születési dátumokat születési dátum szerint csökkenő sorrendben (azonos év esetén vegyük figyelembe a hónapot, ...)
- Hányan laknak a szülővárosukban?
- Kiknek jelenik meg a telefonszámukban a házszámuk?

8. Adott n személy neve és neme.

- Lehetséges-e őket párosítani egy tánchoz, és ha igen, adjunk meg egy ilyen párosítást.
- Adjuk meg azt a párosítást, amelyben a lehető legtöbb Jancsi és Juliska, illetve Csongor és Tünde páros van.

9. Adott két bemeneti állomány. Az egyik tartalmazza n ország nevét és fővárosát, a másik m fővárost és lakosainak számát. Készítsünk kimutatást egy kimeneti állományba azon országokról és fővárosaik lakosságáról, amelyek esetében rendelkezünk ezekkel az adatokkal.

10. Adott egy bemeneti állományban m tanuló neve és ezek osztályfőnökei. Készítsünk kimutatást egy kimeneti állományban arról, hogy az egyes osztályfőnököknek hány diákjuk van.

11. Adott három bemeneti állomány. Az első tartalmazza n tanfolyam nevét és árát, a második m személy nevét és születési évét, a harmadik a beiratkozási nyilvántartást, minden sorban egy tanulót és egy tanfolyamot (egy tanuló maximum 5 tanfolyamra iratkozhat be, minden tanfolyamhoz legalább 4, legfeljebb 15 diák szükséges).

- Ellenőrizzük, hogy a megkötéseket figyelembe vették-e.
- Mely tanfolyamok a leglátogatottabbak, és melyek a legkevésbé?
- Kik látogatják a legtöbb tanfolyamot, és kik a legkevesebbet?
- Mely tanfolyamokat látogatják a legfiatalabb diákok, és melyeket a legidősebbek?
- Melyik tanfolyam hozza a legtöbb pénzt, és mennyi pénz jön összesen be?

- f) Készítsünk kimutatást egy kimeneti állományba a tanfolyamokról, és az egyes tanfolyamokat látogató hallgatókról.
- g) Készítsünk kimutatást egy kimeneti állományba a hallgatókról és a hallgatók választotta tanfolyamokról.

12. Adott három bemeneti állomány. Az első tartalmazza az osztályfőnököket és osztályuk megnevezését, a második a tanulókat és hogy melyik osztályba járnak, a harmadik pedig az osztályfelelősök listáját. Írjuk ki az osztályfelelősöket és osztályfőnökeiket.

13. Adott n egész szám a `long` tartományból. Titkosítsuk a számsorozatot úgy, hogy a számok belső ábrázolásainak bájtjait annyiszor forgatjuk körkörösén, ahányadik a szám a sorozatban. A páratlan pozíciójúakat balra, a páros pozíciójúakat pedig jobbra forgassuk.

14. Adott egy szöveges állományban n valós szám. Titkosítsuk a számsorozatot úgy, hogy a `float`-kénti belső ábrázolásuk középső két bitjét (a 15. és 16. helyértékű biteket) felcseréljük maguk között.

- A titkosítást `union`-ok és bitmezők által valósítsuk meg.
- A titkosításhoz használjuk a bitműveleteket.

8.5. A felsorolás típus (enum)

Mi határozza meg, milyen értékeket vehet fel egy változó? A típusa. Az eddig megismert egyszerű típusok esetében implicite adva volt az értéktartomány. Például egy `char` változó -128 és 127 közötti egészeket kaphat értékként. Az `enum` típus segítségével a felhasználó sorolhatja fel, milyen értékeket vehet fel egy ilyen típusú változó. A felsorolt azonosítók, amint látni fogjuk, valójában egész konstans nevek. Tehát a `#define` direktíva és a `const` módosító jelző mellett az `enum` típus is lehetővé teszi szimbolikus egész konstansok definiálását. Mivel az `enum` típus értéktartományának minden értéke nevet kap, ezért alkalmazása áttekinthetőbb kódot eredményez.

Az alábbi példákban sok minden kiderül a felsorolás típusról:

```
enum napok
{
    hetfo, keddi, szerda, csutortok, pentek, szombat=10,
    vasarnap
} nap;
typedef enum
{
```

```

    tavasz=3, nyar=6, osz=9, tel=12
} evszak;
enum
{
    jan=1, feb, mar, apr, maj, jun, jul, aug, szept, okt,
    nov, dec
};
enum napok szuletesnap;
evszak vakacio;
int ho = jul; /* 1 */
if (ho >= nyar && ho < osz)
    puts("Nyari honap!"); /* 2 */
nap = pentek; /* 3 */
vakacio = 6; /* 4 */
vakacio = aug; /* 5 */
nap++; /* HIBÁS: az enum azonosítóknak nincs bal értékük */
puts(hetfo); /* HIBÁS: hetfo egész azonosító, nem karakterlánc */

```

Megjegyzések:

- A **nap**, illetve **szuletesnap** változók **enum napok** típusúak.
- A **vakacio** változó **evszak** típusú, ahol **evszak** egy **enum** típus neve.
- A felsorolt azonosítók, **hetfo..vasarnap**, **tavasz..tel**, **jan..dec**, a valóságban egész értékeket képviselnek. Az első azonosító a felsorolásban implicite a 0 értéket kapja, a többi pedig eggyel nagyobb, mint az előtte lévő. Az implicit értékeket felül lehet bírálni explicit értékadással. Például **hetfo** értéke 0, **kedd**-é 1, **pentek**-é 5, **vasarnap**-é pedig 11 lesz.
- Mivel a felsorolásban *azonosítók* szerepelnek, ezért érvényes rájuk az a megkötés, hogy csakis az angol ábécé kis- és nagybetűit, a számjegyeket és az aláhúzásjelt tartalmazhatják, és nem kezdődhetnek számjeggyel.
- Mivel a felsorolt azonosítók alapján véve egész értékek, használhatók bárhol, ahol egész értékre van szükség. Ebből adódóan a harmadik **enum** definíciónak is van értelme – bár nem definiáltunk ilyen típusú változót, és nevet se adtunk neki, hogy később hivatkozhattunk rá –, hiszen a felsorolt azonosítók használhatók mint egészek (lásd az 1-es és 2-es programsorokat).

8.5. A FELSOROLÁS TÍPUS (ENUM)

157

- Egy **enum** változó alapvetően a saját felsorolása azonosítóit veheti fel (3-as programsor). Bár működnek a 4-es és 5-ös programsorok is (**warning** kíséretében), ellentétben állnak az **enum** típus céljával.
- Az **enum** típust hasonlóan definiáljuk, mint a **struct**-ot, azzal a különbséggel, hogy opcionális mind a címke-, mind a változólista:
enum [**<címke>**] {**<felsorolás>**} [**<változólista>**];
- Az **enum** változók tárhelyigénye **sizeof(int)**. Úgy foghatók fel, mint az **int** tartomány részhalmazai. Ebből adódik az a megkötés, hogy az **enum** felsorolásokban csakis **int** tartományba eső értékek szerepelhetnek.

8.7. feladat. Készítsünk egy kimutatást arról, hogy a hét melyik napján a legnagyobb egy adott üzlet bevétele. A billentyűzetről **n** nap sorszámát (hétfő: 1, ..., vasárnap: 7) és az aznapi bevételt olvassuk be.

```
enum
{
    hetfo=1, kedd, szerda, csutortok, pentek,
    szombat, vasarnap
};
char * napok[8] = {"", "hétfő", "kedd", "szerda",
    "csütörtök", "péntek", "szombat", "vasárnap"};
unsigned long bevetel[8] = {0}, penz, maxpenz;
int nap, n, maxnap, i;
puts("Add meg a napok számát:");
scanf("%d", &n);
puts("Írd be a napokat és a pénzösszegeket:");
for(i = 1; i <= n; i++)
{
    scanf("%d%lu", &nap, &penz);
    bevetel[nap]+=penz;
}
maxpenz = bevetel[hetfo]; maxnap = hetfo;
for(i = kedd; i <= vasarnap; i++)
    if(bevetel[i] > maxpenz)
        {maxpenz = bevetel[i]; maxnap = i;}
printf("A legnagyobb forgalmú nap: %s", napok[maxnap]);
```

9. FEJEZET

FÜGGVÉNYEK

Egy feladaton keresztül fogjuk érzéltetni a függvények használatának előnyeit.

9.1. feladat. Olvassuk be a billentyűzetről az n és k ($k \leq n$) természetes számokat, és írjuk ki a képernyőre az n elemű halmaz k -ad rendű kombinációinak számát az alábbi képlet segítségével:

$$c(n,k) = n! / (k! * (n-k)!).$$

Megoldás, függvények alkalmazása nélkül:

```
int n, k, i;
unsigned long fn, fk, fn_k, cnk;
puts("Írd be az n és a k értékét:");
scanf("%d%d", &n, &k);
/* az n! kiszámítása */
if (!n) fn = 1;
else for(fn = 1, i = 1; i <= n; i++) fn *= i;
/* a k! kiszámítása */
if (!k) fk = 1;
else for(fk = 1, i = 1; i <= k; i++) fk *= i;
/* az (n-k)! kiszámítása */
if (!(n - k)) fn_k = 1;
else for(fn_k = 1, i = 1; i <= n - k; i++) fn_k *= i;
/* a c(n,k) kiszámítása */
cnk = fn / (fk * fn_k);
printf("A c(n,k) értéke: %lu", cnk);
```

Vegyük észre, hogy a fenti megoldásban háromszor írtuk le a faktoriális számítás kódját, n -re, k -ra és $(n-k)$ -ra. Hasznos lenne egy olyan lehetőség, hogy csak egyszer kelljen megírni, de akárhányszor végre lehessen hajtani, más-más értékekre. Erre biztosítanak lehetőséget a függvények. A függvényt úgy lehet elképzelni, mint egy színdarab forgatókönyvét, amelyet egyszer írnak ugyan meg, de akárhányszor el lehet játszani, különböző színtereken, más-más színészekkel.

Íme a feladat megoldása függvény segítségével:

```
#include <stdio.h>

/* függvény a faktoriális számításra */
/* unsigned int a függvény típusa, faktorialis pedig a neve */

unsigned long faktorialis (int n) /* n, formális paraméter */
{
    int i; unsigned long f; /* i és f, lokális változók */
    if (!n) f = 1;
    else for(f = 1, i = 1; i <= n; i++) f *= i;
    return f; /* a hívó függvénynek eredményként visszatérített érték */
}

main()
{
    int n,k; unsigned long cnk; /* n, k és cnk, a main lokális változói */
    puts("Írd be az n és a k értékét:");
    scanf("%d%d", &n, &k);

    /* a c(n,k) kiszámítása */
    /* n, k, (n-k), aktuális vagy effektív paraméterek */

    cnk=faktorialis(n)/(faktorialis(k)*faktorialis(n-k)); /* 1 */

    printf("A c(n,k) értéke: %lu", cnk);
    return 0;
}
```

Hogyan megy végbe a fenti program?

Minden program a **main** függvény végrehajtásával kezdődik. A **main** függvény, jelen esetben, 3-szor fogja meghívni a **faktorialis** függvényt. Az 1-es programsornál felfüggesztődik a **main** végrehajtása, és a **faktorialis(n)** függvényhívás hatására elkezdődik a **faktorialis** függvény végrehajtása. Létrejön¹ az **n** formális paraméter, és megkapja a **main**-beli **n**-nek – mint aktuális paraméternek – az értékét (**Rvalue**). Létrejönnek az **i** és **f** lokális változók, és a függvény utasításainak végrehajtása nyomán **f**-ben képződik az **n!** értéke. A **return** utasítás visszatéríti a **main** függvénynek az **f** értékét (**Rvalue**), és felszámolódnak² a lokális változók (**i**, **f**) és a formális paraméterek (**n**). Ezennel véget ért a **faktorialis**

1 Helyfoglalás történik számára. Lásd a „Paraméterátadás a STACK-en keresztül” című alfejezetet.

2 Felszabadul a számára lefoglalt memóriaterület.

függvény első végrehajtása, és folytatódhat a **main**. A **main** függvény, a **faktorialis(k)**, illetve **faktorialis(n-k)** hívások hatására, másodszor és harmadszor is meghívja a **faktorialis** függvényt. Ezek a függvényhívások hasonló módon bonyolódnak le, mint az első. A második hívásban az **n** formális paraméter a **main**-beli **k**, a harmadikban pedig az **n-k** értékét kapja meg. Így valósul meg, hogy rendre az **n!**, **k!**, **(n-k)!** értékek kerülnek kiszámításra a **faktorialis** függvény által.

Megjegyzések:

1. Folytatva a szemléltetést a színdarabban, a formális paraméterek a szereplőknek, az aktuális paraméterek pedig a színészeknek felelnek meg. Egy fontos részlet, hogy a függvények végrehajtásának idejére nem az aktuális paraméterek „öltöznek be” és „játsszák el” a formális paramétereknek előírt szerepet, hanem maguk a formális paraméterek „elevenednek” meg mint az aktuális paraméterek valamiféle „dublőrjei”³ (másolatai). Ezért nevezik ezt a mechanizmust *érték szerinti paraméterátadás*nak.
2. A formális paraméterek és a lokális változók között az a közös vonás, hogy a függvény végrehajtásának pillanatában „születnek meg”, a memória Stack-nek nevezett részén, és a függvény befejeztével „megszűnnek létezni”. Tehát csak a függvényvégrehajtás idején léteznek. Csak abban különböznek, hogy a formális paraméterek inicializálódnak a megfelelő aktuális paraméter értékével, a lokális változók viszont nem kapnak implicit módon kezdőértéket.
3. Természetesen a formális („szereplők”) és aktuális („színészek”) paraméterek száma meg kell hogy egyezzen, sőt, mivel a formális paraméterek megkapják a megfelelő aktuális paraméter értékét, ezért típusaik az értékadásra nézve kompatibilisek kell hogy legyenek.
4. Az előbbi példában a **main** függvényt *hívó*, a **faktorialis** függvényt pedig *meghívott függvény*nek nevezzük. Természetesen egy meghívott függvény maga is meghívhat más függvényeket. Sőt, amint látni fogjuk, egy függvény meghívhatja önmagát is (*rekurzió*). Arra is van lehetőség, hogy függvények egymást hívják (*indirekt rekurzió*).
5. Egy függvény típusa a hívó függvénynek eredményként visszatérítendő érték típusa. Ezért a **return** utasításban szereplő kifejezés típusa az értékadás szempontjából kompatibilis kell hogy legyen a függvény típusával.

³ A színészt egyes helyzetekben helyettesítő szereplő.

6. A **return** utasítás az eredmény értékét (**Rvalue**) téríti vissza. Ezért elmondható, hogy az információ átadása a hívó és a meghívott függvény között mindkét irányban érték szerint történik. Ebből adódik az is, hogy sem az aktuális paramétereknek, sem a **return** által visszatérített kifejezésnek nem kell szükségszerűen változónak lennie. Bármilyen megfelelő típusú *kifejezés* lehet, amelynek van jobb értéke (**Rvalue**).
7. Egy függvényben több helyen is szerepelhet **return**. Ilyenkor azt mondjuk, hogy több visszatérési pontja van. Például:

```
unsigned long faktorialis (int n)
{
    int i; unsigned long f;
    if (n == 0) return 1;
    for (f = 1, i = 1; i <= n; i++) f *= i;
    return f;
}
```
8. A függvények általában kiszámítanak valamit, és ezt eredményként visszatérítik a hívó függvénynek. Vannak azonban esetek, amikor a függvény nem kell hogy visszatérítsen semmit a hívó függvénynek, csak el kell végeznie valamit. Ezek a **void** függvények (eljárások). A **void** szónak mint függvénytípusnak az a jelentése, hogy nincs visszatérő érték.
9. A **void** függvényekben is szerepelhet a **return** utasítás („üres” **return**-ként, visszatérítendő kifejezés nélkül), ha több visszatérési pontot szeretnénk a függvényből.
10. Lehetőség van arra is, hogy függvényünk több eredményt térítsen vissza, de ez csakis a paraméterlistán keresztül valósítható meg (lásd a *cím szerinti* paraméterátadást).
11. A függvények alkalmazásának nem csak az az előnye, hogy nem kell ugyanazt többször leírunk. Alkalmazásukkal sokkal áttekinthetőbb kódok írhatók. A függvények a moduláris programozás alapeszközei (12.2. alfejezet).

Egy függvény definiálásának általános szintaxisa:

```
<típus> <függvény_név>(<formális_paraméter_lista>)
{
    <lokális_definiálások>
    <utasítások>
    [return [<kifejezés>];]
}
```

A függvényhívás szintaxisa:

```
<függvény_név>(<aktuális_paraméter_lista>);
```

A C nyelv különbséget tesz egy függvény *definíciója* és *deklarációja* között. Egy függvény deklarációja a függvény fejléce;-vel lezárva. Ezt a függvény *prototípusának* is szokás nevezni. A deklarációnak nem fontos tartalmaznia a formális paraméterek nevét, csupán a típusukat.

A függvénydeklaráció szintaxisa:

```
<típus> <függvény_név> (<formális_paraméterek_típusai>);
```

A faktoriális függvény prototípusa:

```
unsigned long faktoriális (int);
```

Ahhoz, hogy a fordító ellenőrizni tudja egy függvényhívás szintaktikai helyességét, szükséges, hogy ezt megelőzze a függvény definíciója, vagy legalább deklarációja, amely az éppen ehhez szükséges információt tartalmazza. Emlékezzünk, hogy a könyvtári függvények deklarációit a *header* állományok tartalmazzák, és éppen az előbbi okból szükséges ezeknek az `#include` direktívával való beszúrásuk.

A bevezető feladatot megoldó program szerkezete a következő is lehetett volna:

```
#include <stdio.h>
unsigned long faktoriális (int);
main()
{
    ...
    return 0;
}
unsigned long faktoriális (int n)
{
    ...
}
```

9.1. Cím szerinti paraméterátadás

9.2. feladat. Írjunk egy függvényt, amely felcseréli két változó tartalmát.

Vajon helyes-e az alábbi megvalósítás?

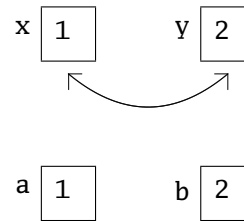
```
#include <stdio.h>
void csere (int, int);
```

9.1. CÍM SZERINTI PARAMÉTERÁTADÁS

163

```
main()
{
    int a = 1, b = 2;
    csere (a,b);
    printf("a=%d, b=%d", a, b);
    return 0;
}

void csere (int x, int y)
{
    int v;
    v = x; x = y; y = v;
}
```



Mi fog megjelenni a képernyőn?

Bár arra számítanánk, hogy **a=2, b=1**, a valóságban **a=1, b=2** jelenik meg.

Mi a magyarázata?

Mivel **x** és **y** csupán másolatai **a**-nak és **b**-nek, ezért attól, hogy ezek tartalmát felcseréljük, **a** és **b** még változatlan marad (érték szerinti paraméterátadás).

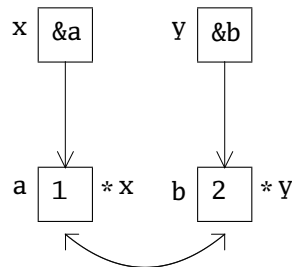
Mi a megoldás?

A cím szerinti paraméterátadás!

Íme a helyes megoldás:

```
#include <stdio.h>
void csere (*int, *int);
main()
{
    int a = 1, b = 2;
    csere (&a,&b);
    printf("a=%d, b=%d", a, b);
    return 0;
}

void csere (int *x, int *y)
{
    int v;
    v = *x; *x = *y; *y = v;
}
```



Hogyan működik a fenti program?

Az **x** és **y** **int**-pointerek, mint formális paraméterek, az **a** és **b** változók címét kapják mint értéket, ezért **a**-ra, illetve **b**-re fognak mutatni. Így felcserélődik ***x** és ***y**, vagyis az **a** és **b** változók.

Úgy is szokták mondani, hogy ha szeretnéd, hogy a függvény megváltoztathassa az aktuális paramétereket, akkor cím szerint kell átadnod őket, azaz a címüket kell átadnod. A hívó függvény azzal, hogy átadja egyes változóinak a címét a meghívott függvénynek, elérhetővé teszi az illető változókat az illető függvény számára, és így az megváltoztathatja őket. Ez nem új mechanizmus, ugyanis úgy valósítottuk meg **a** és **b** cím szerinti átadását, hogy címüket, **&a**-t és **&b**-t érték szerint átadtuk az **x** és **y** pointereknek.

Megjegyzések:

1. Cím szerint átadott paraméterlistában csakis változók szerepelhetnek, hiszen ezeknek van bal értékük, azaz címük (**Lvalue**).
2. Említettük, megtörténhet, hogy egy függvény több eredményt is vissza kell hogy térítsen a hívó függvénynek. Ez cím szerint átadott paramétereken keresztül valósítható meg. Példaként tekintsük az alábbi „beszédes” programot:

```
void szamol(int a, int b, int*ps, int*pk, int*psz,
            int*ph)
{ *ps=a+b; *pk=a-b; *psz=a*b; *ph=a/b; }
main()
{
    int a, b, summa, kulonbseg, szorzat, hanyados;
    scanf("%d%d", &a, &b);
    szamol(a, b, &summa, &kulonbseg, &szorzat,
           &hanyados);
    printf("%d %d %d %d", summa, kulonbseg, szorzat,
           hanyados);
    return 0;
}
```

3. Változók esetében mind a cím szerinti, mind az érték szerinti paraméterátadásnak van egy előnye és egy hátránya. A cím szerinti paraméterátadás általában memóriatakarékosabb, ezért nagyméretű változók esetében célszerű lehet előnyben részesíteni, amennyiben lehetséges. Az érték szerinti paraméterátadás viszont biztonságosabb, abban az értelemben, hogy a meghívott függvény még véletlenül sem tudja „elrontani” a hívó függvény változóit, hiszen csak az értéküket kapja meg. Amikor a megoldás logikája nem követeli meg kifejezetten

valamelyik paraméterátadást, akkor a programozónak kell eldöntenie, melyik szempontot részesíti előnyben.

4. A C nyelvben a tömbök mindig cím szerint adódnak át, hiszen a tömb neve adódik át, ami a valóságban a tömb címe, egy pointer, amely a tömb nulladik elemére mutat.
5. Az alábbi példák több lehetőséget is bemutatnak arra vonatkozóan, miként adhatunk át egy- és kétdimenziós tömböket függvényeknek. Mivel egydimenziós tömbben rendszerint sorozatot, kétdimenziósban pedig mátrixot tárolunk, célszerű átadni a függvénynek az elemek, illetve a sorok és oszlopok számát is:

- a)


```
typedef int vektor[20];
typedef float matrix[20][30];
void f(vektor v, int n, matrix m, int p, int q)
{...}
main()
{
    int n, p, q;
    vektor a, matrix b;
    ...f(a, n, b, p, q); ...
    return 0;
}
```
- b)


```
void f(int v[], int n, float m[][30], int p,
      int q)
{...}
main()
{
    int n, p, q;
    int a[20], float b[20][30];
    ...f(a, n, b, p, q); ...
    return 0;
}
```
- c)


```
void f(int * v, int n, float (*m)[30], int p,
      int q)
{...}
main()
{
    int n, p, q;
    int a[20], float b[20][30];
    ...f(a, n, b, p ,q); ...
}
```

```
    return 0;
}
```

6. A **struct**, illetve **union** típusú változók érték szerint és cím szerint is átadhatók függvényeknek.
7. Bizonyos esetekben szükségessé válhat pointerek cím szerinti átadása. Ezt mutatja be az alábbi példa. A **beolvas** függvény szerepe, hogy létrehozzon dinamikusan egy egydimenziós tömböt, és olvasson bele egy egész számsorozatot adott állományból. A függvény paraméterként kapja meg azon **int**-változó, valamint **int**-pointer *címét*, ahova el kell tárolnia a számsorozat elemszámát, illetve a dinamikusan lefoglalt tömbterület címét. Ugyancsak paraméterként van átadva a függvénynek az állomány neve mint **char**-pointer.

```
void beolvas(int*pn, int**pa, char*allomany)
{
    int i;FILE*f;
    f=fopen(allomany,"r");
    fscanf(f,"%d",pn);
    *pa=(int*)malloc((*pn)*sizeof(int));
    for(i=0; i<*pn; i++)
        fscanf(f,"%d",(*pa)+i);
    fclose(f);
}
main()
{
    int n, *a;
    beolvas(&n, &a, "be.txt");
    ...
}
```

8. A cím szerinti paraméterátadás megértése megmagyarázza, miért kell a **scanf** függvénynek változócímet átadni, és karakterlánc-olvasáskor miért egyszerűen a karaktertömb nevét kapja meg.

9.2. Paraméterátadás a STACK-en keresztül

A hívó függvény a Stack-re teszi fel azokat az adatokat, amelyeket a meghívott függvény rendelkezésére szeretne bocsátani. A klasszikus érték szerinti paraméterátadáskor az aktuális paraméterek értékét, a cím szerinti pedig az aktuális paraméterek címének az értékét teszi fel a Stack-

re. Tehát a paraméterátadás alapvetően érték szerint történik. Így a cím szerinti is érték szerinti, abban az értelemben, hogy a címek érték szerint adódnak át. Például a **csere** függvény esetében úgy adtuk át cím szerint **a**-t és **b**-t, hogy átadtuk érték szerint a címüket, **&a**-t és **&b**-t.

A cím szerinti paraméterátadás azért „erősebb”, mert ha megkapjuk egy változó értékét, ez még nem teszi lehetővé, hogy magához a változóhoz hozzáférjünk, de ha a címe értékét kapjuk meg, ezen keresztül elérhetjük azt. Szemléltetésül: ha rendelkezünk egy festmény másolatával, ez nem jelenti azt, hogy elérhető számunkra az eredeti is. Ha viszont megkapjuk a másolatát annak a címnek, amelyen a festményt őrzik, akkor ez éppen úgy elvezet a festményhez, mint ha az eredeti címmel rendelkeznénk.

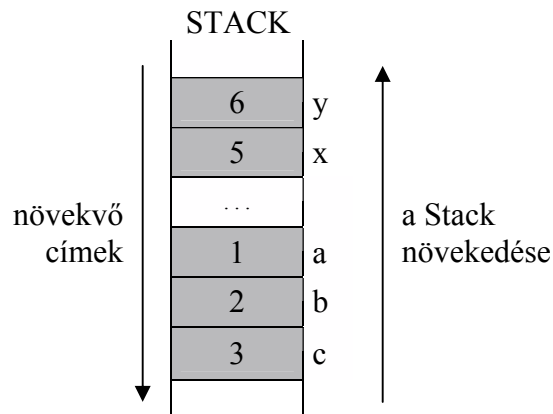
Tehát az érték szerinti paraméterátadáskor a formális paraméterek maguknak az átadandó adatoknak a másolatai, a cím szerintinél pedig a címük másolatai.

Egy másik szempont, amelyet fontos figyelembe venni, hogy a legtöbb fordító esetében függvényhíváskor az aktuális paraméterek jobbról balra értékelődnek ki, és értékeik ebben a sorrendben kerülnek fel a Stack-re. Ezt szemléltetik az alábbi példák (vegyük figyelembe, hogy a Stack kicsi címek fele nő):

```
int f(int *p)
{return ++(*p);}
main()
{
    int a = 1;
    printf("%d, %d, %d",f(&a), f(&a), f(&a));
    return 0;
}
```

A fenti program a következőket írja ki a képernyőre: 4,3,2

```
void g(int a, int b, int c)
{
    int x = 5, y = 6;
    printf("%p, %p, %p\n", &a, &b, &c);
    printf("%p, %p", &x, &y);
}
main()
{
    g(1, 2, 3); return 0;
}
```



A **main** függvény felteszi a Stack-re a függvényhívás aktuális paramétereit (először a 3-ast, azután a 2-est és végül az 1-est). A **g** függvény ugyancsak a Stack-en hozza majd létre a lokális változóit.

A függvénybeli kiírások célja bemutatni, milyen összefüggések állnak fenn a formális paraméterek címe, illetve a lokális változók címe között:

```
&b = &c - sizeof(int)
&a = &b - sizeof(int)
&x < &a
&y = &x - sizeof(int)
```

9.3. Globális és lokális változók

Eddigi programjainkban kizárólag *lokális* változókat használtunk. Ezeket a függvény keretén belül definiáljuk, és csakis az illető függvény végrehajtásának idejére jönnek létre a Stack-en. Ahogy a lokális név is utal rá, egy adott függvény lokális változói egy másik függvényből nem érhetők el. Ebből adódik, hogy a lokális változóikon keresztül a függvények nem tudnak egymással kommunikálni. Ezt a kommunikációt mind ez ideig a paraméterátadás, illetve az eredmény-visszatérítés által valósítottuk meg. Például így kommunikált minden programunk **main** függvénye a **scanf** könyvtári függvénnyel, vagy az imént a **main** függvény az általunk írt **faktorialis** függvénnyel. Bár továbbra is ez marad a függvények közti kommunikálás fő módja, létezik egy másik is a globális változók által.

A *globális változókat* minden függvényen kívül definiáljuk. Ezek a program végrehajtásának megkezdése előtt „születnek meg” (még a fordítás

9.3. GLOBÁLIS ÉS LOKÁLIS VÁLTOZÓK

169

időszakában) az adatszegmensen, és léteznek a teljes programvégrehajtás ideje alatt. A globális név arra utal, hogy minden függvényből elérhetők. Mivel a függvények közösen használják a globális változókat, ezért felhasználhatók a függvények közti kommunikációban.

A különböző függvényeknek nyilván lehetnek azonos nevű lokális változói. De mi a helyzet akkor, ha egy függvényben olyan lokális változót definiálunk, amilyen nevű globális változó már létezik? Ilyen esetben a lokális változó mintegy eltakarja a globálisat, és így az illető függvényből nem lesz elérhető a szóban forgó globális változó.

Az alábbi példa jobban megvilágítja mindezt:

```
int a, b, c; /* a, b és c globális változók */
void f1(int x, int y) /* x, y az f1 formális
    paraméterei (lokálisak f1-ben) */
{
    float t; int a; /* t, a az f1 lokális változói */
    ...
}
int f2(float x) /* x az f2 formális paramétere
    (lokális f2-ben) */
{
    int t, b; float a; /* t, b és a az f2 lokális
        változói */
    ...
    f1(t, b); /* lokális t és b */
    ...
    f1(b, c); /* lokális b és globális c */
    ...
}
main()
{
    float t, c; int t1; /* t, c és t1 a main lokális
        változói */
    ...
    f1(a, t1); /* globális a és lokális t1 */
    ...
    t1 = f2(t); /* lokális t és t1 */
    ...
    return 0;
}
```

Megjegyzések:

- **main** meghívja **f1**-et és **f2**-t. **f1** maga is meghívja kétszer **f2**-t.
- **f1**-ben elérhető a lokális **t** és **a**, az **x** és **y** formális paraméterek, valamint a globális **b** és **c**. A globális **a**-t eltakarja a lokális **a**.
- **f2**-ben elérhető a lokális **t**, **b** és **a**, az **x** formális paraméter, valamint a globális **c**. A globális **a**-t és **b**-t eltakarja a lokális **a** és **b**.
- **main**-ben elérhető a lokális **t**, **c** és **t1**, valamint a globális **a** és **b**. A globális **c**-t eltakarja a lokális **c**.

Hogyan hajtódik végre a példabeli program?

A fordítás fázisában létrejönnek az adatszegmensen az **a**, **b**, **c** globális változók.

Elkezdődik a main függvény végrehajtása.

Létrejönnek a STACK-en a **main** lokális változói: **t**, **c**, **t1**

...

Elkezdődik az f1 függvény végrehajtása (felfüggesztődik a **main** függvény).

Létrejönnek a STACK-en az **f1** formális paraméterei, az **x**, **y**, és megkapják a globális **a**, valamint a **main** lokális **t1**-jének értékeit.

Létrejönnek a STACK-en az **f1** lokális változói: **t**, **a**.

...

Felszámolódnak a STACK-ről az **f1** lokális változói és formális paraméterei.

Befejeződik az f1 végrehajtása, folytatódik a **main** függvény.

...

Elkezdődik az f2 függvény végrehajtása (újra felfüggesztődik a **main** függvény).

Létrejön a STACK-en az **f2** formális paramétere, az **x**, és megkapja a **main** lokális **t**-jének értékét.

Létrejönnek a STACK-en az **f2** lokális változói: **t**, **b**, **a**.

...

Elkezdődik az f1 függvény végrehajtása (felfüggesztődik az **f2** függvény).

Létrejönnek a STACK-en az **f1** formális paraméterei, az **x**, **y**, és megkapják az **f2** lokális **t**-jének és **b**-jének értékeit.

Létrejönnek a STACK-en az **f1** lokális változói: **t**, **a**.

...

Felszámolódnak a STACK-ről az **f1** lokális változói és formális paraméterei.

Befejeződik az f1 végrehajtása, folytatódik az **f2** függvény.

...

Elkezdődik az f1 függvény végrehajtása (újra felfüggesztődik az **f2** függvény).

Létrejönnek a STACK-en az **f1** formális paraméterei, az **x**, **y**, és megkapják az **f2** lokális **b**-jének és a globális **c**-nek értékeit.

Létrejönnek a STACK-en az **f1** lokális változói: **t**, **a**.

...

Felszámolódnak a STACK-ről az **f1** lokális változói és formális paraméterei.

Befejeződik az f1 végrehajtása, folytatódik az **f2** függvény.

...

Felszámolódnak a STACK-ről az **f2** lokális változói és formális paraméterei.

Befejeződik az f2 végrehajtása, folytatódik a **main** függvény.

...

Felszámolódnak a STACK-ről a **main** függvény lokális változói.

Befejeződik a main végrehajtása.

Felszámolódnak az adatszegmensről a globális változók.

9.4. Programtervezés kicsiben

Programtervezéskor többek között a következő kérdések merülnek fel:

1. Mit oldjunk meg a **main** függvényben, és mit osszunk le más függvényeknek? Eddig mindent a **main** függvényben oldottunk meg.
2. A függvények közti kommunikációban mit valósítsunk meg paraméterátadás által, és mit globális változókon keresztül?
3. Mely változók legyenek globálisak és melyek lokálisak?
4. Mely paramétereket adjuk át érték szerint, és melyeket cím szerint?

A következő feladat megoldásában megpróbálunk egy lehetséges választ adni ezekre a kérdésekre.

9.3. feladat. A **matrix.be** szöveges állományban adottak a sorok száma (**n**), az oszlopok száma (**m**), valamint egy **n** × **m**-es mátrix elemei. Ren-

dezzük külön-külön a sorait, majd az így kapott mátrixot írjuk ki a `matrix.ki` szöveges állományba.

Programtervezés:

1. A bemenő adatok beolvasását az `n` és `m` változókba, valamint az `a` kétdimenziós tömbbe, egy `beolvasas` nevű függvényre bízunk.
2. A mátrix kiírására a kimeneti állományba ugyancsak egy függvényt írunk. Legyen a neve `kiiras`.
3. A mátrix sorain a `main` függvényben fogunk végigmenni, de egy adott sor rendezését leosztjuk egy `sor_rendezes` függvénynek.
4. A rendezéshez szükséges csereművelethez a `sor_rendezes` függvény meg fog hívni egy `csere` függvényt.
5. Mivel az `n`, `m` és `a` változókat mind a `main`, mind a `beolvasas`, `kiiras` és `sor_rendezes` függvények kell hogy használják, ezért ezeket globálisnak definiáljuk.
6. A beolvasáshoz szükséges `i` és `j` ciklusváltozók lokálisak lesznek a `beolvasas` függvényben. Hasonló módon a kiíráshoz szükséges `i` és `j`, valamint a mátrix egy adott sorának rendezéséhez szükséges `i` és `j`, illetve a mátrixon sorról sorra való végigszaladáshoz szükséges `i` változók ugyancsak lokálisak lesznek a `kiiras`, `sor_rendezes`, illetve `main` függvényekben.
7. A `main` függvény érték szerint átadott paraméter által fogja közölni a `sor_rendezes` függvénnyel, hogy melyik sort rendezze.
8. A `sor_rendezes` függvény szükségszerűen cím szerint kell hogy átadja a `csere` függvénynek a felcserélendő elemeket.

```
#include <stdio.h>
int n, m, a[50][50];
void beolvasas();
void kiiras();
void sor_rendezes(int);
void csere(int *, int *);
main()
{
    int i;
    beolvasas();
    for(i = 0; i < n; i++)
        sor_rendezes(i);
    kiiras();
    return 0;
}
void beolvasas()
{
```



```

    int i, j; FILE * f;
    f = fopen("matrix.be", "r");
    fscanf(f, "%d%d", &n, &m);
    for(i = 0; i < n; i++)
        for(j = 0; j < m; j++)
            fscanf(f, "%d", &a[i][j]);
    fclose(f);
}
void kiiras()
{
    int i, j; FILE * f;
    f = fopen("matrix.ki", "w");
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < m; j++)
            fprintf(f, "%5d", a[i][j]);
        fprintf(f, "\n");
    }
    fclose(f);
}
void sor_rendezes(int k)
{
    int i, j;
    for(i = 0; i < m - 1; i++)
        for(j = i + 1; j < m; j++)
            if (a[k][i] > a[k][j]) csere(&a[k][i], &a[k][j]);
}
void csere(int * p1, int * p2)
{
    int v;
    v = *p1; *p1 = *p2; *p2 = v;
}

```

A fenti programtervezés indoklása

1. Mivel a függvények globális változókon keresztül jutnak hozzá **n**-hez, **m**-hez és **a**-hoz, ezért kevésbé általánosak. Például a **beolvasas** függvény nem lenne alkalmas egy mátrix adatainak **p** (sor), **q** (oszlop) és **b** (elemek) változókba való beolvasására. De ebben a feladatban erre nincs is szükség.
2. A fentiekkel ellentétben, mivel a **sor_rendezes** paraméteren keresztül kapja meg a rendezendő sor indexét, ezért az **a** tömb bármelyik sorának rendezésére alkalmas, és erre szükség is van.
3. Szükségtelen lett volna a sorindexet cím szerint átadni a **sor_rendezes** függvénynek, hiszen csak az értékére van szüksége.

Ezzel ellentétben a **csere** függvénynek magukra az elemekre van szüksége a felcserélésükhöz.

4. Ha kényelemből (hibásan) globálisnak definiáltuk volna az *i* és *j* változókat, hogy ne kelljen minden függvényben külön deklaráljuk, akkor a **main** függvény *i* ciklusváltozóját „elrontotta” volna a **sor_rendezer** függvény.
5. *Következésképpen, bár a globális változók használata kényelmesnek tűnik, általában kerülendő. Célszerű előnybe helyezni a paraméterátadást és a lokális változókat. Sőt, egyes pedagógusok azt ajánlják a programozni tanulóknak, hogy teljességgel kerüljék a globális változók használatát. Ez egészséges programozói stílus kialakításához vezet, amely többek között azt is magába foglalja, hogy olyan függvényeket írunk, amelyek bármely alkalmazásban értékesíthetők. Továbbá felkészít nagy projektek „csapatmunkával” való megvalósítására. A programozói csapatmunka egyik alapelve, hogy a projekt függvényei kizárólag paraméterlistán keresztül kommunikálnak, és csak lokális változókat használnak.*

Következzék hát a feladat megoldása globális változók nélkül:

```
#include <stdio.h>
void beolvasas(int *, int *, int (*)[50], char *);
void kiiras(int, int, int (*)[50], char *);
void sor_rendezer(int, int, int[][50]);
void csere(int *, int *);
main()
{
    int n, m, a[50][50], i;
    beolvasas(&n, &m, a, "matrix.be");
    for(i = 0; i < n; i++)
        sor_rendezer(i, m, a);
    kiiras(n, m, a, "matrix.ki");
    return 0;
}
void beolvasas(int* pn,int* pm,int(*a)[50],char*allomany)
{
    int i, j; FILE * f;
    f = fopen(allomany, "r");
    fscanf(f, "%d%d", pn, pm);
    for(i = 0; i < *pn; i++)
        for(j = 0; j < *pm; j++)
            fscanf(f, "%d", &a[i][j]);
    fclose(f);
}
```

```

}
void kiiras(int n, int m, int (*a)[50], char * allomany)
{
    int i, j; FILE * f;
    f = fopen(allomany, "w");
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < m; j++)
            fprintf(f, "%5d", a[i][j]);
        fprintf(f, "\n");
    }
    fclose(f);
}
void sor_rendezes(int k, int m, int a[][50])
{
    int i, j;
    for(i = 0; i < m - 1; i++)
        for(j = i + 1; j < m; j++)
            if (a[k][i] > a[k][j]) csere(&a[k][i], &a[k][j]);
}
void csere(int * p1, int * p2)
{
    int v;
    v = *p1; *p1 = *p2; *p2 = v;
}
}
    
```

Megjegyzések ezen második megoldáshoz:

1. A **beolvasas** függvény most már alkalmas lenne egy mátrix adatainak a beolvasására, bármilyen nevű állományból bármilyen nevű memóriaváltozókba. Ugyanez igaz a **kiiras** és **sor_rendezés** függvényekre is.
2. A **beolvasas** függvénynek nyilván cím szerint kellett átadnunk az **n**, **m** és **a** változókat, hiszen a beolvasásnak ezekbe kellett megtörténnie és nem a másolatukba. Azért nem kellett külön jelezni az **a** tömb cím szerint való átadását, mert, amint említettük, a C nyelvben a tömbök kivételesen implicite cím szerint adódnak át. A **kiiras** és **sor_rendezés** függvények esetében elégséges volt **n** és **m** érték szerinti átadása.
3. Úgy is mondhatnánk, hogy a **main** függvény a saját **n**-jét, **m**-jét és **a**-ját bocsátja a cím szerinti paraméterátadás által a **beolvasas** függvény rendelkezésére, míg a másik kettőnek csupán **n** és **m** értékét adja oda.

Ha dinamikusan szeretnénk helyet foglalni egy tömbnek, ezt a feladatot is kioszthatjuk egy függvénynek.

A `letrehoz_1` egy `n` elemű egydimenziós `int` típusú tömböt hoz létre.

```
int * létrehoz_1(int n)
{
    int * tp;
    tp = (int*)malloc(n * sizeof(int));
    return tp;
}
```

Hogyan használható a fenti függvény?

```
int * a;
a = létrehoz_1(n); /* a tömb i-edik elemére való hivatkozás:
a[i] */
```

A `letrehoz_2` `n x m`-es `float` típusú tömböknek foglal helyet. Pontosabban, egy `n` elemű `float`-pointer tömböt hoz létre, amelynek minden eleme egy `m` elemű `float`-tömbre mutat.

```
float ** létrehoz_2(int n, int m)
{
    float ** tpp; int i;
    tpp = (float**)malloc(n * sizeof(float*));
    for(i = 0; i < n; i++)
        tpp[i] = (float*)malloc(m * sizeof(float))
    return tpp;
}
```

Kétdimenziós tömb létrehozása a `letrehoz_2` segítségével:

```
float ** b;
b = létrehoz_2(n, m); /* b[i][j], a tömb i-edik sorának
j-edik eleme */
```

Amíg az `a` tömb törölhető a `free(a)`; utasítással, addig `b` törlését az alábbi függvény valósítja meg:

```
void torol_2(float ** b, int n)
{ int i;
  for(i = 0; i < n; i++) free(b[i])
  free(b);}
```

9.5. Függvénypointerek

Egy függvény típusán eddig a visszatérési értékének típusát értettük. Azonban, ha egészen pontosak akarunk lenni, akkor ez magába foglalja még a paraméterek számát és típusát is.

Legyen a következő függvénydeklaráció:

```
int f(char, float);
```

f egy **char** és **float** paraméterű **int**-függvény. **f** típusa leírva:

```
int (char, float).
```

Hogyan definiálunk egy függvénypointert?

Például így:

```
int (*fp)(char, float);
```

fp, egy **char** és **float** paraméterű **int**-függvénypointer. **fp** típusa leírva: **int(*) (char, float)**.

typedef-et használva nevet adhatunk egy függvénytípusnak vagy függvénypointer-típusnak.

Példa:

```
typedef float FV_TIP (double, int**, char[]);
typedef int (*FV_PTR_TIP) (int, int);
FV_TIP fuggv; /* függvénydeklaráció, prototípus */
FV_PTR_TIP f_ptr; /* függvénypointer-definiálás */
```

Egy függvény neve az illető függvénytípusú pointerkonstansként is felfogható, amely magára a függvényre mutat. Tehát **f** egy **int(*) (char, float)** típusú függvénypointer-konstansnak is tekinthető. Ezért az **f**, illetve ***f** hivatkozások egyenértékűek.

Az **fp** pointer megkaphatja bármely **char** és **int** paraméterű **int**-függvény címét.

Példa:

```
fp = &f; ezzel ekvivalens: fp = f;
```

Az érem másik oldala az, hogy bármely függvénypointer felfogható annak a függvénynek a nevéként, amelyre mutat. Így az **fp** és ***fp** hivatkozások is egyenértékűek.

Az **f** függvényt a következő módokon lehet meghívni:

```
int x; char a; float b;
...x = f(a, b); ... /* klasszikus forma */
...x = (*fp)(a, b); ... /* pointeren keresztül */
...x = (*f)(a, b); ... /* a nevéen mint pointeren kereszt-
                        túl */
...x = fp(a, b); ... /* pointeren mint nevéen keresztül */
```

Megjegyzés. Mivel egy függvény típusa – az adattípusokkal ellentétben – nem határozza meg kódjának memóriaméretét, ezért a függvénypointerekre nem érvényesek a címaritmetika műveletei. Például egy függvénypointerhez nem adható hozzá egy egész szám, vagy az azonos típusú függvénypointerek nem vonhatók ki egymásból.

9.6. Függvény paraméterként való átadása függvénynek

A függvényátadás mindig cím szerint történik.

9.4. feladat. Írjunk egy olyan rendező függvényt, amely rendezi egy tömb elemeit, függetlenül ezek típusától.

Megoldás: Mivel a rendezéshez szükséges összehasonlítás típusfüggő, ezért minden típusra írunk egy összehasonlító függvényt, amelyet majd átadunk a rendező függvénynek. A rendező függvény paraméterként a tömb címét, elemeinek számát és méretét, valamint az összehasonlító függvényt fogja megkapni. Azért, hogy a különböző összehasonlító függvények azonos típusúak legyenek, az összehasonlítandó elemek címét mint **const void** pointereket fogják megkapni.

```
#include<sdtlib.h>
#include<string.h>
int int_cmp(const void * p1, const void * p2)
{ return * (int *) p1 - * (int *) p2; }

int float_cmp(const void * p1, const void * p2)
{ if (* (float *) p1 == * (float *) p2) return 0;
  else if (* (float *) p1 < * (float *) p2) return -1;
  else return 1; }

void sort(void*v,int nr,int size,int(*p_cmp)(const void*,const void*))
{
    int i, j; void *pv, *pi, *pj;
    pv = malloc(size); /* a veder szerepét fogja betölteni */
    for(i = 0, pi=v; i < nr - 1; i++, pi=(char*)pi+size)
    {
        for(j = i + 1, pj=(char*)pi+size; j < nr; j++, pj=(char*)pj+size)
        {
            if (p_cmp(pi, pj) > 0)
            {
                memcpy(pv, pi, size);
                memcpy(pi, pj, size);
            }
        }
    }
}
```

```

        memcpy(pj, pv, size);
    }
}
free(pv);
}
main()
{
    /* Ha az a tömb egy n elemű int sorozatot tartalmaz, b pedig egy
    m elemű float sorozatot, akkor ezek a következőképpen rendezhetők
    a sort függvénnyel */
    sort(a, n, sizeof(int), int_cmp);
    sort(b, m, sizeof(float), float_cmp);
    return 0;
}

```

Megjegyzés. A C nyelv függvénykönyvtárában létezik **qsort** nevű hatékony tömbrendező függvény. Prototípusa hasonló az általunk írt **sort** prototípusához:

```

void qsort(void * tomb_cim,
           int elem_szam,
           int elem_meret,
           int(*cmp)( const void*, const void*));

```

9.7. Változó paraméterszámú függvények

Az eddig írt függvényeinknek volt egy jellemvonásuk: mindig ugyanannyi paraméterrel hívtuk meg őket. Ez természetesnek tűnik is, hisz azt mondtuk, hogy a formális és az aktuális paraméterek száma meg kell hogy egyezzen. A szemléltetésünkben is ez tükröződött: ahány szereplője van egy darabnak, annyi színészre van szükség az eljátszásához.

De hát mi a helyzet akkor az olyan könyvtári függvényekkel, mint a **printf** és a **scanf**, amelyek „bármilyen” paraméterszámmal meghívhatók? A válasz a *változó paraméterszámú* függvényekben rejlik.

Egy konkrét példán fogjuk bemutatni ezt a különleges függvényosztályt.

9.5. feladat. Írjunk egy **osszeg** nevű függvényt, amely paraméterként megkapja az összeadandók számát, valamint magukat az összeadandókat (valós számok), és kiírja az összeget a képernyőre.

Az `double_osszeg` függvényt például az alábbi módokon szeretnénk meghívni:

```
main()
{
    double_osszeg(5, 3.1, 5.2, -2.1, 4.02, 1.0); /* a
        képernyőn megjelenik 11.22 */
    double_osszeg(2, 3.14, 1.1); /* a képernyőn
        megjelenik 4.24 */
    return 0;
}
```

Hogyan történik egy ilyen függvény deklarációja, illetve definiálása?

```
void double_osszeg (int n, ...); /* deklaráció */
void double_osszeg (int n, ...) /* definíció */
{
    double * p, s = 0;
    p = (double *)&n + 1;
    for(; n; n--)
        { s += * p; p++; }
    printf("%5.2lf\n", s);
}
```

Megjegyzések:

1. Legalább egy fix paraméterrel kell hogy rendelkezzenek a változó paraméterszámú függvények is. Például a `printf` és a `scanf` esetében a fix paraméter egy `const char *` típusú pointer, amely a formázósorra mutat.
2. Azok a paraméterek, amelyek vagy megjelennek, vagy nem, a fix paraméterek után kell hogy következzenek. Mivel számuk és típusuk ismeretlen a függvény deklarálásakor, ezért „...”-val értesítjük róluk a fordítót.
3. A kulcsszerep a fix paramétereknek jut, ugyanis ezek elegendő információt kell hogy tartalmazzanak a többi paraméterről. A `double_osszeg` függvény `n` paramétere például megmondja, hány `double` paraméter követi még (`double_osszeg`, egy kifejezetten `double` összeadó). A `scanf` és a `printf` formázósorából nemcsak a további paraméterek száma olvasható ki, hanem a típusaik is.
4. Hogyan érjük el ezeket a további paramétereket a függvényben? Emlékezzünk, hogy paraméter-átadáskor a paraméterek értékei a STACK-en keresztül adódnak át a függvénynek. A STACK-en ezek az értékek egymás után következő memóriarekeszekbe kerülnek. Ahol befejeződik

az egyik paraméter, közvetlenül utána kezdődik a másik. Mivel a fix paraméterek elérhetők, ezért könnyen meghatározható, hol kezdődnek a többiek: az utolsó fix paraméter után. Erre a memóriacímre állítottuk a `p_double` pointert.

5. Tehát a fix paraméterek segítenek annak meghatározásában, hogy hol vannak a memóriában – a függvény végrehajtása idején – a további paraméterek, és hogy hányan vannak. Sőt további információt is hordozhatnak róluk.
6. Az `osszeg` függvény `for` ciklusában összeadjuk azt az `n` darab `double` értéket, amelyek követik `n`-et a `STACK`-en.

Kényelmesebbé tehetik a változó paraméterszámú függvények írását az alábbi könyvtári függvények: `va_start`, `va_arg`, `va_end`. Ezen függvények deklarációi az `stdarg.h` header-állományban találhatók.

Íme az előbbi függvény a segítségükkel:

```
void double_osszeg (int n, ...) /* definíció */
{
    double d, s = 0;
    va_list argptr;
    va_start(argptr, n);
    for(; n; n--)
    {
        d = va_arg(argptr, double);
        s += d;
    }
    va_end(argptr);
    printf("%5.2lf\n", s);
}
```

Megjegyzések:

1. a `va_list` egy mutatótípus, amely az `stdarg.h` állományban van definiálva. Egy ilyen típusú pointert definiáltunk, az `argptr`-t.
2. a `va_start` ráállítja `argptr`-t az utolsó fix paramétert (jelen esetben az `n`-t) követő első paraméterre.
3. a `va_arg` „kiolvas” egy adott típusú értéket (jelen esetben a `double`-t) az `argptr` címről, és lépteti „típusnyit” a pointert.
4. a `va_end` meghívása biztosítja a `STACK` rekonstrukcióját.

9.6. feladat. Írjunk egy `myprintf` nevű változó paraméterszámú függvényt, amely képes kiírni `char`, `int`, `long`, `float`, `double` és karakterlánc

típusú értékeket (az egyszerűség kedvéért megengedjük, hogy használja a `printf`-et).

```
#include<stdarg.h>
#include<stdio.h>
void myprintf(const char *fs, ...)
{
    va_list arg_ptr;
    va_start(arg_ptr, fs);
    while(*fs)
    {
        if (*fs=='%')
        {
            fs++;
            switch(*fs)
            {
                case 'c':printf("%c", va_arg(arg_ptr, int));break;
                case 'd':printf("%d", va_arg(arg_ptr, int));break;
                case 'f':printf("%f", va_arg(arg_ptr, double));break;
                case 's':printf("%s", va_arg(arg_ptr, char*));break;
                case 'l':fs++;
                    if(*fs=='d')
                        printf("%ld", va_arg(arg_ptr, long));
                    else
                        printf("%lf", va_arg(arg_ptr, double));
            }
            fs++;
        }
        else {putchar(*fs);fs++;}
    }
    va_end(arg_ptr);
}

main()
{
    char c='A'; long b=12; float d=3.14;
    myprintf("%c\n%d\n%ld\n%f\n%lf\n%s", c, 256, b, d, 1.41, "alma");
    return 0;
}
```

Magyarázat: A `main` függvénybeli `myprintf` függvényhívásának az aktuális paramétereinek típusai rendre: `char*`, `char`, `int`, `long`, `float`, `double`, `char*`. A két karakterlánc konstans a `string`-literációk táblázatába kerül eltárolásra, és az itteni címük adódik át a `myprintf` függvénynek. A `myprintf` függvény az `fs` fix paraméterben a formázósornak mint karakterlánc-konstansnak a `string`-literációk táblázatbeli címét kapja meg. A `while` ciklus formázó karakterről formázó karakterre lépteti az `fs`

mutatót. A `myprintf` függvény a formázókarakterek számából és típusából szerez információt a többi paraméteréről.

Figyelem!

A régebbi verziókkal való kompatibilitás érdekében a karakterek, függetlenül attól, hogy konstansok vagy változók, `int`-ként adódnak át. Hasonlóképpen a `float`-ok `double`-ként adódnak át.

9.8. Parancssor-argumentumok (a `main` függvény paraméterei)

Felmerül a kérdés: ki hívja meg a `main` függvényt? Az operációs rendszer, hiszen a `main` függvény meghívása jelenti a program elindítását. Ugyancsak az operációs rendszer az, amely megkapja a `main` által esetleg visszatérített értéket. Adhat-e át paramétereket az operációs rendszer a `main` függvénynek? A válasz igen, és ezeket nevezzük *parancssor-argumentum*oknak.

Amikor parancssorból indítjuk el a programunkat, akkor általában begépeljük a nevét és `enter`-t ütünk. A program nevét azonban követhetik paraméterek (argumentumok). Sőt magának a programnak a nevét is úgy tekintjük, mint a 0-dik argumentumot. Az argumentumokat legalább egy `space`-szel, vagy `tab`-bal választjuk el egymástól.

A programindítás szintaxisa parancssorból:

```
<program_nev> <arg_1> <arg_2> ... <arg_n>
```

Ha a C környezetből indítjuk a programunkat, akkor a `parameters` vagy `arguments` párbeszédablakban adhatók meg az argumentumok (a program nevét ide nem kell beírni).

Hogyan érhetjük el a `main` függvényben a parancssor-argumentumokat?

A `main` függvény két paramétere segítségével. Az első paraméter `int` típusú, és a parancssor-argumentumok számát tartalmazza, a második pedig egy `char` pointertömb, amely pointereket tartalmaz felénk mint karakterlánc-konstansok felé.

Az alábbi program kiírja a képernyőre a parancssor-argumentumokat és számukat:

```
main(int argc, char * argv[])
{
    int i;
```

```
printf("Az argumentumok száma: %d\n", argc);
for(i = 0; i < argc; i++)
    printf("Az %d-ik argumentum: %s\n", i, argv[i]);
return 0;
}
```

Megjegyzések:

1. Feltételezve, hogy programunknak a `teszt.c` nevet adjuk, a `teszt` `alma` `korte` `dio` programindítás nyomán, a képernyőn a következő jelenik meg:

```
Az argumentumok száma: 4
Az 0-ik argumentum: teszt
Az 1-ik argumentum: alma
Az 2-ik argumentum: korte
Az 3-ik argumentum: dio
```
2. Természetesen nem kötelező az `argc` (*argument counter*) és az `argv` (*argument vector*) paraméterneveket használni, bár ezek a megszokott elnevezések.
3. Mivel `argv` egy `char` pointertömb neve, ezért alapvetően `char **` típusú, és így is definiálható: `char ** argv`.
4. Gyakran állományneveket adunk át a `main` függvénynek parancssor-argumentumon keresztül. Ha ezek elérési utakat is tartalmaznak, nem kell a `backslash`-eket megduplázni a parancssorban, hiszen ez automatikusan megtörténik a paraméterátadáskor.

9.7. feladat. Adott két bemeneti állomány: egyik egy `n` elemű számsorozat, a másik pedig egy `n × n` méretű mátrixot tartalmaz. Ellenőrizzük, hogy a számsorozat azonos-e a mátrix főátlójának fentről lefele való bejárásával. A *bemeneti állományok nevét parancssorból vesszük át.*

```
main(int argc, char **argv)
{
    FILE * f, * g;
    int i, j, n, m, a[50], b[50][50];
    if (!(f = fopen (argv[1], "r")))
        {puts("Hiba1"); return 0;}
    if (!(g = fopen (argv[2], "r")))
        {puts("Hiba2"); return 0;}
    fscanf(f, "%d", &n);
    for(i=0;i<n;i++)
        fscanf(f, "%d", &a[i]);
    fclose(f);
    fscanf(g, "%d", &m);
```

```

for(i=0;i<m;i++)
for(j=0;j<m;j++)
    fscanf(g, "%d", &b[i][j]);
fclose(g);
if(n!=m)
{puts("Nem találunk a méretek");return 0;}
for(i=0;i<n;i++)
if(a[i]!=b[i][i])
    {puts("Nem FŐÁTLÓ");return 0;}
puts("FŐÁTLÓ");
return 0;
}

```

9.9. Rekurzív függvények

Rekurzivitásról akkor beszélünk, amikor egy függvény önmagát hívja meg (önrekurzió vagy direkt rekurzió). Ha ezt közvetve teszi meg, azaz meghív egy függvényt, amely aztán meghívja őt, akkor *kölcsönös* vagy *indirekt rekurzióról* van szó.

Az A. Függelékben közöljük a szerző egy cikkének a kivonatát, amely egy diáklapban jelent meg, és a következő címet viselte: *Rekurzió egyszerűen és érdekesen*. Mivel a cikk részletesen bemutatja a rekurzió mechanizmusát, itt csupán két példán keresztül mutatjuk be azt.

```

int fibonacci(int n)
{
    if(n==0 || n==1) return n;
    else return fibonacci(n-1)+fibonacci(n-2);
}
typedef void fv(int); /* a függvények típusa */
fv fv_a, fv_b; /* a függvények prototípusa */
void fv_a(int i)
{
    ...
    if(i>0) fv_b(i/2);
    ...
}
void fv_b(int j)
{
    ...
}

```

```
if(j>0) fv_a(j-1);
...
}
```

Feltételezve, hogy az olvasó átnézte a fentebb említett függeléket, legyen az alábbi megoldott feladat:

Írjunk rekurzív függvényt, amely meghatározza egy n elemű bitsorozat paritását. (A paritás 1, ha a sorozat páratlan számú 1-es bitet tartalmaz, különben 0).

1. A bitsorozat egy n számjegyű `unsigned long int` tartománybeli természetes szám számjegyei révén adott (feltételezzük, hogy az első bit 1-es és $n \leq 10$).
2. A bitsorozat az n elemű bináris x számsorozat által adott.
3. A bitsorozatot egy n hosszú s karakterláncként kapja meg a rekurzív függvény.
4. A bitsorozatot egy `unsigned long int` érték (w) belső ábrázolásának az n darab kisebb helyértékű bitje jelenti ($n \leq \text{sizeof}(\text{unsigned long int})$).

Megoldás_1: (Az " n feladatot" visszavezetjük az " $(n/10)$ feladatra")

- A feladat akkor triviális, ha $n == 0$. A nulla paritása 0.
- Általános esetben:
 - n paritásának meghatározása visszavezethető $(n/10)$ paritására.
 - Ha n utolsó számjegye nulla ($(n\%10) == 0$), akkor n paritása azonos $(n/10)$ paritásával, különben annak fordítottja.

```
int paritas1(int n)
{
    if (n == 0) return 0;
    else
    {
        int talca = paritas1(n/10);
        if (n%10 == 1) return !talca;
        else return talca;
    }
}
```

Megoldás_2: (Az " (x,n) feladatot" visszavezetjük az " $(x,n-1)$ feladatra")

- A feladat akkor triviális, ha a számsorozat üres ($n == 0$). Ez esetben a paritás nulla.

- Általános esetben:
 - Az első n elem alkotta számsorozat paritásának meghatározása visszavezethető az első $(n-1)$ elemből álló részsorozat paritására.
 - Ha az n -edik elem nulla ($a[n-1] == 0$), akkor az "(x,n) feladat" paritása azonos az "(x,n-1) feladat" paritásával, különben annak fordítottja.

```
int paritas2(int *x, int n)
{
    if (n == 0) return 0;
    else
    {
        int talca = paritas2(x, n-1);
        if (a[n-1] == 1) return !talca;
        else return talca;
    }
}
```

Megoldás_3: (Az " s feladatot" visszavezetjük az " $(s+1)$ feladatra")

- A feladat akkor triviális, ha a karakterlánc üres ($s[0] == '\0'$). Ez esetben a paritás nulla.
- Általános esetben:
 - Az s címen kezdődő karakterlánc paritásának meghatározása visszavezethető az $(s+1)$ címen kezdődő részkarakterlánc paritására.
 - Ha az s címen lévő karakter '0' ($s[0] == '0'$), akkor az " s feladat" paritása azonos az " $(s+1)$ feladat" paritásával, különben annak fordítottja.

```
int paritas3(char *s)
{
    if (s[0] == '\0') return 0;
    else
    {
        int talca = paritas3(s+1);
        if (s[0] == '1') return !talca;
        else return talca;
    }
}
```

Megoldás_4: (A "w feladatot" visszavezetjük a "(w»1) feladatra")
(Megfigyelhető, hogy w utolsó n bitjének paritása azonos w paritásával)

- A feladat akkor triviális, ha w egyenlő nullával. Ez esetben a paritás nulla.
- Általános esetben:
 - A w paritásának meghatározása visszavezethető a (w»1) paritására.
 - Ha w legkisebb helyértékű bitje 0 ((w & 1UL) == 0), akkor a "w feladat" paritása azonos az "(w»1) feladat" paritásával, különben annak fordítottja.

```
int paritas4(unsigned long int w)
{
    if (w == 0) return 0;
    else
    {
        int talca = paritas4(w»1);
        if ((w & 1UL) == 1) return !talca;
        else return talca;
    }
}
```

9.10. Kitűzött feladatok

(Egyes feladatok esetében szükséges lehet a függvények megtervezése. Például melyek legyenek a paraméterei, hogyan történjen a paraméterátadás stb. Mindenik feladatban, amennyiben erre szükség van, az átlományneveket parancssorból vegyük át.)

1. Helyettesítsd a kérdőjeleket úgy, hogy szintaktikailag helyes, működőképes programot kapj, amelyben nem kerül sor egyetlen implicit típuskonverzióra sem.

```
#include <stdio.h>
?
jonatan alma;
main()
{
    printf("%lf", alma(1,1.0,'A'));
    return 0;
}
```



```
? alma (? a, ? b, ? c)
{return ? a + ? b + ? c ;}
```

2. Írjunk függvényt, amely a paraméterként kapott
 - a) természetes számról eldönti, hogy prím-e,
 - b) természetes számnak visszatéríti a fordítottját,
 - c) két természetes számnak meghatározza a legnagyobb közös osztóját,
 - d) két természetes számnak meghatározza a legkisebb közös többszörösét,
 - e) természetes számból törli a nulla számjegyeket, és visszatéríti a kitörölt számjegyek számát,
 - f) természetes számot úgy alakítja át, hogy rendezve legyenek a számjegyei.
3. Írjunk függvényt, amely paraméterként megkapja egy tömb címét és a benne tárolt számsorozat elemeinek számát, és
 - a) ellenőrzi, hogy a számsorozat elemei növekvő sorrendben vannak-e,
 - b) képezi a számsorozat szimmetrikusát,
 - c) rendezi a számsorozatot az elemek utolsó számjegyei szerint,
 - d) töröl egy harmadik paraméterként kapott értéket a rendezett számsorozatból,
 - e) beszúr egy harmadik paraméterként kapott értéket a rendezett számsorozatba,
 - f) számokat olvas be 0 végjelig, és ezeket beszúrja a rendezett számsorozatba.
4. Írjunk függvényt, amely paraméterként megkap egy polinomot és egy értéket, és visszatéríti a polinom értékét az adott helyen.
5. Írjunk függvényeket, amelyek paraméterként megkapnak két polinomot, és meghatározzák az összeg, különbség, szorzat, hányados és maradék polinomokat.
6. Írjunk függvényt, amely paraméterként megkap
 - a) egy karakterláncot, és eldönti, palindrom-e,
 - b) két karakterláncot és eldönti, hogy egymás anagrammái-e.
7. Írjunk saját függvényeket, amelyek az alábbi könyvtári függvényeket valósítják meg: **strlen**, **strcmp**, **strcpy**, **strcat**, **strchr**, **strstr**
8. Írjunk függvényt, amely megvalósítja egy karakterlánc beszúrását egy másik karakterláncba egy adott helyre.
9. Írjunk függvényt, amely kitöröl egy karakterláncból egy adott helyről adott számú karaktert.

10. Írjunk függvényt, amely egy mátrix sorait, illetve oszlopait úgy rendezi át, hogy a főátlón lévő elemek növekvő sorrendben legyenek. Két sor, illetve oszlop felcseréléséhez használjunk segédfüggvényt.

11. Írjunk függvényeket, amelyek két mátrixnak meghatározzák az összegét, különbségét, szorzatát.

12. Írjunk függvényt, amely dinamikusan létrehoz paraméterként megadott méretű egydimenziós tömböt, feltölti véletlen egészekkel egy paraméterként megadott intervallumból, majd visszatéríti a tömb címét.

13. Írjunk függvényt, amely dinamikusan létrehoz paraméterként megadott méretű kétdimenziós tömböt, feltölti véletlen egészekkel egy paraméterként megadott intervallumból, majd visszatéríti a tömb címét.

14. Írjunk függvényt, amely dinamikusan létrehoz egy olyan karakterláncot, amely az angol ábécé nagybetűit tartalmazza ábécésorrendben.

15. Adott n személy neve és címe (város, utca, szám) egy szöveges állományban. Van-e olyan személy, akinek városneve az utcanéve fordítottja? (Függvénnyel ellenőrizd a két karakterlánc palindromságát.) Hány személy lakik prím szám alatt? (Függvénnyel ellenőrizd egy számról, hogy prím-e.)

16. Írjunk olyan kereső függvényt, amely bármilyen típusú tömbben megkeres egy elemet (paraméterként kapjon egy összehasonlító függvényt).

17. Írjunk olyan kereső függvényt, amely visszatéríti bármilyen típusú tömb legkisebb elemének címét (paraméterként kapjon egy összehasonlító függvényt).

18. Rendezzük `qsort`-tal egy kétdimenziós tömb sorait úgy, hogy az első oszlop elemei csökkenő sorrendben legyenek.

19. Adottak n személy adatai (név, cím, születési dátum). Rendezzük az adatokat `qsort`-tal a következő kritériumok szerint: név, város, születési év, születési dátum.

20. Adottak n személy adatai (név, cím, születési dátum), amelyeket egy `struct` típusú tömbben tárolunk. Egy `struct`-pointer típusú tömb elemeit állítsuk rá a `struct` tömb elemeire. Rendezzük a pointereket `qsort`-tal az általuk megcímzett `struct`-ok következő mezői szerint: név, város, születési év, születési dátum. Írjuk ki a rendezett adatsorokat (a beolvasáshoz és kiíráshoz is használjunk függvényt).

21. Írjunk függvényt, amely bármilyen típusú mátrix elemeit rendezi sorfolytonos bejárás szerint (paraméterként kapjon egy összehasonlító függvényt).

22. Írjuk meg a `myscanf` függvényt néhány típusra.

23. Írjunk rekurzív függvényt, amely meghatározza egy paraméterként kapott természetes szám

- a) számjegyeinek összegét,
- b) számjegyeinek szorzatát,
- c) eldönti róla, hogy tartalmaz-e 5-ös számjegyet,
- d) 1-es számjegyeinek számát,
- e) legkisebb számjegyét,
- f) legnagyobb számjegyét,
- g) fordítottját.

24. Írjunk rekurzív függvényt, amely meghatározza egy paraméterként kapott számsorozat

- a) elemeinek összegét,
- b) legkisebb elemének értékét,
- c) legnagyobb elemének pozícióját,
- d) pozitív elemeinek számát,
- e) tartalmaz-e páros elemet.

25. Írjunk rekurzív függvényt, amely egy bináris számsorozatnak meghatározza a paritását. A paritás 1, ha az egyesek száma páratlan, ellenkező esetben 0.

26. Írjunk rekurzív függvényt, amely beszűrő rendezéssel rendez egy számsorozatot.

27. Írjunk rekurzív függvényt, amely kiírja egy számsorozat elemeit fordított sorrendben.

28. Írjunk rekurzív függvényt, amely egész számokat olvas be nulla végjelig, majd kiírja az utolsó n számot fordított sorrendben.

29. Írjunk rekurzív függvényt, amely kiírja egy számsorozat pozitív elemeit a beolvasás sorrendjében, a negatívakat pedig a beolvasás fordított sorrendjében.

30. Írjunk rekurzív függvényt, amely egész számokat olvas be vakon (írjunk egy ilyen beolvasó függvényt) nulla végjelig, majd kiírja a páratlan sorszámúakat a beolvasás sorrendjében, és alájuk a páros sorszámúakat fordított sorrendben.

31. Írjunk rekurzív függvényt, amely kiírja egy karakterlánc páratlan sorszámú szavait a beolvasás sorrendjében, és alájuk a páros sorszámúakat fordított sorrendben.

32. Írjunk rekurzív függvényt, amely ugyanazt valósítja meg, mint az `strcmp` könyvtári függvény.

33. Írjunk rekurzív függvényt, amely visszatéríti az n -edik fibonacci számot.

34. Írjunk rekurzív függvényt, amely visszatéríti a $C(n, k)$ értékét.
35. Írjunk rekurzív függvényt, amely egy n méter hosszú rudat fel-
darabol egy- és kétméteres rudakra az összes lehetséges módon.
36. Írjunk rekurzív rendező függvényt (*quick sort*) a szétválogatás
algoritmusra (lásd a 6.1. alfejezetet).
37. Írjunk rekurzív rendező függvényt (*merge sort*) az összefésülő
algoritmusra (lásd a 6.1. alfejezetet).
38. Adottak n személy adatai (vezetéknév, keresztnév, születési dá-
tum – év, hónap, nap –, cím – város, utca, házszám). Írj programot,
amely menüből választhatóan rendez az adatokat valamely mező sz-
erint. Használjuk a *qsort* könyvtári függvényt, és készítsünk egy függvény-
pointertömböt, amelynek elemei a mezők szerinti összehasonlító füg-
gvényekre mutassanak.
39. Mit ír ki az alábbi programrészlet (anélkül válaszoljunk, hogy
lefuttatnánk a programot)?

```
int f(int a){return ++a;}
int ff(int *a){return ++(*a);}
int g(int a){return f(++a)+1;}
int gl(int *a){return f(++(*a))+1;}
main()
{
    int a=1;
    printf("%d\n%d\n%d\n%d", f(a), ff(&a), g(++a),
        gl(&a));
    return 0;
}
```

40. Egy szöveges állományból kiolvassunk egy n elemű egész szám-
sorozatot egy dinamikusan lefoglalt tömbbe.
- Határozzuk meg a számok legkisebb számjegyeinek az összegét.
 - Rendezzük (*qsort*) a számokat úgy, hogy legkisebb számjegyeik
szerint csökkenő sorrendben legyenek! A rendezett számsorozatot
írjuk egy kimeneti állományba!
41. Adott n , illetve $n \times n$ tanuló neve és telefonszáma egy szöveges
állományban. A tanulók adatait egy $n \times n$ méretű kétdimenziós **struct**-
tömbben tároljuk el. Írj egy olyan függvényt, amely különböző szá-
molási műveleteket tud végezni a kétdimenziós tömbben eltárolt **struct**-
mátrix főátló alatti háromszögében, attól függően, milyen függvényt kap
paraméterként.

- a) Számoljuk meg azokat a személyeket a főátló alatt, akik neve négysszavas!
- b) Számoljuk meg azokat a személyeket a főátló alatt, akiknek a telefonszáma palindrom?

42. Egy szöveges állományból kiolvassunk egy n elemű egész számsorozatot egy dinamikusan lefoglalt tömbbe.

- a) Határozzuk meg a számok számjegyei szorzatának az összegét.
- b) Rendezzük (**qsort**) a számokat úgy, hogy számjegyeik szorzata szerint növekvő sorrendben legyenek! A rendezett számsorozatot írjuk egy kimeneti állományba!

43. Adott n , illetve n tanuló neve és telefonszáma egy szöveges állományban. A tanulók adatait egy n elemű egydimenziós **struct**-tömbben tároljuk el. Írj egy olyan függvényt, amely különböző ellenőrzési műveleteket tud végezni az egydimenziós tömbben eltárolt adatokon, attól függően, milyen függvényt kap paraméterként.

- a) Ellenőrizzük, hogy van-e olyan személy a főátló felett, akinek a keresztnéve és vezetéknéve azonos!
- b) Ellenőrizzük, hogy van-e olyan személy a főátló felett, akinek a telefonszáma számjegyei növekvő sorrendben követik egymást!

44. Írj egy függvényt, amely bármely elemtípusú és elemszámú tömb elemeit szétválogatja az első elem szerint, anélkül hogy rendezné a tömböt.

45. Adott egy szöveges állományban n apa keresztnéve, gyerekei száma és ezek keresztnéve.

- a) Írjuk ki egy másik állományba az apák adatait gyerekszám szerint csökkenő sorrendbe rendezve (**qsort**).
- b) Hány Barni (Barnika, Barna) nevű gyerek van?
- c) Készíts statisztikát, hogy az egyes apáknak összesen hány Barni fiuk van.

46. Írj függvényt, amely visszatéríti a paraméterként kapott n -nél nagyobb legkisebb prím értékét.

47. Írj függvényt, amely visszatéríti az n -edik Fibonacci számot.

48. Írj függvényt, amely eldönti, hogy szám tökéletes-e. (a szám egyenlő a nála kisebb osztói összegével).

49. Írj függvényt, amely implementálja az Euler függvényt. (meghatározza az n -nél kisebb n -el relatív prím számok számát).

50. Írj függvényt, amely visszatéríti a paraméterként kapott mátrix rangját.

51. Írj függvényt, amely kiszámítja két p számrendszerbeli szám összegét p számrendszerben.

10. FEJEZET

MAKRÓK

Amikor fordítunk egy programot, a szó szerinti fordítást (kompilálást) rendszerint megelőzi egy előfeldolgozásnak (prekompilálásnak) is nevezett művelet. Az előfeldolgozó (preprocesszor), amely ezt végzi, szólunk az általában a programok elején megjelenő úgynevezett direktívák. Ezek a # karakterrel kezdődnek. Ilyen direktívák, például, amelyeket már használtunk, az `#include` és a `#define`.

A következőkben a `#define` direktívával fogunk alaposabban megismerkedni.

Használatának szintaxisa:

```
#define <szimbólum> <helyettesítendő_szóveg>
```

Azért, hogy különváljanak a programban használt azonosítóktól, a szimbólumokat csupa nagybetűvel szokás írni.

Az előfeldolgozás többek között abból áll, hogy az előfeldolgozó átvizsgálja a program forráskódjának minden sorát, hogy tartalmaz-e a `#define` direktívával korábban definiált szimbólumot. Ha talál, kicseréli a helyettesítendő karaktersorozattal. Ez addig ismétlődik, amíg vagy nincs már az adott sorban szimbólum, vagy csak olyan van, amely már egyszer helyettesítve lett (a végtelen rekurzió elkerülése végett).

A alábbi példákból kiderül, hogy a szimbólum definíciót leggyakrabban arra használjuk, hogy *nevet adjunk konstansoknak vagy átnevezzünk típusokat*.

```
#define EOS '\0'
#define TRUE 1
#define FALSE 0
#define YES TRUE
#define bool int
#define MENCOL 20
#define MENULINE 5
#define BORDER 2
#define MENUROW (MENULINE + BORDER)
#define MENU SIZE (MENUROW * MENCOL)
```

Megjegyzések:

1. Az első három példában szereplő szimbolikus konstansokra szinte minden C programban szükség van, ha jól áttekinthető programot szeretnénk írni.
2. A YES-t a TRUE segítségével definiáltuk, ezért kettős helyettesítéssel válik majd 1-gyé: minden YES először lecserélődik TRUE-ra, majd minden TRUE lecserélődik 1-re.
3. Bár a C nyelvben nincs explicit logikai típus, mi létrehozhatunk egy új „alapszót”, a **bool**-t, amelyet ezután természetesen éppúgy használhatunk, mint az eredeti **int**-et (hiszen úgyis arra cserélődik le), de használatával az egyes változók szerepét jobban kidomboríthatjuk. Az alapszójelleg hangsúlyozása érdekében használtunk ennél a szimbólumnál kisbetűt.
4. A további definíciók a szimbólumok leggyakrabban használt területét mutatják be: a különböző „bűvkonstansokat” névvel ellátva egy helyre gyűjtjük össze, világosabbá téve használatukat és megkönnyítve módosításukat.
5. Az utolsó két példából az is kiderül, milyen fontos szem előtt tartani, hogy szöveghelyettesítésről van szó. Ha a MENUROW definíciójából a látványosan fölösleges zárójeleket elhagynánk, akkor MENU SIZE első lépésben nem((MENULINE + BORDER) * MENU COL)-ra, hanem (MENULINE + BORDER * MENU COL)-ra cserélődne le, ami végeredményként a kívánt 140 helyett 45-öt eredményezne mindenütt, ahol MENU SIZE-t használtunk.
6. Mivel a felesleges zárójelek bajt nem okozhatnak, szokjuk meg, hogy a **#define** definíciókban szereplő kifejezéseket állig felzárójelezzük.

A *makrók* lehetővé teszik, hogy a szöveghelyettesítés paraméterezhető legyen:

Példák:

```
#define abs(x) ((x) < 0? ( - (x)): (x))
#define min(a, b) ((a) < (b)? (a): (b))
#define negyzet(x) ((x) * (x))
```

A makróhívások két lépésben hajtódnak végre:

1. lecserélődik a definíciójabeli formális alakra,
2. a formális paraméterek lecserélődnek a hívásbeli aktuális paraméterekre.

Például az

```
alfa = abs(y + 2);
```

utasításban, a makróhívás a következőképpen fog végrehajtódni:

1. lépés után:

```
alfa = ((x) < 0? ( - (x)): (x));
```

2. lépés után:

```
alfa = ((y + 2) < 0? ( - (y + 2)): (y + 2));
```

A fenti példák azt is kiemelik, hogy nemcsak a helyettesítő kifejezést fontos zárójelekkel védeni, hanem a helyettesítendő paramétereket is.

Ha **y** értéke **-3**, akkor **alfa**-ba **1** kerül, mint az várható.

Ha azonban a definíció helyettesítő részében az **x**-et védő zárójeleket elhagytuk volna, akkor az **alfa** **5**-öt kapna értékül.

Még valamire nagyon oda kell figyelnünk. Ha nem vagyunk éberek, makrohívásainknak nem kívánt mellékhatásai lehetnek.

Íme egy példa: a

```
beta = negyzet(++a);
```

utasítás, a makrohívás lecserélése után, a következőképpen fog kinézni:

```
beta = ((++a) * (++a));
```

A nem kívánt mellékhatás az, hogy az **a** változó kétszer kerül növelésre. Az ilyen rendellenességek a legegyszerűbben úgy kerülhetők el, hogy makrohívás paramétereként csak változót használunk. Például növelhettük volna **a**-t a makrohívást megelőzően, hogy a makrót **a**-ra kelljen meghívjuk.

Ha **negyzet**-et függvényként írtuk volna meg, akkor az előbbi problémával nem találtuk volna szemben magunkat, de a makrós megvalósítások mindig gyorsabb kódot eredményeznek, mint a függvények használata. Ennek az a magyarázata, hogy a függvényhívások a futási időből vesznek el, míg a makrohívások már az előfordítás alatt lecserélődnek. Ezért van az, hogy a C rendszerben sok könyvtári „függvény” a valóságban makró (például a **getchar()**).

Egy másik előnye a makróknak a függvényekkel szemben a „típus-függetlenség”. Például, ha függvényként szeretnénk megoldani a négyzetre emelést, akkor kellene írunk külön **int-négyzetre emelőt**, **float-négyzetre emelőt** stb. Mivel a makrók esetében egyszerű szöveghelyettesítésről van szó, ezért „univerzálisak”.

Végkövetkeztetésként elmondható, hogy a makrók használata igen kíváncsú, de figyelmet igényel.

Ha egy szimbólum vagy makró által lefoglalt azonosítót fel szeretnénk szabadítani, azt az

```
#undef <szimbólum>
```

utasítással tehetjük meg.

Ezt követően az előfeldolgozó a szimbólum (makró) további előfordulási helyeit nem kíséri figyelemmel. Természetesen egy újabb **#define** utasítással újradefiniálhatjuk ezt a szimbólumot is.

Megjegyzések:

1. Egy makró definíciós részében két szintaktikai egység összeforrasztására használható a **##**.
Például a **#define VAR(I, J) (I ## J)** makródefiníció esetén a **VAR(x, 5)** szövegből **x5** lesz a kifejtés után.
2. Ha bármely makróargumentum elé **#**-t írunk, az karakterlánccá alakul. Így a makrókifejtéskor a **#arg** alakú paraméterhivatkozások az **"arg"** alakkal helyettesítődnek.
3. A makrók definíciós részében lévő beágyazott makrók kifejtése a külső makró kifejtésekor történik meg, nem pedig a definiálásakor.
4. A makrók két lépésben cserélődnek le, ahogy azt az alábbi példa is mutatja:

delta = min(x + 1, y);

első lépés után:

delta = ((a) < (b)? (a): (b));

második lépés után:

delta = ((x + 1) < (y)? (x + 1): (y));

10.1. feladat. Írjunk egy függvényt, amely egy paraméterként kapott számsorozatot rendez. A rendezéshez szükséges cserét valósítsuk meg makró segítségével.

Első változat

```
# define csere(a,b) a=a+b; b=a-b; a=a-b;

void rendez(int n, int a[])
{
    int i,j;
    for(i=0; i<n; i++)
        for(j=i+1; j<n; j++)
            if(a[i]>a[j]) {csere(a[i], a[j])}
}
```

Második változat

```
# define csere(T,a,b) {T v=a; a=b; b=v;}

void rendez(int n, int a[])
```

```
{
    int i,j;
    for(i=0; i<n; i++)
        for(j=i+1; j<n; j++)
            if(a[i]>a[j]) csere(int, a[i], a[j])
}
```

10.1. Kitűzött feladatok

1. Hogyan működik az alábbi program? (backslash karakterrel több sorba tördelhetők a `#define` sorok.)

```
#include <stdio.h>
#define def_fn(name, operation)\
int name(int a, int b)\
{\
    int c = a operation b;\
    printf("%d %s(%s) %d = %d\n", a, #name, #operation,\
        b, c);\
    return c;\
}\
def_fn(plus,+); def_fn(minus,-);
main()
{
    plus(1,1);
    minus(1,1);
    return 0;
}
```

2. Mit ír ki az alábbi program? Az `M_PI` konstans a pi értékét tartalmazza. A feladatot fejben oldjuk meg!

```
#include <stdio.h>
#define kerulet(r) 2 * M_PI * r
double terület(double r){return M_PI * r * r;}
main()
{
    printf("%lf\n%lf", kerulet(1+2), terület(1+2));
    return 0;
}
```

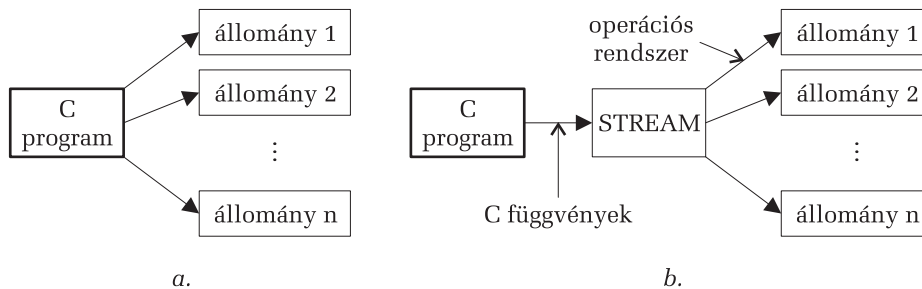
11. FEJEZET

ÁLLOMÁNYKEZELÉS (INPUT/OUTPUT MŰVELETEK)

Egy C program valamely feladat bemenő adatait a billentyűzetről vagy egy állományból olvassa be (Input). Az eredmények a képernyőre, a nyomtatóra vagy egy állományba kerülnek kiírásra (Output). A billentyűzet, valamint a képernyő és a nyomtató is felfogható állományként. A billentyűzet olyan „állomány”, amelyből csak olvasni lehet (input file), és a begépzelt karaktereket tartalmazza. Ezzel szemben a képernyő-, illetve nyomtató-„állományba” csak írni lehet (output file), és a kiírt karaktereket tartalmazza. A háttértárakon (floppy, merev lemez, CD, mágneses szalag stb.) tárolt állományok esetében mind az olvasási, mind az írási művelet lehetséges (input/output file). A háttértárak is két csoportba sorolhatók: véletlen (floppy, merev lemez, CD), illetve szekvenciális (mágneses szalag) elérésűek. A fizikai egységek tehát, ahonnan a bemenő adatok „jönnek”, illetve ahova az eredmények „mennek”, nagyon sokfélék lehetnek, mindeniknek meglévén a specifikus adattárolási módja. Ennek a kényelmetlen sokféleségnek az elrejtése érdekében vezették be a C nyelvben a *stream*-eket (fluxus).

A stream-et úgy kell elképzelni, mint egy irányított adatfluxust.

A C program mindig egy stream-en keresztül lép kapcsolatba egy adott állománnyal. Úgy is mondhatnánk, hogy a stream egy logikai egység, amely beékelődik a C program és valamely fizikai egységen létező állomány közé.



11.1. ábra.

Amíg az állományok magukon viselik az illető fizikai egységek jellegzetességeit, a stream-ek alapvetően egyformák. Mivel a program a valóságban a stream-be ír, illetve innen olvas, ezért minden állományt egyformának lát. Természetesen ez nem jelenti azt, hogy az állományoknak nem marad meg az input, output vagy input/output jellege.

Mi ennek a gyakorlati haszna? Ahelyett, hogy a különböző típusú állományok kezeléséhez külön-külön függvénysort használnánk (11.1. a) ábra), bármilyen állományt kezelhetünk ugyanazzal a függvénysorral (11.1. b) ábra). Például ugyanazokkal a függvényekkel írunk bármilyen állományba, függetlenül attól, hogy azok hol, azaz melyik periférián vannak eltárolva.

Bár az imént azt mondtuk, hogy a stream-ek egyformák, mégis egy bizonyos szempontból két csoportba sorolhatók. Vannak szöveges stream-ek és bináris stream-ek. Ebből adódóan egy állományt kétféleképpen lehet nézni. Szöveges stream-en keresztül úgy látjuk, mint egy karaktersort, a binárison keresztül pedig, mint egy byte-sort. Ez a kétféleség azonban nem változtat azon a tényen, hogy a C nyelv egyetlen függvény-készletet használ bármilyen típusú állomány kezeléséhez. Csupán arról van szó, hogy bizonyos függvények egy kissé másként működnek, ha szöveges stream-en keresztül kezeljük az állományt, mintha bináris által tennénk ezt meg.

Gyakran ez annyit jelent, hogy íráskor, illetve olvasáskor – amennyiben szöveges stream-en keresztül történik az állománykezelés – bizonyos karakter-konverziókra kerülhet sor. Például íráskor a „newline” (`'\n'`) karakter a carriage-return (`<CR>`) és line-feed (`<LF>`) karakterpárrá alakulhat át. Ilyen esetben olvasáskor a fordított irányú konverzióra kerül sor (`<CR> + <LF> -> '\n'`). Így magyarázható, hogy szöveges stream-ek használata esetén az állomány nagyobb méretű lehet (több karaktert tartalmazhat), mint a programunk által beleírt karakterek száma.

Bináris stream-ek esetén semmilyen konverzióra nem kerül sor, és ezért az állomány mérete mindig megegyezik a beleírt byte-ok számával.

Ezt a kétféleséget a programozó munkájának megkönnyítése céljából vezették be. Ha olyan állományról van szó, amely alapvetően egy szöveget tartalmaz, különböző hosszú sorokba rendezett karaktersort, amelyet talán meg szeretnénk jeleníteni a képernyőn vagy ki szeretnénk nyomtatni, akkor nyilván szöveges stream-en keresztül fogjuk kezelni. Miért előnyös ez? A képernyőre való íráskor, illetve a nyomtatáskor, az új sorba való ugrás a `<CR>` és `<LF>` karakterek hatására történik meg. Ha szöveges stream-et használunk, nem kell nekünk gondoskodnunk ezen karakterek

beírásáról minden sor végére, hiszen a ‘\n’ karakter automatikusan ezekké konvertálódik. Amennyiben egy állomány nem szöveget tartalmaz, és nem áll szándékunkban képernyőn vagy nyomtatásban megjeleníteni, célszerű bináris állományként kezelni.

Ugyancsak az állományok e két csoportba sorolhatóságának köszönhető az, hogy egyes függvények a szöveges állományok kezelésére alkalmasabbak, mások pedig a binárisokéra. Például **fprintf**-fel rendszerint szöveges stream-en keresztül írunk egy olyan állományba, amely szöveget tartalmaz. Ellenben az **fwrite**-ot szokás használni, amennyiben bináris stream-be történik az írás.

Hogyan történik – milyen függvények által – egy C programból az állománykezelés?

11.1. Állománykezelő függvények

Egy állományt C programban egy `FILE *` típusú úgynevezett állománypointer képvisel. A `FILE` típus egy *struct* típus, amely az **stdio.h** header-állományban van definiálva.

Az állománypointer definiálásának szintaxisa:

```
FILE * <állomány_pointer>;
```

Az alábbiakban az állománypointert **fp**-vel fogjuk jelölni.

Ahhoz, hogy egy állományon műveleteket tudjunk végezni, először meg kell nyitnunk. Ekkor történik egy stream kijelölése az állomány részére. Az, hogy miként nyitjuk meg az állományt, meghatározza, hogy *milyen* I/O műveletek hajthatók végre rajta (írás, olvasás, írás a végére, írás/olvasás). A kijelölt stream típusa, amint láthattuk, azt befolyásolja, hogy *miként* hajtódnak végre az említett műveletek. Az alábbi függvények a legfontosabb file-műveletek elvégzését biztosítják.

Állomány megnyitása:

```
fp = fopen(<állomány_név>, <mód>);
```

A mód lehet:

| | |
|-------------|----------------------------------|
| rt,r | szöveges mód, olvasás |
| wt,w | szöveges mód, létrehozás és írás |
| at,a | szöveges mód, írás a végére |
| rb | bináris mód, olvasás |
| wb | bináris mód, létrehozás és írás |
| ab | bináris mód, írás a végére |

| | |
|---------------|---|
| r+t,r+ | szöveges mód, olvasás/írás |
| w+t,w+ | szöveges mód, létrehozás és írás/olvasás |
| a+t,a+ | szöveges mód, írás a végére vagy létrehozás és írás/olvasás |
| r+b | bináris mód, olvasás/írás |
| w+b | bináris mód, létrehozás és írás/olvasás |
| a+b | bináris mód, írás a végére vagy létrehozás és írás/olvasás |

Az **r** és **a** módok feltételezik, hogy az állomány létezen a megnyitáskor. Ha **w** módban nyitunk meg egy állományt, és már létezett, tartalma elvész. Ha sikerült a megnyitás, az **fopen** az állománypointert téríti vissza. Sikertelenség esetén **NULL** pointert.

Állomány bezárása:

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

Írás/Olvasás állományba/ból:

– Karakterírás állományba:

```
int fputc(int ch, FILE * fp);
```

Beírja (és vissza is téríti) a **ch** karaktert az állományba.

A régebbi verziókkal való kompatibilitás végett a **ch** karakter **int** típusú.

– Karakterolvasás állományból:

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

Kiolvas egy karaktert az állományból, és eredményként visszatéríti.

A régebbi verziókkal való kompatibilitás miatt a karaktert **int** típusúként téríti vissza.

– Karakterlánc-írás állományba:

```
int fputs(const char * s, FILE * fp);
```

Beírja az **s** címen kezdődő karakterláncot az állományba. Hiba esetén **EOF**-et térít vissza, aminél értéke rendszerint **-1**.

– Karakterlánc-olvasás állományból:

```
char * fgets(char * s, int h, FILE * fp);
```

Kiolvas egy karakterláncot az **s** címre, amíg sorvégjelt (a **gets**-szel ellentétben ez is bekerül a karakterláncba) talál, vagy kiolvasott már **h-1** karaktert, majd lezárja a karakterláncvégjellel. Ha sikerült, a karakterlánc címét téríti vissza, ha nem, **NULL** pointert.

– Írás formátálva állományba:

```
int fprintf(fp, <formázósor>, <kifejezés_lista>);
```

Hasonlóan működik, mint a `printf`, csak hogy az `fp` állományba (általában szöveges) történik az írás. A sikeresen kiírt kifejezések számát téríti vissza.

- Olvasás formatálva állományból:

```
int fscanf(fp, <formázósor>, <változócím_lista>);
```

Hasonlóan működik, mint a `scanf`, csak hogy az `fp` állományból (általában szöveges) történik az olvasás. A sikeresen beolvasott változók számát téríti vissza. Az állományvégjel olvasásakor `-1(EOF)`-et térít vissza.

Az előbbi függvényeket általában szöveges állományok esetében használjuk, az alábbi kettőt viszont kifejezetten binárisoknál. A `size_t` típus az `stdio.h` header-állományban van definiálva, és rendszerint előjel nélküli egész.

- Adott számú adatblokk kiírása állományba:

```
size_t fwrite(const void*<p>, size_t<m>,
              size_t<nr>, FILE * fp);
```

A `p` címről `nr` darab `m` byte méretű blokkot kiír az állományba. Visszatéríti a sikeresen kiírt blokkok számát.

- Adott számú adatblokk beolvasása állományból:

```
size_t fread(void*<p>, size_t <m>, size_t <nr>,
              FILE * fp);
```

Beolvas az állományból a `p` címre `nr` darab `m` byte méretű blokkot. Visszatéríti a sikeresen beolvasott blokkok számát.

Hibalekérdés:

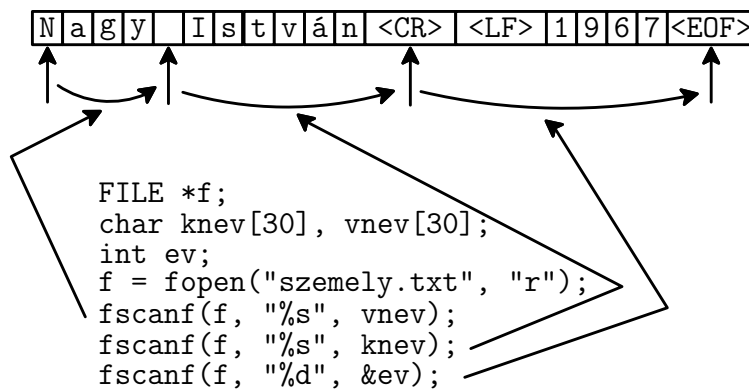
```
int ferror(FILE * fp);
```

„Igaz”-at térít vissza, ha egy állományművelet kapcsán hiba adódott. Ellenkező esetben értéke „hamis”.

Minden olyan állományhoz, amelyhez lehetséges a véletlen hozzáférés (floppy, merevlemez stb.), kapcsolódik egy pozíciómutató, amely az állományban az aktuális pozíciót (karaktert vagy byte-ot) jelöli. Az írás- és olvasásműveletek mindig ettől a pozíciótól kezdődően történnek. A pozíciómutató léptetése automatikusan történik a beolvasott vagy kiírt adatokat követő pozícióra. Ha létrehozásra és írásra nyitunk meg egy állományt (`w`-mód), akkor kezdetben csak az állományvégjelt tartalmazza, és a pozíciómutató ide mutat. Minden kiírást követően a pozíciómutató automatikusan továbblép az állományvégjelre. Hasonlóan működik az állomány végére történő írás is (`a`-mód), csak hogy ez esetben az állomány megnyitáskor

már létezik, a pozíciómutató viszont ugyancsak az állományvégjelre mutat. Olvasásra való megnyitáskor (r-mód) a pozíciómutató kezdetben az állomány elejére mutat.

Például egy szöveges állományt – amely alapvetően egy karaktersor – az alábbi módon képzelhetünk el (feltételeztük, hogy az állomány első sorában egy személy vezeték- és keresztnéve található szóközzel elválasztva, a második sorában pedig az illető személy születési éve). A nyilak azt szemléltetik, miként mozdul el az állománymutató az egymást követő olvasási műveletek által (a `fscanf` függvény működését részletesen tárgyaljuk a 11.2. alfejezetben):



Pozíciómutató-lekérdezés és -mozgatás

Állományvégjel érzékelése:

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

Nullától különböző értéket (logikai igaz) térít vissza, ha az állományvégjelt próbáltuk kiolvasni.

A pozíciómutatónak az állomány elejére állítása:

```
void rewind(FILE * fp);
```

A pozíciómutató beállítása:

```
int fseek(FILE * fp, long <nr>, int
<viszonyítási_pont>);
```

Elmozdítja `nr` byte-tal a pozíciómutatót a `viszonyítási_pont`-hoz képest.

A `viszonyítási_pont` paraméter lehetséges értékei:

`SEEK_SET` (állomány eleje)

`SEEK_CUR` (pozíciómutató aktuális pozíciója)

SEEK_END (állomány vége)

Nullát térít vissza sikeres művelet esetén.

A pozíciómutató lekérdezése:

```
long ftell(FILE * fp);
```

Visszatéríti a pozíciómutató aktuális pozícióját. Hiba esetén **-1**-et térít vissza.

Háttér-információ az állománykezeléssel kapcsolatban:

A belső memória és a háttértárak közötti adatcsere köztudottan lassú. Az adatsín hatékonyabb kihasználása érdekében kidolgoztak egy puffert az I/O műveletek lebonyolítására. Például állományba való íráskor rendkívül lefoglalná az adatsínt, ha minden byte-tal egyenként szaladnánk valamely perifériára. Ennek elkerülése érdekében az állomány megnyitásakor általában létrejön a memóriában az állomány részére egy úgynevezett *memóriapuffer*. Ez úgy működik, mint egy raktár. Az állományba író függvények az adatokat egyelőre csak ezen pufferbe írják be, és ha feltelt a puffer, akkor egyszerre, egyetlen háttértár-hozzáféréssel, a puffer teljes tartalma kiíródik az állományba. Valami hasonló történik olvasáskor is. A háttértárról a pufferbe egyszerre egy egész puffernyi adat kerül beolvasásra, és innen olvasnak majd az állományolvasó függvények.

Az imént felvázolt mechanizmusnak van egy mellékhatása: amit kiírunk az állományba, az csak később – a puffer ürítése után – jut el effektíve a háttértárra (automatikus pufferürítésre kerül sor, valahányszor feltelik a puffer, a program befejeztével, illetve az **fclose** függvényhívás hatására). Létezik azonban egy függvény, amelynek segítségével programból ellenőrizhetjük, hogy mikor történjen a pufferürítés.

Memóriapuffer ürítése:

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

Ha **fp** bemeneti állományt (csak olvasni lehet belőle) képvisel (például a billentyűzetet), akkor egyszerűen törlődik a puffer tartalma. Amennyiben **fp**-ét írásra nyitottuk meg, a puffer tartalma beíródik az állományba. A függvény nullát térít vissza sikeres művelet esetén, és **EOF**-et, azaz **-1**-et, ha valamilyen hiba történt.

Nem puffert I/O műveletek esetén (például képernyőre való írás vagy billentyűzetről való nem puffert olvasás (**getch**, **getche**)) az **fflush** hatástalan.

Példa: Tegyük fel, hogy egy egysoros szöveges állomány az angol ábécé nagybetűit tartalmazza ábécésorrendben. Minden magánhangzót írjunk át kisbetűre.

```
FILE * f; char c;
f = fopen("abc.txt", "r+t");
while (!feof(f))
{
    c = fgetc(f);
    if (!feof(f) && (c=='A' || c=='E' || c=='I' || c=='O'
        || c=='U'))
    {
        fseek(f, -1, SET_CUR);
        fputc(c + ('a' - 'A'), f);
        fflush(f); /* minden karakterátírás után beírjuk
            (visszaírjuk) a puffer tartalmát az állományba */
    }
}
fclose(f);
```

Ha nem szerepelne az `fflush` a kódban, akkor megállítva a programot közvetlen az `fclose` előtt, még semmilyen változás nem lenne észlelhető az `abc.txt` állományban. A fenti példa azt is megmutatja, hogy az `fseek` használatát is a puffertelt állománykezelés teszi lehetővé.

Megjegyzés. Bináris állományok esetén a `-1` érték (az EOF értéke) lehet az állomány valamely elemének értéke is. De egyes függvények az EOF visszatérítésével jelzik a hibát vagy azt, hogy állományvégnél vagyunk. Azért, hogy biztosak legyünk a visszatérített `-1`-es jelentésében, használjuk a hibaellenőrzésre az `ferror`, az állományvégnél érzékelésre pedig az `feof` függvényeket.

Bezárt állományok letörölhetők, illetve átnevezhetők egy C programból.

Állomány törlése:

```
int remove(<állomány_név>);
```

Nullát térít vissza sikeres művelet esetén.

Állomány átnevezése:

```
int rename(<régi_állomány_név>, <új_állomány_név>);
```

Nullát térít vissza sikeres művelet esetén.

Még egy kérdés tisztázásra vár: ha stream-eken keresztül minden állomány (a billentyűzet és a képernyő is) egyformának látszik, és ebből adódóan ugyanazon függvénykészletet használjuk bármilyen állomány kezelésére, akkor mi a helyzet a `printf` és `scanf`, illetve egyéb olyan

függvényekkel, amelyek kifejezetten a billentyűzetről való olvasásra és a képernyőre való írásra szolgálnak?

Létezik az ANSI C-ben három elődefiniált állománypointer (`stream`), `stdin`, `stdout` és `stderr`. `stdin` a billentyűzethez (standard bemenet) rendelt stream-re mutat, `stdout` és `stderr` pedig a képernyőhöz (standard kimenet) rendeltre. A billentyűzet-állomány, illetve a képernyőállomány megnyitása, illetve bezárása (minden ehhez kapcsolódó művelettel) automatikusan történik a programfutás kezdetekor, illetve befejeztével. Így bármely állománykezelő függvényben, ha állománypointerként `stdin`-t, illetve `stdout`-ot használunk, akkor az a billentyűzetre, illetve a képernyőre fog vonatkozni. Következésképpen az alábbi vázlatos függvényhívások teljesen egyenértékűek:

```
fscanf(stdin, ...);    scanf(...);
fprintf(stdout, ...);  printf(...);
fputs(..., stdout);   puts(...);
fgets(..., stdin);     gets(...);
fputc(..., stdout);   putchar(...);
getc(stdin);           getchar();
```

Miért vezették akkor be a második oszlopban lévő függvényeket? A programozó kényelméért.

Egy másik érdekes tulajdonság, amely kihasználható a standard stream-ek esetében, az, hogy (`stdin`, `stdout`, `stderr`) átirányíthatók az `freopen` függvény segítségével.

Stream- (fluxus-) átirányítás:

```
FILE * freopen(<állomány_név>, <mód>, FILE * <stream>);
```

Egy létező stream-hez (állománypointerhez) új állományt rendel.

Példa:

```
char s[80];
freopen("ki.txt", "w", stdout);
printf("Írd be a karakterláncot:"); /* ki.txt-be
    ír */
gets(s); /* billentyűzetről olvas */
printf("%s", s); /* ki.txt-be ír */
```

A fenti példában átirányítottuk az `stdout` stream-et a képernyőről a `ki.txt` állományra, aminek következtében a `printf` az illető állományba ír.

11.2. A **scanf/fscanf**, illetve **printf/fprintf** függvények közelebbről

Csak a **scanf** és a **printf** függvényeket fogjuk bemutatni, hiszen az **fscanf** és az **fprintf** hasonlóan működnek. Az alábbi információk kiegészítik azt, amit a 2. fejezet tartalmaz a szóban forgó függvényekkel kapcsolatban.

11.2.1. A **scanf** függvény

Szintaxisa:

```
int scanf(<formázósor>, <változócím_lista>);
```

Lebonyolítja a megadott változók értékeinek a beolvasását a billentyűzetről a megfelelő memóriacímekre, és eredményként visszatéríti a beolvasott és eltárolt értékek számát. A formázósor % karakterrel bevezetett formázókarakterei jelzik, hogy milyen konverziókra van szükség, hiszen a billentyűzetről karakterlánc formájában érkeznek az adatok, de a memóriában az illető változó típusának megfelelő belső ábrázolásban kell eltárolni őket. A **scanf** puffertelt beolvasást végez, ami azt jelenti, hogy a begépelte adatok a billentyűzet pufferebe kerülnek először, majd az ENTER billentyű leütésekor a **scanf** innen olvassa be őket a memóriába. Tehát a **scanf** alapvetően a pufferből olvas.

Hogyan működik a scanf függvény?

Az eddigi példáinkban a **scanf** függvény formázósora csak % karakterrel bevezetett formázókaraktereket tartalmazott, minden változó részére egy-egy formázókaraktert. A valóságban viszont egyéb (bármilyen) karakterek is szerepelhetnek a formázósorban. Éppen ez indokolja a % karakter használatának szükségességét a formátumleíró karakterek bevezetésére. Tehát a formázósor általános alakja a következő:

```
[egyéb karakterek][formátumleíró][egyéb karakterek]
[formátumleíró] ...
```

Sőt általános esetben a formátumleíró is több adatot tartalmazhat, mint egyszerűen a % karaktert és a formázókaraktert. Általános alakja:

```
%[*][mezőszélesség][h/l/L]formázókarakter
```

Ha szerepel a formátumleíróban a * karakter, akkor ugyan kiolvasódik a billentyűzet puffereből a megfelelő adat, de nem kerül eltárolásra a memóriába. Úgy is mondhatnánk, hogy átugorjuk az illető értéket. Természetesen az átugrott adatok nem számolódnak bele a **scanf** függvény által visszatérített értékbe.

11.2. A SCANF/FSCANF, ILLETVE PRINTF/FPRINTF FÜGGVÉNYEK 209

A mezőszélességgel azt jelezhetjük a `scanf`-nek, hogy legfennebb ennyi karaktert olvasson be az illető változó értékeként.

A leggyakrabban használt formázókaraktereket (c, d, i, u, o, x, f, s, p) bemutattuk a 2. fejezetben, a teljes listát pedig megtaláljuk a Help-ben. Két példán keresztül megemlíünk két különleges esetet:

11.2.1.1. A `%n` formázókarakter

```
int a, b, c;
scanf("%d%d%n", &a, &b, &c);
printf("%d %d %n", a, b, c);
```

Az `n` formázókarakter hatására nem a pufferből történik értékolvasás a megfelelő változóba, hanem a `scanf` függvény által addig beolvasott karakterek száma tárolódik el benne. A fenti példában az 12 789 bemenetre a kimenet 12 789 6 lesz. Mivel a `scanf`, a `%n` megjelenése pillanatáig, 6 karaktert olvasott be (1, 2, SPACE, 7, 8, 9), ezért a `c` változó értéke 6 lett.

11.2.1.2. A `scanset` használata

```
int i; char s1[80], s2[80];
scanf("%d%[abcdefg]s", &i, s1, s2);
printf("%d %s %s", i, s1, s2);
```

A szögletes zárójelek közé írt karakterhalmazt *scanset*-nek nevezzük. Ha karaktertömbbe való olvasáskor scanset-et használunk az `s` formázókarakter helyett, akkor a karakterbeolvasás addig tart, míg ezek elemei az illető halmaznak. Például, ha a fenti esetben a billentyűzetpufferbe az 123abcdtye karaktersort gépeltük be, akkor a képernyőn 123 abcdtye fog megjelenni. Mivel `t` nem eleme a scanset-nek, az `s1` tömbbe való olvasás befejeződik, amikor a `t` karakterhez értünk. Így hát a fennmaradt karakterek `s2`-be lesznek beolvasva. Ha a scanset a `^` karakterrel kezdődik, akkor ez azt jelzi a `scanf` függvénynek, hogy csak olyan karaktereket fogadjon el, amelyek *nem* elemei a scanset-nek. A scanset ily módon való használata lehetővé teszi többszavas mondatok beolvasását `scanf`-fel. Ezt a lehetőséget mutatják be az alábbi példák:

```
char mondat[80];
scanf("%s", mondat);
```

Az `alma` a `fa` alatt bemenetre a `mondat` tömbbe csupán az `alma` szó olvasódik be, mivel a `scanf` csak az első fehér karakterig fog olvasni.

```
char mondat[80];
```

```
scanf("%[\\n]", mondat);
```

Ugyanarra a bemenetre ez esetben a **mondat** tömb tartalma **alma a fa alatt** lesz, mivel a **scanf** minden karaktert olvasni fog, kivéve a new line karaktert (**\\n**).

A **scanf** függvény működésének pontos megértése érdekében tekintsük az alábbi két karakterláncot:

1. A billentyűzet pufferébe begépett karakterláncot.
2. A **scanf** függvény formázósorát.

Mindkét karakterlánchoz képzeletben rendeljünk egy nyilat (pointert), amely az aktuális karakterre mutat. Kezdetben mindkét nyíl a megfelelő karakterlánc elejére mutat. Az **ENTER** billentyű leütésekor a **scanf** az alábbi algoritmus szerint bonyolítja le a beolvasást. Megfigyelhető, hogy alapvetően három „üzemmódban” dolgozik, attól függően, hogy a formázó-sorban az aktuális karakter

1. nem fehér karakter és nem % karakter,
2. fehér karakter,
3. % karakter.

Ha a formázósor aktuális karaktere nem fehér karakter és nem % karakter,

akkor ellenőrzi, hogy a puffer aktuális karaktere ugyanaz a karakter-e,

ha igen, **akkor** mindkét nyíl átlépi az illető karaktert,

ha nem, **akkor** megszakad a beolvasás, és a **scanf** visszatéríti az addig sikeresen beolvasott értékeket.

Ha a formázósor aktuális karaktere fehér karakter,

akkor mindkét nyíl átlépi az összes útjában lévő fehér karaktert, és az ezeket követő első nem fehér karakteren állapodnak meg. (Ha a pufferben nem fehér karakter következett, akkor a puffer nyílja nem mozdul.)

Ha a formázósor aktuális karaktere a % karakter,

akkor ellenőrzi, hogy az ezt követő karakterek megfelelnek-e a formátumleíró szintaxisának,

ha nem felenek meg, **akkor** a % karaktert úgy kezeli, mint akármelyik nem fehér karaktert,

ha megfelelnek, **akkor** annak megfelelően jár el, hogy milyen típusú adatot kell beolvasni;

ha karaktert kell beolvasni (**c** a formázókarakter), **akkor** beolvassa a puffer aktuális karakterét

11.2. A SCANF/FSCANF, ILLETVE PRINTF/FPRINTF FÜGGVÉNYEK 211

(függetlenül attól, hogy fehér vagy sem), és ennek ASCII kódját eltárolja a megfelelő változóba. A puffer nyílja a következő karakterre, a formázósor nyílja pedig a formátumleíró követő karakterre fog mutatni.

Ha egyéb típusú értéket kell beolvasni (egész, valós, karakterlánc), **akkor** a puffer nyílja átugorja az útjában álló esetleges fehér karaktereket, majd beolvasásra kerülnek az ezeket követő karakterek egészen az első – a típusra nézve – „illegális” karakterig. Ilyen „illegális” karakterek például a fehér karakterek (a **scanf**-fel beolvasott karakterláncok sem tartalmazhatnak fehér karaktereket), vagy numerikus típus esetén a nem számjegy karakterek, sőt nyolcas számrendszer esetén a 8-as és 9-es számjegyek is stb. Ha a formátumleíróban mezőszélességet is megadtunk, akkor legfennebb ennyi karakter kerül beolvasásra. Más szóval, ha az első „illegális” karakter később következne, mint a mezőszélességben megadott karakterszám, akkor csak mezőszélességnyi karakter kerül beolvasásra az illető változó értékeként. A beolvasott karaktersort a **scanf** átalakítja a soron következő változó típusának megfelelően, majd eltárolja a megjelölt memóriacímre (ez elmarad, ha a formátumleíróban szerepel a * karakter; lásd fennebb). A puffer nyílja az utolsó kiolvasott karaktert követő karakterre fog mutatni, a formázósoré pedig a formátumleíró követő karakterre. Ha egyetlen karakter sem lett beolvasva, átalakítva és eltárolva, akkor a beolvasás megint csak megszakad, és a **scanf** az addig sikeresen beolvasott és eltárolt értékek számát téríti vissza.

Megjegyzés. Megtörténhet, hogy a billentyűzetről puffertelt beolvasást biztosító függvények (**scanf**, **getchar**, **gets**) nem találják üresen a billentyűzetpuffert, mert egy előző beolvasás „szemetet” hagyott maga után, és ez megzavarhatja a működésüket. Ennek elkerülése érdekében használjuk pufferürítésre a **fflush(stdin)**; függvényhívást minden beolvasás előtt.

Az alábbi példák tanulmányozása segíthet még tisztábban látni a **scanf** függvény működését. A pufferben található **SPACE** karaktereket **_** karakterrel jelöltük. A formázósorban normális szóközt használtunk.

| | Puffer tartalma | Változók értéke |
|---|--|---|
| int a,b; scanf("%ld", &a); scanf("%ld %*ld %ld", &a, &b); | 123 123 | a=1 a=1, b=3 |
| char a,b; scanf("%c %c", &a, &b); scanf("%c%c", &a, &b); scanf("1%c", &a); | p_ _ _ _ q p_ _ _ _ q 12 21 | a='p', b='q' a='p', b=' ' a='2' a nem kap értéket |
| float a;double b; long double c; scanf("%f %lf %Lf", &a, &b, &c); scanf("%f%lf", &a, &b); scanf("%3f", &a); scanf("%f 1 %lf", &a, &b); | 2.5_2.7_4.3 2_3 1.25 1.2s1s-1.7 | a=2.5,b=2.7, c=4.3 a=2.0, b=3.0 a=1.2 a=1.2, b=-1.7 |
| char a,b; int c; float d; scanf(" %c%3d%c 12 %f", &a, &c, &b, &d); | _ _ a _ _ 123412-41.7 | a='a', c=123 b='4', d=-41.7 |

11.2.2. A printf függvény

Szintaxisa:

```
int printf(<formázósor>, <kifejezés_lista>);
```

A formázósor általános alakja:

```
[egyéb karakterek][formátumleíró][egyéb  
    karakterek][formátumleíró]...
```

A formátumleíró általános alakja:

```
%[igazítás,előjel][mezőszélesség] [.pontosság]  
    [h/l/L/#]formázókarakter
```

Csak azokat a szempontokat fogjuk bemutatni, amelyeket nem érinttünk a 2. fejezetben.

11.2. A SCANF/FSCANF, ILLETVE PRINTF/FPRINTF FÜGGVÉNYEK 213

A kiírás mindig a képernyő aktuális pozíciójától kezdődően történik. A **mezőszélesség** egy minimális számú karakterpozíciót jelent, amelyen a kiírás megvalósul. Ha a kiírandó kifejezés értéke kevesebb karaktert tesz ki, mint a mezőszélesség, akkor ezen belül a kiíratás történhet jobbra vagy balra igazítottan (leginkább akkor élünk ezzel a lehetőséggel, ha táblázatos formában szeretnénk adatokat megjeleníteni). Implicit **SPACE**-ekkel töltődnek fel a fennmaradt pozíciók, de ha a mezőszélesség értékét megelőzi egy nulla, akkor számok jobbra igazítása esetén a feltöltő karakter a nulla lesz. Az is szabályozható, hogy pozitív numerikus értékek esetén megjelenjen-e a + előjel. (A példákban _ jellel jelöltük a képernyőn megjelenő **SPACE** karaktereket.)

Az **igazítás előjel** mezőhatása:

- balra igazítás történik, és jobbról **SPACE**-ekkel pótolódik ki mezőszélességig;
- + pozitív numerikus érték esetén is megjelenik az előjel;
- SPACE** pozitív numerikus érték esetén az előjel helyett **SPACE** jelenik meg.

Példa:

```
printf("%d,%-5d,%05d,%+d,% d", -13, 13, 13, 13);
```

A kiíratás eredménye:

```
-13,13____,00013,+13,_13
```

A **pontoságmező** hatása változik a kiírandó adat típusától függően:

- Valós típusok esetén azt határozza meg, hogy hány tizedes pontosággal történjen a megjelenítés (implicit 6 tizedes pontosággal).

Példa: `printf("%lf,%5.2lf", 3.1415, 3.1415);`

Az alábbi karaktersor jelenik meg a képernyőn:

```
3.141500,_3.14
```

- Karakterlánc kiíratásakor a maximális mezőszélességet jelenti, amelyen a megjelenítés megtörténhet. Hosszabb karakterlánc esetén a plusz karakterek levágódnak. Ha a karakterlánc rövidebb, figyelmen kívül marad a pontoság értéke.

Példa:

```
char *szo="abc";
```

```
printf("%s,%5s,%.2s,%5.2s",szo, szo, szo, szo);
```

A kimenet az alábbi lesz:

```
abc,___abc,ab,____ab
```

- Ha valamely egész típusra alkalmazzuk, akkor azt a minimális számjegyet jelenti, amelyen a szám megjelenítésre kerül. Kevesebb számjegyű egészek esetén a szám elejéhez nullák ragadnak. Ha több

számjegyű a kiírandó érték, mint a megadott pontosság, akkor figyelmen kívül marad.

Példa:

```
int a=12;
printf("%d,%05d,%5d,%.4d,%5.4d",a, a, a, a, a);
```

A képernyőn a következők jelennek meg:

```
12,00012, _12,0012, _0012
```

A **mezőszélesség** és a **pontosság** helyett szerepelhet a ***** karakter. Ilyenkor a kifejezéslista soron következő értékeit fogja a **printf** függvény **mezőszélesség** és **pontosság** értékeként értelmezni.

Példa: `printf("%*.*f",10, 4, 1234.55);`

A kimenet a következő lesz: `_1234.5500`

Kiegészítések a *formázókarakterek*et illetően:

- Ha az **x** vagy **X**, illetve **o** formázókarakterek előtt a **#** módosítót használjuk, akkor a hexaértékek **0x**, az oktálértékek pedig **0** prefixszel fognak megjelenni.
- **double** értékek tudományos alakban való kiíratásához használjuk az **e/E** formázókaraktereket.
- Valós értékek esetén a **g/G** formázókarakterek hatására a **printf** választ az **f**, illetve **e/E** használata között annak alapján, hogy melyik biztosít rövidebb alakot.

Példa: `double f;`

```
for(f=0.1; f<1.0e+10; f*=10) printf("%g ", f);
```

Az alábbiak jelennek meg a képernyőn:

```
1 10 100 1000 10000 100000 1e+06 1e+07 1e+08 1e+09
```

- A **printf** formázósorában is szerepelhet a **%n** formázókarakter. Használata azt feltételezi, hogy a kiírandó kifejezések között – a megfelelő helyen – jelenjen meg egy egész változó címe, ahova a **printf** eltárolhatja azon karakterek számát, amelyeket a **%n**-t megelőzően írt ki.

Példa:

```
int a=12345, b;
printf("Az %d%n szám ", a, &b);
printf("%d számjegyű ", b-3);
```

A képernyőn a következő jelenik meg:

Az 12345 szám 5 számjegyű.

11.3. Megoldott feladatok

11.1. feladat. Írjunk programot, amely karaktereket olvas be a billentyűzetről állományvégjelig (<ctrl>Z), és beírja őket egy **szoveg.txt** nevű szöveges állományba.

```
FILE * f; char c;
if (!(f = fopen ("szoveg.txt", "w")))
    {puts("Hiba"); return 0;}
while (!feof(stdin))
{
    c = getchar();
    if(!feof(stdin)) fputc(c, f);
}
fclose(f);
```

11.2. feladat. Írjunk programot, amely kiírja a **szoveg.txt** nevű szöveges állomány tartalmát a képernyőre.

```
FILE * f; char c;
if (!(f = fopen ("szoveg.txt", "r")))
    {puts("Hiba"); return 0;}
while (!feof(f))
{
    c = fgetc(f);
    if (!feof(f)) putchar(c);
}
fclose(f);
```

11.3. feladat. Írjunk programot, amely a **szoveg.txt** nevű szöveges állomány sorait sorszámozza. *Ötlet:* Létrehozunk egy másik állományt, amely sorszámozva tartalmazza a sorokat, majd az eredeti állományt letöröljük, és az újat átnevezzük az eredeti nevére. Így úgy fog tűnni, *mintha* az eredeti állományt módosítottuk volna.

```
FILE * f, * t; char s[100]; int i = 0;
if (!(f = fopen ("szoveg.txt", "r")))
    {puts("Hiba1"); return 0;}
if (!(t = fopen ("temp.txt", "w")))
    {puts("Hiba2"); return 0;}
while (!feof(f))
{
    i ++;
    fgets(s, sizeof(s), f);
```

```

    if (!feof(f)) fprintf(t, "%d) %s", i, s);
}
fclose(f); fclose(t);
remove("szoveg.txt");
rename("temp.txt", "szoveg.txt");

```

11.4. feladat. Írjunk programot, amely a **szoveg1.txt** nevű szöveges állományt, a **szoveg2.txt** nevű szöveges állomány után fűzi.

```

FILE * f, * g; char c;
if (!(f = fopen ("szoveg1.txt", "r")))
    {puts("Hiba1"); return 0;}
if !(g = fopen ("szoveg2.txt", "a"))
    {puts("Hiba2"); return 0;}
while (!feof(f))
{
    c = fgetc(f);
    if (!feof(f)) fputc(c, g);
}
fclose(f);
fclose(g);

```

11.5. feladat. Írjunk programot, amely a **be.txt** nevű szöveges állományból beolvassa egy számsorozat elemeinek számát, illetve az elemeket (int típusú értékeket), és létrehoz belőlük egy **adat.dat** nevű bináris állományt. Ezután kódolja a bináris állományt úgy, hogy minden bitjét tagadja. Végül kiírja a kódokat a **ki.txt** szöveges állományba.

```

void be_txt(char *, int *, int ** );
void ki_bin(char *, int, int *);
void titkosit(char *);
void bin_to_txt(char *,char *);
main()
{
    int n, * a;
    be_txt("be.txt", &n, &a);
    ki_bin("adat.bin", n, a);
    titkosit("adat.bin");
    bin_to_txt("adat.bin", "ki.txt");
    free(a); return 0;
}
void be_txt(char * s, int * pn, int ** pa )
{
    FILE * f; int i;

```

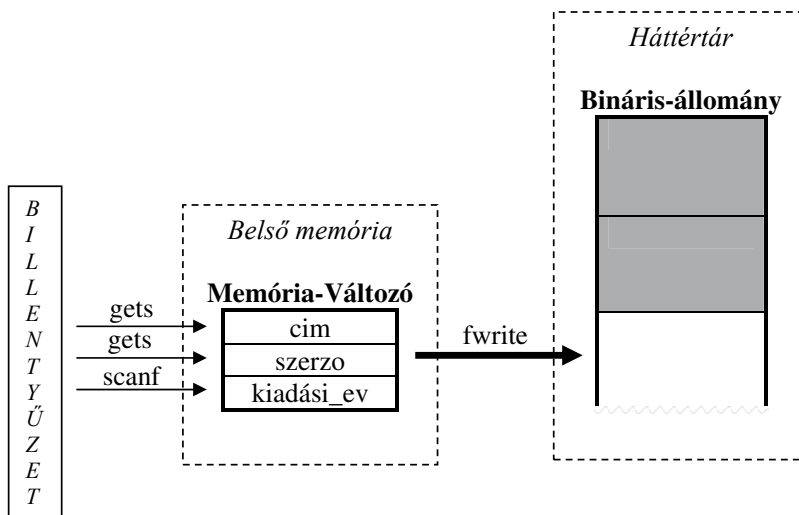
```

    f = fopen(s, "r");
    fscanf(f, "%d", &pn);
    *pa = (int *) malloc((*pn) * sizeof(int));
    for(i = 0; i < *pn; i++)
        fscanf(f, "%d", (*pa) + i);
    fclose(f);
}
void ki_bin(char * s, int n, int * a)
{
    FILE * f;
    f = fopen(s, "wb");
    fwrite(a, n, sizeof(int), f);
    fclose(f);
}
void titkosit(char * s)
{
    FILE * f; int x;
    f = fopen(s, "r+b");
    for(;;)
        {if(!fread(&x, 1, sizeof(int), f)) break;
         fseek(f, -sizeof(int), SEEK_CUR);
         x = ~x;
         fwrite(&x, 1, sizeof(int), f);}
    fclose(f);
}
void bin_to_txt(char * s1, char * s2)
{
    FILE * f, *g; int x;
    f = fopen(s1, "rb"); g = fopen(s2, "w");
    for(;;)
        {if(!fread(&x, 1, sizeof(int), f)) break;
         fprintf(g, "%d\n", x);}
    fclose(f); fclose(g);
}

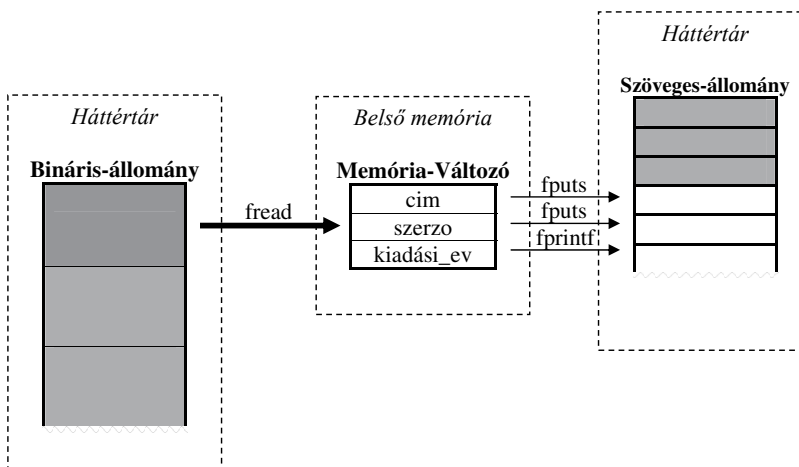
```

11.6. feladat. Írjunk programot, amely menüből megvalósítja az alábbi műveletek: beolvassa egyenként könyvek adatait (cím, szerző, kiadási év), kilistázza a nyilvántartásban lévő könyveket, töröl egy könyvet (amelyet a címe által azonosítunk) a nyilvántartásból. A könyvek adatait a billentyűzetről olvassuk be, egy konyv.dat bináris állományban tartjuk nyilván őket, a listát pedig egy lista.txt szöveges állományba írjuk ki.

Magyarázat: A main függvényben "kitesszük" a menüt, és várunk egy billentyűleütést. Attól függően, hogy a felhasználó 1-es, 2-es vagy 3-as karaktert ütött be, meghívjuk a beolvasas(), listazas(), illetve torles()



11.2. ábra.



11.3. ábra.

eljárásokat. Mindezt addig ismételjük míg a billentyűzetről 4-es karakter kerül beütésre (Kilépés). Továbbá lásd a kódban található kommentárokat.

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

typedef struct
{
    char cim[50],szerzo[50];
    int kiadasi_ev;
}KONYV;

void menu();
void beolvasas();
void listazas();
void torles();

int main()
{
    char c;
    do
    {
        menu();
        c = getch();
        switch(c)
        {
            case '1': beolvasas(); break;
            case '2': listazas(); break;
            case '3': torles(); break;
        }
    }
    while (c!='4');
    return 0;
}

void menu()
// A menüpontokat a sorszámaik szerint aktiváljuk
{
    printf("1:Beolvasas\n");
    printf("2:Listazas\n");
    printf("3:Torles\n");
    printf("4:Kilepes\n");
}

void beolvasas()
```

```
{
    FILE* f;
    KONYV k;
    // Ha az állomány már létezik, akkor a végére írunk,
    // ha nem, akkor új állományt hozunk létre
    f=fopen("konyv.dat","ab");
    if(!f) f=fopen("konyv.dat","wb");
    // Egy könyv adatainak beolvasása a billentyűzetről
    // A k változóba mezőnként olvasunk
    printf("Egy új könyv beolvasása:\n\n");
    printf("cím: ");gets(k.cím);
    printf("szerző: ");gets(k.szerző);
    printf("év: ");scanf("%i",&k.kiadasi_ev);
    fflush(stdin); // üríti a billentyűzet puffert
    // A k munkaváltozó tartalmának beírása a bináris állományba
    // A k változót úgy tekintjük, mint egyetlen adatblokk
    fwrite(&k,1,sizeof(KONYV),f);
    fclose(f);
}

void listazas()
{
    FILE *be, *ki;
    KONYV k;
    be=fopen("konyv.dat","rb");
    ki=fopen("lista.txt","wt");
    if(!be)
    {
        printf("Nincs egy könyv sem a könyvtárban");
        return;
    }
    while(!feof(be)) // Amíg nem értünk az állomány végére
    {
        // Kiolvassuk a k munkaváltozóba egy könyv adatait
        // (mint egyetlen adatblokkot)
        fread(&k,1,sizeof(KONYV),be);
        if (!feof(be)) // Ha nem az EOF jelt próbáltuk olvasni
        {
            // Beírjuk a könyv adatait mezőnként a szöveges állományba
            fputs(k.cím, ki);
            fputs(k.szerző, ki);
            fprintf(ki, "%i\n", k.kiadasi_ev);
            fprintf(ki, "\n");
        }
    }
    fclose(be);
}
```



```

    fclose(ki);
}

void torles()
// Létrehozunk egy új állományt, amelyik tartalmazza az
// összes könyvet, kivéve a törlendőt. Az első állományt
// letöröljük, majd az újat átnevezzük az eredeti nevével
{
    char torlendo_cime[50];
    KONYV k;
    FILE *f, *t;
    f = fopen("konyv.dat", "r");
    if(!f)
    {
        printf("Nincs egy könyv sem a könyvtárban");
        return;
    }
    t = fopen("temp.dat", "wb");
    printf("Melyik könyvet akarod törölni(cime):");
    gets(torlendo_cime);
    while(!feof(f))
    {
        fread(&k, 1, sizeof(KONYV), f);
        if (!feof(f) && strcmp(k.cim, torlendo_cime))
        {
            fwrite(&k, 1, sizeof(KONYV), t);
        }
    }
    fclose(f);
    fclose(t);
    remove("konyv.dat");
    rename("temp.dat", "konyv.dat");
}

```

11.4. Alacsony szintű állománykezelés

Mivel a C nyelvet eredetileg az UNIX operációs rendszerrel párhuzamosan fejlesztették, gyakorlatilag minden implementációja a nyelvnek tartalmaz egy második állománykezelő rendszert is, amelyet *Unix-féle file-kezelő rendszernek* vagy *alacsony szintű file-kezelésnek* neveznek.

Az alacsony szintű file-kezelés nem puffert, azaz a rendszer nem tartalmaz beépített puffereket. Így például a `read` és `write` függvények (lásd lennebb) közvetlenül a memóriából olvasnak, illetve ide írnak, valahányszor meghívjuk őket.

Bár a UNIX-féle rendszer nem ANSI standard, amint már említettük, gyakorlatilag minden C fordító elfogadja. Viszont a C nyelv sok implementációja nem engedi meg az ANSI és UNIX függvények együttes használatát ugyanabban a programban.

A magas szintű file-kezeléstől eltérően, az alacsony szintű file-kezelés nem `FILE`-pointert, hanem `int` típusú *file-leíró* (`fd` – *file descriptor*) használ az állományok azonosítására a C program keretén belül.

Az alábbiakban összefoglaljuk az alacsony szintű file-kezelés főbb függvényeit. Prototípusaik az `io.h` header-állományban találhatóak meg. Egyesek közülük igénylik a `fcntl.h` header-állományt is.

Új állomány létrehozása:

```
int create(const char * <állománynév>, int <mód>);
```

A `<mód>` az alábbi makrók egyike lehet (a `fcntl.h` header-állományban vannak definiálva):

`O_RDONLY` Olvasás

`O_WRONLY` Írás

`O_RDWR` Olvasás/Írás

Ha a `create` függvénynek sikerül létrehozni az állományt, akkor az állományleíró (`fd`) téríti vissza. Sikertelen művelet esetén `-1`-et.

Létező állomány megnyitása:

```
int open(const char * <állománynév>, int <mód>);
```

Ugyanazok a megjegyzések érvényesek, mint az előbbi függvény esetén.

Példa olvasás/írásra való állománymegnyitásra:

```
int fd;
if(fd=open("be.dat", O_RDWR)==-1)
{puts("Nem tudom megnyitni az állományt");
return;}
```

Állomány bezárása:

```
int close(int fd);
```

Írás állományba:

```
int write(int fd, const void *<puffer>, unsigned
        <szám>);
```

Kiír az `fd` file-leíróval azonosított állományba `<szám>` darab bájtot, a `<puffer>` címről.

Olvadás állományból:

```
int read(int fd, const void *<puffer>, unsigned
        <szám>);
```

Beolvas az `fd` file-leíróval azonosított állományból `<szám>` darab bájtot, a `<puffer>` címre.

Állomány törlése:

```
int unlink(const char * <állománynév>);
```

Véletlen hozzáférés az állományhoz:

```
long lseek(int fd, long <nr>, int <kezdet>);
```

Elmozdítja `<nr>` byte-tal a pozíciómutatót `<kezdet>`-hez képest.

A `kezdet` paraméter lehetséges értékei:

`SEEK_SET` (állomány eleje)

`SEEK_CUR` (pozíciómutató aktuális pozíciója)

`SEEK_END` (állomány vége)

A pozíciómutató aktuális értékét téríti vissza sikeres művelet esetén. Ellenkező esetben `-1`-et.

12. FEJEZET

ÁTFOGÓ KÉP A C NYELVRŐL

Ennek a fejezetnek az a célja, hogy segítsen az olvasónak fölé emelkedni mindannak, amit eddig tanult, és ezáltal jobb rálátást nyerni a C programozás bizonyos területeire.

12.1. Strukturált programozás

Az algoritmus

Valahányszor számítógéppel szeretnénk megoldani egy feladatot, a feladat elemzése után a feladatmegoldás egy hatékony algoritmus keresésével kezdődik.

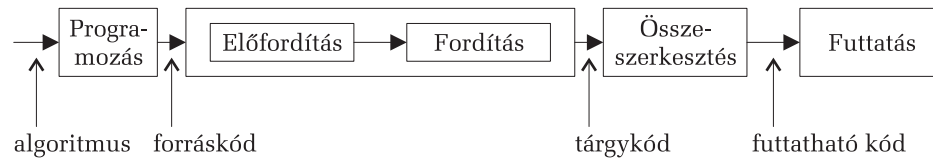
A strukturált programozás azt jelenti, hogy algoritmusunk az alábbi három alapstruktúrából épül fel (ezek bárhogy kombinálhatók, sőt egymásba is ágyazhatók):

1. szekvencia (két vagy több művelet szekvenciálisan követi egymást),
2. elágazás (döntési struktúra),
3. ismétlés (elől-tesztelő ciklus).

Böhm és *Jakopini* híres tétele matematikailag igazolta, hogy bármely algoritmus, amelynek egyetlen bemenete (start) és egyetlen végpontja (stop) van, felépíthető a fent említett három szerkezet segítségével.

Amennyiben több algoritmus is rendelkezésre áll, idő- és tárhelyigény alapján lehet kiválasztani a legmegfelelőbbet.

Hogyan jutunk el az algoritmustól a végrehajtható kódig?



Ezen lépések közül elsődleges fontossággal bír a forráskód elkészítése. Ennek megvalósítása teljes mértékben a programozó feladata. A többi lépést (fordítás, összeszerkesztés, futtatás) a számítógép végzi „parancsra”.

Programozáskor az algoritmus által előírt műveleteket utasításokkal helyettesítjük. A strukturált programozás alapelvei¹ tükröződnek a C nyelv alaputasítás-készletében is:

- egymás utáni kifejezésutasítások szekvenciát alkotnak,
- az **if** és a **switch** két-, illetve többelágazású döntések,
- a **while**, **for**, illetve **do-while** ciklusutasítások.

12.1.1. Feltételes fordítás

Bár a fordítás "automatikusan" történik, az előfeldolgozón keresztül egy bizonyos mértékben befolyásolható. Ez az előfeldolgozó feltételes fordítási direktívái által érhető el. A feltételes fordítás lehetővé teszi, hogy a forrásállomány bizonyos részei csak adott feltételek mellett kerüljenek be az előfeldolgozó által fordításra előkészített állományba. A leggyakrabban használt feltételes fordítási direktívák az **#if**, **#else**, **#elif**, **#endif**, **#ifdef**, **#ifndef**. (A makrókkal foglalkozó fejezetben megemlítettük, hogy a fordítási direktívákat a **#** karakter előzi meg). Az alábbi példákban könnyen megérthető, hogy miként használhatók az előbbieken felsorolt fordítási direktívák.

1. *példa*: Az 1-essel jelölt programsor csak akkor kerül be a fordításra előkészített állományba, ha az N szimbolikus konstans értéke kisebb mint 1000. (Jelen esetben bekerül, hiszen N=150)

```
#include <stdio.h>
#define N 150
main()
{
    #if N<1000
        puts("Lefordítva 1000-nél kevesebb elemű
            tömbre!"); //1
    #endif
    . . .
    return 0; }
```

¹ Véleményem szerint a **break** és **continue** ugró utasítások használata még összhangban van a strukturált programozás alapelveivel, a **goto**-é viszont már nem, ezért általában kerülendő.

2. *példa*: Ha az N szimbolikus konstans értéke kisebb mint 1000, akkor az 1-essel jelölt programsor kerül be a fordításra előkészített állományba, különben a 2-essel jelölt. (Jelen esetben a 2-essel jelölt kerül be, hiszen N=1500)

```
#include <stdio.h>
#define N 1500
main()
{
    #if N<1000
        puts("Lefordítva 1000-nél kevesebb elemő
        tömbre!");//1
    #else
        puts("Lefordítva 1000-nél több (vagy egyenlő)
        elemű tömbre!");//2
    #endif
    . . .
    return 0;
}
```

3. *példa*: Attól függően, hogy a SAPIENTIA_KAR szimbolikus konstans értéke a KOLOZSVAR, MAROSVASARHELY vagy CSIKSZEREDA szimbolikus konstansokkal egyenlő (azaz 1, 2 vagy 3), a fordításra előkészített állományba a diakletszam globális változó definíciója (kezdőértékadással) az 1-es, 2-es vagy 3-as variánsban kerül be. (Jelen esetben a második változatban)

```
#define KOLOZSVAR 1
#define MAROSVASARHELY 2
#define CSIKSZEREDA 3
#define SAPIENTIA_KAR MAROSVASARHELY
#if SAPIENTIA_KAR == KOLOZSVAR
    int diakletszam = 500;//1
#elif SAPIENTIA_KAR == MAROSVASARHELY
    int diakletszam = 1500;//2
#else
    int diakletszam = 2000;//3
#endif
main()
{
    . . .
    return 0;
}
```

}

4. *példa*: Attól függően, hogy a PI szimbolikus konstans definiálva van vagy sem, a fordításra előkészített állományba az 1-es vagy a 2-es programsor kerül be. (Jelen esetben az 1-essel jelölt). Továbbá, annak függvényében, hogy a GYOK2 szimbolikus konstans nincs definiálva vagy van, a fordításra előkészített állományba az 3-as vagy a 4-es programsor kerül be. (Jelen esetben az 4-essel jelölt).

```
#include <stdio.h>
#define PI 3.14
#define GYOK2 1.41
main()
{
    #ifdef PI
        puts("Definiálva van a PI szimbolikus
        konstans!"); //1
    #else
        puts("Nincs definiálva a PI szimbolikus
        konstans!"); //2
    #endif
    #ifndef GYOK2
        puts("Nincs definiálva a GYOK2 szimbolikus
        konstans!"); //3
    #else
        puts("Definiálva van a GYOK2 szimbolikus
        konstans!"); //4
    #endif
    . . .
    return 0;
}
```

Megjegyzések:

1. Az `#if`, `#elif`, `#ifdef`, `#ifndef` direktívák esetén a feltétel csakis konstans kifejezés lehet, amely, nyilván, logikailag értékelődik ki. (Ha az értéke nulla, akkor hamis, különben igaz)
2. Az `#elif` direktíva az "else-if" szerkezetet implementálja.
3. Az `#if`, `#elif`, `#ifdef`, `#ifndef` direktívák egymásbaágyazhatók. (Az ANSI C standard szerint legalább nyolc szinten)
4. Az `#ifdef` direktíva helyett használhatunk `#if` direktívát és `defined` operátort.
Az `#ifdef` `<makroazonosító>` alak azonos

az `#if` `defined` <makro_azonosító> alakkal.

Hasonlóképpen az `#ifndef` <makro_azonosító> alak azonos

az `#if!defined` <makro_azonosító> alakkal.

Az ANSI C standard az előfeldolgozó részére az alábbi direktívákat tartalmazza még (ezek részletes leírásuk végett lásd a szakirodalmat)

`#define` - lásd még a 10-dik fejezetet

`#undef` - lásd még a 10-dik fejezetet

`#include` - lásd még a 12.3 alfejezetet

`#error`

`#line`

`#pragma`

Felsoroljuk továbbá az ANSI C standardban elődefiniált makró-konstansokat (mindenik előtt és után két aláhúzásjel található):

`_LINE_` - a forráskód lefordított sorainak száma

`_FILE_` - a forráskód neve, mint karakterlánc

`_DATE_` - a forráskód lefordításának napja, mint karakterlánc (hónap/nap/év)

`_TIME_` - a forráskód lefordításának időpontja, mint karakterlánc (óra/perc/másodperc)

`_STDC_` - ha a forráskód ANSI C standard szerint lett implementálva, akkor értéke 1.

12.2. Moduláris programozás

Egy C program lehetséges szerkezete:

<preprocesszor direktívák>

`#include` direktívák

`#define` direktívák

<globális definiálások>

típusdefiniálások (`typedef`)

változó definiálások

sajátfüggvény deklarációk (prototípusok)

<függvénydefiníciók>

`main` függvény

sajátfüggvények

Egy C függvény szerkezete:

<típus> <név> (<formális_paraméter_lista>)


```
{
    <lokális_definiálások>
    <utasítások>
}
```

12.2.1. Modulok, blokkok

Eddig minden C programunk egyetlen állományban került implementálásra. Úgy is mondhatnánk, egy modulból állt. Lehetőség van egy nagyobb méretű program több modulra való szétbontására azáltal, hogy függvényeit – bizonyos szempont szerint csoportosítva – szétszjtjuk több állományba. Ezen modulok (állományok) külön fordíthatók, és ennek következtében külön tesztelhetők is. Ez különösen hasznos, amikor több programozó dolgozik együtt egy nagyobb méretű projekt megvalósításán.

A moduláris programozás alapelve:

- a feladatot leosztjuk részfeladatokra,
- külön modulokban implementáljuk a részfeladatokat megoldó függvényeket; a főmodul fogja tartalmazni a main függvényt,
- összeszerkesztjük a modulokat egyetlen programmá.

Egy modulon belül létezik az úgynevezett *blokk-hierarchia*. *Blokk*nak nevezzük egy kapcsos zárójelek közé zárt programrészt. Az első szinten a függvények törzsét képező blokkok vannak. Mivel a C nyelv nem teszi lehetővé függvény definiálását függvényben, ezért a függvényblokkok mind ugyanazon a hierarchiaszinten vannak. Ennek ellenére létezik egyfajta alárendeltség a függvények között, annak alapján, hogy melyik függvény melyiket hívja meg.

A függvényblokkokon belül lehetnek alblokkok, amelyeket összetett utasításoknak neveztünk. Ahogy erre a hierarchia kifejezés is utal, bármely blokknak lehetnek további alblokkjai.

Egy modulra nézve globális definiálások minden függvényen kívül történnek, a blokkokra nézve lokális definiálások pedig az illető blokk elején.

A fenti elemzés a következő szemléltetésre ihlet: egy C program az emberi szervezethez hasonlítható. A testtagok a modulok, az ezeket alkotó sejtek pedig a függvények. Egy jól megírt program jól áttekinthető és hatékony, akár egy egészséges ember. Egy gyenge program, még ha működőképes is, olyan lehet, mint egy beteg (rokkant) ember.

12.3. Definíció és deklaráció

A C nyelv különbséget tesz definíció és deklaráció között.

Egy program végrehajtása szempontjából *objektumnak* nevezünk egy névvel azonosított memóriaterületet.

A függvények és változók ilyen objektumok.

- Minden függvény névvel rendelkezik, és biztosítva van számára egy memóriaterület, amely a függvény kódját tartalmazza (ezért tudtunk beszélni a függvény címéről).
- Hasonlóképpen, a változók is névvel rendelkező memóriaterületek, amelyek tartalma viszont – a függvényekkel ellentétben – változhat.

Akkor beszélünk egy objektum *definálásáról*, amikor tárhelyet foglalunk neki, és meghatározzuk bizonyos jellemzőit. *Deklaráláskor* csak bejelentjük az illető objektumot a fordítónak, hogy hivatkozni tudjunk rá. Természetesen minden definíció egyben deklaráció is, de fordítva nem igaz.

A függvények kapcsán eddig is különbséget tettünk deklaráció (prototípus) és definíció (a függvény megírása) között, a változók esetében azonban kizárólag definíciót használtunk.

Ahhoz, hogy hivatkozni tudjunk egy függvényre, nem volt szükség arra, hogy föltétlenül definiálva legyen azt megelőzően, a fordító beírta azzal is, ha találkozott már a deklarációjával.

Mivel a könyvtári függvények deklarációi a *header*- (fej-) állományokban vannak csoportosítva, ezért szükséges volt bizonyos headerállományok beékelése a programunkba. Az `#include` direktíva hatására a preprozessor csak logikailag építi be a headerállományokat a forráskódba (tehát a forráskód változatlan marad).

Az `#include` direktíváról megemlíjtük még, hogy lehetővé teszi bármilyen állomány logikai beékelését a forráskódunkba. Ebben az esetben az állomány nevét idézőjelek közé tesszük, mint a karakterlánckonstansek esetében szokás. Ha elérési utat is megadunk, nem kell a back-slash karaktereket megdupláznunk (a C programban ez kötelező volt).

A sajátfüggvények deklarációit célszerű a **main** függvény előtt felsorakoztatni, és a definíciójukat a **main** után megírni. Sőt – különösen, ha több modulból áll a programunk – ajánlatos egyes modulokhoz saját headerállományt készíteni, amely tartalmazza az illető modulban megírt függvények deklarációit. Ha így teszünk, akkor ahhoz, hogy valamely modul függvényei meghívhassák egy másik modul függvényeit, csak arra van szükség, hogy az első modul headerállományát beékeljük a második modulba. (Lásd a megoldott feladatot a fejezet végén.)

Ahhoz, hogy a változók esetében átlássuk a különbséget a definíció és deklaráció között, alaposabban meg kell ismerkednünk a változó fogalmával.

12.4. A változók osztályozása

A változóknak négy jellemzőjével ismerkedtünk meg eddig: név, cím, típus (méret és értéktartomány) és érték. További kérdések merülnek azonban fel, amelyek a változók további jellemzőihez vezetnek el:

- Milyen memóriaterületen történik a helyfoglalás?
- Mely programsorokból érhető el?
- Mikor jönnek létre és mikor számolódnak fel?

1. *Memóriaosztálya* (azt határozza meg, hogy hol kerül eltárolásra az adott változó a memóriában):
 - a) egy adatszegmensben,
 - b) a mikroprocesszor valamely regiszterében,
 - c) a heap-en,
 - d) a stack-en.
2. *Láthatósági tartománya* (azaz a forráskód mely programsoraiból érhető el az illető változó; az határozza meg, hogy hol kerül definiálásra a program keretén belül):
 - a) blokkszintű,
 - b) modul- (állomány-) szintű.
3. *Élettartam* (az az időszak, ameddig egy változó létezik – tárhely van lefoglalva számára – a memóriában; egy változó élettartamát a tárolási osztálya határozza meg):
 - a) statikus élettartam:
 - i. az adatszegmensben történik tárhelyfoglalás számukra,
 - ii. léteznek a memóriában a teljes programfutás alatt,
 - iii. automatikusan nullával inicializálódnak;
 - b) lokális élettartam:
 - i. a stack-en vagy a mikroprocesszor valamely regiszterében történik a helyfoglalás számukra,
 - ii. egy adott blokk (függvény vagy összetett utasítás) végrehajtásának ideje alatt léteznek;
 - c) dinamikus élettartam:
 - i. heap-en jönnek létre,

- ii. a helyfoglalás és helyfelszabadítás a programfutás alatt történik speciális függvények által (**malloc**, **calloc**, **free**).

Egy változó imént bemutatott jellemzői kétféleképpen határozhatók meg:

- implicit módon: attól függően, hogy hol kerülnek definiálásra,
- explicit módon: az **extern**, **static**, **register** és **auto** módosító jelzők által.

A fenti jellemzők alapján a változók három csoportba sorolhatók:

- *globális változók*,
- *lokális változók*,
- *statikus változók*.

A globális változók tulajdonságai:

(Nevüket onnan kapták, hogy láthatósági tartományuk kiterjeszthető a teljes programra – amennyiben több modulból áll.)

- minden függvényen kívül történik a definiálásuk,
- létrejönnek már a programfutás előtt az adatszegmensen, és léteznek a teljes programfutás alatt,
- implicit statikus élettartamúak,
- implicit inicializálva vannak nullával,
- láthatósági tartományuk a modul (állomány), amelyben definiáltuk őket, és elérhetőek bárhol, a definiálásuk helyétől lefele az állomány végéig,
- láthatósági tartományuk kiterjeszthető más modulokra is. Hogyan?

Globális változóra hivatkozhatunk olyan modulokban is, amelyek nem tartalmazzák a definiálását, amennyiben bejelentjük deklarálás által.

Ezen deklarálás szintaxisa

extern <típus> <változó>;

Az **extern** szó arra utal, hogy <változó> más modulban lett definiálva. Ez az egyetlen helyzet, amikor sor kerül egy változónak a *definiálásától* különválasztott *deklarálására*.

Példa:

1. állomány

```
int a = 10; /* definíció */
void nyomtat(); /* deklaráció */
main()
{
    nyomtat(); return 0;
}
```

2. állomány

```
#include <stdio.h>
extern int a; /* deklaráció */
void nyomtat() /* definíció */
{
    printf("%d", a);
}
```

Felmerül egy kérdés: ha egy változó külső (**extern**) egy modulban, mekkora a láthatósági tartománya?

Attól függ, hogy hol van deklarálva! Ha minden függvény blokkján kívül, akkor a deklaráció helyétől lefele az állomány végéig látható, ha viszont valamely blokkon belül, akkor a deklaráció helyétől lefele az illető blokk végéig.

A lokális (automatikus) változók tulajdonságai:

(Nevüket onnan kapták, hogy *láthatósági tartományuk blokkszintű és nem terjeszthető ki*; csak definiálhatók.)

- definiálásuknak valamely blokk elején kell történnie, és láthatósági tartományuk az illető blokk területe, anélkül hogy kiterjeszthető lenne,
- definiálásukat megelőzheti az **auto** kulcsszó, de ennek sok értelme nincs, hiszen implicit automatikus minden olyan változó, amelyet valamely blokkon belül definiálunk,
- általában a stack-en jönnek létre, de ha definiálásukat megelőzi a **register** kulcsszó, akkor lehetőség szerint a mikroprocesszor valamelyik regiszterében történik helyfoglalás számukra,
- tárhelyfoglalás számukra programfutás közben történik, amikor a programfutás a definiálásukhoz érkezik,
- az illető blokk befejeztével felszámolódnak (felszabadul a számukra lefoglalt memóriaterület).

Példa:

```
#include <stdio.h>
int f ()
{
    int b = 3, a = b;
    return ++a;
}
```

```

}
main()
{
    printf ("a = %d\n", f());
    printf ("a = %d", f());
    return 0;
}
    
```

Magyarázat: A program két 4-est fog kiírni a képernyőre. Miért? Hiába volt kétszer meghívva az `f` függvény, hiszen minden függvényhívás végén a lokális változók felszámolódnak, és újabb hívás alkalmával mások jönnek létre.

A statikus változók tulajdonságai:

(Ha egy változódefiniálás elé odaírjuk a `static` szót – legyen az globális vagy lokális – statikussá változtatjuk.)

- létrejönnek már a programfutás előtt az adatszegmensben, és léteznek a teljes programfutás alatt,
- implicit inicializálva vannak nullával,
- láthatósági tartományuk nem terjeszthető ki,
- ha minden függvényen kívül definiáltuk őket, akkor láthatósági tartományuk modulszintű (külső statikus változó),
- ha valamely blokk elején definiáltuk őket, akkor blokk szintű a láthatósági tartományuk (belső statikus változó),
- tárolási osztályukat (adatszegmens) és élettartamukat (statikus) tekintve megegyeznek a globális változókkal, de láthatósági tartományukat illetően különböznek ezektől, hiszen láthatósági tartományuk – még ha ez modulszintű is – nem terjeszthető ki más modulokra,
- a blokkokon belül definiált statikus változók láthatósági tartományuk tekintetében megegyeznek a lokális változókkal, de tárolási osztályukat (adatszegmens) és élettartamukat (statikus) illetően különböznek ezektől; léteznek a teljes programfutás alatt, de nem érhetők el, csak az illető blokkon belül.

Példák:

```
#include <stdio.h>
static double a;
main()
{
    ++a;
    printf("a=%.21f",a);
    return 0;
}
```

```
#include <stdio.h>
main()
{
    {static double a;}
    ++a;
    printf("a=%.21f",a);
    return 0;
}
```

```
#include <stdio.h>
int f ()
{static int a;
return ++a;}
main()
{
    printf("a=%d\n",f());
    printf("a=%d",f());
    return 0;
}
```

Magyarázat:

- Az első program 1-et fog kiírni a képernyőre, ellenőrizve ezáltal, hogy **a** tényleg inicializálva lett nullával. Az **a** változó csak abban különbözik ez esetben a globális változóktól, hogy láthatósági tartománya nem terjeszthető ki más modulokra.
- A második program szintaktikailag hibás, hiszen **a** láthatósági tartománya a blokk, amelyben definiáltuk.
- Az utolsó program demonstrálja, hogy a statikus változók élettartama a programfutas végéig tart, bár láthatósági tartományuk blokkszintű. A program **a=1**-et, majd **a=2**-t fog kiírni a képernyőre. Fontos észrevenni, hogy **a** értéke megmaradt, bár kiléptünk a függvényből. Azért nem zavaró, hogy kétszer is átmentünk a definiálásán, mert a statikus változók részére a helyfoglalás, akár a globális változók esetében, programfutas előtt történik.

A fenti tulajdonságok magyarázattal szolgálnak arra is, hogy miért inicializálhatók a lokális változók változóértékekkel is (programfutas alatt jönnek létre), szemben a globális és statikus változókkal, amelyek csakis állandókat kaphatnak kezdőértékként (programfutas előtt jönnek létre).

Még egy kérdés: lehet-e két változónak ugyanaz a neve?

Ha különbözik a láthatósági tartományuk, akkor igen. Ezért tudtunk azonos nevű lokális változókat definiálni más-más függvényekben, sőt azonos nevű globális és lokális változókat is. Amikor két azonos nevű változó láthatósági tartománya átfedődik (globális és lokális változók), akkor a kisebb láthatósági tartományú változó úgymond eltakarja a nagyobb láthatósági tartományút.

Példák:

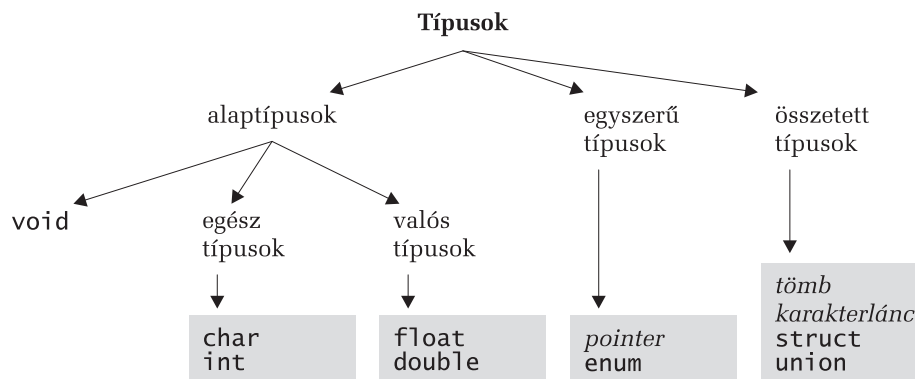
```
#include <stdio.h>
int a = 2;
main()
{
    int a = 3;
    printf ("a = %d", a);
    return 0;
}
```

```
#include <stdio.h>
int a = 1;
main()
{
    {
        int a = 2;
        {
            int a = 3;
            printf ("a=%d\n",a);
        }
        printf ("a = %d\n", a);
    }
    printf ("a = %d", a);
    return 0;
}
```

Magyarázat: Az első program 3-at, a második pedig 3, 2, 1-et ír ki a képernyőre. A második programban az első `printf` végrehajtása alkalmával létezik a memóriában mind a három `a` változó, de a 3-as értékű az aktív. A második `printf`-nél már csak az 1, illetve 2 értékű változók léteznek, és a lokális az aktív. Végül az utolsó `printf`-nél egyedül a globális `a` létezik, így nyilván ennek az értéke kerül kiírásra.

12.5. A C nyelv adattípusai felülnézetből

Az alábbi ábra átfogó képet nyújt a C nyelv adattípusairól:



Típusmódosítók az alaptípusokhoz: signed, unsigned, short, long.

Módosító jelzők: **const**, **volatile**, **typedef**.

Az első kettő a változókhoz való hozzáférést szabályozza. A **const** módosító jelző megakadályozza az illető változó megváltoztatását programból. Természetesen használatának csak akkor van értelme, ha definiáláskor kezdőértéket adtunk a változónak. Ha használjuk a **volatile** módosító jelzőt, akkor ezzel engedélyezzük, hogy a változó értéke megváltoztatható legyen a programon kívüli okokból is. Például, ha szeretnénk egy globális változóban eltárolni a valós időt, ezt úgy valósíthatjuk meg, hogy cím szerint átadjuk az operációs rendszer megfelelő időrutinjának. Egy ilyen változót **volatile**-nek kell definiálni. Együtt is használható a **const** és **volatile**, ha azt szeretnénk, hogy csak külső hatások tudjanak megváltoztatni egy változót.

A **typedef**-et alaposan bemutattuk a megfelelő fejezetben.

12.6. Egy összetett feladat

Azért, hogy gyakorlatiasabb legyen mindaz, ami eddig bemutatásra került ezen fejezet keretén belül, alkalmazzuk a tanultakat egy konkrét feladat elvégzésében.

Gyakran találkoztunk olyan feladatokkal, amelyek egy szám számjegyein, illetve egy számsorozat elemein végeztek különféle műveleteket.

12.1. feladat. Készítsünk az alábbiakban két modult: az első (**szamjegy.c**) tartalmazza azokat a függvényeket, amelyek számjegyenként dolgozzák fel a paraméterként kapott természetes számot, a második (**szamsor.c**) pedig azokat, amelyek elemenként foglalkoznak a paraméterként kapott számsorozattal (a legtöbb függvény rekurzívan van implementálva a rekurzió begyakorlása érdekében). Mindkét modul részére készítsünk egy-egy headerállományt (fejállomány) (**szamjegy.h**, **szamsor.h**), amely tartalmazza az illető függvények deklarációit (prototípusait). Egy harmadik modul (**feladat.c**), használva az előzők függvényeit, oldjon meg egy konkrét feladatot.

```
szamjegy.c

#include<stdio.h>

int osszeg(long n)
{
```

```

        if(n<10) return n;
        return osszeg(n/10)+n%10;
    }
    int szorzat(long n)
    {
        if(n<10) return n;
        return szorzat(n/10)*(n%10);
    }
    int szamjegyek(long n)
    {
        if(n<10) return 1;
        return szamjegyek(n/10)+1;
    }
    float atlag(long n)
    {
        return (float)osszeg(n)/szamjegyek(n);
    }
    int min_szamjegy(long n)
    {
        int t;
        if(n<10) return n;
        t=min_szamjegy(n/10);
        return t<(n%10)?t:(n%10);
    }
    int max_szamjegy(long n)
    {
        int t;
        if(n<10) return n;
        t=max_szamjegy(n/10);
        return t>(n%10)?t:(n%10);
    }
    int van_adott_szamjegy(long n, int x)
    {
        if(!n) return 0;
        if(n%10==x) return 1;
        return van_adott_szamjegy(n/10, x);
    }
    int adott_szamjegyek(long n, int x)
    {
        if(!n) return 0;
        return adott_szamjegyek(n/10,x)+(n%10==x);
    }
    void forditva(long n)
    {
        printf("%d", n%10);
        if(n>0) forditva(n/10);
    }

```

}

szamsor.c

#include<stdio.h>

int beolvas(**int***np, **long****ap, **char***all)

```
{
    FILE *f; int i;
    if(!(f=fopen(all, "r"))) return 0;
    fscanf(f,"%d",np);
    (*ap)=(long*)malloc((*np)*sizeof(long));
    for(i=0; i<n; i++)
        fscanf(f,"%ld",(*ap)+i);
    fclose(f); return 1;
}
```

int kiir(**int** n, **long** *a, **char** *all)

```
{
    FILE *f; int i;
    if(!(f=fopen(all, "w"))) return 0;
    fprintf(f, "%d\n",n);
    for(i=0; i<n; i++)
        fprintf(f,"%ld ",a[i]);
    fclose(f); return 1;
}
```

void utolsok_forditva(**int** n, **long** *a, **int** m, **int** i)

```
{
    static k;
    if(i<n)
    {
        utolsok_forditva(n, a, m, i+1);
        if(k<m){printf("%ld ", a[i]); k++;}
    }
}
```

void beszur(**int** n, **long** *a, **long** x)

```
{
    while(n>0 && a[n-1]>x)
    {
        a[n]=a[n-1];
        n--;
    }
    a[n]=x;
}
```

void rendez(**int** n, **long** *a)

```
{
    if(n>1)
```

```

    {
        rendez(n-1,a);
        beszur(n-1,a,a[n-1]);
    }
}
int szamol(int n,long*a,int(*c)(long))
{
    int i, k=0;
    for(i=0; i<n, i++)
        if(c(a[i]))k++;
    return k;
}

```

szamjegy.h

```

int osszeg(long);
/* kiszámítja a paraméter számjegyeinek összegét */

int szorzat(long);
/* kiszámítja a paraméter számjegyeinek szorzatát */

int szamjegyek(long);
/* kiszámítja a paraméter számjegyeinek számát */

float atlag(long);
/* kiszámítja a paraméter számjegyeinek átlagát */

int min_szamjegy(long);
/* meghatározza a paraméter legkisebb számjegyét */

int max_szamjegy(long);
/* meghatározza a paraméter legnagyobb számjegyét */

int van_adott_szamjegy(long, int);
/* ellenőrzi, hogy megtalálható-e a második paraméter
   mint számjegy az első paraméterben */

int adott_szamjegyek(long, int);
/* megszámolja, hogy hányszor fordul elő a második
   paraméter mint számjegy az első paraméterben */

void forditva(long);
/* kiírja a paramétert fordítva */

```

szamsor.h

```
int beolvas(int *,int **,char *);
/* beolvassa az elemek számát (az első paraméterként
   kapott címre), helyet foglal a számsorozatnak
   (a második paraméterként kapott címre eltárolja
   a tömb címét), majd beolvassa az elemeket
   (a harmadik paraméterként megadott állományból) */

int kiir(int, int *, char *);
/* kiírja az elemek számát (első paraméter), majd
   az elemeket (a második paraméterként kapott tömbből)
   a megadott állományba (harmadik paraméter) */

void utolsok_forditva(int,int*,int,int);
/* kiírja a számsorozat adott számú utolsó elemét:
   első paraméter: elemek száma
   második paraméter: a tömb
   harmadik paraméter: hányat írjunk ki?
   negyedik paraméter: a számsorozat első elemének indexe
   a tömbben */

void beszur(int, int *, int);
/* beszúrja az utolsó paraméter értékét a számsorozatba:
   első paraméter: elemek száma
   második paraméter: a tömb */

void rendez(int, int *);
/* rendez a számsorozatot:
   első paraméter: elemek száma
   második paraméter: a tömb */

int szamol(int, int *, int (*)(int));
/* megszámlolja az adott tulajdonságú elemeket
   a számsorozatban:
   első paraméter: elemek száma
   második paraméter: a tömb
   harmadik paraméter: egy függvény, amely ellenőrzi
   az illető tulajdonságot */
```

12.2. feladat. Adott egy számsorozat a `be.txt` állományban. Írjuk ki a `ki.txt` állományba a rendezett számsorozatot, majd a képernyőre a következőket:

- a rendezett számsorozat utolsó `m` elemét,
- a rendezett számsorozat minden elemének tükrözését,
- hány eleme van, amelyekre teljesül, hogy:

1. számjegyeinek összege páros,
2. számjegyeinek szorzata páratlan,
3. számjegyeinek átlaga egész,
4. legkisebb számjegye nulla,
5. tartalmaz nulla számjegyet,
6. 9-es és 0 számjegyeinek száma azonos.

```
feladat.c

#include <stdio.h>
#include "szamjegy.h"
#include "szamsor.h"

int c1(long n)
{return !(osszeg(n)%2);}

int c2(long n)
{return szorzat(n)%2;}

int c3(long n)
{
    int t;
    t=atlag(n);
    return (int)t==t;
}

int c4(long n)
{return min_szamjegy(n)==0;}

int c5(long n)
{return van_adott_szamjegy(n,0);}

int c6(long n)
{return adott_szamjegyek(n,0)==adott_szamjegyek(n,9);}

main()
{
    int n, *a, m, i; long *a;
    beolvas(&n,&a,"be.txt");
    rendez(n,a);
    kiir(n,a,"ki.txt");
    scanf("%d", &m);
    utolsok_forditva(n,a,m,0);
    printf("\n");
    for(i=0;i<n;i++)
        printf("%ld ", forditva(a[i]));
```

```
printf("\n");
printf("%d\n", szamol(n,a,c1));
printf("%d\n", szamol(n,a,c2));
printf("%d\n", szamol(n,a,c3));
printf("%d\n", szamol(n,a,c4));
printf("%d\n", szamol(n,a,c5));
printf("%d\n", szamol(n,a,c6));
return 0;
}
```

Hogyan lehet a fenti modulokat egyetlen végrehajtható programmá szerkeszteni?

Léteznek parancssor-orientált fordítók (ez a klasszikus változat) és programozási környezetbe ágyazott fordítók. Ha programozási környezetben programozunk, akkor programunk moduljait összefoghatjuk egy **project**-be. Ha parancssor-fordítóval dolgozunk, akkor egy külön szövegszerkesztővel kell megszerkesztenünk a moduljainkat, ezután egyenként le kell fordítanunk, majd össze kell szerkesztenünk őket. Például a Linux alatti **cc** (C-Compiler) C-fordító segítségével a következő parancsokkal állítható elő a futtatható kód (ha a modulállományok már elkészültek):

```
cc -c szamjegy.c /* fordítás: eredménye a szamjegy.o tár-
gykód */
cc -c szamsor.c /* fordítás: eredménye a szamsor.o tár-
gykód */
cc -c feladat.c /* fordítás: eredménye a feladat.o tár-
gykód */
cc szamjegy.o szamsor.o feladat.o -o feladat.x /* sz-
erkesztés: a futtatható kód neve legyen feladat.x */
./ feladat.x /* a futtatható kód végrehajtása */
```


A. FÜGGELÉK

REKURZIÓ EGYSZERŰEN ÉS ÉRDEKESEN

A tanulás legyen teljesen gyakorlatias,
teljesen szórakoztató, . . . , olyan, hogy
általa az iskola valóban a játék helyévé,
vagyis az egész élet előjátékává váljon.
(Comenius)

Tegyük fel, hogy egy bizonyos engedélyt szeretnél kiváltani a polgármesteri hivataltól. Az első irodában közlik veled, hogy az engedély megszerzése feltételezi egy másik engedély birtoklását, amelyet egy másik irodában állítanak ki. Amikor belépsz oda, ugyanazt a választ kapod, mint az előző irodában. És ez így folytatódik addig, míg egy olyan engedélyhez nem jutsz, amelyik megszerzése már nem feltételezi egy további engedély birtoklását. Minekutána ezt kiváltottad, folytathatod a félbehagyott kísérleteidet – fordított sorrendben –, míg minden szükséges engedélyt meg nem szerzel. Végül az első irodában fogják a kezébe adni azt az engedélyt, amiért beléptél a hivatal ajtaján.

Rekurzió a matematikában

Bár a fenti kálváriához hasonlót tapasztalhattál már, mégis, a rekurzió fogalmával valószínűleg matekórán találkoztál először, a rekurzív képletek kapcsán. Klasszikus példa erre a faktoriális rekurzív képlete. A matematikusok az első n ($n > 0$) természetes szám szorzatát n *faktoriális*nak nevezik és $n!$ -lel jelölik. A $0!$ értéke megegyezés szerint 1.

$$n! = \begin{cases} 1, & \text{ha } n = 0, \\ 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n, & \text{ha } n > 0. \end{cases} \quad (13.1)$$

Ha a fenti képletben az $1 \cdot 2 \cdot \dots \cdot (n-1)$ szorzatot $(n-1)!$ -lel helyettesítjük, akkor eljutunk a faktoriális rekurzív képletéhez.

$$n! = \begin{cases} 1, & \text{ha } n = 0, \\ (n-1)! \cdot n, & \text{ha } n > 0. \end{cases} \quad (13.2)$$

Ezt a képletet azért nevezzük rekurzívnak, mert az $n!$ kiszámítását ($n > 0$ esetén) visszavezeti $(n-1)!$ kiszámítására, egy *hasonló, de egyszerűbb* (eggyel kevesebb szorzást feltételező) feladatra. Természetesen $(n-1)!$ is hasonló módon visszavezethető $(n-2)!$ -re és így tovább, míg eljutunk $0!$ -ig.

$$\begin{aligned} n! &= (n-1)! \cdot n = (n-2)! \cdot (n-1) \cdot n = \dots = \\ &= 0! \cdot 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = 1 \cdot 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n \end{aligned}$$

Rekurzió az informatikában

Tegyük fel, hogy két függvény, f_1 és f_2 célul tűzik ki, hogy kiszámítsák a paraméterként kapott n faktoriálisának értékét. f_1 a (13.1) képlet alapján lát hozzá a feladathoz, és a következőképpen oldja meg:

```
int f1 (int n)
{
    if (n == 0) return 1;
    else
    {
        int p=1;
        for(int i=1; i<=n; i++)
            p *= i;
        return p;
    }
}
```

Megfigyelhető, hogy f_1 felvállalja az egész feladatot, ami azt jelenti, hogy elvégzi az $n!$ kiszámításához szükséges valamennyi n szorzást. Az $n > 0$ esetben ezt egy **for** ciklussal oldja meg. Ezt a megközelítést – amikor bizonyos utasítások ismétlését ciklus segítségével valósítjuk meg – *iteratív* módszernek nevezzük.

f_2 viszont ennél sokkal kényelmesebb. A faktoriális rekurzív képletéből véve az ötletet, a következőképpen gondolkodik:

1. Ha $n == 0$, akkor én is készségesen megoldom a feladatot – elvégre ez csak annyit jelent, hogy bemondom a $0!$ értékét, az **1**-et.
2. Ha viszont $n > 0$, nem vállalom fel a teljes feladatot, túl fárasztó lenne nekem n szorzást elvégezni. A következőképpen fogok eljárni:
 - a) Átruházom „valaki”-re az első $n-1$ szorzás elvégzésének feladatát – ami valójában az $(n-1)!$ értékét jelenti –, és nyújtok neki egy „tálcát”, amelyre rátegye az eredményt.
 - b) A tálcán kapott eredményt még megszorozom n -nel, és máris büszkélkedhetem az $n!$ értékével.

Ez mind jó és szép, de ki lesz ez a „valaki”, és mi lesz a „tálca”?

És most jön a hideg zuhany f_2 úrfinak. Mivel ez a „valaki” az $(n-1)$! kiszámításánál hasonlóképpen fog eljárni, mint ő, ezért a „valaki” ő maga lesz.

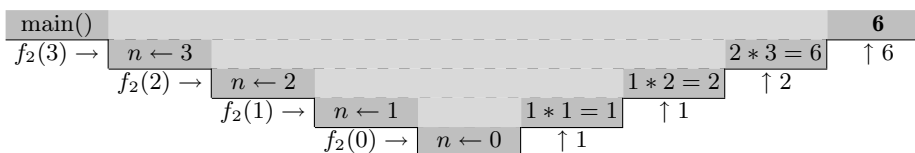
- Hogyhogy? Azt jelentse ez, hogy a feladatomban orozslánrészét átruházom magamra, és annak orozslánrészét megint csak magamra, és így tovább, míg eljutok a $0!$ -ig?
- Igen, pontosan erről van szó!
- De hát ez lehetetlen, hiszen ez azt feltételezné, hogy klónozzak n példányt magamból.
- Pedig, amint látni fogod, valami hasonlóról van szó.
- És a tálcá?
- Mivel „mindenik f_2 ” a *saját* tálcáját kell hogy nyújtsa a következőnek, ezért a tálcá szerepét egy – a függvény típusával azonos típusú – *lokális (saját)* változó fogja betölteni.

Íme az f_2 függvény:

```
int f2 (int n)
{
    int talca;
    if (n == 0) return 1;
    else
    {
        talca = f2(n-1);
        return talca*n;
    }
}
```

Ez tényleg elegáns, de hogyan működik?

Tegyük fel, hogy a $3!$ értékét szeretnénk kiíratni a képernyőre az f_2 függvény segítségével. Hogyan bonyolódik le az $f_2(3)$ függvényhívás? Az alábbi ábra – az úgynevezett lépcsőmódszert alkalmazva – grafikusán ábrázolja mindazt, ami egy rekurzív függvény hívásakor a háttérben történik. A „rekurzió lépésről lépésre” rész pedig mintegy kézen fogva vezet végig a „rekurzió útján”, és egyben magyarázattal is szolgál az ábra megértéséhez.



Az f_2 függvény „útja” az ábrán:

- Indul a „tetőről” (main függvény).
- Lemegy a „lépcsőkön” a „földszintre” (banális eset), hogy megtalálja a $0!$ értékét.
- Visszafele jövet minden szinten kiszámítja az „emeletnek” megfelelő faktoriális értéket (az alatta lévő szintről hozott értéket megszorozza az „emelet számával”).
- Visszaérkezve a tetőre, a kezében van az $n!$ értéke.

Következtetések az ábra segítségével:

1. Az $n!$ kiszámításához f_2 $n+1$ -szer lesz meghívva és végrehajtva.
2. A függvényhívások fordított sorrendben fejeződnek be, mint ahogy elkezdődtek.
3. Ha t -vel jelöljük a függvény utasításainak végrehajtásához szükséges időt $n > 0$ esetén, és t_0 -val $n=0$ esetén, akkor az $f_2(i)$ függvényhívás ($0 \leq i \leq n$) végrehajtása $i*t + t_0$ ideig tart, amiből $i > 0$ esetén $(i-1)*t + t_0$ időre fel van függesztve.
4. Mivel mindenik függvényhívásnak meg kell hogy legyen a saját n -je és $talca$ -ja, ezért n föltétlenül érték szerint átadott paraméter, $talca$ pedig lokális változó kell hogy legyen.

Nem nehéz átlátni, hogy a $talca$ változó használata nem föltétlenül szükséges, hiszen az `if` utasítás `else` ága egyszerűen így nézhetne ki:

```
else return f2(n-1)*n;
```

Mégis, azért vezettük be a $talca$ változót, hogy az általános eset ($n>0$) kezelésénél – az `else` ágon – világosan különválasszuk az *átruházott orosz-lánrész* megoldását, amely a rekurzív hívás által történik, a *saját résztől*, amikor is a tálcán kapott értékből felépítjük az eredeti feladat megoldását.

Befejezésül állítsuk szembe a két függvény stratégiáját. f_1 úgy építette fel a megoldást, hogy az *egyszerűtől haladt a bonyolult felé*. Vett egy p változót, amelybe kezdetben 1-et tett, a $0!$ értékét. Ezután pedig a $0!$ értékéből kiindulva a p változóban sorra előállította 1-től n -ig a természetes számok faktoriálisait: először $1!$ -t, azután $2!$ -t, és így tovább, míg eljutott $n!$ -ig.

f_2 pont fordítva látott hozzá a feladathoz: egyből nekiszökött az $n!$ kiszámításának. A rekurzió mechanizmusa által először *lebontotta a feladatot a bonyolulttól haladva az egyszerű felé*, majd pedig *felépítette a megoldást az egyszerűtől haladva a bonyolult felé*. ...és beigazolódik a közmondás, miszerint a rest kétszer fárad.

Akkor hát melyik a jobb stratégia, az f_1 -é vagy az f_2 -é, az iteratív vagy a rekurzív? Tény, hogy az iteratív módszer gyorsabb és kevésbé memóriaigényes. De hát akkor szól-e valami is a rekurzió mellett? Két-

ségtelenül! A rekurzív megközelítés egyszerű, elegáns és ezért könnyen programozható. Amint majd tapasztalni fogjuk, vannak feladatok, amelyeknek iteratív megoldása rendkívül bonyolult, rekurzívan viszont néhány soros programmal megoldhatók.

Ezekután, ha majd legközelebb belépsz valamely hivatal ajtaján, mit fogsz mondani? Azt, hogy kezdődik a kálvária, vagy azt, hogy kezdődik a rekurzió?

Rekurzió lépésről lépésre

Megszakad a main függvény végrehajtása, és válaszként az **f2(3)** hívásra elkezdődik az **f2** függvény végrehajtása:

1/1 megszületik az **f2** függvény **n** nevű formális paramétere a Stack-en;

– az **n** formális paraméter megkapja a függvény végrehajtását kiváltó hívás aktuális paraméterét, a 3-at;

1/2 megszületik a *talca* nevű lokális változó a Stack-en;

az utasításrész végrehajtása:

1/3 mivel $3 \neq 0$, az **if else** ágán folytatódik a függvény végrehajtása;

1/4 a **talca=f2(2)** utasítást megintcsak nem tudjuk végrehajtani az **f2(2)** érték hiányában. Ezért **felfüggesztődik az f2 függvény végrehajtása** is – ebben a pontban –, és az **f2(2)** hívásra válaszolva **elkezdődik az f2 függvény egy újbóli végrehajtása, amely a második:**

2/1 megszületik a második átjárás saját **n** nevű formális paramétere a Stack-en;

– ez az **n** megkapja a jelen végrehajtást kiváltó hívás aktuális paraméterének az értékét, a 2-t;

2/2 megszületik a második végrehajtás saját **talca** nevű változója;

a második átjárás utasításrészének végrehajtása:

2/3 mivel $2 \neq 0$, az **if else** ágán folytatódik a függvény második átjárása is;

2/4 a **talca=f2(1)** utasítást megintcsak nem tudjuk végrehajtani – ez alkalommal az **f2(1)** érték hiányában. Ezért **felfüggesztődik az f2 függvény második átjárása** is – ebben a

pontban –, és válaszként az **f2(1)** hívásra, **elkezdődik az f2 függvény egy további átjárása, immár a harmadik:**

3/1 megszületik a harmadik átjárás saját **n** nevű formális paramétere a Stack-en;

– ez az **n** megkapja a jelen végrehajtást kiváltó hívás aktuális paraméterének az értékét, az **1**-et;

3/2 megszületik a harmadik végrehajtás saját **talca** nevű változója;

a harmadik átjárás utasításrészének végrehajtása:

3/3 mivel **1!=0**, az **if else** ágán folytatódik a függvény harmadik átjárása is;

3/4 a **talca=f2(0)** utasítást most sem tudjuk végrehajtani – ez alkalommal az **f2(0)** érték hiányában. Ezért **felfüggesztődik az f2 függvény harmadik átjárása** is – ebben a pontban –, és válaszként az **f2(0)** hívásra, **elkezdődik az f2 függvény egy további átjárása, immár a negyedik:**

4/1 megszületik a negyedik átjárás saját **n** nevű formális paramétere a Stack-en;

– ez az **n** megkapja a jelen végrehajtást kiváltó hívás aktuális paraméterének az értékét, a **0**-t;

4/2 megszületik a negyedik végrehajtás saját **talca** nevű változója;

4/3 mivel ez esetben **0 == 0**, az **if then** ágán fog folytatódni a függvény ezen negyedik végrehajtása, a banális feladat megoldásával;

4/4 a függvény neve megkapja a visszatérítendő értéket, a banális feladat eredményét, az **1**-et. *Bingó, megvan a 0! értéke!*

Befejeződik a függvény negyedik átjárása:

eltűnik a Stack-ről a negyedik átjárás **talca** változója;

eltűnik a Stack-ről a negyedik átjárás **n** nevű formális paramétere;

3/5 **folytatódik az f2 függvény harmadik átjárása** abban a pontban, ahol annak idején felfüggesztődött. A **talca** nevű változóba bekerül a most már rendelkezésre álló **f2(0)** érték, ami nem más, mint a negyedik átjárás által visszatérített eredmény, a **0!** értéke, vagyis **1**;

3/6 a függvény neve megkapja a **talca*n** szorzat értéket. Mivel ezen harmadik átjárásnak az **n**-je 1, így az általa visszatérített érték $1*1=1$ lesz. *Ez nem más, mint az 1! értéke.*

Befejeződik a függvény harmadik átjárása:

eltűnik a Stack-ről a harmadik átjárás **talca** változója;
eltűnik a Stack-ről a harmadik átjárás **n** nevű formális paramétere;

2/5 **folytatódik az f2 függvény második átjárása** abban a pontban, ahol annak idején felfüggesztődött. A **talca** nevű változó megkapja a most már rendelkezésre álló **f2(1)** értéket, ami nem más, mint a harmadik átjárás által visszatérített eredmény, az 1! értéke, vagyis az 1;

2/6 a függvény neve megkapja a **talca*n** szorzat értéket. Mivel ezen második átjárásnak az **n**-je 2, így az általa visszatérített érték $1*2=2$ lesz. *Ez nem más, mint a 2! értéke.*

Befejeződik a függvény második átjárása:

eltűnik a Stack-ről a második átjárás **talca** változója;
eltűnik a Stack-ről a második átjárás **n** nevű formális paramétere;

1/5 **folytatódik az f2 függvény első átjárása** abban a pontban, ahol annak idején felfüggesztődött. A **talca** nevű változó megkapja a most már rendelkezésre álló **f2(2)** értéket, ami nem más, mint a második átjárás által visszatérített eredmény, a 2! értéke, vagyis a 2;

1/6 a függvény neve megkapja a **talca*n** szorzat értékét. Mivel ezen első átjárásnak az **n**-je 3, így az általa visszatérített érték $2*3=6$ lesz. *Ez nem más, mint a 3! értéke.*

Befejeződik a függvény első átjárása:

eltűnik a Stack-ről az első átjárás **talca** változója;
eltűnik a Stack-ről az első átjárás **n** nevű formális paramétere.

Folytatódik a main függvény abban a pontban, ahol annak idején felfüggesztődött, ugyanis most már rendelkezésre áll az **f2(3)** érték, ami nem más, mint a függvény első átjárása által visszatérített eredmény, a 3! értéke, vagyis 6.

Rekurzív függvények

Az előbbieken arról volt szó, hogy miként közelíti meg egy rekurzív függvény az $n!$ kiszámításának feladatát. Foglaljuk össze:

Ha a feladatot banálisnak (triviálisnak) találta ($n==0$), akkor felvállalta a teljes feladat megoldását. Ellenkező esetben viszont ($n>0$) két részre osztotta a feladatot: egy oroszlánrészre (az $(n-1)!$ értékét biztosító első $n-1$ szorzás), amit rekurzív hívás által átruházott, és egy saját részre (az n -edik szorzás), amit felvállalt.

Próbáljuk meg általánosítani a fenti megközelítési módot. A következő sablont sikeresen lehet alkalmazni:

```
<típus> f(<a feladat paraméterei>)
{
    <típus> talca;
    if (<banalitás feltétele>) <banális eset kezelése>
    else
    {
        talca = f(<átruházott rész paraméterei>);
        <saját rész>
    }
}
```

Egyszerű függvények esetén igen jól használható a fenti sablon. Gyakran egyebet sem kell tenni, csak kitölteni a sablont, és máris megvan a feladatot megoldó rekurzív függvény. Segítséget adhat ebben, ha feltesszük a következő három rávezető kérdést:

1. kérdés

Hogyan vezethető vissza a feladat egy *hasonlóképpen megoldható*, de *egyszerűbb* feladatra? Az erre a kérdésre adott válasz világosan el fogja határolni a rekurzívan *átruházandó oroszlánrészt* a *felvállalt saját résztől*. Továbbá nyilvánvalóvá fogja tenni mind a fő feladat, mind az átruházott feladat paramétereit.

2. kérdés

Miután tálcán megkapjuk az átruházott rész eredményét, *hogyan építhető fel* ebből a teljes feladat eredménye a felvállalt saját rész megoldása által?

3. kérdés

Mikor tekintjük a feladatot annyira *banálisnak*, hogy teljesen felvállaljuk a megoldását anélkül, hogy valamit is rekurzívan átruháznánk belőle?

Ez a megközelítés nagyon eredményes, például olyan feladatok esetében, amelyek *egy szám számjegyenkénti* vagy *egy számsorozat elemeinkénti* feldolgozását követelik meg.

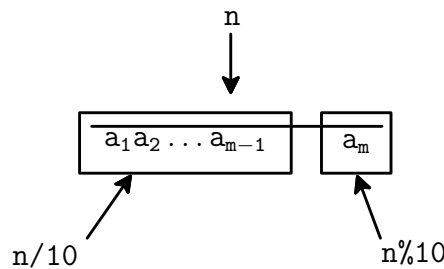
Például, legyen a következő feladat:

1. feladat. Írjunk rekurzív függvényt az n természetes szám számjegyei összegének kiszámítására.

Kezdjük azzal, hogy megválaszoljuk a három kulcskérdést:

1. *válasz*

Az n számjegyei összegének kiszámítása visszavezethető az $n / 10$ (az n szám, az utolsó számjegye nélkül) számjegyei összegének kiszámítására. Ez egy *hasznos* feladat, hiszen ugyancsak egy természetes szám számjegyei összegének kiszámítását jelenti, de *egyszerűbb*, mert $n / 10$ eggyel kevesebb számjegyű szám, mint n . Tehát a *rekurzívan átruházandó orozzlánrész* az $n / 10$ számjegyei összegének kiszámítása lesz, a *saját rész* pedig az utolsó számjegynek ($n \% 10$) a kezelése. Az alábbi ábra ezt szemlélteti (n egy m számjegyű természetes szám):



2. *válasz*

Miután a **talca** változóban megkapjuk $n / 10$ számjegyeinek az összegét, egyszerűen annyit kell még tennünk, hogy hozzáadjuk n utolsó számjegyét, az $n \% 10$ értékét.

3. *válasz*

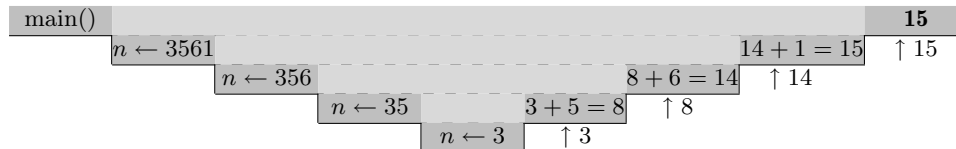
A feladat banálisnak tekinthető már akkor is, ha n egyszámjegyű ($n < 10$), vagy akkor, amikor „elfogyott” ($n == 0$). Az első esetben a megoldás maga a szám lesz, a második esetben pedig 0.

És most következzen a feladatot megoldó rekurzív függvény:

```
int szamjegy_osszeg (int n)
{
    int talca;
    if (n < 10) return n;
    else
```

```
{   talca = szamjegy_osszeg (n/10);
    return talca+n%10;
}
```

Az alábbi ábra nyomon követi a `szamjegy_osszeg(3561)` függvényhívás végrehajtását:



A sablon azt ajánlja, hogy először a feladat orozslánrészét oldjuk meg a rekurzív hívás által, és azután a felvállalt saját részt. Azonban vannak olyan esetek – ilyen a következő feladat is –, amikor célszerűbb először megvizsgálni a saját részt. Miért? Azért, mert bizonyos esetekben a saját rész vizsgálata megoldhatja a teljes feladatot, szükségtelenné téve ezáltal az orozslánrész megoldását. Íme egy példa:

2. feladat. Írjunk rekurzív függvényt, amely egy logikai értéket térít vissza attól függően, hogy egy n elemű számsorozat, amelynek elemei egy a tömbben vannak eltárolva,

- tartalmaz-e nullát?
- szimmetrikus-e?

Kezdjük újra azzal, hogy megválaszoljuk a három kulcskérdést először az a) pontra, majd pedig a b) pontra vonatkozóan. Feltételezzük, hogy a számsorozat az a tömb $a[1]$, $a[2]$, \dots , $a[n]$ (rövidítve $a[1..n]$) elemeiben van eltárolva.

1. válasz

- Az $a[1..n]$ tömb vizsgálata (tartalmaz-e nullát) visszavezethető az $a[1..n-1]$ résztömb vizsgálatára, majd pedig az $a[1..n-2]$ résztömbére, és így tovább, míg a feladat banálissá nem zsugorodik.

$a[1..n] \gg a[1..n-1] \gg a[1..n-2] \gg \dots$

általánosan: $a[1..j] \gg a[1..j-1]$.

- Egy másik lehetőség, ha a számsorozat elejéről hagyjuk el sorra az elemeket:

$a[1..n] \gg a[2..n] \gg a[3..n] \gg \dots$

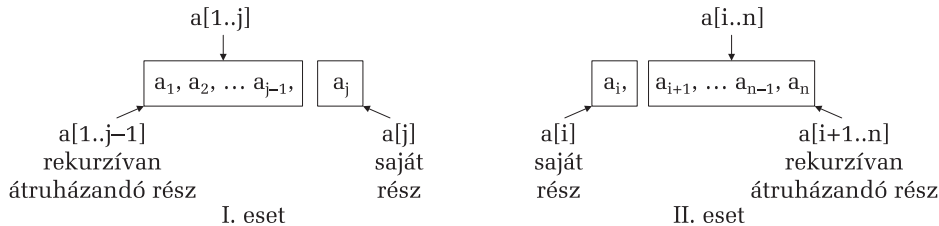
általánosan: $a[i..n] \gg a[i+1..n]$.

- Egyes feladatok természete megköveteli, hogy ötvözzük az előbbi két módszert (ilyen például a feladat b) pontja is).

$a[1..n] \gg a[2..n-1] \gg a[3..n-2] \gg \dots$

általánosan: $a[i..j] \gg a[i+1..j-1]$.

Az alábbi ábrák az első két módszert mutatják be, hiszen ezek ajánlják magukat a feladat a) pontja esetén:



2. válasz

Miután a **talca** változóban megkapjuk vagy az $a[1..j-1]$, vagy pedig az $a[i+1..n]$ szakasz vizsgálatának eredményét, így gondolkodhatunk: az $a[1..j]$ vagy az $a[i..n]$ tömbszakaszban akkor található nulla, ha vagy a **talca** értéke logikai igaz, vagy a megmaradt elem (első esetben az $a[j]$, másodikban az $a[i]$) nulla. Bár a megadott sablon alapján ez a kézenfekvő gondolkodásmód, mégis figyeljük meg a következőt:

Ha a saját részt képviselő elem nulla, akkor nyilvánvaló, hogy van a számsorozatban nulla, és nem szükséges az orozslánrész vizsgálata. Ha viszont a saját részt képviselő elem nem nulla, akkor szükséges az orozslánrész vizsgálata is, sőt ez esetben teljesen tőle fog függeni a feladat eredménye.

3. válasz

A feladat akkor tekinthető banálisnak, ha az $a[1..j]$ vagy az $a[i..n]$ tömbszakasz vagy egy elemet tartalmaz ($j == 1$ vagy $i == n$), vagy egyet sem ($j == 0$ vagy $i > n$). A második változat jelen esetben előnyösebb, mert az eredmény magától értetődően logikai hamis (egy nem létező számsorozatban nincs nulla). Az első változat esetében szükséges azon egyetlen elem megvizsgálása, hogy nulla-e vagy sem.

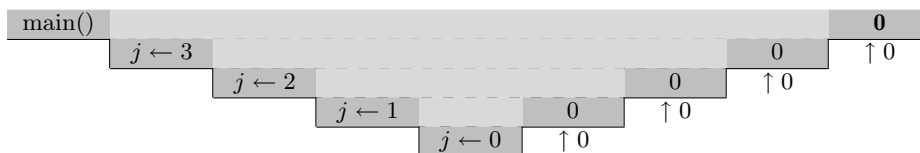
Íme a feladatot megoldó rekurzív függvény két változatban.

```
int van_nulla (int i)
{
  /* ellenőrzi, hogy az a[1..j]
     tömbszakasz tartalmaz-e
     nullát */
  /* int talca; */
  if (j == 0) van_nulla=0;
  else
  {
    if(a[j]==0) van_nulla=1;
    else
      van_nulla=van_nulla(j-1);
  }
}
```

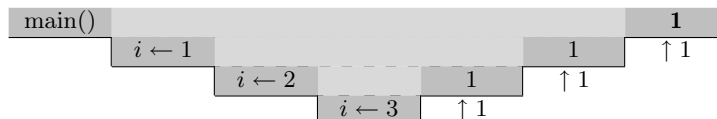
```
int van_nulla (int i)
{
  /* ellenőrzi, hogy az a[i..n]
     tömbszakasz tartalmaz-e
     nullát */
  /* int talca; */
  if (i==n) if (a[i]==0) return 1;
            else return 0;
  else
  {
    if (a[i] == 0) return 1;
    else
      return van_nulla (i+1);
  }
}
```

Az alábbi ábra nyomon követi a `van_nulla(1,n)` függvényhívás végrehajtását.

1. eset: $n=3$, $a[1..3]=\{7, -3, 15\}$

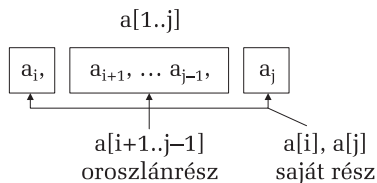


2. eset: $n=4$, $a[1..4]=\{7, -3, 0, 15\}$



Figyeljük meg, hogy a második példában a lépcső egy fokkal alacsonyabb. Miért van ez? Mert felcseréltük az oroslánrész és a saját rész megoldásának sorrendjét. Miért ez a válasz?

A feladat b) pontjához csak az ábrát fogjuk megadni és a kész függvényeket, az olvasóra bízva, hogy tanulmányozza és megértse őket.



```
int szimmetrikus (int i, int j)
{
    /* ellenőrzi, hogy az a[i..j]
       tömbszakasz szimmetrikus-e */
    /* int talca; */
    if (i >= j) szimmetrikus=0
    else
    {
        if (a[i]!=a[j]) return 0;
        else
            return szimmetrikus(i+1,j-1);
    }
}
```

```
int szimmetrikus (int i, int j)
{
    /* ellenőrzi, hogy az a[i..j]
       tömbszakasz szimmetrikus-e */
    /* int talca; */
    if (i >= j) return 1;
    else
    {
        if (a[i] != a[j]) return 0;
        else
            return szimmetrikus (i+1,j-1);
    }
}
```

Rekurzív eljárások

A függvények és az eljárások (void függvények) között az alapvető különbség az, hogy a függvény kiszámít valamit, és ezt visszatéríti mint eredményt, az eljárás pedig elvégez valamit. A rekurzív eljárásoknak is fő jellegzetessége, hogy meghívják önmagukat. Természetesen itt is érvényes az a megkötés, hogy a rekurzív hívásnak feltételhez kötöttnek kell lennie, hogy biztosítva legyen a rekurzív hívások láncolatából való kiszabadulás. Ebből adódóan a rekurzív eljárásokban is általában van egy úgynevezett kulcs if utasítás, amelynek egyik ága rekurzív (itt kerül sor a rekurzív hívásra), a másik pedig nem.

Ha úgy fogjuk fel a rekurzív eljárást, mint amely ezen „kulcs” if köré épül, akkor a következő vázat kapjuk:

```
void rek_elj (<paraméter_lista>)
{
    <lokális definiálások>
    a zóna
    if <feltétel>
    {
        b zóna
        rek_elj (<paraméter_lista>);
        B zóna
    }
```

```

else
  X zóna
  A zóna
}
    
```

Bár a fenti sablon a rekurzív eljárásoknak egy nagyon leegyszerűsített vázlat, mégis sokat segíthet a tanulmányozásukban.

Amint megfigyelhető, a kulcs `if` rögzítése öt területet határolt el a rekurzív eljárásban:

- a – a kulcs `if` előtti terület,
- A – a kulcs `if` utáni terület,
- b – a kulcs `if` rekurzív ágán a rekurzív hívás előtti terület,
- B – a kulcs `if` rekurzív ágán a rekurzív hívás utáni terület,
- X – a kulcs `if` nem rekurzív ága.

A rekurzív eljárás bármely utasítása csakis a fenti területek valamelyikére kerülhet.

Ha gondolatban lefuttatjuk a `rek_elj` eljárást, akkor könnyen nyomon követhetjük, hogy az egyes területek utasításai mikor, hányszor és milyen sorrendben hajtódnak végre. Mindez abban segíthet, hogy világosan átlásuk, milyen hatása van annak, ha egy utasítást egy bizonyos területre írunk. Tegyük fel, hogy futtatása során a `rek_elj` eljárás – a rekurzió következményeként – n -szer fog meghívódni, ennyiszor lesz átjárva.

A következő ábra bemutatja, hogy a különböző zónák hányszor, milyen sorrendben, valamint mely átjárások során kerülnek végrehajtásra.

$\begin{array}{ccccccc} \mathbf{a_1 f_1 b_1} & \mathbf{a_2 f_2 b_2} & \dots & \mathbf{a_{n-1} f_{n-1} b_{n-1}} & \mathbf{a_n f_n X_n A_n} & \mathbf{B_{n-1} A_{n-1}} & \dots & \mathbf{B_2 A_2} & \mathbf{B_1 A_1} \\ \mathbf{I} & \mathbf{I} & & \mathbf{I} & \mathbf{H} & & & & \end{array}$

ahol:

- f = feltétel,
- I = azt jelzi, hogy a feltétel értéke logikai IGAZ,
- H = azt jelzi, hogy a feltétel értéke logikai HAMIS.

Az indexek jelzik, hogy a rekurzív eljárás hányadik átjárásáról van szó, például B_2 jelentése: sor kerül – a második átjárásban – a B zóna utasításainak végrehajtására.

A fenti ábra fontos következtetésekhez vezethet el. Milyen következménye lesz, ha egy utasítás egy bizonyos területre kerül?

a terület: a *hívások sorrendjében* hajtódik végre, *annyiszor*, *ahány átjárásra* kerül sor.

A terület: a hívások *fordított* sorrendjében hajtódik végre, *annyiszor*, *ahány átjárásra* kerül sor.

b terület: a *hívások sorrendjében* hajtódik végre, de *eggyel kevesebb-szer*, mint az *a terület*, hiszen az utolsó átjárásban nem kerül sor a végrehajtására.

B terület: a hívások *fordított* sorrendjében hajtódik végre, de *eggyel kevesebb-szer*, mint az *A terület*, hiszen az utolsó átjárásban nem kerül sor a végrehajtására.

X terület: csak az utolsó átjárásban hajtódik végre.

A következő részben bemutatjuk egy konkrét feladaton, miként használható ez a megközelítés rekurzív eljárások írására!

Például, szolgáljon a fenti váz használatára a következő feladat:

3. feladat. Írjunk rekurzív eljárást, amely karaktereket olvas be vakon, addig, amíg '*' karakter olvasott, a képernyőre pedig a beolvasás fordított sorrendjében írja ki őket:

Például, ha a bemenet: ABCD*

Kimenet:

- a) *DCBA
- b) DCBA
- c) ABCDDCBA
- d) ABCD**DCBA
- e) ABCD*DCBA

Íme a megoldás:

```
void eljA()
{
    char c; /* mindenik átjárásnak meglesz a saját c-je */
    c=getch(); /* beolvasás vakon az a zónában */
    if (c!='*') eljA;
    putchar(c); /* kiírás az A zónában */
}
```

```
void eljB()
{
    char c; /* mindenik átjárásnak meglesz a saját c-je */
    c=getch(); /* beolvasás vakon az a zónában */
    if (c!='*')
    {
        eljB;
        putchar(c); /* kiírás a B zónában */
    }
}
```

```
void eljC()
{
    char c; /* mindenik átjárásnak meglesz a saját c-je */
    c=getch(); /* beolvasás vakon az a zónában */
    if (c!='*')
    {
        putchar(c); /* kiírás a b zónában */
        eljC;
        putchar(c); /* kiírás a B zónában */
    }
}
```

```
void eljD()
{
    char c; /* mindenik átjárásnak meglesz a saját c-je */
    c=getch(); /* beolvasás vakon az a zónában */
    putchar(c); /* kiírás az a zónában */
    if (c!='*') eljD;
    putchar(c); /* kiírás az A zónában */
}
```

```
void eljE()
{
    char c; /* mindenik átjárásnak meglesz a saját c-je */
    c=getch(); /* beolvasás vakon az a zónában */
    if (c!='*')
    {
        putchar(c); /* kiírás a b zónában */
        eljE;
        putchar(c); /* kiírás a B zónában */
    }
    else putchar(c); /* kiírás az X zónában */
}
```


B. FÜGGELÉK

AZ ADATOK BELSŐ ÁBRÁZOLÁSA A SZÁMÍTÓGÉP MEMÓRIÁJÁBAN

1. Átalakítások számrendszerek között

A belső ábrázolási módszerek megértéséhez elengedhetetlenül szükséges a 2 (bináris), 8 (oktál), 10 (decimális) és 16-os (hexadecimális) számrendszerek közötti átalakítási műveletek ismerete.

1.1. Számjegyek 2-es, 8-as, 10-es és 16-os számrendszerekben:

Bináris: 0, 1

Oktál: 0, 1, 2, 3, 4, 5, 6, 7

Decimális: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Hexadecimális: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A(10), B(11), C(12), D(13), E(14), F(15)

1.2. Átalakítás 10-es számrendszerből p alapú számrendszerbe

1.2.1. Egész számok átalakítás 10-es számrendszerből p alapú számrendszerbe: Osszuk a számot p -vel amíg nullává válik, majd összeszedjük a maradékokat az osztások fordított sorrendjében.

$$(23)_{10} = (?)_2$$

$$\underline{23} : 2 = \underline{11} : 2 = \underline{5} : 2 = \underline{2} : 2 = \underline{1} : 2 = 0$$

$$\begin{array}{cccccc} 1 & 1 & 1 & 0 & 1 \\ \leftarrow \end{array}$$

$$\text{Tehát: } (23)_{10} = (10111)_2$$

$$(1978)_{10} = (?)_8$$

$$\underline{\mathbf{1978}} : 8 = \underline{\mathbf{247}} : 8 = \underline{\mathbf{30}} : 8 = \underline{\mathbf{3}} : 8 = \underline{\mathbf{0}}$$

$$\underline{\mathbf{2}} \quad \quad \quad \underline{\mathbf{7}} \quad \quad \quad \underline{\mathbf{6}} \quad \quad \quad \underline{\mathbf{3}}$$

$$\text{Tehát: } (1978)_{10} = (3672)_8$$

$$(933)_{10} = (?)_{16}$$

$$\underline{\mathbf{933}} : 16 = \underline{\mathbf{58}} : 16 = \underline{\mathbf{3}} : 16 = \mathbf{0}$$

$$\underline{\mathbf{5}} \quad \quad \quad \underline{\mathbf{10}} \quad \quad \quad \underline{\mathbf{3}}$$

$$\text{Tehát: } (933)_{10} = (3A5)_{16}$$

Tört számok átalakítása 10-es számrendszerből p alapú számrendszerbe:

1.2.2. Szorozzuk a tizedes rész p -vel, amíg ez nullává válik, majd összeszedjük a szorzatok egész részeit a szorzásokkal megegyező sorrendben.

$$(0.75)_{10} = (?)_2$$

$$\begin{aligned} \mathbf{0.75} \times 2 &= \mathbf{1.5} \\ \mathbf{0.5} \times 2 &= \mathbf{1.0} \end{aligned} \quad \downarrow$$

$$\text{Tehát: } (0.75)_{10} = (0.11)_2$$

$$(0.625)_{10} = (?)_8$$

$$\begin{aligned} \mathbf{0.625} \times 8 &= \mathbf{0.5} \\ \mathbf{0.5} \times 8 &= \mathbf{4.0} \end{aligned} \quad \downarrow$$

$$\text{Tehát: } (0.625)_{10} = (0.04)_8$$

$$(0.25)_{10} = (?)_{16}$$

$$\mathbf{0.25} \times 16 = \mathbf{4.0} \quad \downarrow$$

$$\text{Tehát: } (0.25)_{10} = (0.4)_{16}$$

Összefoglalva:

$$\begin{aligned} (23.75)_{10} &= (10111.11)_2 \\ (1978.0625)_{10} &= (3672.04)_8 \\ (933.25)_{10} &= (3A5.4)_{16} \end{aligned}$$

1.3. Átalakítás p alapú számrendszerből 10-es számrendszerbe

Az alábbi példákból könnyen kihámozható az átalakítási algoritmus:

$$(101011.101)_2 = (?)_{10}$$

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 43.625$$

$$(101011.101)_2 = (43.625)_{10}$$

$$(734.4)_8 = (?)_{10}$$

$$7 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 + 4 \times 8^{-1} = 476.5$$

$$(734.4)_8 = (476.5)_{10}$$

$$(3AF.C)_{16} = (?)_{10}$$

$$3 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 + 12 \times 16^{-1} = 943.75$$

$$(3AF.C)_{16} = (943.75)_{10}$$

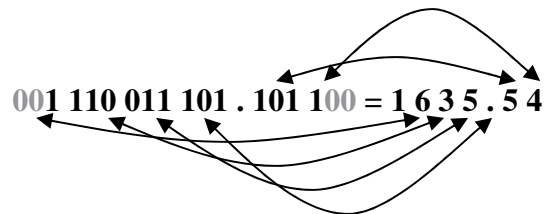
1.4. Átalakítás 2-es számrendszerből 8-as, illetve 16-os számrendszerekbe

A oktál és hexadecimális számjegyek bináris alakja három, illetve négy pozíción:

| | 2^2 | 2^1 | 2^0 | | 2^3 | 2^2 | 2^1 | 2^0 |
|----|-------|-------|-------|----|-------|-------|-------|-------|
| 0– | 0 | 0 | 0 | 0– | 0 | 0 | 0 | 0 |
| 1– | 0 | 0 | 1 | 1– | 0 | 0 | 0 | 1 |
| 2– | 0 | 1 | 0 | 2– | 0 | 0 | 1 | 0 |
| 3– | 0 | 1 | 1 | 3– | 0 | 0 | 1 | 1 |
| 4– | 1 | 0 | 0 | 4– | 0 | 1 | 0 | 0 |
| 5– | 1 | 0 | 1 | 5– | 0 | 1 | 0 | 1 |
| 6– | 1 | 1 | 0 | 6– | 0 | 1 | 1 | 0 |
| 7– | 1 | 1 | 1 | 7– | 0 | 1 | 1 | 1 |
| | | | | 8– | 1 | 0 | 0 | 0 |
| | | | | 9– | 1 | 0 | 0 | 1 |
| | | | | A– | 1 | 0 | 1 | 0 |
| | | | | B– | 1 | 0 | 1 | 1 |
| | | | | C– | 1 | 1 | 0 | 0 |
| | | | | D– | 1 | 1 | 0 | 1 |
| | | | | E– | 1 | 1 | 1 | 0 |
| | | | | F– | 1 | 1 | 1 | 1 |

Csoportosítjuk a bináris számjegyeket a tizedes ponttól jobbról és balra hármanként/négyenként, és minden csoportot helyettesítünk a megfelelő 8-as, illetve 16-os számrendszerbeli számjeggyel.

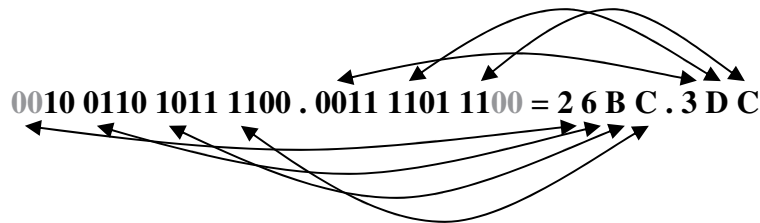
$$(1110011101.1011)_2 = (?)_8$$



14.1. ábra.

$$(1110011101.1011)_2 = (1635.54)_8$$

$$(10011010111100.0011110111)_2 = (?)_{16}$$



14.2. ábra.

$$(10011010111100.0011110111)_2 = (3DC)_{16}$$

1.5. Átalakítás 8-as, illetve 16-os számrendszerekből 2-es számrendszerbe

Minden 8-as, illetve 16-os számrendszerbeli számjegyet helyettesítünk a megfelelő három/négy elemű bináris alakkal. Szemléltetésként tekintsük a fenti példákat jobbról balra olvasva.

1.6. Aritmetikai műveletek 2-es, 8-as és 16-os számrendszerekben

Az alapelv ugyanaz, mint 10-es számrendszerben. Emlékezzünk rá, hogy ha a_p és b_p p alapú számrendszerbeli számjegyek, akkor a c_p eredmény-számjegy

összeadás esetén:

AZ ADATOK BELSŐ ÁBRÁZOLÁSA A SZÁMÍTÓGÉP MEMÓRIÁJÁBAN 265

$c_p = (a_p + b_p) \text{MOD} p$, ahol a MOD operátor osztási maradékot jelent
 átvitel = $(ap + bp) \text{DIV} p$, ahol a DIV operátor osztási hányados
 jelent

szorzás estén:

$c_p = a_p * b_p = (a_p * b_p) \text{MOD} p$, ahol a MOD operátor osztási
 maradékot jelent

átvitel = $(a_p * b_p) \text{DIV} p$, ahol a DIV operátor osztási hányados jelent

kivonás esetén:

$c_p = a_p - b_p$, ha $a_p \geq b_p$

$c_p = (p + a_p) - b_p$, ha $a_p < b_p$ (kölsönkéntünk a kisebbítendő
 aktuális számjegyének bal oldali szomszédjától 1-et; ez az aktuális
 számjegyre nézve p -t jelent)

Példák bináris összeadásra, kivonásra és szorzásra:

| | | |
|-------------------------|------------------|------------------------|
| $0 + 0 = 0$ | $0 \times 0 = 0$ | $0 - 0 = 0$ |
| $0 + 1 = 1$ | $0 \times 1 = 0$ | $0 - 1 = 1$ (kölsön 2) |
| $1 + 0 = 1$ | $1 \times 0 = 0$ | $1 - 0 = 1$ |
| $1 + 1 = 0$ (átvitel 1) | $1 \times 1 = 1$ | $1 - 1 = 0$ |

Összeadás:

| | | |
|----------------|---------------------------|---|
| átvitel | 1 1 1 1 1 0 0 0 0 . 0 1 0 | |
| 1. összeadandó | 1 1 1 0 1 1 0 1 . 1 0 1 | + |
| 2. összeadandó | 1 0 1 1 0 1 0 . 0 0 1 | |
| összeg | 1 0 1 0 0 0 1 1 1 . 1 1 0 | |

Kivonás (*-val jelöltük azokat a pozíciókat ahol kölsönre volt szükség):

| | | |
|--------------|-----------------------|---|
| kölsön | * * * * * | |
| kisebbítendő | 1 0 0 0 1 0 1 . 1 1 0 | + |
| kivonandó | 1 1 1 0 1 0 . 0 1 1 | |
| különbség | 1 0 1 1 . 0 1 1 | |

Szorzás:

| | | | |
|-----------|-------------|---------|-----------|
| | 1 1 0 . 1 1 | × | 1 0 . 1 1 |
| 1 | 1 0 1 1 | | |
| 1 1 | 0 1 1 | | |
| 0 0 0 | 0 0 | | |
| 1 1 0 1 | 1 | | |
| 1 0 0 1 0 | . | 1 0 0 1 | |

Előjel nélküli egészek (unsigned) fix pontos belső ábrázolása N biten

Az alábbi táblázat összefoglalja, hogy az előjel nélküli egész típusok esetében mennyi az N értéke, illetve milyen tartományból tárolhatók értékek az illető típusú változókbán:

| Típus | N | Értéktartomány |
|--------------------|-----|----------------------|
| unsigned char | 8 | $0 \dots 2^8 - 1$ |
| unsigned short int | 16 | $0 \dots 2^{16} - 1$ |
| unsigned long int | 32 | $0 \dots 2^{32} - 1$ |

Ábrázolási algoritmus:

(Legyen x az ábrázolandó természetes szám; $0 \leq x \leq 2^N - 1$)

1. Átalakítjuk x -et kettes számrendszerbe.
2. Kipótoljuk x bináris alakját bal felől nullás bitekkel. (ha szükséges)

Példák:

- a.) A 19 belső ábrázolása 8 biten (unsigned char típusú változóban)
 1. lépés: $(19)_{10} = (10011)_2$
 2. lépés: **00010011**
- b.) A 0 belső ábrázolása 8 biten (unsigned char típusú változóban)
 1. lépés: $(0)_{10} = (0)_2$
 2. lépés: **00000000**
- c.) A 255 belső ábrázolása 8 biten (unsigned char típusú változóban)
 1. lépés: $(255)_{10} = (11111111)_2$
 2. lépés: **11111111**

2. Előjeles egészek (signed) fix pontos belső ábrázolása N biten

Az alábbi táblázat összefoglalja, hogy az előjeles egész típusok esetében mennyi az N értéke, illetve milyen tartományból tárolhatók értékek az illető típusú változókbán:

| Típus | N | Értéktartomány |
|------------------|-----|----------------------------|
| signed char | 8 | $-2^7 \dots 2^7 - 1$ |
| signed short int | 16 | $-2^{15} \dots 2^{15} - 1$ |
| signed long int | 32 | $-2^{31} \dots 2^{31} - 1$ |

Ábrázolási algoritmus:

(Legyen x az ábrázolandó előjeles egész szám; $2^{N-1} \leq x \leq 2^{N-1} - 1$)

Ha $x \geq 0$, akkor

AZ ADATOK BELSŐ ÁBRÁZOLÁSA A SZÁMÍTÓGÉP MEMÓRIÁJÁBAN 267

1. ábrázoljuk x -et N biten, mint előjel nélküli egészset.

Ha $x < 0$, akkor

1. ábrázoljuk $|x|$ -et N biten, mint előjel nélküli egészset.

2. képezzük a kapott N bites alaknak a 2-es komplementjét

a. minden 0 bitet 1-esre, és minden 1-es bitet 0-ra cserélünk. (1-es komplement)

b. a kapott értékhez hozzáadunk 1-et.

Megjegyzés: Előjeles egészek belső ábrázolásakor a legnagyobb helyértékű bit előjel bitként viselkedik. (1 - negatív szám; 0 - pozitív szám)

Példák:

a.) A 19 belső ábrázolása 16 biten (mint előjeles egésznek short int típusú változóban)

1. lépés: **0000000000010011**

b.) A (-19) belső ábrázolása 16 biten (mint előjeles egésznek short int típusú változóban)

1. lépés: a 19 belső ábrázolása 16 biten: 0000000000010011

2. lépés: Képezzük a fenti alak 2-es komplementjét

a. **111111111101100**

b. **111111111101101**

c.) A (-1) belső ábrázolása 16 biten (mint előjeles egésznek short int típusú változóban)

1. lépés: az 1-es belső ábrázolása 16 biten: 0000000000000001

2. lépés: Képezzük a fenti alak 2-es komplementjét

c. **111111111111110**

d. **111111111111111**

d.) A 0 belső ábrázolása 16 biten (mint előjeles egésznek short int típusú változóban)

1. lépés: **0000000000000000**

e.) A $2^{15}-1$ belső ábrázolása 16 biten (mint előjeles egésznek short int típusú változóban)

1. lépés: **0111111111111111**

f.) A (-2^{15}) belső ábrázolása 16 biten (mint előjeles egésznek short int típusú változóban)

1. lépés: a 2^{15} belső ábrázolása 16 biten: 1000000000000000

2. lépés: Képezzük a fenti alak 2-es komplementjét

a. **0111111111111111**

b. **1000000000000000**

g.) Melyik előjeles egésznek a short int típusként való belső ábrázolása a következő 16 bitből álló sorozat: 111111111111101?

- Mivel a legnagyobb helyértékű bit 1-es, ezért biztosan negatív számról van szó.
- Tekintettel arra, hogy $-(-x) = x$, ha képezzük az adott 16 bit 2-es komplementjét, megkapjuk a keresett szám abszolút értékét.
 - Minden 0-ás bitet 1-re, és minden 1-es bitet 0-ra cserélve (1-es komplement), majd a kapott értékhez hozzáadva 1-et, a következő alakhoz jutunk: 0000000000000011.
- Mivel a fenti bitsorozat a 3-as számnak (mint pozitív egésznek) a "short int"-kénti belső ábrázolása, ezért a keresett szám: $x = -3$.

4. Valós számok lebegő pontos belső ábrázolása

Az alábbi táblázat összefoglalja, hogy a valós típusok esetében hány biten történik a belső ábrázolás:

| Típus | N | Pontosság |
|----------------------------------|-----|---------------|
| float (egyszerű pontosság) | 32 | 6-7 tizedes |
| double (dupla pontosság) | 64 | 15-16 tizedes |
| long double (bővített pontosság) | 80 | 19 tizedes |

Ábrázolási algoritmus egyszeri és dupla pontosság estén:

(Legyen v az ábrázolandó valós szám)

1. lépés: Az x számot átírjuk 2-es számrendszerbe.
2. lépés: Normalizáljuk a 2-es számrendszerbeli alakot:

$$v = +/ - 1 \cdot f \times 2^{+/-e}.$$

(eltoljuk a tizedes pontot a 2-es a számrendszerbeli alakban oly módon, hogy az egész rész 1 legyen; innen jön a "lebegő" pontos ábrázolás" elnevezés)

s - a szám előjele (+/-)

f - tizedes rész

$+/- e$ - exponens

$+/- 1 \cdot f$ - mantissza; megfigyelhető, hogy $1 \leq |1 \cdot f| < 2$ (normalizálási egyenlőtlenség)

3. lépés: Tároljuk $N_s = 1$ biten a v szám előjelét (1 - negatív, 0 - pozitív), N_e biten az exponenst és N_f biten a tizedesrészt ($N = N_s + N_e + N_f$). Mivel minden valós szám normalizált alakjában az egész rész 1, ezért ezt szükségtelen eltárolni. Figyeljük meg, hogy az exponens elvileg a $\{-(2^{N_e-1} - 1), \dots, 2^{N_e-1}\}$ halmazból vehet fel értékeket (az exponens lehet úgy pozitív, mint negatív érték). Azért, hogy ne kelljen előjeles egészeket ábrázolni, bevezették a karakterisztika (k) fogalmát: $k = (2^{N_e-1} - 1) + e$. Észrevehető,

AZ ADATOK BELSŐ ÁBRÁZOLÁSA A SZÁMÍTÓGÉP MEMÓRIÁJÁBAN 269

hogy a karakterisztika értéke pontosan úgy lett megválasztva, hogy miközben e értéke $-(2^{N_e-1} - 1)$ és 2^{N_e-1} között mozog, k megfelelő értékei a $\{0, \dots, 2^N - 1\}$ halmazból vegyenek értékeket. Tehát az exponens helyett a valóságban a karakterisztika kerül eltárolásra N_k (a karakterisztikának fenntartott terület mérete) biten. Az N_k érték nyilván azonos N_e -vel.

A tizedes részt az N_f széles bitterületen balra igazítottan tároljuk kipótólva jobbról nullás bitekkel (ha szükséges).

Az IEEE standard szerint:

Egyszerű pontosságnál (float típus): $N_s = 1, N_k = 8, N_f = 23; k = 127 + e$.

| | | | | |
|----|---------------------|----|----|------------------|
| 31 | 30 | 23 | 22 | 0 |
| s | Karakterisztika (k) | | | Tizedes rész (f) |

Dupla pontosságnál (double típus): $N_s = 1, N_k = 11, N_f = 52; k = 1023 + e$.

| | | | | |
|----|---------------------|----|----|------------------|
| 63 | 62 | 52 | 51 | 0 |
| s | Karakterisztika (k) | | | Tizedes rész (f) |

Példák: a.) A -124.0625 belső ábrázolása dupla pontossággal (double típusú változóban)

1,2. lépések: $(124.0625)_{10} = (1111100.0001)_2 = 1.1111000001 \times 2^6$

$s = 1$ (negatív szám)

$f = 1111000001$

$e = 6$

$k = 6 + 1023 = 1029 = (10000000101)_2$

3. lépés:

| | | | | |
|----|-------------|----|----|---------------------------------------|
| 63 | 62 | 52 | 51 | 0 |
| 1 | 10000000101 | | | 11110000010000000000000000000000...00 |

b.) Fordítva: Melyik valós szám egyszerű pontosság szerinti belső ábrázolása az alábbi 32 bitből álló sorozat?

11000000110011110000111100100001

Emlékezzünk, hogy egyszerű pontosság esetén: $N_s = 1, N_k = 8, N_f = 23; e = k - 127$.

Tehát a 32 bit a következőképpen csoportosítható: az s előjelbit félkörvér, a karakterisztika szürke (8 bit) és a fennmaradt 23 bit a tizedes rész.

11000000110011110000111100100001 Mivel az előjel bit 1-es, ezért a szám negatív;

$k = (10000001)_2 = (129)_{10}; e = k - 127 = 2; |$ Tehát a keresett valós szám: $v = -1.10011110000111100100001 \times 2^2$;

Azaz: $v = -110.011110000111100100001$;

Átalakítva a fenti értéket 10-es számrendszerbe az alábbi eredményhez jutunk:

$$\begin{aligned} v &= -(2^2 + 2^1 + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-10} + 2^{-11} + 2^{-12} + \\ &\quad + 2^{-13} + 2^{-16} + 2^{-21}) = -6.470596790313721 \end{aligned}$$

SZAKIRODALOM

BENKŐ Tiborné–BENKŐ László–TÓTH Bertalan

2003 *Programozzuk C nyelven.* Budapest, ComputerBooks

KERNIGHAM, Brian W.–RITCHIE, Dennis M.

2003 *A C programozási nyelv.* Budapest, Műszaki

SCHILDT, Herbert

1998 *C manual complet.* București, Teora

CATRINA, Octavian–COJOCARU, Iuliana

1994 *Turbo C++.* București, Teora

SORIN, Tudor

Bazele programării în C++. București, L&S Infomat

WWW

C programozás BORLAND C++-ban.

www.eet.bme.hu/publications/e_books/progr/cpp