Bevezetés a programozásba

Dr. Szabó Levente

Tartalomjegyzék

1. fejezet

Bevezető

A C programozási nyelvet eredetileg – még a Unix rendszerhez – Dennis Richie alkotta meg, majd barátja, Brian Kernighan segítségével a ”*The C programming language*” címet viselő könyvben foglalta össze 1978-ban. A C volt az első olyan magasszintű nyelv, melynek segítségével operációs rendszert is lehetett írni, mely tényből sejthető, hogy igen hatékony programozási nyelvről beszélünk. Azokban az időkben a Unix – köszönhetően a hordozhatóságának – szerfelett hatékony rendszernek bizonyult, mely tulajdonságát főként annak köszönhette, hogy szinte az egész rendszer C nyelven lett megírva.

Megalkotása óta – szemben más nyelvekkel – a C nyelv alig-alig módosult, feltehetően azért, mert már a kezdet kezdetén is módfelett átgondoltan szerkesztették meg. Tekintve, hogy a Unix rendszercsalád minden tagját C-ben írták, nem meglepő, hogy az összes Unix rendszer rendelkezik egy C fordítóval. E fordítók segítségével – lefordítva a C nyelven írt programjainkat – állíthatunk elő *futtatható, végrehajtható (executable) állományokat*. (Egyébként, a C-ben írt programunkat tartalmazó fájlt, forrás fájlnak, magát a kódot pedig forrás kódnak (source code) nevezzük.) Mivel a Unix és Linux rendszerek között igen szoros a kapcsolat, nem meglepő, hogy – hasonlóan a Unix-hoz – a Linux rendszerek is tartalmaznak egy GNU C compiler-nek (GNU C fordító) vagy röviden gcc-nek nevezett C fordítót.

Túl azon, hogy lefordítja C programjainkat a gép számára is érthető gépi kódra, a GNU C fordító simán elbánik C++, Objective-C, Fortran vagy akár Java nyelven írt kódokkal is. Fordításkor a gcc az ANSI és ISO által előírt szabványokhoz tartja magát. Természetesen, a gcc-n kívül is van élet, így vannak más lehetőségeink is arra, hogy C kódjainkat gépünkkel megértessük, ugyanis mind a Linux, mind a Unix – ahogyan a Windows és több más operációs rendszer is – lehetővé teszi

számunkra számos, különböző fajta IDE (Integrated Development Environment, Integrált Fejlesztő Környezet) telepítését, így használhatjuk azokat is C kódjaink megalkotásához, fordításához, futtatásához. Példának okáért, kedvenc Linux disztribúciónkat (tehát: *operációs rendszert*) használva, anélkül, hogy egy IDE, grafikus felületen (GUI) való üzemeltetésével terhelnénk a processzort, rendkívül energia- és időtakarékosan dolgozhatunk úgy, hogy akár ki sem nézünk a parancssorból, de természetesen, ha úgy tartja kedvünk igénybe vehetjük a grafikus felületet is[1], ahol számos IDE közül választhatunk, úgy mint Anjuta, Code:Blocks, CodeLite, Eclipse, CDT, XEmacs, Geany, KDevelop, stb. Az említett IDE-k általában Linux és Windows alatt is működőképesek (kivéve a KDevelop és az Anjuta, melyek csak Linux alatt futnak). Rendszerint egy IDE mindenképp tartalmaz egy szövegszerkesztőt, ahol kódolhatunk, s többnyire valamilyen C fordítóval is el van látva.

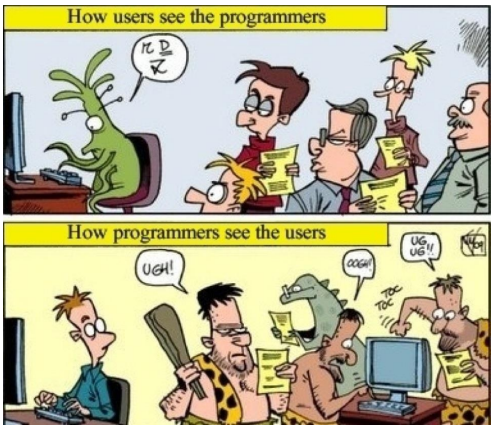
Jelen kurzus során nem használunk fejlesztő környezetet, de természetesen aki akar, kódolhat abban is.

1.1. Előszó

A C programozási nyelvet nem túl könnyű elsajátítani. Dacára annak, hogy a nyelv szerkezete nem túl bonyolult, valamint viszonylag kevés elemet tartalmaz, *jó* C programot írni meglehetősen nehéz. Ha el akarjuk sajátítani ezt a nyelvet (vagy egyáltalán, a programozás alapvető skill-jeit), meglehetősen sok időt kell eltöltenünk *gyakorlással*.

E félév során – többek között – az alapvető algoritmusok, C nyelven történő megírását (kicsit okoskodóan kotnyeleskedve: implementálását) fogjuk megtanulni. A gyakorlás folyamán felmerülő sok-sok probléma megoldása közben megismerkedünk több fajta adatszerkezettel is, attól függően, melyekre lesz épp szükségünk az aktuális probléma hatékony kezeléséhez. Elég sok feladat található e kis könyvecskében, melyek közül majdnem mindhez megoldást is mellékeltem. Ezzel kapcsolatban fontos szem előtt tartani, hogy a megadott ”kulcsok”, az adott kód megírásának nem az egyetlen lehetséges módját prezentálják, ahogy többnyire egy hegy csúcsára sem csak egyetlen út vezet.

[1] Ami nem azért elterjedt, mert annyira hatékony, hanem azért, mert még az is könnyen használhatja, aki szerint egyébként a Föld lapos...



A lelkes hallgatóknak szóló tanácsom – persze, csak, ha elfogadják – a következő: mielőtt bárki megnézne egy megadott megoldást, mindenképpen próbálja meg először önállóan kitalálni azt! Ha nem sikerül elsőre, semmi gond! Ilyenkor böngésszük át a megadott kódot, s miután megértettük azt, essünk neki újra a feladatnak (de persze közben ne puskázzunk..)! Ha kell újra és újra fussunk neki, de semmiképp se nyugodjunk bele a kudarcba! Addig "nyüstöljük" magunkat amíg össze nem hozunk egy működő megoldást, lett légyen az bármilyen ronda is esetleg! Így eljárva, minél többet gyakorlunk, annál jobban/könnyebben/olajozottabban fogunk kódolni. Mottónk: *többet ésszel, mint erővel*... vagy mégsem.. legyen inkább: *a számítógép nem azt teszi, amit szeretnénk, hanem azt, amire utasítjuk.* Ha megértjük, hogy ez pontosan mit jelent, van esélyünk rá, hogy programozók legyünk.

E segédlet vonalvezetése sokban támaszkodik Pere László *UNIX – GNU/Linux Programozás C nyelven* címet viselő könyvének tematikájára, így aki szükségét érzi, lapozgassa bátran az említett művet, melyet tanulmányai folyamán e sorok írója is gyakran forgatott, de tartsa szem előtt, hogy sokban el is térünk a fent hivatkozott irodalomtól, hiszen e kis segédlet megírását nem pontosan ugyanaz a cél motiválta, mint az említett könyv elkészítését. A benne található programokat nem azért írtam, hogy a C nyelv teljes arzenálját felhasználván, azon keresztül, az elérhető legfrappánsabb problémakezelést valósítsam meg, mivel célom e kis példaprogramokkal mindössze az volt, hogy az épp adott tudásszintről, a következő – magasabb, ha tetszik: mélyebb – nívóra való átlépést – az ehhez szükséges ismeretek elmélyítése által – a tőlem telhető módon elősegítsem, helyenként áldozván akár a pontatlanság, illetve pongyolaság oltárán is. Hitem szerint ugyanis könnyebb az alapok megértetését követően kiküszöbölni az említett hiátusokat, mint azok – a lényegi tudást elfedő – sulykolása után megértetni a programozás fő fogásainak vázát képező ismereteket.

A kurzus során nem ismerjük meg a C nyelv teljes apparátusát, mivel a célunk ”mindössze” az, hogy megalapozzunk egy ”masszív”, az algoritmusok kigondolása és implementálása terén a hallgatót hatékony munkára alkalmassá tevő programozási készséget, melynek később igen nagy hasznát veszi majd. Gondolhatunk itt természetesen egy jövőbeli állásinterjúra, de akár már a következő szemeszterek is szóbajöhetnek (Programozás II., Programozás III., PHP, Javascript, stb).

Kódjaink írásakor tehát mindig csak azokat az elemeit használjuk a C nyelvnek, melyek már elégségesek az aktuális probléma kezeléséhez.

2. fejezet

”Hello world!”, avagy az első kódunk

Első programunkat – csak, hogy tisztelegjünk kicsit a hagyomány[1] előtt is – öntsük az alábbi formába!

```
#include<stdio.h>

main()
{
  printf("Hello␣world!\n");
}
```

A fenti program a következő üzenetet írja ki a konzolra: *Hello world!*. Felmerül persze a kérdés: mégis mit tegyünk, hogy ezt a gép meg is csinálja nekünk? Egy tetszőleges szövegszerkesztő segítségével, alkossuk meg a fent látható forráskódot! Mentsük el a programunkat egy `.c` végződésű fájlba[2], amit elnevezhetünk például `first.c` -nek. Ezt a kódot még nem érti a számítógépünk, segítsünk hát neki, s fordítsuk le azt a gép saját nyelvére, ezúton létrehozva a *futtatható, végrehajtható (executable)* állományt[3], mely már a számítógép számára is érthető nyelven/formában tartalmazza programunkat. Egy Linux rendszer parancssorában – a GNU C fordítót használva – a `gcc` begépelése által juthatunk futtatható állományhoz, az alábbi módon:

[1] Az első, C nyelvről szóló könyvükben, Brian Kernighan és Dennis Richie a ”Hello world!” programmal kezdték a C nyelv bemutatását.

[2] Linuxunk ebből fogja tudni, hogy a fájl egy C programot tartalmaz.

[3] Egy ilyen állomány neve rendszerint `.exe`-re végződik, de a Linux megismeri enélkül is.

```
Bash$ gcc first.c -o hello
```

, ahol az `-o` a `gcc` parancs egyik kapcsolója, melynek segítségével tetszőleges névre keresztelhetjük el futtatható állományainkat (itt most épp `hello` lett). Ha ki szeretnénk próbálni újdonsült programunkat, írjuk be a parancssorba, hogy `./hello`, ahol a `./` a munkakönyvtárat jelenti. Enélkül nem futna le a programunk, ugyanis a Linux – biztonsági okokból – a futtatandó állományt sosem keresi a munkakönyvtárban. Miután elkövettünk minden fent javasolt dolgot, képernyőnk valahogy így fog festeni:

```
Bash$ gcc first.c -o hello
Bash$ ./hello
Hello world!
```

Ha történetesen valaki Windows használata által szeretné megnehezíteni az életét, használhatja a Dev-C++ fejlesztői környezetet is, ami ugyan *szerfelett elavult*, viszont C-t tanulni megfelel, főleg azért, mert *nagyon kicsi* az erőforrás-igénye. Maga a Dev lehalászható például innen: http://www.bloodshed.net/dev/devcpp.html

A Dev esetében az első leküzdendő akadály, amivel szembesülünk, az a „pontosvessző-hiba"... a dolog lényege abban áll, hogy a Dev–C++ –ban magyar billentyűzettel nem lehet pontos vesszőt írni, ami elég ciki...már feltéve, ha C programot szeretnénk készíteni. Tekintve, hogy az „*AltGr*” és a „ , ” jel billentyűkombináció hatása nem a pontosvessző előcsalogatása, hanem a programsor dekommentelése, meg kell változtatnunk ezt a beállítást! Nosza, essünk hát neki..... (*Eszközök* » *Gyorsbillentyűk Konfigurálása* » Kijelöljük a „*Szerkesztés: Megjegyzés Ki*” sort, majd az „*Esc*”-kel kivégezzük.... („*Ok*”-t is nyomjunk neki!)) Dolgozhatunk hát végre! Jupppíí!

Ha e trauma sem szegte kedvünket és továbbra is Windows alatt, Dev-ben szeretnénk pötyögni, akkor a megnyitása után a következő a teendő: *Fájl* » *Új* » *Forrás fájl* » Ide írjuk a kódot. Ha kész vagyunk, akkor *Futtatás* » *Fordítás* (Itt a fájl típusát válasszuk „C source files”-nak, majd *mentés*!) Ha jó a kódunk, s nem kapunk hibajelzést, akkor lesz egy `mentésinév.c` forrás fájlunk, amiből a fordító már le is gyártotta mellé a futtatható állományt (nyilván abba a mappába, ahová a forrást is mentette).

Nem árt tudni, hogy a fenti módon eljárva nem épp úgy működik a program, ahogy várnánk (aki szeretné, próbálja csak ki!), ezért módosítanunk kell az eredeti kis kódunkat, mégpedig az alábbira:

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

main()
{
 printf("Hello␣world!\n");

 system("pause");
}
```

Ezután már úgy fog viselkedni, ahogy szeretnénk. Ehelyütt nem megyünk bele a fenti módosítások jelentésébe, viszont megjegyezzük, hogy általuk a kód hordozhatósága sérül[4]. Természetesen, bárki bármilyen környezetben dolgozhat, ahol csak szeretne, viszont tartsa szem előtt az esetleges buktatókat, melyekbe jelen kurzus keretei között nem áll módunkban mélyebben betekinteni. Minden környezethez található részletes tutorial a weben, így, ha valaki úgy dönt, hogy IDE-ben dolgozik, tanulmányozza át alaposan a vele kapcsolatos tudnivalókat.

Térjünk azonban vissza az eredeti kódocskánkhoz, s vonjuk le a nyilvánvaló tanulságokat!

- Amint az látható, a programozó meglehetősen nagy szabadsággal bír a sorok, illetve utasítások elhelyezését, s nemkülönben a köztük lévő kihagyások betoldását illetően.
- Résen kell lennünk a fejállományokkal kapcsolatban! Ilyen például az `#include<stdio.h>` is. Ezekről még ejtünk (jó)pár szót.
- A **main** függvényről is értekezünk még eleget. (Amúgy ő a kurzus főszereplője.)
- A **printf** függvény.....hát igen, ha a **main** Jockey, akkor ő nagyjából Bobby a Dallas –ból.
- Egyáltalán... *mik azok a függvények a C nyelvben?*

A fentebb vázoltakra természetesen még visszatérünk, azonban mielőtt vigyázó szemünket újfent rájuk vetnénk – fókuszunkat élesítendő – megéri tennünk egy

[4]Azért sérül, mert az egyes C fordítók, sok esetben saját ”dialektust beszélnek”, mely nem feltétlenül része a mindenki által megértett ANSI szabványnak. Ilyen például a `system("pause");` is. Épp ezért, e kurzus folyamán – többnyire – nem mozdulunk ki a bárhol, bármikor felhasználható, alapvető ANSI szabvány kényelmes aklából.

tiszteletkört, ami után könnyebben tudjuk majd értelmezni az eddigiek során már részben kifejtetteket.

3. fejezet

Függvények és változók

Mik azok a függvények a C nyelvben, s mik a változók? Ha matekból már ismerősek az említett fogalmak, (s miért ne lennének?), akkor egészen nyugodtan induljunk ki az azokra vonatkozó, bennünk élő intuitív képből. Megtehetjük ezt, mert a C nyelv függvényei hasonlóan viselkednek a már általunk megismert függvényekhez, amennyiben előbbiek – akárcsak az utóbbiak – a kapott feldolgozandó paraméter, valamint a függvény leírásától függően (azok *függvény*ében) állítanak elő valamilyen eredményt.

```
main(){
int a, b;

a=1;
b=a+1;
}
```

A fenti program – túl azon, hogy kiszámolja $1 + 1$ értékét – igazából nem más, mint a **main** nevű függvény leírása.

Sok tanulság néhány sorban!

- Először is: a függvény neve (esetünkben: `main`) után egy `()` következik mindig, ugyanis a fordítónak ily módon jelezzük, hogy függvényt hozunk létre a továbbiakban, melynek leírása (hogy mit is csinál), a `{}` jeleken belül található.
  - Tehát: a C-beli függvények – voltaképp programrészletek – névvel bír-

nak, s létrehozásuk a fent vázoltak szerint zajlik.

- Másodszor: a `main` egy spéci függvény! C-ben ez a szó jelöli a program belépési pontját, magyarán, a program végrehajtása nem más, mint a `main` függvény végrehajtása. Fontos, hogy programunkban mindig legyen egy (és csakis egy) függvény, mely a `main` névre hallgat. A program a `main` végrehajtásával kezdődik, s véget is ér utána legott.

  – Amúgy a fenti program nem igazán „vallásos" módon íródott..... ugyanis a `main()` használata nem volt épp szabályszerű, mivel a `main()`-nek mindig kell rendelkeznie visszatérési értékkel, mégpedig `int` típusúval (a típusokról később többet), akkor is, ha egyébként nem etettük meg semmivel (nem fogadott paramétert)! Ebből következik, hogy nem a

    ```
    main(){}
    ```

    a legegyszerűbb – úgymond – semmittevő C program amit írhatunk, hanem az

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

    Az már csak a mi szerencsénk, hogy a fordítók zöme elfogadja a pongyolább, ám egyszerűbb formát..... (Megnyugtatásul: a eddigiek során emlegetett egyéb fogalmakkal, mint például a *visszatérési érték*, *paraméter(ek)*, vagy egyáltalán, maguk a *függvények*, foglalkozunk még bőven, így senki se keskenyedjen el, ha momentán nem érti, miről van szó. Nem véletlen, hiszen még nem is magyaráztam el ezeket.)

  A lényeg, hogy a mi kis példaprogramunk a 2. sorral kezdődik és az ötödikkel ér véget.

- Harmadszor: mint azt már említettem volt, a programozó keze meglehetősen nagy szabadságnak örvend a program „szövegszerkesztésének" elkövetésekor. Látszik, hogy itt is éltünk ezzel, ugyanis egész jó módja volt a logikailag elkülönülő szereppel bíró sorok közé bepasszintani egy üres sort, hiszen amíg az egyikben csupán létrehoztuk a változókat, addig a másikban már azokkal kapcsolatos parancsokat osztogattunk. Máshol és máskor majd pedig esetleg a sorok beljebb való kezdésével jelezhetjük az egymásbaágyazottságot, ezáltal is áttekinthetőbbé formázván a forrást.

- Negyedszer: változót tehát, mint az látható fentebb, úgy hozhatunk létre (pontosabban szólva: úgy deklarálhatunk), hogy megadjuk a típust majd a nevet (neveket), s a dolog végére mindig **pontosvesszőt** teszünk**!** (Önmagában, a deklarálással csak helyet foglalunk a változónak a memóriában, viszont az értéke bármi lehet, ami a lefoglalt helyen volt épp (pl az ott lévő memóriaszemét). Ezért fontos, hogy az értékadásig (inicializálásig) egy változó értéke rendszerint határozatlan. A két művelet persze akár össze is vonható, például: `int a=1;`)

  (Természetesen külön sorban is létrehozhattuk volna `a`-t és `b`-t, de a látott módon talán elegánsabb, mint szétválasztva azt, ami összetartozik.)

- Ötödször: értékadás.... A 4. és az 5. sorban található módon kell eljárnunk, ha a már létrehozott változóinknak értéket is akarunk adni! (Itt is ott a pontosvessző!!) **Nagyon fontos**, hogy *a változók létrehozásának mindig meg kell előznie az utasításokat*! **Figyelem!** Nem arról van szó, hogy egy adott változót kezelő utasítást – lett légyen az valahol a programban – kell megelőznie az adott változó létrehozásának, hanem bárminemű utasítás kiadása előtt már „le kell tudnunk" a változók létrehozását! Másként fogalmazva: **utasítás kiadása után változó létrehozása már nem megengedett!**

  Az alábbi módon eljárni tehát szigorúan tilos és életveszélyes (legalábbis a kapcsolódó folyamatra nézve...)!

```
main(){
        int a;
        a=1;

        int b;
        b=a+1;
}
```

  Fenti módon tehát SOHA..... !!!!! Most persze – csak a magunk élvezetére – kipróbálhatjuk, mi lesz a fordítás eredménye.... A `gcc` ugyan elhümmög rajta, s szabad folyást enged a program futásának, továbbá, jó eséllyel más fordítók is "zöld"-et adnak neki, de erre nem építhetünk, ugyanis ez csak az időközben napvilágot látott bővítmények következménye. Az alapvető C szabvány nem enged meg efféle huncutkodást, úgyhogy jobban teszszük, ha tartjuk magunkat a fent kifejtett ökölszabályhoz! Egyébiránt, ha bármikor hibaüzenetbe szaladunk, s több is van belőle, mindig az elsőre fókuszáljunk, ugyanis lehet, hogy a többi már csak annak folyománya!

Most pedig evezzünk kissé zavarosabb vizekre, s hozzunk létre egy teljesen új függvényt, mely százalékot, pontosabban százalékértéket számol!

```
#include <stdio.h>

int szazalekertek(int a, int b){
    return(a*b/100);
   }

main(){
  int szalap;
  int szlab;
  int szertek;

  szalap=50; szlab=75;
  szertek=szazalekertek(szalap,szlab);

  printf("%d\n",szertek);
  }
```

Láthatóan a függvényt úgy hoztuk létre, hogy még a neve (`szazalekertek`) előtt megadtuk a visszatérési érték típusát (`int`), majd a neve után – zárójelek közé szorítva – a paraméterlistát, azok típusával egyetemben. Ezek után pedig, `{}`-be foglaltuk a függvényt leíró utasítást. Ezt a függvényt használtuk fel később a programban, midőn egy kifejezés jobb oldalán "akcióba" hívtuk. (A `printf` függvényről egy kicsit később értekezünk.) Ha egyszer összehoztunk egy függvényt, úgy bármilyen kifejezésbe beépíthetjük azt! Hát nem szuper? De! :)

Fontos apróságok Úgy vélem, hogy jó ha még idejekorán rávilágítunk arra, hogy **miért is írunk függvényeket**, hiszen az eddigiek alapján bárkiben felmerülhet a jogos kérdés, miszerint: nem lenne egyszerűbb ha – mellőzvén a függvényekkel való bíbelődést – az általuk tartalmazott utasításokat egy az egyben belegyömöszölnénk a `main`–be?

- Nos, nem feltétlenül. A helyzet ugyanis az, hogy rendszerint **egy** adott **program**on **több programozó** is dolgozik, midőn mindenki a rábízott problémát megoldó algoritmust kódolja, melyet elment egy állományba ahonnan a fő program (a `main` (vagy egy másik függvény)) bármikor „behívhatja" egy kis munkára, mely „behívás" nem más, mint a *függvény hívása*. Ezért is van rá szükség, hogy megtanuljunk „függvényül" :) . . . de nem csak ezért!
- Szintén fontos szempont, hogy a `main` rövid és könnyen áttekinthető legyen! Képzeljük csak el, hogy mi lenne akkor, ha a `main`–ben megadott adatokat először egy sok-sok soros feldolgozó eljárásnak vetnénk alá, majd

az így kapott eredményeket megint csak hasonlóan összetett kódú algoritmussal újra „megdolgoznánk”, stb, stb! Ez esetben rettenetesen hosszúra nyúlhatna a főprogramunk, a `main`, ráadásul nem lévén kivehetőek a főbb moduljai, teljesen elvesznénk a részletekben annak áttekintésekor. Egy ilyen ”ömlesztett”, kaotikus kódban az egyes – szintaktikai vagy ”csupán” működésbeli – hibák felderítése szinte lehetetlen vállalkozás.

- Sokkal egyszerűbbé és áttekinthetőbbé válik a kódunk, ha a `main`–t több modulból építjük fel, még akkor is, ha csak mi dolgozunk a programon. Ily módon a hibák is sokkal könnyebben lokalizálhatóak, továbbá, ha egy-egy függvény munkáját több helyen is igénybe vennénk, akkor ahelyett, hogy újra és újra beírnánk az általa tartalmazott utasításokat a `main`-be, elég ha csak – a neve beírása által – újra hívjuk azt. Maga a függvényhívás művelete, mint az fentebb látható, rendkívül egyszerű: csak annyit kell tennünk, hogy beírjuk a függvény nevét a kódba oda, ahol szükségünk van a munkájára. (Nyilván persze szükség lehet a paraméterek megadására/átadására is, stb, mely kérdéseket nemsokára alaposan körüljárjuk még.)

Visszatérve a ”százalékos” függvényt tartalmazó programocskánkhoz.... **álljunk meg egy szóra!** Jó ez így? Számoljunk már gyorsan utána fejben! Mi lehet a gond? Frissítsünk az alábbi verzióra!

```
#include <stdio.h>

float szazalekertek(float a, float b){
    return(a*b/100);
   }

main(){
  float szalap;
  float szlab;
  float szertek;

  szalap=50; szlab=75;
  szertek=szazalekertek(szalap,szlab);

  printf("%f\n",szertek);
  }
```

Ugye, ugye? Nem figyeltünk a típusokra, s csípőből „`int`”–et adtunk meg, a szó szoros értelmében nem számolván a visszatérési érték minden lehetséges számhalmazbéli besorolásával, illetve annak a változó típusára gyakorolt hatásával! Ez

a kérdés amúgy megér egy misét, így foglalkozunk is majd vele bővebben, ahogy a fejállományokkal is.

A fenti programot ezzel még azért nem filéztük ki teljesen! Beszélnünk kell még példának okáért a **return** utasításról! Igazából a függvénynek **átadott** (bemenő paraméter(ek)), s az általa **visszaadott érték** (az eredmény) kérdésére hegyezzük ki az alábbiakat.

A zárójelben találhatóak az `a` és `b` változók. Ezek értékét a függvénynek – a későbbi felhasználás során – **átadott** értékek határozzák meg, majd miután szabadon masszírozta azokat a **return** utasítás adta keretek között (vagy akár a **return** előtt!), a **return** hatására véget ér a függvény – nem a program!! – futása, s visszaadja a zárójelben megszült értéket a hívó függvénynek. **Ez** a **visszatérési érték**.

Egy függvény esetében tehát bármit is írunk a **return** után (akárcsak számot vagy egy változót, melynek értékét még a **return** előtti részben számolta ki a függvény), ha végrehajtja a program, a tékozló vezérlés visszatér a hívóhoz, s átadja néki a kapott értéket. Fent vázolt esetünkben, mint az látható, meghívtuk az általunk készített, százalékértéket számító függvényt, s annak visszatérési értéke ide – a kifejezés jobb oldalába – helyettesítődött be. Magyarán, a kifejezés bal oldalán ácsingózó változó kapta meg azt érték gyanánt.

Fontos apróságok

- Jó ha tudjuk, hogy a `return` utasítás használható másként is, például olyan esetben, ha valamilyen feltétel teljesülése esetén kiléptetnénk a vezérlést a hívott függvényből, s szeretnénk, hogy visszatérjen (*return*(!)) a hívás helyére, mely esetben a visszatérési értéket akár hibajelzésre is használhatjuk.

- Nem mellékes továbbá az sem, hogy egy függvénynek nem feltétlenül kell rendelkeznie visszatérési értékkel, tehát `return` utasítással sem. Ilyen többek között például az az eset is, amikor mondjuk csak ki szeretnénk íratni vele valamit vagy csak át szeretnénk írni pár változó értékét (ehhez persze már a **mutatók** nyújtotta lehetőségek igénybevételére is szükségünk van), stb. A vezérlés, a függvény leírásának végét jelző „`}`” jelhez érvén ugyanis automatikusan visszaugrik a `main`, hívás helyét követő részéhez, s a program fut tovább.

- Az ilyen – visszatérési érték nélküli – függvényeket eljárásnak hívjuk (de becézés gyanánt akár nyugodtan le is függvényezhetjük őket, ez nem olyan nagy hiba), s a szigorú értelemben vett függvényektől úgy érdemes őket

megkülönböztetnünk, hogy létrehozásukkor oda, ahol eddig a visszatérési érték típusa szerepelt (pl.: `float szazalekertek`) azt írjuk, hogy `void`. Ha például – fenti kódunknál maradva – csak ki szeretnénk íratni a százalékértéket, akkor a zárójelben megidézett függvény, s vele a program is az alábbi formában tündökölne tovább:

```
#include <stdio.h>

void szazalekertek(float a, float b){
printf("%f\n", a*b/100);
}

main(){
float szalap;
float szlab;
float szertek;
szalap=50; szlab=75;
szazalekertek(szalap, szlab);
}
```

Nagyon fontos, hogy a függvényt mindig annak felhasználása előtt hozzuk létre! Azért fontos ez, mert a fordító meg szeretné vizsgálni a függvény paramétereinek típusát, azért mégpedig, hogy egyeztesse azokat a híváskor átadott változók típusával. Tehetnénk persze még ezt is:

```
#include <stdio.h>

float szazalekertek(float a, float b);

main(){
float szalap;
float szlab;
float szertek;
szalap=50; szlab=75;
szertek=szazalekertek(szalap,szlab);
printf("%f\n",szertek);
}

float szazalekertek(float a, float b)
{
return(a*b/100);
}
```

Látszólag megszegjük a saját szabályunkat, pedig nem. Csak, hogy jól értsük a fentieket: *a fordító kizárólag arra kíváncsi, hogy a függvény paramétereinek típusa összeegyeztethető-e a hívó állandóival, változóival*.

Vegyük észre, hogy fentebb, a függvény paramétereinek típusát meghatároztuk mielőtt hívtuk (felhasználtuk) azt. **Ezért** nem jelzett hibát a fordítónk! Nyilván furcsának tűnhet ilyen szeparált módon írni a függvényeket (előbb a paramétere-rek, s csak máshol következik a függvény leírása), helyette rendszerint a „minden összetartozót együtt" utat szeretjük követni.

Igazából mindkét út járható, de jó, ha tudjuk, hogy vannak bizonyos előnyei a fentebb látható módszernek. Arról, hogy pontosan mik is ezek az előnyök, majd valamikor később ejtek szót, most igyekezzünk egyelőre az itt tárgyaltakat minél alaposabban megérteni!

Feladat:
Írjunk programot, mely egy általunk készített függvény segítségével kiszámolja egy adott sugarú kör területét! A π legyen 3.14!

Lehetséges megoldás:

```
#include <stdio.h>

float korte(float a){
return (a*a*3.14);
}

main(){
float sugar, terulet;
sugar=4;

terulet=korte(sugar);
printf("terulet=%f\n", terulet);
}
```

3.1. Kiíratás

Eljött hát a `printf()` függvény ideje. A C nyelv kevés eszközzel bír, de – valószínűleg épp ezért (is) – meglehetősen rugalmas. Talán meglepő, de egyebek mellett, nincs benne mód az üzenetek képernyőre – szabványos kimenetre – való

kiíratására sem. Szerencsére, réges régen, egy messzi messzi galaxisban, az egyik szerfelett ügyes faj, a programozók fajának, egy meglehetősen pimasz, egyben kihívásokat kedvelő alfaja, a C programozók, az ilyen és ehhez hasonló problémákat már jó ideje kiküszöbölték, az azokat megoldó függvények megírása által. Emlékszünk még, ugye? Függvényt már mi is írtunk! :)

Namármost, a sok-sok megírt függvényt – lévén amúgy kedves népek – elmentették a C könyvtárba, hogy mindenki kedvére tobzódhasson bennük, s imígyen, ne kelljen a kereket újra és újra feltalálni.

Lehetőségünk van hát arra, hogy programjainkba beépítsük őket anélkül, hogy előtte megírnánk bármelyiket is. Emlékszünk még az alábbi „körünkre”?

```
#include <stdio.h>

main(){
printf("Hello world!\n");
}
```

Naszóval.....az első sor egy – úgynevezett – előfeldolgozónak szóló utasítás, melyre azért van szükség, hogy a mások által már megírt függvények **típus**ának meghatározását betöltsük a fordítás előtt. Jelen esetben az **stdio.h** fejállománnyal kacérkodunk, ami egyébiránt a **standard input/output** „család” nevének rövidítése, magyarabbúl szólva: ő a **szabványos bemenet kimenet**. Magát a meghívót az **#include**-dal kézbesítjük, míg a címzést a **<>** jellel oldjuk meg, ami közli a fordítóval, hogy ne a munkakönyvtárban, hanem a fejállományokat tároló könyvtárban keresgéljen.

Ennyit hát röviden a fejállományok népes csapatáról. Térjünk át a `printf` nevelésének módozataira! Maga a `printf` a szabványos kimenetre való **formázott** kiíratásra szolgál. Kiírathatunk szöveget, vagy akár változók értékét is megjeleníthetjük a segítségével. Pislogjuk meg az alábbi példát!

```
#include <stdio.h>

main(){
int alfa, beta, ceta;                    /* :-) */
alfa=1;
beta=2;
ceta=alfa+beta;

printf("alfa=%d\nbeta=%d\nceta=%d\n",alfa,beta,ceta);
}
```

Mi lesz a képernyőn? Először is: de csálén áll az az `alfa`! Adjunk neki a forrásban egy **space**-t! Nnnna... mindjárt más, egyúttal a **space** – idézőjelen belüli – szerepét is megismerhettük!

Nézzük a többi leszűrendőt!

A `printf` hívásakor három változó értékét írattuk ki a képernyőre. A `printf` függvény ezúttal négy paraméterrel bír.

Az első paraméter a **formátumot előíró** szöveges érték, amit mindig meg kell adnunk! Ezt látjuk az idézőjelen belül. Itt írjuk elő, hogy az e paramétert követő paraméterek, esetünkben a másik három (`alfa`, `beta`, `ceta`), milyen formában kerüljön a képernyőre. Lássuk, mi mindent tartalmazhat ez az első paraméter! (Tehát még mindig az idézőjelek között vagyunk!)

Lehetnek benne **egyszerű karakterek**, melyeket a `printf` változtatás nélkül kiír a kimenetre. Arra kell csak nagyon figyelni, hogy a `%` és a \ jelek különleges jelentéssel bírnak a `printf` függvény számára, illetve más karakterek is kerülhetnek ilyen helyzetbe, amennyiben eme két karakter **után** állnak!

Tartalmazhat **rövidítéseket**, melyek abban az esetben használatosak, amikor valamilyen karaktert nem tudunk beírni a forrásprogramba. Ezek kiírásához használjuk a kétbetűs rövidítéseket, amelyek mindig a \ jellel kezdődnek.

Megjegyezzük mellékesen, hogy a fenti kis kódban tettenérhető a kommentelés módja is, jelesül, * *\ jelek közé kell tenni azt, ami egy másik programozónak vagy a jövőbeli énünknek szól, s szeretnénk, ha a fordító figyelmen kívül hagyná.

Feladat: Vizsgáljuk meg, mi lehet a `\n` esetében az `n` szerepe a \ után! Nézzük meg azt is, hogy például miért felelős a `\t` jelölés!

Megoldás: új sor, tabulátor. Egyébiránt, csak, hogy érzékeltessük a `printf` rugalmasságát, a fent masszírozott program ölthette volna akár az alábbi formát is:

```
#include <stdio.h>

main(){
int alfa, beta;
alfa=1;
beta=2;

printf("alfa=%d\nbeta=%d\n", alfa, beta);
printf("ceta=%d\n", alfa+beta);
}
```

Próbáljuk csak ki! Végül pedig elérkeztünk – még mindig az első paraméteren belül – a **formátumleírók**hoz. **Ezekkel hivatkozunk az első paraméteren belül, a többi paraméter értékére**, úgy mégpedig, hogy a `%` jel után biggyesztjük az épp aktuális betűt, ami esetünkben a `d` volt, (tehát: `%d`) ezzel utasítva a `printf`-et, hogy az adott helyre írja ki a ***következő*** **paramétert**, mégpedig tízes számrendszerben. A lényeg, hogy a `printf` minden formátumleíróhoz (lásd: `%d`), hozzárendeli a ***következő*** **paraméter** értékét és **azt írja be a helyére**. *Feladat:* játszogassunk el egy két lépés erejéig, példának okáért az elhelyezéssel!

Feladat: Írjunk olyan progarmot, ami átváltja a beírt összegű Forintot Euróba, az általunk megadott árfolyam szerint! **Oldjuk meg függvény írásával!**

Lehetséges megoldás:

```
#include <stdio.h>

float valto(float a, float b){
return(a/b);
}
main(){
float Forint, Euro, arfolyam;

Forint=15;
arfolyam=306;

Euro=valto(Forint, arfolyam);
printf("%f␣Forint=%f␣Euroval\n", Forint, Euro);
}
```

....és anélkül!

Lehetséges megoldás:

```
#include <stdio.h>

main(){
float Forint, Euro, arfolyam;

Forint=15;
arfolyam=306;

Euro=Forint/arfolyam;
printf("%f␣Forint=%f␣Euroval\n", Forint, Euro);
}
```

.....vagy akár:

```
#include <stdio.h>

main(){
float Forint, Euro, arfolyam;

Forint=15;
arfolyam=306;

printf("%fForint=%fEuroval\n",Forint,Forint/arfolyam);
}
```

A `printf`-ről és a rá vonatkozó ismeretekről beszélünk még, bár azt jobbára az épp aktuális feladathoz szükséges tulajdonságaival kapcsolatosan fogjuk elkövetni. Egyelőre legyen elég ennyi, s átülvén egy zsírosabb falatokat kínáló asztalhoz, legyen a következő fogás a

3.2. Formázott bemenet

-et felügyelő `scanf`! Fontos ezt a függvényt megismernünk, ugyanis e függvény ismeretével felvértezve dinamikusabbá tehetjük programjainkat, szemben az eddig írt statikus kódokkal, melyekre az íráson és indításon kívül nem bírtunk semmiféle befolyással. A formázott bemenet nyújtotta lehetőséget kihasználva módunkban áll a program által elvégzett műveleteket, illetve azok sorrendjét, stb-t, annak futása folyamán macerálgatni. Magáról a `scanf` függvényről, tanulmányaink jelen szakaszában csak a legszükségesebbeket árulom el, épp csak annyit, hogy alkalmazni tudjuk azon programjaink írásakor, melyek – lényegüket tekintve – a **vezérlőszerkezetek**kel való bánásmód elsajátítását hivatottak számunkra elősegíteni. A homályban hagyott részekről, majd egy – jóval később megejtendő – másik órán lebbentem fel a fátylat.

A **lényeg**, hogy a `scanf` – bizonyos szempontból legalábbis – hasonlít a `printf`-hez. Az egyik legfontosabb különbség közöttük a működési „vektoruk” iránya, ugyanis a két irány – vagy inkább: a két értelem – egymással pontosan 180^o-ot zár be. Vegyük észre, hogy a `printf` **kiír** valamit, amit előtte mi az általunk óhajtott formába masszíroztunk (ezért is értekeztünk **formázott** kimenetről). Namármost, a `scanf` viszont **„beír”** valamit (az idézőjel nem véletlen, de erről majd később csevegünk!) a szabványos bemenetről (például a billentyűzetről), amit előtte.....úgy van: mi már megformáztunk $\rightarrow$ így kapunk **formázott** be-

menetet.

Nézzünk egy példát:

Az alábbi programocska bekér tőlünk két egész számot, s kiszámolja a különbségüket. Nosza, próbáljuk is ki! Fontos: mindkét beírt szám után nyomjunk „Enter”–t!

```
#include <stdio.h>

main(){
int elso, masodik;
printf("Irj␣be␣ket␣egesz␣szamot,␣");
printf("s␣megmondom␣a␣kulonbseguket!\n");
scanf("%d%d", &elso, &masodik);
printf("A␣szamok␣kulonbsege:␣");
printf("%d␣-␣%d␣=␣%d\n", elso, masodik, elso-masodik);
}
```

Miről is van itt szó.....? Első körben az a legfontosabb, hogy megértsük a `scanf("%d%d", &elso, &masodik);` turpisságait!

Nincs nehéz dolgunk, elég csak arra gondolnunk, amit a `printf`-ről már módunkban áll tudni, gondolok itt például annak kitalálására, hogy mit kellene tennünk akkor, ha ki akarnánk íratni a két változó (`elso` és `masodik`) értékét.

Hogy is nézne az ki?

Hát, valahogy így: `printf("%d%d", elso, masodik);`

Jól van, tudom, hogy lenne mit formázni rajta, de most nem ez a lényeg, hanem az, amit – ha megnézzük a két előző programsort, s látunk is – rögtön észrevehetünk...

Na? Dereng már? Pontosan! A `scanf` függvény első paramétere (tudjátok, az idézőjelek közötti rész) pontosan olyan viszonyban van – legalábbis a `%` jelet, meg az utána következő betűt tekintve – az azt követő paraméterekkel (itt ezek most az `elso` és a `masodik`), mint azt a `printf`-nél volt szerencsénk már megtapasztalni. Arra, hogy a `scanf` esetében az „idézőjeles” paramétert követő paramétereknél miért kell beelőznünk a változókat az `&` jellel, majd a mutatók tárgyalásakor visszatérek. Addig is, érjétek be ennyivel: csak :)

Második körben pedig – már magát a teljes programot tekintve – láthatjuk, hogy visszaköszön néhány már tanult dolog.

Incselkedvén újsütetű tudásunkkal, nézzük meg, hogy festett volna az általunk már megírt ”körte–függvényes” program, ha felépítésekor, a `scanf`-et is beledolgozzuk!

```
#include <stdio.h>

float korte(float a)
{
 return (a*a*3.14);
}

main()
{
float sugar, terulet;

printf("Kerlek␣ird␣be␣a␣kor␣sugaranak␣erteket,␣");
printf("hogy␣kiszamithassam␣a␣teruletet!\n");

printf("sugar:");
scanf("%f", &sugar);

terulet=korte(sugar);

printf("terulet=%f\n", terulet);

}
```

Próbáljuk csak ki!

Most már „menet közben” megadhatjuk az általunk tudni vágyott területérték kiszámításához szükséges sugarat! Egész izgi, nem? Ha nem találjuk annak, akkor gondoljunk bele, hogy `scanf` nélkül, minden egyes alkalommal nekünk kellene „kézzel beletenni” a `sugar` változó értékét a forrásfájlba! ... Akkor már inkább egy számológép! :) A fenti módon viszont ugyanazon forrásfájl alkalmazásával képesek vagyunk kezelni az összes szóbajöhető kör esetét!

(Ez persze így nem igaz, hiszen – típusuknál fogva – a `float` változóink sem vehetnek fel bármekkora értéket, de ezen lendüljünk túl, mert itt és most nem ezt a problémát gyüszmékeljük!)

3.3. Vezérlőszerkezetek

Az e fejezetben bemutatott szerkezetek fontos közös tulajdonsága abban áll, hogy segítségükkel megváltoztathatjuk az utasítások végrehajtásának sorrendjét. (Vegyük észre, hogy ha a `scanf` használata által dinamikussá tettük a programunkat, akkor a *vezérlőszerkezetek* alkalmazásával máris „kettesbe” kapcsolunk!) Kurzusunk e hétre eső fejezetében az alább soroltak kerülnek kifejtésre:

- *feltételes utasításvégrehajtás* (igen, jól gondoljátok, ennek az `if`-es témakör a részét képezi, de ide soroljuk a többszörös elágazás (esetszétválasztás), avagy a `switch` területén való jártasságot is.)
- *ciklusok* (velük kapcsolatban merülnek majd fel az elöl- (`while`), illetve hátultesztelő (`do... while`) ciklusokkal kapcsolatos tudnivalók, akárcsak az egyszerűsített ciklusszervezés, melynek használata feltételezi már a `for` ismeretét is.

3.3.1. Feltételes utasításvégrehajtás

Ha az az `if` nem volna...

Kezdjük mindjárt az elején a *feltételes utasításvégrehajtás*sal! Avagy, hogyan közöljük a vezérléssel a „**Ha** ez van, ezt csináld, **egyéb** esetben azt!” típusú vágyainkat? Nem is olyan nehéz ezt megtennünk, mint gondolnánk! Először is hívjuk segítségül az angol nyelv területén való kalandozásunk során szerzett tapasztalatunkat, miszerint a
HA = IF és az EGYÉB = ELSE!
A következő kis program segítségével igyekszem bemutatni a feltételes utasításvégrehajtás alkalmazásának módját. Jöjjön hát az `if`! Vegyük újra elő a már jól ismert „körtés” programunkat és tegyük egy kicsit szemtelenebbé!

```
#include <stdio.h>

float korte(float a)
{
 return (a*a*3.14);
}

main()
{
float sugar, terulet, n=0;

printf("Kerek␣egy␣0-nal␣nagyobb␣sugarerteket");
printf("␣a␣teruletszamitashoz!"); scanf("%f", &sugar);

if (sugar>n){
        terulet=korte(sugar);
        printf("terulet=%f\n", terulet);
        }else{
      printf("Mondom:␣nullanal␣nagyobb!\n");
      }
}
```

Látható ugye? Az `if` utáni `( )`-ben (a feltétel „fejében”) egy *logikai kifejezés* áll `(sugar>n)`, melynek igaz, illetve hamis értékétől (ez a *feltételvizsgálat*) függ, hogy az *igaz ág* által tartalmazott utasítások hajtódnak-e végre, vagy pedig a *hamis ág*on találhatóak végzik-e el a rájuk rótt feladatot. Az *igaz, illetve hamis ág „tennivalóit”*, mint az észrevehető, `{}` jelek közé illik csomagolnunk! Természetesen egy adott ágra nem csak egy, hanem *több utasítást is ráaggathatunk*, viszont amennyiben csupán egykéket pátyolgatunk, tudatában kell lennünk annak, hogy akár el is hagyhatjuk a `{}` jeleket. Sőt esetenként még a hamis ág megírásától is eltekinthetünk!

Nézzünk egy ilyen példát is!

```
#include <stdio.h>


float korte (float a)
{
 return (a*a*3.14);
}
```

```
main(){
float sugar, terulet, n=0, m=100;

printf("Kerek␣egy␣0-nal␣nagyobb,␣szaznal␣kisebb␣");
printf("sugarerteket␣a␣teruletszamitashoz!");
scanf("%f", &sugar);

 if (sugar<=n)
 {
 printf("Haho,␣0-nal␣nagyobbat!\n");
 }else{
        if(sugar<m)
        {
         terulet=korte(sugar);
         printf("terulet=%f\n",terulet);
        }
       }

 if(sugar>=m)
 {
  printf("Erettsegi␣milyen␣lett?\n");
 }
}
```

Bizony, bizony, túl azon, amit fent írtam a hamis ágról, amint az látszik, *a feltételek akár még egymásba is ágyazhatóak*! (Bár túl sok feltételt nem tanácsos egymásba gyömöszölni, mert pillanatok alatt áttekinthetetlenné válhat a kódunk...)

Természetesen, a fenti feladatot akár egy darab `if` alkalmazása által is megoldhattuk volna, valahogy így:

```
#include <stdio.h>

float korte (float a){return (a*a*3.14);}

main()
{
float sugar, terulet, n=0, m=100;

printf("Kerek␣egy␣0-nal␣nagyobb,␣szaznal␣kisebb␣");
printf("sugarerteket␣a␣teruletszamitashoz!");
scanf("%f", &sugar);
```

```
 if (sugar<=n || sugar>=m)
 {
  printf("0-nal␣nagyobbat,␣szaznal␣kisebbet!\n");
 }else
  {
   terulet=korte(sugar);
   printf("terulet=%f\n", terulet);
  }
}
```

Világos, hogy fentebb a feltétel fejének `(sugar<=n || sugar>=m)` módon való megfogalmazásával – ahol a `||` jel egy „logikai vagy”-ot jelent, mely által a feltétel „megigazul”, ha a `||` jel egyik *vagy* másik oldalán álló kifejezés igaz – két legyet is kivontunk a forgalomból (a nempozitív és a 100- tól 'jobbra' lévő számtartományt), s így már nincs szükségünk egy másik `if`-es kifejezésre, továbbá a „hamis ág” is leegyszerűsödik.

Fontos apróság ami szemet szúrhatott egyeseknek itt az `if`-ekben való tobzódásunk közepette, hogy a feltételek fejébe pőre számok helyett, az adott számértékkel „megáldott” változókat lapátoltam be. Természetesen felvetődik a kérdés, hogy mi értelme van ennek? Nem lenne egyszerűbb beírni egy számot, mint „felesleges” változókkal bíbelődni, s tárolásukkal terhelni még a memóriát is?

Nos, egy ilyen inci-finci kódnál még valóban értelmetlennek tűnhet a forrás ilyetén módon való megfogalmazása, ám gondoljunk csak bele, hogy mi lenne akkor, ha mondjuk egy „masszívabb” kódban, egy - több feltétel esetében is használt - szám értékét meg kellene változtatnunk!

Úgy lenne vajon egyszerűbb a dolgunk, ha külön-külön ki kellene guberálnunk a kódból az összes feltételes kifejezést, ahol az adott szám szerepel, majd egyenként át kellene írnunk mindenütt azt az új értékre vagy esetleg akkor járnánk-e jobban, ha eleve úgy alkotnánk meg a programot, hogy egy változóban tárolva az adott számot, az egyes feltételekbe csak a változó nevét (fent `n` és `m`) írnánk be a konkrét szám helyett?

Nyilván az utóbbi esetben könnyebb az életünk, hiszen egy ilyen kódban elég *egyszer* átírni a számot, mégpedig az őt képviselő változó értékadásának a helyén.

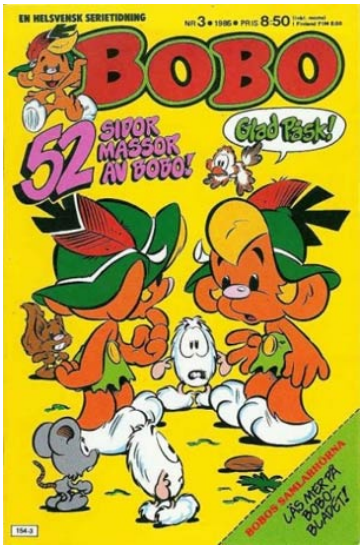
Azt az eléggé el nem ítélhető cselekedetet pedig, amikor valaki konkrét számot ír bele egy feltételbe (vagy például egy ciklus fejébe) egy változó neve helyett, *beégetés*nek becézzük, s tűzzel-vassal igyekszünk küzdeni ellene.

Jegyezzük meg jól: **kódba számot nem égetünk bele!**

(E sorok írója tanúja volt már olyasminek, amikor egy jobb sorsra érdemes kollégájának majdnem a teljes munkanapját felfalta egy nagyon hosszú, valamikor régen, valaki által *beégetett* módon megírt kódban való hiba-keresgélés. A mai napig nem vagyok róla meggyőződve, hogy őszinte mosoly volt az arcán a több órás - más hibája által motivált - „küldetés” végén, ami elkerülhető lett volna egyébként. . .)

3.3.2. Ciklusok

"Addig forgasd a gyűrűket, amíg (WHILE) azt nem látod, 'hopp'!"



A fenti kis idézet, megboldogult, ártatlan gyermekkorom egyik képregényéből való, s hogy mégis mi köze van az alább tárgyalandókhoz, arra hamarost fény derül. Hősünk, aki ezúttal demostratíve is közreműködik, elősegítendő hallgatóink szellemi épülését, jobb oldalt látható. Háttérsztorink lényege az, hogy ez a Bobo talált egy távcsövet, melynek teleszkópszerűen kihúzható gyűrűin egy-egy betű volt olvasható, mégpedig sorrendben a következők: H, O, P, P. Ha belenézett a kukkerba és nekiállt tekergetni a gyűrűket, s velük nyilván a betűket is, azok egy vonalba kerülésekor (HOPP), azonmód ott termett ahová épp nézett a látcsővel. Hoppá! :) S hogy ennek mi köze van jelen tanulmányainkhoz? Hát csak az, hogy – jóllehet maga sem tudta, de – szegény pára voltaképp egy úgynevezett *hátultesztelő ciklust* csavargatott mit sem sejtő kiskorú rajongói esti kakaójába! Mert hát miről is van szó? A gyűrűk tekergetésére való felbujtás értelmezhető ciklusmagba tett utasítás gyanánt is, kiváltképp és különösen azért, mert *ciklikusan* újra és újra meg kell *csinálni* (`do`), *amíg* (`while`) egy bizonyos feltétel igaz logikai értékkel bír. Vesézzük csak ezt ki jobban! Lássuk, hogyan is fest **Bobo ciklusa**!

- **Csináld/do**! Tekerj egyet valamelyik gyűrűn!
- Addig, **amíg/while** (nem látod, hogy HOPP.)

Nem más ez bizony, mint egy tipikus **hátultesztelő ciklus**, azért mégpedig, mert

először egyszer végrehajtja a ciklusmagba épített teendőt, s csak *aztán* (utána, mögötte, tehát: *hátul*) teszteli, hogy a feltétel igaz, illetve hamis logikai értékkel bír-e. Amennyiben igaz, tehát (Nem látod, hogy HOPP.) lefuttatja újra a ciklust, ha viszont már látszik a HOPP, tehát a feltétel már nem igaz, befejezi a tekergetést, s HOPP, kilép a ciklusból.

Nézzük meg, hogyan működik ez az egész a C nyelv eszközkészlete által nyújtott lehetőségeket felhasználván megfogalmazva! Lássunk egy hátultesztelő ciklust! Az alábbi programocska bekér egy egész számot, s kiszámolja annak faktoriálisát, újra és újra „köröztetve” egy hátultesztelő ciklust.

```
#include<stdio.h>

main(){
int n, faktorialis=1, i=1, min=0;
/*valtozok deklaralasa illetve inicializalasa*/

printf("Adj␣meg␣egy␣egesz␣szamot,␣s␣megmondom");
printf("␣a␣faktorialisat!");
scanf("%d", &n);
/*ama szam bekerese, melynek tudni szeretnenk
a faktorialisat*/

if(n<min){
printf("Nem␣nyert!␣:)\n");
}/*bemenet ellenorzese, a beirt
 szam nem lehet kisebb nullanal*/
else{
  do{
     faktorialis=faktorialis*i; /*faktorialis szamitas*/
     i=i+1;                     /*valtozo leptetese*/
     }while(i<=n);
     printf("%d␣faktorialisa=%d\n",n, faktorialis);
   }
}
```

Világos, ugye? :) Na jó, „megtávcsövezzük” azért egy kicsit. Első blikkre, foglalkozzunk csupán magával a ciklussal! Mindenekelőtt a `do`-val megadjuk a ciklusmagban elvégzendőket, amit *egyszer mindenképp lefuttat, mellőzve a feltétel ellenőrzését(!)*, hisz annak legelőször csak a ciklus első befutása *után* ejti szerét a vezérlés. Ezért is becézgetjük *hátultesztelő ciklus*nak, melynek kapcsán jó, ha tudatában vagyunk a benne rejlő kockázatoknak. :) *Fontos* leszűrhető ismeret továbbá a *kommentelés módja*, jelesül; a `/* */` jelek közé szendvicselt információ ki-

zárólag a mi tájékozódásunkat hivatott segíteni, a fordítót teljesen hidegen hagyja. Egyébiránt a léptetés `i=i+1;` módon való kivitelezése meglehetősen ritka, helyette jobbára a `i++;` vagy a `++i;` az ami „dívik”. Ezt majd - a többi hasonló finomsággal egyetemben - még megbeszéljük bővebben, mert megéri a ráfordított időt, ráadásul bizonyos esetekben egyáltalán nem mindegy, hogy melyiket alkalmazzuk! Addig is - egyebek mellett - még hátravan az

3.1. ábra. A hátultesztelő ciklus veszélyei :)

Elöltesztelő ciklus

, melynek esetén Bobo, *előbb* megnézi a betűk elhelyezkedését (ez a *feltétel vizsgálat*a), s csak *aztán*, abban az esetben kezdi meg a ciklus magjába foglalt utasítások *végrehajtás*át, ha a feltétel igaz. *Ezért* hívjuk *elöltesztelő* ciklusnak a szóbanforgó szerkezetet. Nézzük meg, hogyan tudnánk átfazonírozni a már fentebb összefésült hátultesztelő ciklust magába foglaló programocskát, elöltesztelő ciklussal rendelkezővé! Nos, valahogy így:

```
#include<stdio.h>

main(){
int n, faktorialis=1, i=1, min=0;
printf("Adj␣meg␣egy␣egesz␣szamot,␣s␣megmondom");
printf("␣a␣faktorialisat!"); scanf("%d", &n);

if(n<min)
 {
  printf("Nem␣nyert!␣:)\n");
  }else{
        while(i<=n)
        {
          faktorialis=faktorialis*i;
          i=i+1;
         }
         printf("%d␣faktorialisa=%d\n",n, faktorialis);
        }
}
```

Ugye milyen egyszerű? A `do` helyére a `while` került, maga a `do` pedig - térhajtó-

műveit beizzítva - végleg elhagyta ezt a dimenziót. Ha kipróbáljuk, láthatjuk, hogy mindkét program elvárásainknak megfelelően üzemel, tehát a két, egymásba átírható forma, egymással egyenértékű megfogalmazásai ugyanannak a feladatnak[1][2].

A ciklust, illetve a programot tekintve, mindössze annyi történt tehát, hogy az alábbi formát

```
do{
   faktorialis=faktorialis*i;
   i=i+1;
   }while(i<=n);
```

a lenti megjelenéssel bíró kifejezéssé ijesztettük össze.

```
while(i<=n){
            faktorialis=faktorialis*i;
            i=i+1;
            }
```

Magyarán, minden egyebet változatlanul hagyva, a `while(i<=n);` - úgymond - kiütötte a `do`-t. De vajon tényleg *mindig ilyen egyszerű átírni egyik fajta ciklust a másikba?* **Sajnos nem!** *Legyünk nagyon, de nagyon óvatosak, s nem csak a ciklusok átírásakor, hanem már azok megfogalmazásakor is! Kerüljük el az aknák – átgondolatlan ciklushasználat általi – magunk alá telepítését!* Mondanivalómat megvilágítandó megemlítem, hogy az elöltesztelő "módot" megadhattuk volna az alább vázolt formában is!

```
while(i<n){
          i=i+1;
          faktorialis=faktorialis*i;
          }
```

Vegyük észre, hogy, jóllehet a *feltétel megváltozott*, az *utasítások sorrendje nemkülönben*, imígyen biztosítandó a ciklus változatlan módon való futását. Ha az *új*

[1] Vigyázzunk azonban, ugyanis már most előrebocsátom, hogy nagy szerencsénk volt ezzel a programmal, mivel az "egymásba való átírás" korántsem ilyen egyszerű minden esetben, ahogy azt hamarosan látni is fogjuk!

[2] No persze felvetődhet az amúgy teljesen jogos kérdés, hogy mi végre a két fajta ciklus, ha mindkettő használható? A válasz egyszerű. Noha mindkettő használható, vannak feladatok, melyek egyszerűbben kezelhetők az egyikkel, míg más problémák, inkább a másik módszer alkalmazása által önthetők áttekinthetőbb formába. Például, vegyük a helyzetet, amikor bekérünk valamit - mondjuk egy számot - , amit megad a felhasználó, s miután megadta, a ciklus végén leellenőriztetjük a vezérléssel, hogy valóban szám volt-e a beírt karakter. Ha nem stimmel, akkor nem lépünk ki a ciklusból, hanem újra bekérünk egy számot (ilyenkor persze célszerű lehet egy figyelmeztetést is képernyőre pötyögtetni a programmal). Ezt a bekérést egyszer mindenképp (tehát: feltétlenül) le kell futtatnunk, ezért célszerű ilyenkor a hátultesztelő ciklust alkalmaznunk.

*ciklus*t behelyettesítjük az eredeti program megfelelő soraiba, annak *működése továbbra is meg fog felelni* a vonatkozó elvárásainknak! Magyarán, a probléma C nyelven *elöltesztelő ciklusként*, amint az lentebb látható, (legalább) *kétféleképpen megfogalmazható*. Egymás alá téve a releváns részeket, valamint szem előtt tartván a tényt, miszerint a `faktorialis`, ahogy az `i` értéke is, már eleve 1–ről indul, könnyen összehasonlíthatjuk működésüket. Íme a két ciklus:

- Az egyik:

```
while(i<=n){
            faktorialis=faktorialis*i;
            i=i+1;
           }
```

- A másik:

```
while(i<n){
            i=i+1;
            faktorialis=faktorialis*i;
           }
```

És most jön a LÉNYEG! Ha az alsót is úgy írjuk át hátultesztelőbe, mint azt a felsőnél már elkövettük volt (a `while(i<n)` lemegy alulra, s helyére beül a `do`, miközben más nem változik), majd az így kapott,

```
do{
   i=i+1;
   faktorialis=faktorialis*i;
   }while(i<n);
```

részt behelyettesítjük a programba, keservesen csalódunk, ugyanis szembesülhetünk a *hibátlan, ám nem a szándékunk szerint való működés* átkával. (0 vagy 1 bevitele esetén nagy szamárság lesz a jutalmunk. . . .) A megfelelő működés visszanyeréséhez bizony át kell írni a ciklus magját, ahogy magát a feltételt is! Így mégpedig:

```
do{
   faktorialis=faktorialis*i;
   i=i+1;
   }while(i<=n);
```

Igaza van a szemfüles hallgatónak, aki azt észrevételezi, hogy a legutóbbi ciklus már eddig is a kis házi arzenálunkat gyarapította, de itt és most nem ez a lényeg,

hanem az, hogy észrevegyük, és meg is tanuljuk, hogy ciklust, annak egyik fajtájából a másikba átírni, egyáltalán nem magától értetődő mutatvány, ami megúszható, egy szimpla cserebere által.

Az elöltesztelő és hátultesztelő ciklus nem ugyanaz, hanem - értő kézzel - egymásba átírható, két különböző szerkezet!

Ez tehát:

```
while(i<=n){
            faktorialis=faktorialis*i;
            i=i+1;
            }
```

egyenértékű ezzel:

```
do{
  faktorialis=faktorialis*i;
  i=i+1;
  }while(i<=n);
```

ahogy ezzel is:

```
while(i<n){
           i=i+1;
           faktorialis=faktorialis*i;
         }
```

viszont ezzel nem(!):

```
do{
   i=i+1;
   faktorialis=faktorialis*i;
   }while(i<n);
```

Vegyük szemügyre az utóbbi négy ciklust alaposan, elemezzük őket, s nem utolsósorban okuljunk!

S végül - csak, hogy némileg egyszerűbb legyen az élet - elmondom az

Egyszerűsített ciklusszervező utasítás

meséjét is, melyet ciklusaink minél egyszerűbbé tételét célzó igyekezetünkben alkalmazhatunk látványos eredménnyel. Alább egy ilyen példa látható:

```
#include <stdio.h>

main()
{
 int i, n=11;
 for(i=1; i<n; i=i+1)
 {
  printf("%d␣\t␣%d␣\n", i, i*i);
 }
}
```

A ciklus a fenti program 7. sorában kezdődik és a 9. sorban ér véget. (Próbáljunk meg rájönni mit csinál a program!)

A ciklus fejében - a `( )` jeles kifejezésben - három rész található, pontosvesszővel elválasztva egymástól. Az *első* egy (esetleg több, vesszővel elválasztott) utasítást tartalmaz(hat), amit a program a ciklus *végrehajtása előtt, egyszer* hajt végre, a *második* egy *feltétel*, melynek igaz volta esetén a ciklusmag végrehajtódik, a *harmadik* rész pedig egy utasítás (esetünkben egy léptetés), amit a program mindig a ciklusmag végrehajtása *után* hajt végre.

Könnyen felismerhető, hogy a `for` ciklus *elöltesztelő* ciklus, ezért a `while`-os elöltesztelő simán átírható `for`-t tartalmazó ciklusba, így lesz például ebből

```
while(i<=n){
faktorialis=faktorialis*i;
i=i+1;
}
```

ez

```
for(;i<=n; i=i+1)
{
 faktorialis=faktorialis*i;
}
```

Ki is próbálhatjuk. Tegyük be a programba, s láthatjuk, hogy működik, ahogy azt is, hogy *nem mindig kell kitöltenünk a fej minden részét*, a lényeg, hogy létüket azért jelezzük a pontosvessző beírásával! Persze az egészet összegyűrhetjük az alábbi - ciklusmagot nélkülöző - formába is:

```
for(;i<=n;faktorialis=faktorialis*i, i++);
```

Sőt, akár - végképp sportot űzvén a tömörítésből - így is eljárhattunk volna:

```
for(;i<=n;faktorialis*=i++);
```

Ha valakinek a fenti sor nem világos, ne keskenyedjen el! :) Később még lesz szó ilyesmikről.

Amennyiben ciklusmagot nélkülöző `for` ciklust hozunk létre, a ciklus feje után mindig ki kell tennünk a `;`-t, mint az fent is látható! *Vigyázzunk* azonban a következőre! Írhatunk maggal rendelkező `for` ciklust a `{}` jelek kitétele nélkül is, ami - távolról ráhunyorítva - hasonlónak tűnhet a "magtalan" kivitelre, de a kettő nem ugyanaz, ugyanis a maggal rendelkező, de `{}` jelek nélküli ciklus így néz ki:

```
for(;i<=n;  i=i+1)
    faktorialis=faktorialis*i;
```

Ezt az alakot akkor és *csak akkor* alkalmazhatjuk, *ha* a ciklusmag mindössze *egy darab* utasítást foglal magába, ugyanis ilyen ciklusforma esetén, a vezérlés kizárólag a ciklusfej utáni első utasításra tekint ciklusmagként.

(Egyébként több változó is elnyerheti kezdeti értékét a `for` ciklus fejének első részében, mint azt a következő feladatban látni is fogjuk, arra kell csak figyelnünk, hogy közéjük tegyünk vesszőt (ne pontosvesszőt!).)

Feladat:
Írjuk meg a faktoriális értéket számoló programot úgy, hogy mind a szám bekérését, mind magát a számolást, mind pedig az eredmény kiírását egy-egy függvény végezze el! (Ezúttal nem kell vizsgálnunk, hogy megfelelő számot adott-e meg a felhasználó!) Valami ilyesmi lehet a végeredmény:

```
#include<stdio.h>


int beker()
    {
     int a;
     printf("Adj␣meg␣egy␣egesz␣szamot,");
     printf("␣s␣megmondom␣a␣faktorialisat!");
     scanf("%d",&a);
     return(a);
    }

int faktor(int b)
    {
     int i, f;
     for(f=1,i=1;i<=b;i++)
```

```
            {
             f=f*i;
            }
     return(f);
    }

void kiir(int c, int d)
     {
      printf("%d faktorialisa= %d\n", c, d);
     }

main(){
int n, fakt;
n=beker();
fakt=faktor(n);
kiir(n, fakt);
}
```

Láthatjuk fentebb, hogy a `beker()` függvénynek nincs bemenő paramétere, amin - ha az eddigieket figyelmesen olvasgattuk - meg sem lepődünk (hiszen elég csak a `main()`-re gondolnunk).

Feltűnhet továbbá az is, hogy a `kiir` függvény nem rendelkezik visszatérési értékkel, ami nem baj, ugyanis csupán annyi a dolga, hogy megjelenítsen a képernyőn egy másik függvény által már kiszámolt eredményt.

Az ilyen - visszatérési értékkel nem rendelkező - függvényeket nevezzük a C nyelvben *eljárás*nak. Ilyetén mivoltukat oly módon jelezzük a fordító számára, hogy az eljárás neve elé - oda, ahol a függvényeknél eddig a visszatérési érték típusát írtuk a `void` szót gépeljük be.

Nézzük, hogy mi történik a fenti programban!

Először is, a `beker()` függvény bekéri a számolandót, aminek értékét átadja az `n` változónak.

Ezután, a `fakt=faktor(n);` sorban a `faktor` függvény megkapja az `n` változó értékét, mely értéket bemásolja a saját (csakis általa "látható") helyi/lokális `b` változójába. A lokális változóival (`b`, `i`, `f`) elvégzi a faktoriális értékének kiszámítását, majd a kapott eredményt a visszatérési értékbe csomagolva átadja a `main` függvény `fakt` névre hallgató változójának.

Mindezek után végül akcióba lendül a `kiir` eljárás is, hogy elvégezze a rábízot-

takat.

Természetesen a `faktor` függvény helyi változóit (`b`, `f`) nevezhetnénk akár `n`-nek és `fakt`-nak is úgy, ahogy a `main`-beli megfelelőiket elkereszteltük s, hogy most mégsem így tettünk, annak mindössze az az oka, hogy ezúton is kihangsúlyozzuk: ezek nem ugyanazok, nem egymással megegyező változók, hanem két különálló függvény saját (helyi, lokális) változói, még akkor is, ha esetleg ugyanaz a nevük!

Természetesen, a fenti kódban sok minden átalakítható, már jelen tudásunk keretei között is. Ilyen például a `for` ciklus is, sőt az igazán bátrak a függvényhívásokat is átrendezhetik, akár úgy is, hogy - puszta számológéppé degradálván a programot - meg sem őrzik a faktoriális értékét.....valahogy így:

```
#include<stdio.h>

int beker()
    {
     int a;
     printf("Adj␣meg␣egy␣egesz␣szamot,");
     printf("␣s␣megmondom␣a␣faktorialisat!\n");
     scanf("%d",&a);
     return(a);
    }

int faktor(int b)
    {
     int i, f;
     for(f=1,i=1;i<=b;f*=i++);
     return(f);
    }


void kiir(int c)
     {
      printf("A␣beirt␣szam␣faktorialisa:␣%d\n", c);
     }

main(){kiir(faktor(beker()));}
```

A függvényhívások fent látott módjával kapcsolatban fontosnak tartom megemlíteni, hogy így azért nem szokás elkövetni azt.....mármint függvényből természetesen hívhatunk függvényt, de függvényt hívni ott, ahol egy másik függvény bemenő paramétert vár (lásd: `kiir(faktor(beker()));`) ..... legalábbis nem

túl ildomos.

Jóllehet - példánknál maradva - az egyik függvény elfogadja ugyan bemenő paraméter gyanánt a „benne” hívott másik függvény visszatérési értékét (pl.: `faktor(beker());`), továbbá a fordító sem jelez hibát, mégis, ha lehet, tartózkodjunk az ilyen "viselkedéstől", ha másért nem, legalább az áttekinthetőség kedvéért!

Feladat:
Egy követ v_0 kezdősebességgel elhajítunk felfelé. Milyen magasságig fog felemelkedni? A feladatot ne analitikusan (algebrai egyszerűsítés), hanem numerikusan oldjuk meg: vizsgáljuk dt időközönként (dt megadható kis szám) a kő helyzetét, és keressük meg, hogy mikor/hol fordul meg! Írjunk programot a feladat megoldására, melynek bekért változói legyenek a `dt`, mint időköz, a `g`, mint nehézségi gyorsulás és a `v0`, mint kezdősebesség! Kivételesen(!), megengedett ugyan a megoldás teljes kódjának `main`-be való ömlesztése, de erről igyekezzünk minél előbb leszokni!

Lehetséges megoldás:

```
#include <stdio.h>

main(){
float g, h, v0, v, dt, n=0;
printf("Nehezsegi␣gyorsulas(m/s2):␣"); scanf("%f", &g);
printf("\nKezdosebesseg(m/s):␣"); scanf("%f", &v0);
printf("\nIdobeli␣felbontas(s):␣"); scanf("%f", &dt);

for (h=0, v=v0; v>n; v=v-g*dt)
{
 h=h+v*dt;
}

printf("A␣legnagyobb␣magassag:␣%f␣m.␣\n",h);
}
```

A megoldás magyarázata: (*Aki érti, hogy miért az adott összefüggéseket használtuk, az ugorja át nyugodtan e magyarázatot!*)

Namármost: az ugye a kérdés, hogy adott v_0 kezdősebesség esetén, milyen magasra emelkedik a kavicsunk? Hogy lehet ezt megválaszolni? Nos, első körben látnunk kell, hogy mi most távolságot szeretnénk számolni (a magasság is távol-

ság). Ehhez, mindenekelőtt azt kell tisztáznunk, hogy milyen típusú mozgással van dolgunk, hiszen az adott mozgástípus esetén érvényes formulákkal juthatunk csak el adekvát megoldáshoz. A kő mozgása ebben az esetben (eltekintve a sebességfüggő közegellenállástól vagy a Földön, mint forgó rendszerben fellépő – mozgást befolyásoló – tehetetlenségi hatásoktól, mint például a Coriolis-erő, stb) jó közelítéssel tekinthető egyenes vonalú, egyenletesen változó mozgásnak (ahol az a gyorsulás állandó).

Mint azt már fentebb rögzítettük volt, mi most távolságot akarunk számolni. Mivel a mozgás pályája egyenes, elnevezhetjük az (f)eldobás helyétől mért távolságot mondjuk x-nek. Ez az x nyilván időtől függő mennyiség, tehát $x(t)$-nek illik jelölni. Mivel tisztáztuk, hogy egyenes vonalú, egyenletesen változó mozgással van dolgunk, nézzük, hogyan néz ki $x(t)$ erre a mozgásfajtára:

$$x(t) = x_0 + v_0 t + \frac{a}{2} t^2 \tag{3.1}$$

Esetünkben $x_0 = 0$m, a gyorsulás pedig negatív, így írhatjuk azt, hogy

$$x(t) = v_0 t - \frac{a}{2} t^2 \tag{3.2}$$

Azt gondolván, hogy a tanulók nehezen váltanak a nézőpontot, a tankönyvek szerzői a feladatban szereplő mozgást elnevezték függőleges hajításnak[3]. Így lett $x(t)$-ből $h(t)$, a-ból pedig g. A használandó összefüggés tehát:

$$h(t) = v_0 t - \frac{g}{2} t^2 \tag{3.3}$$

Minket az érdekel, hogy mekkora magasságot ér el addig, amíg emelkedik, tehát a 3.3 formula az alábbira módosul:

[3] Ami szerintem oktatási szempontból nem feltétlenül hasznos dolog, mert egyfelől, így külön többlet tanórákat kell szentelni ennek a mozgásfajtának, miközben pontosan ugyanarról van szó, amit már előtte egyszer megtanultak az "egyenes vonalú, egyenletesen változó mozgás" címszó alatt, másfelől pedig, épp azt a készséget ölik ki ezáltal a diákból, melynek segítségével megtanulhatná használni a tananyagban szereplő ismereteket (no meg az eszét is), hiszen ráébredne, hogy a függőleges mozgások esetén elég csak 90 fokkal elfordítani a fejét a vízszintes tengely körül (magyarán: elforgatni a vonatkoztatási rendszerét, tehát: nézőpontot változtatni) és az új mozgásforma (a függőleges) máris "vízszintessé" válik, így a vízszintesre megtanultak nyilván itt is érvényesek lesznek. Szóval, megtanulhatná visszavezetni ismertre, az ismeretlent.

$$h_{max} = h(t_{em}) = v_0 t_{em} - \frac{g}{2} t_{em}^2 \quad , \tag{3.4}$$

ahol t_{em} az emelkedés ideje, h_{max} pedig nyilván a maximális magasság. Na de mennyi is az? Az 3.4 formulából ezt nem tudhatjuk meg. Máshol kell hát keresgélnünk!

Először is, derítsük ki, miből tudható meg, hogy a kő már elérte a maximális magasságot, h_{max}-ot? A válasz az, hogy amikor a maximális magasságon van, akkor – az egyébként időtől függő – sebessége nulla, vagyis: $v(t) = 0\frac{\text{m}}{\text{s}}$.

Ez eddig gyönyörű, de minket most az érdekel, hogy mekkora az a magasság, ahol mindez teljesül? Nos, mindössze annyit kell tennünk, hogy kiszámoljuk, hogy a gyorsulás (ami itt negatív (tehát: lassulás)) adott értéke mellett mennyi idő alatt csökken le 0-ra a kezdősebesség v_0 értéke! Újfent csak az adott mozgástípusra érvényes összefüggések között kell keresgélnünk! Azt kell megtalálnunk, amelyik a sebesség időfüggését írja le! Ez a következő:

$$v(t) = v_0 + at \quad , \tag{3.5}$$

ahol az a gyorsulást egyfelől g-vel jelölik, másfelől előjele negatív, így a 3.5 formula az alábbira módosul:

$$v(t) = v_0 - gt. \tag{3.6}$$

Azt tudjuk, hogy az emelkedési idő (t_{em}) alatt lassul le a kő $0\frac{\text{m}}{\text{s}}$-ra, tehát $v(t_{em}) = 0\frac{\text{m}}{\text{s}}$. Ebből következően:

$$v(t_{em}) = 0 = v_0 - gt_{em} \quad \Rightarrow \quad gt_{em} = v_0, \tag{3.7}$$

ahonnan világos, hogy a keresett emelkedési idő:

$$t_{em} = \frac{v_0}{g}. \tag{3.8}$$

Örülünk, mert most már csak vissza kell helyettesítenünk a kapott t_{em}-et a 3.4 formulába, ami után az alábbi összefüggést kapjuk a maximális magasságra:

$$h_{max} = h(t_{em}) = v_0 \frac{v_0}{g} - \frac{g}{2}\frac{v_0^2}{g^2}. \tag{3.9}$$

A 3.9 kifejezés triviális egyszerűsítést követően a következő formába töpörödik:

$$h_{max} = \frac{v_0^2}{2g}. \tag{3.10}$$

Nnnna... *ez* volt a feladat *analitikus* megoldása, hiszen függvényeken keresztül jutottunk el eddig, s közben algebrai egyszerűsítést is elkövettünk.

Térjünk rá most a programozással kapcsolatos részre! Sok jelenség esetében, egyszerűen nem ismerjük a mennyiségek egymástól való függésének pontos alakját, így ilyenkor okosan megtervezett közelítő számításokhoz folyamodunk[4]. Ennek egyik módja lehet például az, hogy veszünk egy esetet, ami hasonló a kezelendő problémához, ám attól eltérően, ismerjük a rá jellemző pontos összefüggéseket. Hisszük vagy sem, de például legfeljebb két, egymás erőterében mozgó test mozgását tudjuk csak időben, analitikus számításokkal lekövetni (ez a kéttestprobléma). Több test esetén már numerikus közelítést kell alkalmaznunk. Maga a közelítés pontossága viszont tetszőleges mértékű lehet, jelemzően akkorára lőjük be, amekkorára az adott keretek között szükségünk van.

A mi feladatunkra vonatkoztatva ezt most úgy lehet elképzelni, hogy nem ismerjük az adott mozgástípust (egyenes vonalú, egyenletesen változó) leíró függvények pontos alakját, csak az egyenes vonalú egyenletes mozgását, ezért megpróbáljuk azzal közelíteni az egyenletesen változó mozgást. Ezt úgy érjük el, hogy veszünk egy pici időközt, dt-t, s úgy tekintjük, hogy dt alatt egyenletes sebességgel mozgott a kavics, s kiszámoljuk, hogy ezzel az állandó sebességgel haladva mekkora utat tesz meg dt alatt. A következő dt időtartam alatt szintén egyenletesen mozog, ám ott már kisebb sebességgel halad, mint az előző dt alatt. Kiszámoljuk itt is, hogy mennyit haladt, majd hozzáadjuk az eredményt az előző időszakaszra kiszámolt távolsághoz. Addig csináljuk ezt, amíg a sebesség előjele pozitív. Ha negatív lesz, akkor az azt jelenti, hogy volt már 0 is, tehát már megjárta a maximális magasságot, ahonnan már vissza is fordult. A fentieket hajtja végre a programban szereplő `for` ciklus:

```
for(h=0, v=v0; v>n; v=v-g*dt){h=h+v*dt;}
```

[4] A kvantummechanikában gyakran előfordul ilyesmi. Ekkor jön be a képbe például a perturbáció számítás.

Látjuk, hogy a ciklus, fejének harmadik részében minden körben csökkenti a sebességet, s a ciklus magjában ezzel az *állandónak* vett sebességgel számolva kiszámolja a megtett távot ($v * dt$), amit hozzáad az addig elért magassághoz, így növelve annak értékét ($h = h + v * dt$).

Világos, hogy a $v * dt$ összefüggés egyenes vonalú egyenletes mozgásra érvényes csak, így a feldobásra már nem, de nem is az egész emelkedésre alkalmazzuk, csak annak pici dt-k alatt megtett szakaszaira, s minden újabb dt időszakaszra kisebb értékkel számolunk. Ezt a kisebb értéket a ciklus fejében a $v = v - g * dt$ utasítás állítja be minden újabb körre. Nyilván csak addig, amíg a v értéke pozitív ($v > n$), hiszen addig megy csak felfelé.

Tegyünk egy próbát! Legyen a kezdősebesség $10\frac{\mathrm{m}}{\mathrm{s}}$, a nehézségi gyorsulás $g = 10\frac{\mathrm{m}}{\mathrm{s}^2}$. Erre az esetre az analitikus úton kapott megoldásunk 5m-es maximális emelkedést jósol. Adjuk meg a megadott megoldásban látható program futásakor ezt a két 10-es értéket, s adjunk meg az időbeli felbontásra (dt-re) mondjuk 0.1-et! A legnagyobb magasságra nem pontosan 5-öt kaptunk, de majdnem. Növeljük a közelítés pontosságát egy nagyságrenddel, s adjunk meg a következő körben dt-re 0.01-et! Hoppá, ez már sokkal pontosabb, sokkal közelebb van az analitikus úton kalkulált 5-ös értékhez. Ha tovább finomítunk, tetszőleges pontosságúvá tehetjük a közelítő számításunkat, akkorává, amekkorára csak szükségünk van.

Egyébként erre az egészre csak azért volt itt szükség, hogy mindenképpen gyakorolni kelljen a ciklus megírását, hiszen analitikus módszerrel csupán egy képletbe kellett volna behelyettesíteni a dolgokat, s voila... :)

Feladat:
Egyfelől: igyekezzünk "függvényesíteni" a fenti programot oly módon, hogy legalább a számolási részt bízzuk rá egy e célból megalkotott függvényre! Másfelől: tegyük interaktívabbá programunkat, úgy mégpedig, hogy annyiszor fusson le, ahányszor csak szeretnénk!

Lehetséges megoldás:

```
#include <stdio.h>

float maxmagassag(float g, float v0, float dt)
{
 float h, v, n=0;
 for (h=0, v=v0; v>n; v=v-g*dt){h=h+v*dt;}
 return(h);
}
```

```
main()
{
int ujra;
float g, v0, dt, h;

do
{
 printf("Nehezsegi␣gyorsulas(m/s2):␣"); scanf("%f", &g);
 printf("\nKezdosebesseg(m/s):␣"); scanf("%f", &v0);
 printf("\nIdobeli␣felbontas(s):␣"); scanf("%f", &dt);

 h=maxmagassag(g,v0,dt);

 printf("A␣legnagyobb␣magassag:␣%fm␣\n\n",h);

 printf("Meg␣egy␣szamolas?␣(i/n)␣");

 ujra=getchar(); if(ujra=='\n'){ujra=getchar();}

}while(ujra=='i');
}
```

Látható, hogy itt kifizetődött a hátultesztelő ciklus alkalmazása, mivel egyszer mindenképp le kell futnia a programnak. (Különben miért indította volna el a felhasználó?)

Az `ujra=getchar();`-ban szereplő `getchar()` függvénnyel még igazából nem akartam előhozakodni (kiválthattuk volna `scanf`-fel is), de múlt órán felmerült valakinél, így hát miért is ne? :) A lényeg, hogy ez a függvény a következő karaktert olvassa be a szabványos bemenetről és azt egész számként adja vissza (tehát: a visszatérési értékének a típusa: `int`). No de mi köze lehet egy általunk beírt karakternek, például egy `i` betűnek egy egész számhoz. A válasz egyszerű: a `getchar()` függvény megeszi a beírt karaktert és annak az ASCII kódját (ami egy egész szám) adja vissza, mint visszatérési értéket. Ezt vizsgáljuk a `while(ujra=='i')` feltételben, ahol az `'i'`-nél látható `'` jelek a közéjük tett karakter (esetünkben az `i` betű) ASCII kódértékére hivatkoznak. (*Fontos*: egyenlőség *vizsgálatakor* == -t használunk, tehát kettő egyenlőség jelet, s nem csak egyet!)

(Az

```
if(ujra=='\n'){ujra=getchar();}
```

feltételre/utasításra elvileg nem lenne szükség, mégsem árt kitenni, ugyanis könnyen előfordulhat, hogy a billentyűzet pufferen marad egy újsor karakter (aminek ASCII kódja a

'\n'

egész szám), s így kapásból azt nyeli be a `getchar()`, amit úgy érzékelnénk, mintha átugrotta volna a vezérlés a `getchar()`-t, pedig nem.)

Végül pedig, bezárván körünket, visszaíveljük fejezetünk fonalát a starthoz, s újfent elővesszük a feltételes utasításvégrehajtást. Ám ezúttal nem ragadunk le a kétágú villákkal ("**Ha** ez van, ezt csináld, **más** esetben azt!") való bíbelődésnél, hanem - szemben az eddigi kettő, egy időben kezelhető lehetőséggel - a továbbiakban képesek leszünk tetszőleges számú eset, egyetlen nekirugaszkodás általi megkülönböztetésére. A minket ebben segítő vezérlőszerkezet neve a

Többszörös elágazás, avagy "hetet egy csapásra"

Persze, becézhetjük *esetszétválasztás*nak is. A többszörös elágazás lényege - mint azt már a neve is sejteti - abban áll, hogy egyazon szerkezeten belül, a vezérlést több (tehát nem csak kettő (igaz ág, hamis ág)), gyakorlatilag tetszőleges számú esetre (`case`) felkészíthetjük, az azok előfordulásának esetén végrehajtandó utasítások megadása által.

Igen, helyes a meglátás, mely szerint incselkedtünk már ilyesmivel "`if`-es" módon is, hiszen hasonló célt szolgált a fejezet első részében bemutatott "`if-else-if`"-es egymásba ágyazás, viszont jó ha észrevesszük azt is, hogy már egy hármas elágazás létrehozása is milyen macerával járt, mert hát csupán egy `if`-es szerkezet elégtelen volt annak kivitelezéséhez.

Az alábbiakban azt fogom megmutatni, hogyan kezelhetünk egy szerkezeten belül több, mint két lehetőséget, sőt, akár "hetet egy csapásra".

Tegyük fel, hogy írunk egy százalékszámítást végző programot, mely - mielőtt megkezdené munkáját - megkérdezi tőlünk, hogy egyáltalán mit szeretnénk számoltatni vele, százaléklábat, százalékalapot vagy esetleg százalékértéket. Ez ugye *három lehetőség*....., amit kezelhetünk ugyan "`if`-es" stílusban (például egymásbaágyazással, meg persze, igazság szerint, anélkül is), de talán kifinomultabb ízlésre vallana, ha *egyszerre* "tudnánk le" mindhárom lehetőséget, *egyazon szinten* tudatva a vezérléssel a hozzájuk kapcsolódó összes teendőt.

Lesz tehát egy programunk, egy *egy pontból kiinduló hármas elágazással*, mely-

nek ágain az adott ág szimbolizálta eset bekövetkeztéhez rendelt utasítások gubbasztanak várván, hogy vadászni küldje őket a vezérlés, mégpedig valahogy így:

```
#include <stdio.h>

main()
{
int a; float b, c;
printf("Mit␣szamoljak?␣(%%-alap:1,␣%%-lab:2,␣%%-ertek:3)");
scanf("%d", &a);
/*a printf, %%-kal tudunk %-jelet kiiratni*/

 switch(a)
 {
 case 1: printf("Kerem␣%%-labat␣es␣a␣%%-erteket!\n");
 scanf("%f%f", &b, &c);
 printf("%f␣a(z)␣%f-nak/nek", c, c*100/b);
 printf("␣a␣%f␣szazaleka.\n", b);
 break;

 case 2: printf("Kerem␣a␣%%-alapot␣es␣a␣%%-erteket!\n");
 scanf("%f%f", &b, &c);
 printf("%f-nak/nek␣a(z)␣%f", b, c);
 printf("␣a(z)␣%f␣szazaleka.\n", c*100/b);
 /*Ugyanaz a keplet, de csak azert,
 mert a "b" itt mast jelol!!*/
 break;

 case 3: printf("Kerem␣a␣%%-alapot␣es␣a␣%%-labat!\n");
 scanf("%f%f", &b, &c);
 printf("%f␣%f␣szazaleka␣%f.\n", b, c, b*c/100);
 break;
 }
}
```

Nézzük - kizárólag a lényegre összpontosítva figyelmünket -, mi játszódik le újdonsült programunkban!

- Először is: az `a` változó értéke 1, 2, illetve 3 lehet. Ettől az értéktől függ, hogy a program a továbbiakban mihez kezd a `b` és a `c` változók általunk megadott értékével. Ezt - mármint, hogy mely változó, illetve kifejezés értékétől függ a további teendő - a `switch` kulcsszó segítségével adjuk a vezérlés tudtára, úgy mégpedig, hogy az utána következő gömbölyű zárójel-

be csomagoljuk azt. Nagyon fontos, hogy *csak egész típusú* változó, illetve kifejezés szerepelhet az említett gömbölyű zárójelek között.

- Másodszor: ha sikeresen túl vagyunk a gömbölyű zárójelekkel vívott csatánkon, legott szembesülnünk kell a `{` személyében érkező erősítéssel. Innentől kell felvázolnunk a vezérlés számára szép sorjában - egészen a `}`-ig - a különböző esetekhez (az esetszétválasztás alapjául szolgáló változó, illetve kifejezés - esetünkben:`a` - különböző értékeihez) kapcsolódó vágyainkat, mégpedig oly módon, hogy minden esethez (minden ághoz), még a legelején beírjuk a `case` kulcsszót, majd azt az értéket, melynek megvalósulása esetén, az adott ágon várakozó utasításokat szeretnénk végrehajtatni. Rögtön az érték begépelése után tegyünk *kettőspontot* (tehát NEM pontosvesszőt!)!

- Harmadszor: fontos megemlékeznünk a fent látható `break` utasítás, `switch`-ben betöltött szerepéről! Miután a vezérlés megtalálta az `a` értékéhez társítható ágat és elvégezte az ott rábízott feladatot, abban az esetben ha `break`-kel találkozik, azonmód kilép a `switch` szerkezetből, s átadódik az azt követő sorra (...tehát a `}` jel utánra...). Magyarán; az elágazás egy-egy ágát a `break`-kel zárhatjuk le. Jó ha tudjuk azonban, hogy van élet a `break` után, pontosabban nélkül is! Könnyen lehet ugyanis, hogy egy probléma megoldása során több értékhez kell egyszerre odavarrnunk ugyanazt a tennivalót. Ha, példának okáért, egy `a` változó 2, 3, 4 és 5 értéke esetén ugyanaz az utasítás, akkor ahelyett, hogy minden ágra külön-külön beírnánk a vonatkozó tennivalókat (esetünkben most épp a "Hello world!" kiíratását), megtehetjük az alábbiakat is:

```
.
.
.
case 2:
case 3:
case 4:
case 5:
printf("Hello world!\n");
break;
.
.
.
```

Igen, a 2, 3, 4, valamit az 5 érték ágai voltaképp *egyetlen* ágat alkotnak, melyhez több érték tartozik, s melyet az említett értékek esetén fut be a vezérlés. Itt fontos megemlítenem, hogy egyazon érték viszont nem tartozhat több ághoz egyszerre!

- Negyedszer: Kezelhető az az eset is, melynek kapcsán a vezérlés nem talál az ágak között olyan értéket, amit az esetszétválasztást megalapozó változó képvisel. Ilyenkor lép színre a `default`!

A `default` használatát demonstrálandó, visszatekintve a százalékos programocskánkra, tegyük fel, hogy valaki véletlenül, vagy esetleg pusztán a huncutkodás kedvéért, nem 1-et, 2-t vagy 3-at ír be, hanem egy másik számot! Programunkat csiszolandó, egészítsük ki azt, egy `default`–ot tartalmazó sor `switch`-be való beszúrása által, majd miután az alábbi formát öltötte, próbáljuk is ki annak működését!

```
#include <stdio.h>

main(){
int a; float b, c;
printf("Mit␣szamoljak?␣(%%-alap:1,␣%%-lab:2,␣%%-ertek:3)");
scanf("%d", &a);

 switch(a)
 {
 case 1: printf("Kerem␣%%-labat␣es␣a␣%%-erteket!\n");
 scanf("%f%f", &b, &c);
 printf("%f␣a(z)␣%f-nak/nek", c, c*100/b);
 printf("␣a␣%f␣szazaleka.\n", b);
 break;

 case 2: printf("Kerem␣a␣%%-alapot␣es␣a␣%%-erteket!\n");
 scanf("%f%f", &b, &c);
 printf("%f-nak/nek␣a(z)␣%f", b, c);
 printf("␣a(z)␣%f␣szazaleka.\n", c*100/b);
 /*Ugyanaz a keplet, de csak azert,
 mert a "b" itt mast jelol!!*/
 break;

 case 3: printf("Kerem␣a␣%%-alapot␣es␣a␣%%-labat!\n");
 scanf("%f%f", &b, &c);
 printf("%f␣%f␣szazaleka␣%f.\n", b, c, b*c/100);
 break;

 default: printf("Csak␣1-et,␣2-t,␣3-at␣fogadok␣el!\n");
 }
}
```

....mint látjuk, a `default` után is kettőspont írandó!

4. fejezet

Néhány fontos apróság

Elérkezett az idő, hogy tartsunk egy kis pihenőt és az eddig leírtak, az ismeretek lényegét képező tudnivalók megértése által bennetek felépült nyers váz „masszájából” – némi csiszolgatás árán – előcsalogassuk a finomabban metszett, árnyaltabb éleket, részleteket is. Ebben a fejezetben azokról a – részemről, órán már megmegemlített – dolgokról lesz szó, melyek nem voltak ugyan elengedhetetlenek a tárgyalt szerkezetek, illetve utasítások működésének megértéséhez (sőt, szerintem a figyelem szétforgácsolása által, csak nehezítették volna azt), viszont mindenképpen részét illik képezniük az alapvető eszköztárunknak. Legfőbb ideje hát, hogy megadjuk nekik a megérdemelt figyelmet!

(Természetesen még a kurzus végén sem leszünk mindentudó programozók, viszont meg lesznek az alapjaink, melyekre támaszkodva tovább építhetjük ezirányú tudásunk egyre lakájosabb bázisát. Mindazonáltal meglehet, hogy valaki úgy gondolja, boldogul az elkövetkező oldalakon sorjázó ismeretekben való elmélyedés nélkül is. Lelke rajta, viszont jó, ha tudja, hogy igénytelensége folytán tudása megmarad a felszínes, "féltudás" állapotában, maga pedig viselni kényszerül annak minden kockázatát.)

4.1. Elemi adattípusok

Jelen alfejezetben eltekintünk a majdan általunk létrehozható, céljainkhoz idomított összetett adattípusok ismertetésétől, beérjük egyelőre a C nyelv beépített adattípusainak, nemkülönben azok néhány jellemzőjének rövid áttekintésével. Alább bemutatom, a minden – lényeget érintő – információ egy helyre való gyömöszölé-

sének iskolapéldáját... a táblázatot: Először is, lássuk az egész számok tárolására használt alaptípusokat! (Megjegyzés: van több is, de nekünk ennyi bőven elég!)

Típus	Megjeleníthető értéktartomány	Méret (byte)
char	$[-128, 127]$	1
unsigned char	$[0, 255]$	1
short int	$[-32768, 32767]$	2
unsigned short int	$[0, 65535]$	2
int	$[-2147483648, 2147483647]$	4
unsigned int	$[0, 4294967295]$	4

Vegyük észre, hogy egy – `unsigned` alkalmazása által – előjelétől megfosztott egész típus, felhasználván az ily módon felszabadított előjeltároló bitet, képes nagyobb – ám kizárólag nemnegatív – számot tárolni, mint amire egyébként mérete által rendeltetve lenne! Egyébként az *egész* típusosztály minden tagja *előjeles*!

Másodszor pedig, mutatkozzunk be a *lebegőpontos* számokkal is elbíró típuscsalád tagjainak!

Típus	Megj. é.	Méret (byte)	Pontosság (jegy)
float	$[3.4 * 10^{-38}, 3.4 * 10^{-38}]$	4	6
double	$[1.7 * 10^{-308}, 1.7 * 10^{308}]$	8	15
long double	$[3.4 * 10^{-4932}, 3.4 * 10^{4932}]$	10	19

Mint látjuk, a lebegőpontos típusok esetében nélkülözni kényszerülünk az előjeltől való megfosztás általi (pozitív irányba történő) méretnövelést lehetőségét (nem használhatjuk az unsigned-ot), ugyanis a *lebegőpontos változótípusok mindig ábrázolják az előjelet*. Alább következő értekezésünk folyamán

4.1.1. a változók változó méretéről

ejtünk néhány szót. Nos, ha valaki veszi a fáradságot és kipróbálja a lenti programocskát, meglepő tapasztalatra tehet szert.

```
#include <stdio.h>

main()
{
```

```
  char a;
  unsigned char b;

  a=200;
  b=200;

  printf("signed␣char:␣%d\n", a);
  printf("unsigned␣char:␣%d\n", b);
}
```

Mi történt itt?

Jóllehet, mind `a` (signed char), mind pedig `b` (unsigned char) értékének 200-zal kellene egyenlőnek lennie, mégis azzal voltunk kénytelenek szembesülni, hogy míg az előjelétől megfosztott változó vígan befogadta a megadott számot, addig az előjellel rendelkező által felvett érték (-56), előzetes várakozásunknak merőben ellentmondani látszik.

A helyzet a következő:

az előjellel bíró, `char` típusú változó a $[-128, 127]$ tartományon legelészhet csak, míg az előjelétől megszabadított (`unsigned`) már a $[0, 255]$ mezőn egerészik, magyarán „eléri” az általunk beadott 200-as értéket is.

Felmerülhet persze bárkiben, az amúgy teljesen jogos kérdés: az előjelesnek meghagyott változó (`a`) miként tett szert az értékére, s hogyan lett az épp -56?

A válasz egyszerű és nagyon tanulságos. A -56-ot eredményező jelenség neve *túlcsordulás*, mely elnevezés az esemény lefolyásáról alkotott, bennünk élő intuitív képben gyökeredzik. Ahhoz, hogy a lényeget megértsük elég mindössze azt elképzelnünk, amit a 99999 km-ig távolságot mérni képes kilométer-számláló esetén tapasztalunk, ha – mondjuk – 100002 km-t kocsikázunk. Mennyit fog mutatni? 2 km-t! (Azért épp ennyit, mert amikor $99999,0$ km-ről továbbpördül $00000,0$ km-re, akkor (m)ért el 100000 km-ig.)

Valami hasonló történik a `char`-ral is, azzal a különbséggel, hogy nála 127 a plafon, s mikor azt elérte, kezdi újra a „számolást” az általa felvehető legkisebb értékről, ami az ő esetében nem 0, hanem -128. Gondoljuk csak végig! *Plusz* 200 volt a beadott érték, viszont nemnegatív tartományban csak 127-ig tudott elszámolni, ami nem volt elég a 200-hoz! Továbbpördült – *túlcsordult* – hát, s kezdte a

legkisebb értékről (-128) és innen lépett még annyit, amennyi az előbb hiányzott a 200-hoz, míg végül ezúton elért -56-ig, s ezt az értéket tette magáévá. . . .200 helyett!

Az egészben az a hajmeresztő, hogy *a C fordítók nem ellenőrzik a változók túlcsordulását*!

Ha például írunk egy számoló függvényt, ilyen esetben *nem kapunk hibaüzenetet*, s nyugodtan hátradőlve székünkben, számoltatunk az általunk készített programmal, miközben nem szembesülünk azzal, hogy egy túlcsordult változó már alattomos, sumák módon szétbombázta az összes általa felvett értékre épülő számítást! *Ez bizony a programozó felelőssége*!

Egyébként – ha már a méreteknél tartunk – az `int` mérete (ilyen példának okáért a `char` is) a C nyelvben *nincs előírva*, így az, az alkalmazott processzortól is függhet! Nem kevésbé fontos tény továbbá, hogy – ilyen szempontból – hasonló jellemzőket mutat a `long double` típus is!

Ahogy már néhány sorral feljebb azt kifejtettem volt, a *mi felelősségünk*höz tartozik annak biztosítása, hogy a változók ilyetén huncutságai ne nyilvánulhassanak meg alattomos hibák okozóiként! Na de mégis

mit tehetünk?

Természetesen a `sizeof()` kulcsszavat függvényszerűen használva – módunkban áll megtudni az *aktuális méreteket bájtban*, úgy mégpedig, hogy beírjuk a következő programot, mely „elbánik” néhány típussal:

```
#include <stdio.h>

main()
{
 printf("char: %d byte\n", sizeof(char));
 printf("short int: %d byte\n", sizeof(short int));
 printf("int: %d byte\n", sizeof(int));
 printf("long int: %d byte\n", sizeof(long int));
}
```

Lekérdezhetjük a legnagyobb és legkisebb értékeket is, valahogy így:

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

```
main()
{
 printf("char␣min.:␣%d\n", SCHAR_MIN);
 printf("char␣max.:␣%d\n", SCHAR_MAX);
 printf("short␣int␣min.:␣%d\n", SHRT_MIN);
 printf("short␣int␣max.:␣%d\n", SHRT_MAX);
 printf("int␣min.:␣%d\n", INT_MIN);
 printf("int␣max.:␣%d\n", INT_MAX);
 printf("long␣int␣min.:␣%ld\n", LONG_MIN);
 printf("long␣int␣max.:␣%ld\n", LONG_MAX);
}
```

Fentebb, a `printf` első paraméterében a `%ld` a `long int`-ért "felel".

A `sizeof()` egyébként nem csak a típusok kulcsszavával használható, hanem változók nevével is:

```
#include <stdio.h>

main()
{
 int valtozo_neve;
 printf("int␣merete:␣%d␣byte\n", sizeof(valtozo_neve));
}
```

Nem szóltam még a lebegőpontos típusok méretének lekérdezési módjáról! Nos, ilyetén szándékunk esetén a következő a teendő:

```
#include<stdio.h>

main()
{
printf("float:␣%d␣byte\n", sizeof(float));
printf("double:␣%d␣byte\n", sizeof(double));
printf("long␣double:␣%d␣byte\n", sizeof(long double));
}
```

Ami itt most még – a lebegőpontos típus emlegetése kapcsán – hirtelen eszembe jut, az annak a csúnya megjelenésnek a meg(re)formálása, amit eddig a `float` típus használatakor tapasztaltunk, ugyanis lehetőségünk van arra, hogy gátat vessünk a tizedesjegyek parttalan tobzódásának, más szóval meghatározzuk a változó értéke által elfoglalt mező szélességét, ahogy a megjelenített tizedesjegyek számát is!

Lássunk egy kapcsolódó példát, melyben megelégszünk 2 tizedesjegy pontossággal történő kiírással is! Alább a lényeget `printf`-ben látható, a % jel és az `f` betű között.

```
#include<stdio.h>

main()
{
float f=224.47785;
printf("%1.2f\n", f);
}
```

A fenti progam futtatásakor láthatjuk, hogy az nem csupán „levágta” a harmadik tizedes jegyet, kettőt hagyván a képernyőn, de le is kerekítette az értéket a kiírt tizedesekre. A számok tárolására hivatott adattípusokról, most áttérünk egy már régóta magunk előtt görgetett hiányosság kiküszöbölésére, s tágítjuk némileg

4.2. a `printf()` függvényhez

köthető ismereteink körét, annak kis mértékben való kiegészítése által. Alább vázolom, hogy a `printf()` függvény idézőjelbe tett részébe (tehát annak első paraméterébe) mikor milyen betűt kell tennünk – a behelyettesítendő típustól függően – a % jel után. Igen, újabb táblázat:

Az 1. paraméter karakterei	A behelyettesítendő típusok
%d	decimális egész
%u	előjel nélküli decimális egész
%f	lebegőpontos
%c	karakter
%s	sztring vagy karakter tömb (karakterlánc)

4.3. Értékadás, s egyebek...

Ezúttal tényleg csak rettenetesen tömören álljon itt pár szó azokról a dolgokról, melyekkel megkönnyíthetjük az életünket a fenti címmel fémjelzett területen való barangolásunk folyamán...

4.3.1. Összehasonlítás

A következő – valószínűleg már ismerős – jelek segítségével tudunk különböző értékeket összehasonlítani:

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

Két változó, illetve kifejezés összehasonlítása esetén az eredmény `int` típusú lesz, 1 értékkel, ha igaz a kifejezés, 0-val pedig amikor hamis. Összetett feltételek használatakor a tagadás kifejezése `!` jellel, a logikai „és” `&&` jelekkel, a „vagy” pedig || szimbólumokkal jeleníthető meg.

4.3.2. Léptetés

Találkoztunk már ilyesmivel, csak még az alábbi formáját nem ismerjük. Egész típus esetén, az eggyel való növelés $++$ jellel, míg a csökkentés $--$ beírásával érhető el. Nem minden esetben mindegy persze, hogy a változó elé vagy mögé írjuk a két jelet!

Félreértés ne essék! Ha – teszem azt – az `i` változó értékét eggyel szeretnénk növelni, mind az `i++`, mind pedig a `++i` megnöveli eggyel az `i` aktuális értékét, „a baj nem itt van, nem most”[1].

Szemünket felnyitandó, írjunk egy pici programocskát, amely egyfelől egyről indulva, egymás alá egyesével növekedő módon oszlopba rendezi a számokat egészen a bekért pozitív `n` számig, mellyel aztán zárja is a sort (oszlopot), másfelől egy tőle „tab-nyi” távolságban lévő újabb oszlopban – a megfelelő sorokban – az előbbi oszlopot alkotó számok második hatványát jeleníti meg!

```
#include<stdio.h>

main()
{int n, i=1;
 printf("n␣erteke:␣");
 scanf("%d", &n);

 while(i<=n)
 {printf("%d\t%d\n", i, i*i);
  i+=1;
 }
}
```

[1] Obi-Wan Kenobi (Baljós árnyak)

Mondanom sem kell ugye, hogy az `i+=1;` az `i=i+1;` helyett áll, mint összevont forma... Nézzük meg mi történik, ha az `i+=1;`-et lecseréljük `i++;`-ra, vagy akár `++i;`-re! Nos, mindkét esetben megfelelően működik a program, látszólag tehát teljesen közömbös, melyik utasítás kiadása által növeljük változónk értékét. Az `i++;` illetve `++i;` általi léptetés, ciklusmagba tett módon való használata azonban lebutított változat, ugyanis betehetnénk azt akár magába a feltételbe is (persze ilyenkor `;` nélkül). Vegyük alaposan szemügyre a fenti program alábbi változatát!

```
#include<stdio.h>

main()
{
 int n, i=0;
 printf("n␣erteke:␣");
 scanf("%d", &n);

 while(++i<=n)
 {
  printf("%d\t%d\n", i, i*i);
 }
}
```

Jól látható, hogy mit nyertünk a `++i;` alkalmazásából fakadóan. (Jól van tudom, voltaképp vesztettünk egy sort, amivel viszont helyet nyertünk.) Azért követhettük el a fentebb művelt dolgokat, mert a ++-szal vagy a −-szal való léptetés beírható különböző kifejezésekbe vagy akár – mint azzal esetünkben élni volt szerencsénk – feltételekbe is! Most pedig szántsuk fel fürkésző tekintetünkkel a fenti program sorait, s értsük meg a működését!

A legfontosabb újdonság a léptetés beültetése a ciklus fejének részét képező logikai kifejezésbe, a feltételbe. Természetesen – tekintve, hogy maga a léptetés már a feltételben lezajlik – hibásan működne a program, ha az `i` változó kezdőértéke továbbra is 1 lenne, hiszen ebben az esetben a ciklusmag már az első lefutás alkalmával is `i=2` értéket kapna bejövő adat gyanánt. Ezért adtam `i`-nek 0 kezdőértéket. Most pedig, hogy tisztáztuk a nyilvánvalót, elértünk a *lényeg*hez! Futtassuk le programunkat újra, de előbb írjuk át a `++i`–t, `i++`-szá!

Bizony, bizony, egy kicsit túllőtt a célon! Pontosabban, eggyel túlfutott azon az `n` értéken, amit beadtunk a futás folyamán. A dolog magyarázata egyszerű és nagyon, de nagyon megjegyzendő!

Amikor az `i` értékét `++i` módon növeltük a feltételben, akkor amellett, hogy az

eggyel nagyobbá vált, már magába a feltételbe is ez a megnövelt érték került bele, ezzel szemben, amikor `i++`-t írtunk a feltételbe, változónk ugyan szintén nagyobb értékre tett szert, viszont az azt tartalmazó feltételbe még az előző – a léptetést megelőző – érték lett behelyettesítve! A fenti állítások nem csak feltételekben való – ily módon történő – léptetések esetén igazak, hanem minden olyan helyzetben, amikor a fent tárgyalt léptetési módot alkalmazzuk, lett légyen az akár feltételben, akár más kifejezésben. Mindkét mód léptet tehát, viszont amíg az `i++` az azt tartalmazó kifejezésbe a léptetés előtti értéket helyettesíti be, addig a `++i` a már megnöveltet!

4.3.3. Értékadás tömören

A gyakorlás megkezdése előtti utolsó táblázat következik, melyben egymás mellett láthatóak a legalapvetőbb értékadások normál és tömör formái, sorról sorra. Íme:

Normál mód	**Tömör változat**
a=a+b	a+=b
a=a−b	a−=b
a=a∗b	a∗=b
a=a / b	a /=b
a=a ^b	a ^=b

A továbbiakban pedig – tekintve, hogy meglehetősen hosszú ideig tartottam fejeteket az elmélet tengerének felszíne alatt – ideje, hogy lélegzethez jussatok végre, s a kifejtetteket elmélyítendő, gyakoroljatok, ezáltal megtanulván alkalmazni is az eddig oly hosszan vázoltakat!

4.4. Gyakorlás

Átmozgatás gyanánt kezdjük azzal, hogy önállóan megírjuk azt a pici programocskát, amely egyről indulva, egymás alá, egyesével növekedő módon oszlopba rendezi a számokat, egészen a bekért pozitív `n` számig, mellyel aztán zárja is a sort (oszlopot), s egyúttal egy tőle „tab-nyi” távolságban lévő újabb oszlopban – a megfelelő sorokban – az előbbi oszlopot alkotó számok második hatványát jeleníti meg! Ez volt a mondás akkor tájt, rögtön azután, hogy átírtuk a léptetés módját:

```
#include<stdio.h>

main()
{int n, i=0;
 printf("n␣erteke:␣");
 scanf("%d", &n);

 while(++i<=n)
 {
  printf("%d\t%d\n", i, i*i);
 }
}
```

Feladat: írjuk át a fenti példát úgy, hogy az `i` 1-et kapjon kezdőértéknek, viszont a léptetés továbbra is a feltételben maradjon!

Lehetséges megoldás:

```
#include<stdio.h>

main()
{
 int n, i=1;
 printf("n␣erteke:␣"); scanf("%d", &n);

 do
 {
  printf("%d\t%d\n", i, i*i);
 }while(++i<=n);
}
```

Feladat: írjuk át for ciklusba a fenti programot!

Lehetséges megoldás:

```
#include<stdio.h>
main(){
 int n, i;
 printf("n␣erteke:␣"); scanf("%d", &n);
 for(i=1;i<=n;i++){printf("%d\t%d\n", i, i*i);}
}
```

Feladat: Kegyelemdöfés gyanánt, írjuk át a fenti programot oly módon, hogy most egyfelől csak az első oszlopot pötyögtesse teli, mégpedig az `i` aktuális értékeit egymás alá írogatva, másfelől pedig ne kérje be `n` értékét, hanem addig fusson a ciklus, amíg az `i` túl nem csordul! Ekkor a vezérlés lépjen ki a ciklusból, s írja ki `i` értékét! Tegyük meg mindezt kizárólag elemi eszközöket igénybe véve!

Lehetséges (rettenetesen ocsmány) megoldás:

```
#include<stdio.h>

main()
{
 int i, n=1;

 for(i=1;n;i++)
 {
 if(i<0)
  {
  printf("Megtelt!\n"); break;
  }else
       {
        printf("%d\n", i);
       }
  }
 printf("i=%d\n", i);
}
```

A fenti kódban, egy feltétel segítségével láthatóan azt vizsgáltuk, hogy `i` értéke – a túlcsordulás hatására – mikor csap át a negatív tartományba. Maga a ciklus feltétele ezúttal csak az `n`-t – illetve annak értékét – tartalmazta, ami `1`, s mely érték, mint tudjuk, *igaz* logikai értéknek minősül, hiszen nem `0`, magyarán a ciklus, az általa tartalmazott `if` hiányában örökké futna, a vezérlés sosem lépne ki belőle. A fenti `break`-kel kapcsolatban fontos megjegyezni, hogy az csak abból

a szerkezetből ugratja ki a vezérlést, amelyben maga is szerepel. Esetünkben ez a `for` ciklus volt. Kettő egymásba ágyazott ciklus esetén például, ha a belsőben van a `break`, akkor csak a belső ciklusból lép ki a vezérlés. (Ha nem győzzük kivárni a túlcsordulást, átírhatjuk a `i++`-t mondjuk `i+=100000`-re.)

Szemfülesebb hallgatóink – továbbra is kizárólag elemi eszközöket igénybe véve – találhatnak persze kevésbé "szószátyár" kódot tartalmazó megoldást is, például, valami ilyesmit:

```
#include<stdio.h>

main()
{
 int i, n=0;

 for(i=1;i>n;i++);
 printf("Megtelt!\n");
 printf("i=%d\n", i);
}
```

(Itt is átírhatjuk a `i++`-t, `i+=100000`-re.)

Feladat: írjunk programot, mely bekéri egy háromszög oldalainak hosszát, azokat paraméterként átadja egy függvénynek (eljárásnak), mely megvizsgálja, hogy megszerkeszthető-e az adott méretekkel büszkélkedő síkidom, majd közli velünk, hogy mire is jutott!

Lehetséges megoldás:

```
#include <stdio.h>

void haromszog(float d, float e, float f)
{

 if(d+e>f && d+f>e && e+f>d)
 {
  printf("A haromszog megszerkeszthető.\n");
 }
  else
     {
      printf("Ilyen haromszog nem letezik.\n");
     }
}
```

```
main()
{
 float a, b, c;
 printf("Oldalak␣hossza␣(a,b,c).\n");
 scanf("%f%f%f", &a, &b, &c);
 haromszog(a,b,c);
}
```

Feladat: írjunk programot, amelyen belül egy függvény meghatározza a billentyűzetről általa beolvasott `n` darab szám közül a legnagyobbat, melyet mint visszatérési értéket átad a `main`-nek, ami ki is írja azt!

Lehetséges megoldás:

```
#include <stdio.h>

float maximum(int m)
{
 int i;
 float max, szam;
 printf("1.␣szam:␣"); scanf("%f", &max);

 for(i=2;i<=m;i++)
 {
 printf("%d.␣szam:␣", i); scanf("%f", &szam);
 if (szam>max){max=szam;}
 }
 return(max);
}

main()
{
 float legnagyobb;
 int n;
 printf("Mennyi␣szamot␣fog␣beirni?␣");
 scanf("%d", &n);
 legnagyobb=maximum(n);
 printf("%1.1f␣volt␣a␣legnagyobb.\n" , legnagyobb);
}
```

Természetesen az alábbi függvény is megfelel, csak arra ügyeljünk, hogy az `if` feltételében, az `i==1` résznél 2 db = jelet írjunk, mivel itt az *egyenlőség vizsgálatáról*, s nem értékadásról van szó! (Tessék csak kipróbálni elrettentő példaként, mi lesz, ha csak egy = jelet alkalmazunk!)

Lehetséges megoldás:

```
float maximum(int m)
{
 int i; float max, szam;
 for(i=1;i<=m;i++)
 {
 printf("%d. szam: ", i); scanf("%f", &szam);
 if (i==1 || szam>max){max=szam;}
 }
 return(max);
}
```

Feladat: Módosítsuk függvényünket úgy, hogy jelezze, ha több, mint egy darab maximum érték van a beadott számok között, továbbá írja ki azt is, hogy mennyi volt belőlük!

Lehetséges megoldás:

```
float maximum(int m)
{
 int i, maxdb=1;
 float max, szam;
 printf("1. szam: "); scanf("%f", &max);

 for(i=2;i<=m;i++)
 {
  printf("%d. szam: ", i); scanf("%f", &szam);
  if(szam==max){maxdb++;}
  if (szam>max){max=szam; maxdb=1;}
 }

 if(maxdb>1)
 {
  printf("%d db ", maxdb);
  printf("maximum volt es ");
  /*"%d volt a legnagyobb."
  (main-ben printelve)*/
 }
 return(max);
}
```

vagy egy másik megoldás:

```
float maximum(int m)
{
 int i, maxdb=1; float max, szam;

 for(i=1;i<=m;i++)
 {
 printf("%d. szam: ", i); scanf("%f", &szam);
 if(szam==max && i!=1){maxdb++;}
 if (i==1 || szam>max){max=szam; maxdb=1;}
 }

 if(maxdb>1){/*a tobbi ugyanaz...*/}
 return(max);
}
```

Feladat: írjunk programot, mely miután bekéri a kamatoztatás végett betenni kívánt összeget, és a kamatlábat, egy hívott függvény (eljárás) segítségével megmondja, hogy az adott év növekménye hányadik évben haladja meg a legelső év elején betett összeg 30 százalékát! *Kitétel:* Ciklus alkalmazásával oldjuk meg a feladatot (valamint `#include<math.h>` nélkül, mivel úgy elég lenne egy képlet, de most itt ciklusokat gyakorlunk)!

Lehetséges megoldás:

```
#include<stdio.h>

void kamatozo(float be, float r)
{
 float be30=be*0.3; int ev=1;

 while(be*(1+r/100)-be<=be30){
  be=be*(1+r/100); ev++;
 }
 printf("%d. evben haladja meg a novekmeny ", ev);
 printf("az alaposszeg 30%%-at.\n");
}

main(){
 float betett, kamatlab;
 printf("Betett osszeg: "); scanf("%f", &betett);
 printf("Kamatlab: "); scanf("%f", &kamatlab);
 kamatozo(betett, kamatlab);
}
```

Feladat: írjunk programot, melyben egy függvény kiír a képernyőre egy tetszőleges $n*m$-es szorzótáblát és az utolsó szorzat eredményét átadja a `main`-nek, ahol az kiírásra kerül!

Lehetséges megoldás:

```
#include <stdio.h>

int szorzo()
{
 int n, m, i, j;
 printf("n? m?\n");
 scanf("%d%d", &n, &m);

 for(i=1;i<=n;i++)
 {
  printf("\n");
   for(j=1;j<=m;j++)
   {
    printf("%d*%d=%d\t",i,j,i*j);
   }
  printf("\n");
 }
 return --i*--j;
 /*Azert van itt --i es --j, mert i es j mar
 tulment n es m erteken, igy meg a szorzas
 elvegzese elott kell csokkenteni oket*/
 }

main()
{
 printf("\n%d volt a visszateresi ertek.\n", szorzo());
}
```

(No igen, kicsit talán túltoltuk a dolgot a függvény javára. :))

Feladat: írjunk programot, mely bekér egy `a` és egy `b` számot, s abban az esetben, ha `a` nagyobb, mint `b`, kicseréli a kettő értékét!

Lehetséges megoldás:

```
#include <stdio.h>

main(){
 float a, b, segedv;
```

```
  printf("a?␣b?\n"); scanf("%f%f", &a, &b);
  if(a>b)
  {
   segedv=b;
   b=a;
   a=segedv;
   printf("Mostantol␣a=%1.2f␣es␣b=%1.2f\n", a, b);
   }else
   {
    printf("Tovabbra␣is␣a=%1.2f␣es␣b=%1.2f.\n", a, b);
   }
}
```

Természetesen felmerülhet a kérdés, hogy vajon valóban feltétlenül szükséges-e ilyen esetben egy harmadik – ideiglenes, segéd- – változó igénybevétele? Nem „férne bele” a csere az eredeti két változónkba? Nos, ha az ember egy kicsit elmereng a dolgon, könnyedén rájöhet, hogy megoldható a csere kizárólag a két eredeti változó (`a` és `b`) felhasználása által is. A teendőnk mindössze annyi, hogy a fenti programban látható

```
   segedv=b;
   b=a;
   a=segedv;
```

sorokat az alábbiakkal váltjuk ki:

```
a=a-b;
b=a+b;
a=b-a;
```

Ez a megoldás rém egyszerű ugyan, mégsem nagyon találkozhatunk vele, aminek valószínűleg az lehet az oka, hogy talán a folyamat szempontjából hatékonyabb egy új változó bevezetése – még ha több helyet is foglalunk így a memóriában – , mint a többlet műveletek elvégzése a kizárólag két változóval dolgozó megoldás esetén.

Feladat: Írjunk egy olyan programot, amely segít döntenünk a bankban! Tegyük fel, hogy a kamatozás módja állít minket választás elé, ugyanis míg az egyik esetben a számlánkon nyilvántartott összeget az eredetileg betett összeg adott százalékával növeli évente a bank (nevezzük ezt most fix többletnek), addig a másikban, az éves növekmény mindig a már a számlánkon addig összegyűlt összeg adott százalékától függ (ez utóbbi a kamatos kamat).

A programnak, feladata szerint – miután közöltük vele a két esethez tartozó ka-

matlábak értékét, a betenni kívánt összeget, valamint a lekötés időtartamát – ki kell tudni számolni, majd közölni velünk a lekötés, általa javasolt jövedelmezőbb formáját! (A feladatot most is ciklus alkalmazásával oldjuk meg, viszont ezúttal nem kötelező függvényt írnunk hozzá (leszámítva persze a `main`-t :))!

Lehetséges megoldás:

```
#include<stdio.h>

main()
{

float betett, klfix, klkk, bfk, bfknov;
int ido, i;

printf("Betett␣osszeg:␣"); scanf("%f", &betett);

printf("Fix␣tobblethez␣tartozo␣kamatlab:␣");
scanf("%f", &klfix);

printf("Kamatos␣kamathoz␣tartozo␣kamatlab:␣");
scanf("%f", &klkk);

printf("Lekotes␣idotartama:␣");
scanf("%d", &ido);

bfk=betett; bfknov=bfk*klfix/100; i=0;


while(++i<=ido)
{
 bfk+=bfknov; betett*=(1+klkk/100);
}
/*Ezt persze nyilvan lehet maskeppen is, pl igy:

for(bfk=betett, bfknov=bfk*klfix/100, i=0;
++i<=ido; bfk+=bfknov, betett*=(1+klkk/100));

;-)*/

if(bfk>betett)
        {
        printf("A␣fix␣tobblettel␣jar␣jobban.\n");
        }
```

```
        else
        {
         printf("Inkabb␣a␣kamatos␣kamat!.\n");
         }

}
```

Feladat: írjunk programot, mely tartalmaz egy függvényt, ami kiszámolja a futásidőben bekért sugárral rendelkező kör területének – szintén a futásidőben bekért finomságú „pontosságtól" függő – közelítő értékét, úgy mégpedig, hogy a számoláshoz nem használja a π számot, illetve annak kerekített értékét sem! Az `#include<math.h>` viszont, ahol az `sqrtf` és a `powf` függvény is lakik, már játékba hozható. Az `sqrtf` négyzetgyököt von a paraméterül kapott számból, míg a `powf` az első paramétert, a másodikként beírt értékkel megegyező kitevőre emeli. Mindkettő `float` típust dob vissza.
Fontos: a `math.h`-ban lakó függvények akkor használhatók/hívhatók, ha "belevesszük" a programunkba az `m` könyvtárat is a fordítás során. Úgy tehetjük ezt meg, hogy igénybe vesszük a `gcc -lm` kapcsolóját. Így, ha pl a forrás neve `terulet`, s így szeretnénk elnevezni a futtatható állományunkat is, plusz még "képbe akarjuk hozni" az `m` könyvtárat is, akkor az alábbiakat kell begépelnünk a parancssorba:

```
gcc terulet.c -lm -o terulet
```

Lehetséges megoldás:

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

float terulet(float r, float dx)
{
 int i=1; float t=0;
 while(i*dx<=r)
 {
  t+=sqrtf(powf(r,2)-powf(i++*dx,2))*dx*4;
 }

/*or:
 for(;i*dx<=r;t+=sqrtf(powf(r,2)-powf(i++*dx,2))*dx*4);
*/

 return(t);
}
```

```
main()
{float sugar, felbontas;
 printf("Sugar es 'pontossag' (<<1, de nagyobb, mint 0): ");
 scanf("%f%f", &sugar, &felbontas);
 printf("Terulet: %1.2f\n", terulet(sugar, felbontas));
}
```

Megjegyzés: a fentebb megadott megoldás kódjában a ciklus szándékosan ilyen tömör, mivel itt a feladat – azok számára, akik nem találtak ki saját megoldást – épp az, hogy a kódot tanulmányozva *megértsék*, hogy miként működik az. Normál esetben – az áttekinthetőség végett – természetesen célszerű "felhigítani" egy kicsit a ciklus kódját.

Feladat: Írjunk programot, mely miután bekéri Jancsi és Juliska – négyszögletű kerek erdőben érvényes – Descartes-koordinátáit, hív egy függvényt, mely kiszámolja a kettőjük távolságával megegyező oldalhosszúságú négyzet területét, amit – miután megkapta azt – a `main` 3 tizedes jegy pontossággal ki is ír a képernyőre! *Kitétel:* `#include<math.h>` nem játszik!

Lehetséges megoldás:

```
#include<stdio.h>

float jj(float a, float b, float c, float d)
{
 return((a-c)*(a-c)+(b-d)*(b-d));
}

main()
{
 float a, b, c, d, area;
 printf("Jancsi x, y koordinatai?\n");
 scanf("%f%f", &a, &b);
 printf("Juliska x, y koordinatai?\n");
 scanf("%f%f", &c, &d);
 printf("A negyzet terulete= %1.3f egyseg.\n", jj(a,b,c,d));
}
```

5. fejezet

Rekurzió

Az alábbiakban – ahogy tettük azt már eddig is – továbbra is gyakorolni fogunk, viszont mielőtt nekikezdenénk, úgy vélem mindenképp el kell magyaráznom azt, amit az órákon általam itt-ott már megemlített *rekurzió* fogalma takar. Abból kiindulva, hogy egy függvényből hívhatunk egy másik függvényt, az emberben – már puszta érdeklődésből is – felvetődhet a kérdés, hogy mi történne akkor, ha egy függvényből nem egy másikat, hanem önmagát hívnánk (...esetleg – közvetett módon – egy függvényben hívott függvényen belül kerülne sor az eredeti függvény hívására..). Van egyáltalán értelme az ilyesminek? (Elvégre feltehetőleg több erőforrást igényel, mint egy – szintén valamilyen műveletismétlést végrehajtó – ciklus beüzemelése, mivel a függvény minden egyes újbóli hívásakor, annak lokális (s egyben ideiglenes) változói számára helyet kell, hogy foglaljon a vezérlés. Ciklus esetén – mint tudjuk – nincs ilyen probléma.)

Nos, a fenti kérdésre a válasz egyértelműen: igen. Azért mégpedig, mert – ha most még nem is, de – később, már haladó programozóként könnyen belefuthatunk olyan feladatba, melynek kezelése egyszerűbb rekurzió igénybevétele által, mint anélkül. Tipikusan ilyen feladat lehet például egy fastruktúra (ami egy még nem tanult adatszerkezet) bejárása. Ezért jó hát, ha még most, idejekorán – legalább alapszinten – megismerkedünk ezzel a módszerrel.

A rekurzió működése szépen szemléltethető a már jól ismert, alaposan körüljárt,

faktoriálist számító függvény megújítása által. Írjunk hát egy kis programot, melyben egy rekurzív függvény számítja ki a bekért szám faktoriálisának az értékét!
Lehetséges megoldás:

```
#include<stdio.h>

int faktor(int m)
{
 if(m>=2){return(m*faktor(m-1));}
 else{return 1;}
}

main()
{
 int n;

 do{
    printf("n?␣(Nemnegativ!)␣");
    scanf("%d", &n);
    }while(n<0);

 printf("%d␣faktorialisa␣%d.\n", n, faktor(n));
}
```

Látható, hogy a `faktor` függvény a visszatérésre adott `return` utasítás ”hasában” újra és újra hívja önmagát a `return(m*faktor(m-1));` parancsnak megfelelően, így a `return`-höz érve ahelyett, hogy befejezvén a működését visszaadna egy értéket a hívónak, újra akcióba lendül, mégpedig úgy, hogy a kezdetben kapott paraméter értékét szorozza valamivel.

Ez a valami, a kezdetben kapott paraméter értékének, új hívás által sorra kerülő (itt csökkentett) értéke lesz. Mindez addig ismétlődik, amíg, az `if`-nek hála, végre fellélegezhet, s nem kell magát újra hívnia – hanem az 1-es számmal lezárván az addigi ”szorzásfolyamot” – egy konkrét számot dobhat vissza a `main`-beli `printf`-nek, ahol eredetileg hívtuk.

Egyelőre – számos kapcsolódó ismeret tárgyalását mellőzve, kezdet gyanánt – ennyit említenék a rekurzióról.

(Vegyük azért észre a `main`-ben szerénykedő `do{}while()`-t is aminek hála, „felhasználóbiztos” módon működik a bekérés, mivel csak nemnegatív számok szivároghatnak át a feltétel által reprezentált szűrőn.

Most pedig pötyögjük végig az alábbi feladatokat, s ha lehet, törekedjünk arra,

hogy **saját** kútfőből merítve alkossuk meg azok megoldásait! Jöjjenek hát a további

5.1. gyakorló feladatok!

Feladat: Írjunk egy programot, ahol egy függvény `n` darab szám bekérése után kiszámolja azok átlagát, majd a kapott eredményt visszadobja a `main`-be, ahol az megjelenítésre kerül!

Lehetséges megoldás:

```
#include<stdio.h>

float atlag(int m)
{
 int i=1;
 float szam, osszeg=0;

 while(i<=m)
 {
 printf("%d. szam: ", i++);
 scanf("%f", &szam);
 osszeg+=szam;
 }

 return osszeg/m;
}

main()
{
 int n;
 printf("Mennyi szam lesz? ");
 scanf("%d", &n);
 printf("Az atlaguk: %1.2f\n", atlag(n));
}
```

Ezúttal így gondoskodtunk a ciklusváltozó növeléséről:
`printf("%d. szam: ", i++);`
(ily módon is megemlékezve a léptetéssel kapcsolatos tudnivalókról).

Feladat: Írjunk egy programot, ahol egy függvény/eljárás egy nxn-es négyzet –

képernyő szerinti – jobb felső és bal alsó sarkát összekötő átlója feletti részét teleszórja csillagokkal (* jelekkel)!

Lehetséges megoldás:

```
#include<stdio.h>

void csillag(int m)
{
 int i, j;
 for(i=0;i<m;i++)
 {
  for(j=0;j<m-i;j++){printf("*");}
  printf("\n");
 }
}

main()
{
int n;
 printf("Hany karakter legyen a negyzet oldala? ");
 scanf("%d", &n);
 csillag(n);
}
```

Természetesen számos, különböző módon is kitekerhetjük a megoldás nyakát, például a belső ciklus fejébe `for(j=0;j<m-i;j++)` helyett
`for(j=-i;0<m+j;j--)` -t is írhatunk, stb, stb.

Feladat:
Írjuk át a fenti eljárást úgy, hogy most a bal felső és jobb alsó sarkok közötti átló alatt tevékenykedjen!

Lehetséges megoldás (csak az eljárás, s abban is csak a belső ciklus feje, mert a többi megegyezik):
`for(j=0;j<=i;j++)`

A feladat, mint mindig, most is többféleképpen megoldható.

Feladat:
Írjunk egy programot, ahol egy eljárás kiírja egy 2-nél nagyobb egész szám osztóit és azok átlagát!

Lehetséges megoldás:

(Ami persze bőven optimalizálható még, hisz ebben az esetben elegendő a szám négyzetgyökéig vizsgálódni.)

```
#include<stdio.h>

void oszto(int m)
{
 int i, j; float osszeg;

 for(i=1,j=0,osszeg=0;i<=m;i++)
 {
  if(m%i==0)
  {
   printf("\n%d", i);
   osszeg+=i;
   j++;
   }
 }

 printf("\n\nAZ osztok atlaga: %1.2f\n", osszeg/j);
}


/*A javitott valtozat
(amihez viszont mar szukseges a math.h betoltese is):
void oszto(int m)
{
 int i, j; float ossz, gyokm=sqrtf(m);

 for(i=1, j=0, ossz=0;i<gyokm;i++)
        {
         if(m%i==0)
         {
          printf("%d\t%d\n", i, m/i);
          ossz+=i+m/i; j+=2;}
        }

 if(i==gyokm){printf("%d\n", i); ossz+=i; j++;}
 printf("\n\nAz osztok atlaga: %1.2f.\n", ossz/j);
}*/


main()
```

```
{
 int n;
 do
 {
  printf("Vizsgalando szam: "); scanf("%d", &n);
  if(n<=2){printf("2-nel nagyobb kell!\n");}
 }while(n<=2);

 oszto(n);
}
```

A fenti kódban, a `m%i==0` kifejezésben látható `%` jel az `m` és az `i` hányadosának *maradékát* képezi (Például: `10%8=2`, `10%5=0`, stb.) (A `math.h`-val kapcsolatban emlékezzünk rá, hogy ilyenkor a `gcc`-nél használni kell a `-lm` -et is!) Világos, hogy elegendő csak az adott szám négyzetgyökéig vizsgálódnunk. Ezt valósítja meg az `oszto` javított változata.

Feladat:
Földönkívüli intelligenciára vadászván, olyan csillagközi jeleket keresünk, melyeken keresztül prímszámok érkeznek hozzánk, ezúton sem hagyván kétséget bennünk az adás mesterséges eredetét illetően. Megkönnyítendő az észlelést, írjunk egy olyan programot, melyben egy eljárás kiírja az adott, zárt intervallumban található prímszámokat![1] **Mindenesetre kezdjük ezúttal rendhagyóan a megoldást!** Első körben elég, ha megmondjuk egy beadott számról, hogy prímszám-e (csak eggyel és önmagával osztható természetes szám, kivéve az 1-et) vagy sem.

Egy lehetséges (nem túl hatékony) megoldás:

```
#include<stdio.h>

void prima(int m)
{
 int i;
 for(i=2;i<m;i++)
 {
  if(m%i==0){break;}
 }
 if(i==m){printf("Prim.\n");}
 else{printf("Nem prim.\n");}
}
```

[1] Oké, rendben, ez nem pont azt tenné, amire a csillagászok áhítoznak, mivel ők prímszámokból álló sorozatoknak örülnének, de akkor is, tessék megoldani a feladatot! :-)

```
main()
{int n;
 do{printf("Vizsgalando␣szam␣(>=2):␣");
 scanf("%d", &n);}while(n<2);
 prima(n);
}
```

Fejlesszük tovább a rutinunkat, hiszen elegendő, ha csak a szám négyzetgyökéig vizsgálódunk, ami – minél nagyobb a szám, relatíve annál – kevesebb lefutást tesz szükségessé a fenti ciklus esetében!

Egy lehetséges (az előzőnél hatékonyabb) megoldás:
Csak a ”teteje” változott, így itt most csak azt mutatom meg újra:

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

void prima(int m)
{
 int i; float gyok=sqrtf(m);

 for(i=2;i<gyok;i++)
 {
  if(m%i==0){break;}
 }

 if(i>gyok){printf("Prim.\n");}
 else{printf("Nem␣prim.\n");}
}
```

Természetesen ez még tovább javítható, hiszen amennyiben a szám négyzetgyöke egész, akkor már eleve a ciklust sem kell ”bepörgetni”, mivel biztosan tudható, hogy a szám nem prímszám. Magától értetődne a `if(gyok%1==0)` feltételvizsgálat, ám itt a C#-os módszer nem segít, ugyanis ez a kifejezés ott működik, itt viszont már (még) sajnos nem.

A C nyelv ugyanis megköveteli, hogy a `%`-jel két egész típus közé legyen „beszendvicselve”, ami ebben az esetben gondot okoz, hiszen a `gyok` változó lebegőpontos szám, mivel az `sqrtf()` függvény annak ”dobta” vissza. A dolgot a `modf` függvénnyel kerülhetjük ki, ami képes rá, hogy egy lebegőpontos számnak megadja az egészrészét, valahogy így:

.
.

```
float lebeg=2.89; double egeszresz;
.
modf(lebeg, &egeszresz);
.
printf("%f", egeszresz);
.
.
```

A `double`, a `modf` és a `&` részekre külön ügyelni kell! A függvény első paramétere maga a lebegőpontos szám – `lebeg` – lesz, a második pedig az a *double* típusú lebegőpontos változó – `double egeszresz` –, ahová a függvény beírhatja a `lebeg` értékének egészrészét. A fenti `printf` eredménye a 2.000000 szám kiírása lesz.

Az eddigiek `modf`*-fel bővített, még hatékonyabb változata:*

```
void prima(int m)
{
 int i; float gyok=sqrtf(m); double egesz;
 modf(gyok, &egesz);
 for(i=2;i<gyok;i++)
 {
   if(gyok-egesz==0 || m%i==0){break;}
 }

 if(i>gyok){printf("Prim.\n");}
 else{printf("Nem prim.\n");}
}
```

Kérdés: lehet ezt még vajon tovább ragozni? Igen is, meg nem is :-) , ugyanis amennyiben a vizsgált szám nagyobb, mint 2 és még páros is, úgy kizárt, hogy prím szám legyen. Mindössze annyi lenne tehát a teendőnk, hogy a fenti ciklusban lévő feltételt kiegészítjük, valahogy így:

```
if(gyok-egesz==0 || m%i==0 || m%2==0){break;}
```

De vajon tényleg szükség van erre a fenti kódban? Nos, nem igazán, hiszen a ciklusváltozó (az `i`), már eleve 2-ről indul, így nincs szükség az `m%2==0` -ra, mert azt magába foglalja az `m%i==0` rész is, amikor `i` még épp 2.

Most pedig, hogy ilyen prímán megy már a prímszámtesztelés, ideje, hogy rátérjünk az *eredeti feladat megoldására*, avagy egy adott, zárt `[a, b]` intervallumban lakó prímszámok kiírására.
Lehetséges megoldás:

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

```
#include<math.h>

void prima(int c, int d)
{
 int i, j; float gyok; double egesz;

 for(i=c;i<=d;i++)
 {
  gyok=sqrtf(i);
  modf(gyok, &egesz);

   for(j=2;j<gyok;j++)
   {
    if(gyok-egesz==0 || i%j==0){break;}
   }

  if(j>gyok){printf("\n%d\n",i);}
 }
}

main()
{
 int a, b;
 do
 {
 printf("Intervallum eleje(>=2) es vege(>=2)?\n");
 scanf("%d%d", &a, &b);
 }while(a<2 || b<2);

 prima(a,b);
}
```

Örvendezhetnek hát csillagász barátaink, bár kicsit belegondolva rájöhetünk, hogy talán eleve ”lefutott ügyről” van szó, hiszen ”**az a legbiztosabb jele annak, hogy létezik intelligens élet a Földön kívül, hogy még nem próbáltak kapcsolatba lépni velünk**”. ;-) *Megjegyzés:* természetesen több módon is megfogalmazhatnánk a fenti kis algoritmust megvalósító kódot, de talán érdemes lehet áttanulmányozni az alábbi megvalósítási módot is:


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

void prima0(int e)
{
 int i; float gyok; double egesz;
 gyok=sqrtf(e); modf(gyok, &egesz);

   for(i=2;i<gyok;i++)
   {
       if(gyok-egesz==0 || e%i==0){break;}
    }

 if(i>gyok){printf("\n%d\n",e);}
}

void prima(int c, int d)
{
  int i;
  for(i=c;i<=d;i++){prima0(i);}
}
```

```
main()
{
 int a, b;
 do
 {
  printf("Intervallum␣eleje(>=2)␣es␣vege(>=2)?\n");
  scanf("%d%d", &a, &b);
 }while(a<2 || b<2);

 prima(a,b);
}
```

Világos, hogy itt is két ciklus ágyazódik egymásba, ám ezt most úgy valósítottuk meg, hogy a `prima()` függvény ciklusában hívtuk a `prima0()` függvényt, mely szintén egy ciklust üzemeltetett, ily módon ő maga vált a `prima()` függvény ciklusának belső ciklusává.

Mindenképp érdemes kitérnünk arra az apróságra is, hogy ha esetleg a `prima0()` függvényt a `prima()` ”alá” helyeznénk a forrásban, valahogy így:

```
void prima(int c, int d)
{
 blablabla
}

void prima0(int e)
{
 blablabla
}

main()
{
 blablabla...
}
```

akkor könnyen lehet, hogy a fordító – legalábbis, amelyik példának okáért a `c99`-es dialektust beszéli csak – a forrás vonatkozó helyének megjelölése után valami olyasmit reklamálna, hogy ”*warning: conlicting types for 'prima0' [enabled by default]*”, illetve ’*note: previous implicit declaration of 'prima0' was here*”.

Ennek az az oka, hogy a függvényeket nem a ”megfelelő sorrendben” írtuk meg, így a fordító – a kódban haladván – olyan függvény hívásával szembesült (a `prima()`-ban a `prima0()` hívásával), amivel addig még nem ”találkozott” a kódban, így nem tudta megvizsgálni például azt sem, hogy a hívott függvénynek

(a `prima0()`-nak) átadott paraméter típusa passzol-e ahhoz, amire a függvényt felkészítettük... Talán még emlékszünk rá, hogy kitértünk már erre e jegyzet elején is, amikor az alábbi módon igyekeztem felhívni a figyelmet a szóban forgó problémára:

"**Nagyon fontos, hogy a függvényt mindig annak felhasználása előtt hozzuk létre!!!** *Azért fontos ez, mert a fordító meg szeretné vizsgálni a függvény paramétereinek típusát, azért mégpedig, hogy egyeztesse azokat a híváskor átadott változók típusával.*" Nos, itt épp ezt a szabályt sikerült megszegnünk...

Nyilván eléggé nyögvenyelős lehet egy több – egymást is hívogató – függvényből álló kód írásakor még azzal is bajlódni, hogy a függvényeink megfelelő sorrendben legyenek a kódba illesztve, ezért természetesen felvetődik a kérdés, vajon nincs-e rá mód, hogy ettől a kellemetlenségtől megszabaduljunk valamiképpen?

Nos, a válasz az, hogy természetesen van, mégpedig a – segédletünk elején – már felvázolt módon, melynek kapcsán "szétszedtük" a függvényt, úgy mégpedig, hogy a `main()` "fölött" csak a függvény "fejét" írtuk meg, míg a teljes leírását a `main()` alatt követtük el. Akkor azt írtam, hogy "*Igazából mindkét út járható, de jó, ha tudjuk, hogy vannak bizonyos előnyei a fentebb látható módszernek. Arról, hogy pontosan mik is ezek az előnyök, majd valamikor később ejtek szót...*"

Úgy vélem, akár most is szerét ejthetném ennek a rövid mesének, azzal a megjegyzéssel, hogy nem csupán az alábbiak miatt előnyös a függvényeket így (mármint, úgymond: "szétszedve") megírni. (A többi előnyt szintén később fejteném ki valamikor, egyelőre elég, ha ezt, itt, most megértjük.)

Tehát, amennyiben a függvények "fejeit" a `main()` "fölé" tesszük, a leírásukat pedig "alá", akkor mentesülünk a sorrendre való figyelés kényszere alól, hiszen a fordító minden függvény esetén megkapja a bemenő paraméterekre, illetve a visszatérési értékre vonatkozó típusokat, így azokat már képes lesz összevetni a `main()`-ben, illetve a többi függvényben található hívások alkalmával átadott paraméterek típusaival, s ugyanezt megteheti a visszatérési értékek esetében is, már, ha vannak egyáltalán. Ugyanis, mint azt valahol már e segédlet elején leírtam: "*a fordító kizárólag arra kíváncsi, hogy a függvény paramétereinek típusa összeegyeztethető–e a hívó állandóival, változóival.*"

Ebből az következik, hogy a függvények paramétereinek a `main()` "fölött" való "megemlítése" által, sikerült megnyugtatnunk a fordítót, így a `main()` "alatt" már kedvünk szerinti sorrendben helyezhetjük el a függvényeink leírását. Nosza, éljünk hát ezzel a lehetőséggel! Ez esetben a fenti kis kódunk az alábbi formát ölti, ahol már nem számít milyen sorrendben követik egymást a függvények leírásai:

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

void prima(int c, int d);
void prima0(int e);

main()
{
 int a, b;
 do
 {
  printf("Intervallum␣eleje(>=2)␣es␣vege(>=2)?\n");
  scanf("%d%d", &a, &b);
 }while(a<2 || b<2);

 prima(a,b);
}


void prima(int c, int d)
{
  int i;

  for(i=c;i<=d;i++)
   {
    prima0(i);
    }
  }



void prima0(int e)
{
 int i; float gyok; double egesz;
 gyok=sqrtf(e); modf(gyok, &egesz);

  for(i=2;i<gyok;i++)
   {
    if(gyok-egesz==0 || e%i==0){break;}
   }

  if(i>gyok){printf("\n%d\n",e);}
}
```

Feladat:
Írjunk egy progit, amiben egy eljárás kiírja ama hely Descartes-féle (x, y) koordinátáit, melyhez közeledve az $y = x^2$ és az $y = 1/x$ függvények görbéi egyre közelebb kerülnek egymáshoz! Kitétel: `math.h` nem játszik, numerikusan közelítve, ciklussal oldjuk meg a feladatot! A közelítés "finomságát" – egy tizedes törtet – bekérhetjük akár a `main`-ben is.

Lehetséges megoldás (ha "balról közelítünk", mert tudjuk, hogyan néznek ki a görbék):

```
#include<stdio.h>

void kozelito(float dx)
{
 int i;
 for(i=1;1/(i*dx)>(i*dx)*(i*dx);i++);
 printf("x=%1.2f, y=%1.2f\n", i*dx, i*dx*i*dx );
}

main()
{
 float a;
 printf("Kozelites felbontasa? (tizedes tort)\n");
 scanf("%f", &a);

 kozelito(a);
}
```

Ha nem tudjuk, hogy hogyan néznek ki a görbék, akkor nyilván próbálkoznia kell a programnak, például akár úgy is, hogy megnézi egy helyen a görbék y értékeinek a különbségét, majd attól balra és jobbra is megteszi ugyanezt, s amerre csökken ez a különbség, arrafelé kezd el "oldalazni" a ciklus. Természetesen lehetne még tovább is általánosítani a program működését (pl.: az x tengely nempozitív részével is kellene "kezdeni valamit" vagy más, tetszőleges lefutású függvényekre kiterjeszteni az eljárást), ezt ki-ki belátása szerint megteheti, ha tovább szeretne gyakorolni.

Feladat:
Írjunk egy programot, ahol a `main` bekéri egy pont két koordinátáját (Descartes-rendszer: (x, y)) és egy origó középpontú kör sugarát, majd ezeket átadja egy függvénynek, ami visszatérési értékként megadja a pont és a kör egymástól való távolságát!
Lehetséges megoldás:

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

float tav(float r, float x, float y)
{
 return(sqrt(x*x+y*y)-r);
}

main(){
 float sugar, x, y;
 printf("Sugar? x? y?\n");
 scanf("%f%f%f", &sugar, &x, &y);
 printf("Kor-pont tavolsag: %1.2f.\n", tav(sugar, x, y));
}
```

Feladat:
Írjunk egy programot, amiben egy függvény a neki paraméterül adott pozitív számot egy Descartes-féle rendszer x koordinátájaként értelmezi, majd tetszőleges felbontással kiszámolja az $y = x^2$ függvény görbéje alatti területnek, az $x = 0$ és a beadott x érték közé eső részét, amit visszaad a `main`-nek! Kitétel: `math.h` nem használható, ahogy az ismert analitikus formulák sem, stb, stb.

Lehetséges megoldás:

```
#include<stdio.h>

float terulet(float x, float dx)
{
 int i; float ter=0;
 for(i=1;i*dx<=x;i++)
  {
   ter+=((i*dx)*(i*dx))*dx;
  }
 return(ter);
}

main(){
 float x, dx;
 printf("x? Felbontas?\n"); scanf("%f%f", &x, &dx);
 printf("Az x=0 es x=%1.2f kozti reszterulet: ", x);
 printf("%1.2f\n", terulet(x, dx));
}
```

6. fejezet

Egy program több állományban

Fejezetünk elején induljunk ki egy, rendszerint a szemeszter első felében fel/kiadott ”első zh-ra” hajazó mini-zh-szerű feladatsorból, melynek megoldott feladataira építjük majd a fejezet címében szereplő téma diszkusszióját. Tehát, a feladatok:

1/a feladat
Írjunk egy függvényt, melynek egyetlen feladata, hogy a hívása után bekér egy egész típusú számot, s teszi ezt addig, újra és újra, amíg a beadott szám nem nagyobb kettőnél. Amennyiben a kapott szám nagyobb kettőnél, a függvény – befejezvén a működését – visszatér a szóban forgó számmal.

1/b feladat
Írjunk egy függvényt, mely – miután a `main`-ben meghívtuk – meghívván az imént írt függvényünket, értéket ad két helyi (`a`, `b`) változójának, s kiír a képernyőre egy ’`a`’ oszloppal és ’`b`’ sorral bíró szorzótáblát (a*b=...) elemekkel. Ha ’`a`’ vagy ’`b`’ közül valamelyik nagyobb, mint 5, akkor még a kiíratás előtt állítsa be mindkét változó értékét 4-re! A függvény – ki is írandó – visszatérési értéke legyen a*b! E visszatérési értéket pedig vegye fel a `main` egyik helyi változója!

1/c feladat
Oldjuk meg, hogy a `main`-beli kód annyiszor fusson le, ahányszor csak szeretnénk (a program újraindítása nélkül, természetesen)!

2. feladat
Írjunk egy függvényt, mely paraméter gyanánt kapja meg az előző függvény visszatérési értékét, utána kérjen be ennyi számot, válassza ki a legnagyobbat, valamint

számolja ki a számok átlagát, s az eredményeket írja is ki! Legyen a legnagyobb talált szám a függvény visszatérési értéke!

3. feladat
Újabb megírandó függvény, melynek bemenő paramétere legyen az előbb megírt függvény visszatérési értéke! Ha ez nagyobb, mint 10 akkor a függvény – mielőtt bármi mást tenne – állítsa be 10-re! A függvény feladata a kapott szám faktoriálisának kiszámolása, kiírása, majd az eredmény visszatérési értékként való „elkönyvelése".

4. feladat
A most megírandó eljárás kezelje az előző függvény visszatérési értékét egy kör sugaraként, mellyel kapcsolatos teendője az, hogy a π szám, illetve annak bárminemű kerekített, közelítő, stb értékének felhasználása nélkül – bekért pontosságú ($0 < dx << 1$) közelítéssel – számolja ki a kör területét, majd a kapott eredményt jelenítse is meg!

Lehetséges megoldás:

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


int beker()
{
 int n, limit=2;
 do{
  printf("Kerek␣egy␣szamot␣(>2)!␣");
  scanf("%d", &n);
  }while(n<=limit);
 return (n);
}


int szorzo()
{
 int a, b, i, j, limit=5;

 a=beker();
 b=beker();

 if(a>limit || b>limit){a=4; b=a;}
```

```
 for(i=1;i<=a;i++)
 {
  for(j=1;j<=b;j++)
  {
   printf("%d*%d=%d\t", i, j, i*j);
  }
  printf("\n\n");
 }

 printf("Visszateresi ertek: %d\n", a*b);
 return(a*b);
}


int max(int n)
{
 int i, legnagyobb, szam;
 float osszeg=0;

 for(i=1;i<=n;i++)
 {
  printf("%d. szam: ", i); scanf("%d", &szam);
  if(szam>legnagyobb || i==1){legnagyobb=szam;}
  osszeg+=szam;
 }

 printf("A szamok atlaga: %1.2f.\n", osszeg/n);
 printf("%d volt a legnagyobb.\n", legnagyobb);
 return(legnagyobb);
}


int faktor(int n)
{
 int i=1, fakt=1, limit=10;
 if(n>limit){n=limit;}

 while(i<=n){fakt*=i++;}

 printf("%d faktorialisa: %d\n", n, fakt);
 return(fakt);
}
```

```
void kor(int r)
{
 float dx, ter=0; int i=1;
 printf("Szamitas␣\"pontossaga\"␣(0<dx<<1)?\n");
 scanf("%f", &dx);

 while(i*dx<r){ter+=dx*sqrtf(powf(r,2)-powf(i++*dx,2))*4;}

 printf("A␣kor␣terulete:␣%1.2f.\n", ter);
}


main(){
 int a, b;

 do{
    a=szorzo();
    a=max(a);
    a=faktor(a);
    kor(a);
    printf("Meg␣egy␣menet?␣(i/n)␣"); b=getchar();
    if(b=='\n'){b=getchar();}
    }while(b=='i');
 }
```

Természetesen megoldhattuk volna másképpen is a `main`-t (ahogy az egész ZH-t is), de most készakarva lett ilyen, amilyen, azért mégpedig, hogy ily módon is követhetőbbé tegyem a függvények közötti értékátadást.

Látható, hogy a `main`-beli 'a' paraméter kezdetben nem kapott határozott értéket, csupán deklaráltuk. Értéket, a `szorzo` függvény hívása után kap csak, mégpedig annak visszatérési értékét.

Ezt az értéket kapja meg a `max` függvény, bemenő paraméter gyanánt, s miután befejezte a működését, a visszatérési értékét 'a'-ba másolja a vezérlés, mely 'a' lesz a `faktor` függvény bemenő paramétere.

Befejezvén a munkáját, a `faktor` függvény visszatérési értéke 'a'–ban talál gazdára, amit végül a `kor` nevű eljárás használ fel saját céljaira.

Vegyük észre, hogy ebben a feladatban mind a bemenő paraméterek, mind a visszatérési értékek típusa végig `int` volt, így nem okozott gondot az, hogy a

`main`-ben mindössze egy darab változó értékét írogattuk felül újra és újra a függvények hívása által, mivel a típusa végig megfelelt céljainknak!

Előfordulhat(!), hogy a `kor` viszonylag nagy bemenő értéket kap, amit mi tovább tetézhetünk egy nagyon pici értékű "pontossági" szám megadása által. Ilyenkor meglehetősen hosszú időre "gondolkodóba eshet" a vezérlés. Nem kell megijednünk, nem múlt ki a folyamat, csak félrevonult számolni egy kicsit. Ilyenkor, a szám beadása után csak azt látjuk, hogy a kör területének kiírása helyett csak "vár" a terfminál, miközben villog a kurzor. :-)

(Nyilván egy ilyen zh-ban nem csupán a fentiek, de az összes – a segédletekben eddig szereplő – feladat is előfordulhat, sőt, természetesen olyan is, amit még nem oldottunk meg (, de a jelenlegi apparátusunkkal és egy kis gondolkodással megoldható).)

6.1. Egy gondolat a függvényekről

Talán fel sem tűnt, de az eddigiek során igencsak kötöttük magunkat az "amilyen-sorrendben-hívjuk-a- függvényeket-olyan-sorrendben-írjuk-is-meg-őket" módszerhez, melynek felrúgása ezúttal(!) ugyan nem feltétlenül okozna gondot, ám nem kell, hogy ez mindig így legyen. Van ugyanis olyan helyzet, ahol a nem egyező sorrend probléma forrása lehet, példának okáért, próbáljuk csak ki, hogy mi lesz, ha a `beker()` függvényt áthelyezzük a `szorzo()` alá! Amennyiben például a fordítónk a `c99`-es sztenderdet használja, nem lesz túl boldog a módosított kód láttán. Biztos, ami biztos, ügyeltünk hát a sorrend egyeztetésére, mivel így nem hibázhattunk.

Hogy kicsit messzebbre lássunk, próbáljuk ki azt, ami végül is a szemeszter elején – még a függvényeknél – tárgyaltakból egyébként is következne, s amit már az előző fejezet prímkereső kódjának a végén is megpendítettem volt! Az ott említettekből kiviláglott, hogy a fordítót – midőn fentről lefelé halad a kódban – kizárólag az érdekli, hogy egy függvény bemenő paramétereinek és visszatérési értékének típusa "illik"-e a hívás helyén talált állandó/változó típusokhoz. Ezt kihasználva megtehetjük azt is, hogy a program `main` előtti/fölötti részében csupán a függvények paramétereit és visszatérési értékét adjuk meg (tehát, csak a függvények típusát). Ezek után a `main`-ben használjuk a függvényeket, míg azok leírását (a testüket) a program végére hagyjuk. Ha így járunk el, akkor a fenti ZH az alábbi formába alakul át:

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

int beker();
int szorzo();
int max(int n);
int faktor(int n);
void kor(int r);


main(){
 int a, b;

 do{
   a=szorzo();
   a=max(a);
   a=faktor(a);
   kor(a);
   printf("Meg egy menet? (i/n) "); b=getchar();
   if(b=='\n'){b=getchar();}
   }while(b=='i');
 }


int szorzo()
{
 int a, b, i, j, limit=5;
 a=beker();
 b=beker();
 if(a>limit || b>limit){a=4; b=a;}

 for(i=1;i<=a;i++)
 {
  for(j=1;j<=b;j++)
  {
   printf("%d*%d=%d\t", i, j, i*j);
  }
  printf("\n\n");
 }

 printf("Visszateresi ertek: %d\n", a*b);
 return(a*b);
}
```

```
int max(int n)
{
 int i, legnagyobb, szam;
 float osszeg=0;

 for(i=1;i<=n;i++)
 {
  printf("%d. szam: ", i); scanf("%d", &szam);
  if(szam>legnagyobb || i==1){legnagyobb=szam;}
  osszeg+=szam;
 }

 printf("A szamok atlaga: %1.2f.\n", osszeg/n);
 printf("%d volt a legnagyobb.\n", legnagyobb);
 return(legnagyobb);
}


int faktor(int n)
{
 int i=1, fakt=1, limit=10;
 if(n>limit){n=limit;}

 while(i<=n){fakt*=i++;}

 printf("%d faktorialisa: %d\n", n, fakt);
 return(fakt);
}


void kor(int r)
{
 float dx, ter=0; int i=1;
 printf("Szamitas \"pontossaga\" (0<dx<<1)?\n");
 scanf("%f", &dx);

 while(i*dx<r){ter+=dx*sqrtf(powf(r,2)-powf(i++*dx,2))*4;}

 printf("A kor terulete: %1.2f.\n", ter);
}
```

```
int beker()
{
 int n, limit=2;
 do{
  printf("Kerek egy szamot (>2)! ");
  scanf("%d", &n);
  }while(n<=limit);
 return (n);
}
```

A programok fent látható módon történő megfogalmazása teljesen mentesíti a programozót a sorrend egyeztetése alól. Nézzük csak meg például, hogy a `beker` függvényt teljes lelki nyugalommal tehettük be alulra, noha a hívására már az elsőként megírt `szorzo()`-ban sor került.

A szemeszter további részében, már csak a saját munkánkat megkönnyítendő is, ezt az utat fogjuk követni (mármint azt, hogy szétválasztjuk a függvények típusát és testét).

A fejezet újdonságot jelentő része most következik:

6.2. Programok több állományban

Beszéltünk már arról, hogy egy programon rendszerint több kódoló szokott dolgozni, ki-ki a saját szubrutinján, ha tetszik függvényén/függvényein, melyeket – miután elkészültek velük – mindnyájan elmentik a saját állományukba.

Nyilván életközelibb lenne így készíteni a programjainkat, de mivel nulláról indulva kell nem kevés dolgot elsajátítanotok egy amúgy sem túl tágra szabott óraszámban, nem szívesen forgácsolnám szét a figyelmeteket jobban a kelleténél, így a laborgyakorlatok alatt nem követelem meg a programok ilyetén módon való megírását. *A majdani beadandóra viszont nem érvényes az előbbi kitétel, mert azt már így kell majd elkészíteni!* S hogy ez az *így* ne maradjon lógva a levegőben, alább meglessük, miként működne az, ha a fenti zh függvényeit külön állományba tennénk. Nosza!

Először is, mentsük el a függvényeinket egy külön fájlba, melynek adjuk mondjuk a `zh.c` nevet. A `zh.c` így az alábbi tartalommal bír majd:

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


int szorzo()
{
 int a, b, i, j, limit=5;
 a=beker();
 b=beker();
 if(a>limit || b>limit){a=4; b=a;}

 for(i=1;i<=a;i++)
 {
  for(j=1;j<=b;j++)
  {
   printf("%d*%d=%d\t", i, j, i*j);
  }
  printf("\n\n");
 }

 printf("Visszateresi ertek: %d\n", a*b);
 return(a*b);
}


int max(int n)
{
 int i, legnagyobb, szam;
 float osszeg=0;

 for(i=1;i<=n;i++)
 {
  printf("%d. szam: ", i); scanf("%d", &szam);
  if(szam>legnagyobb || i==1){legnagyobb=szam;}
  osszeg+=szam;
 }

 printf("A szamok atlaga: %1.2f.\n", osszeg/n);
 printf("%d volt a legnagyobb.\n", legnagyobb);
 return(legnagyobb);
}
```

```
int faktor(int n)
{
 int i=1, fakt=1, limit=10;
 if(n>limit){n=limit;}

 while(i<=n){fakt*=i++;}

 printf("%d faktorialisa: %d\n", n, fakt);
 return(fakt);
}


void kor(int r)
{
 float dx, ter=0; int i=1;
 printf("Szamitas \"pontossaga\" (0<dx<<1)?\n");
 scanf("%f", &dx);

 while(i*dx<r){ter+=dx*sqrtf(powf(r,2)-powf(i++*dx,2))*4;}

 printf("A kor terulete: %1.2f.\n", ter);
}


int beker()
{
 int n, limit=2;
 do{
  printf("Kerek egy szamot (>2)! ");
  scanf("%d", &n);
  }while(n<=limit);
 return (n);
}
```

Láthatóan csak azt ”inklúdoljuk be”, amire ”szüksége van” az adott kódnak! A függvények típusainak meghatározásait pedig mentsük el mondjuk egy `zh.h` nevű állományba, mely így a következő tartalommal bír majd:

```
int beker();
int szorzo();
int max(int n);
int faktor(int n);
void kor(int r);
```

A program fennmaradó része – ahol a függvényeket hívjuk – kerüljön egy `program.c` nevű állományba, mely így az alábbi módon fog festeni:

```
#include<stdio.h>

main(){
 int a, b;

 do{
   a=szorzo();
   a=max(a);
   a=faktor(a);
   kor(a);
   printf("Meg egy menet? (i/n) "); b=getchar();
   if(b=='\n'){b=getchar();}
   }while(b=='i');
 }
```

Ennyivel persze nem ússzuk meg a dolgot, hiszen még gondoskodnunk kell a program részeit tartalmazó fájlok összefűzéséről! Valamiképp rá kell vennünk a fordítót, hogy a szétdarabolt programot összeillessze! Ahogy sejthetjük, használnunk kell a C fordító előfeldolgozó utasításait (ilyen például az `include` is)!

Mindenekelőtt a `zh.h` tartalmát be kell illesztenünk a `program.c` ”fejébe”! Ily módon, mire a fordító találkozik a függvények hívásaival, azok bemenő paramétereinek száma, típusa, illetve a visszatérési típusaik is ismertek lesznek a `gcc` számára, hiszen már megtalálta a típus-definíciókat a `zh.h`-ban. Mostantól tehát a `program.c` kódja a következőképpen módosul:

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

main(){
 int a, b;

 do{
   a=szorzo();
   a=max(a);
   a=faktor(a);
   kor(a);
   printf("Meg egy menet? (i/n)"); b=getchar();
   if(b=='\n'){b=getchar();}
   }while(b=='i');
 }
```

Így tehát, még a `program.c` fordításának megkezdése előtt, a fordító beszúrja a `zh.h` tartalmát a programba. Ugyanezen ok által motiválva, a `zh.c`-be is be kell "inklúdolnunk" a `zh.h`-t, hiszen a `szorzo()` függvényben hívjuk a `beker()` függvényt, melynek típusa a `zh.h`-ban van definiálva. A `zh.c` tartalma tehát az alábbi formára módosul:

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

A tobbi resz ugyanaz, mint eddig volt.
```

Ezen a ponton a *fájlokkal* való munkánk véget ért. Most már "csak" le kell fordítanunk őket! Ezzel kapcsolatban csak az a bökkenő, hogy ha az eddig bevett módszert alkalmazva kíséreljük meg lefordítani a fájlokat, kapunk pár hibajelzést az arcunkba, hiszen – egy kivétellel – egyik fájl sem tartalmaz `main`-t. Az az egy pedig, amelyik igen, az meg nem tartalmazza a hívott függvényeket... A probléma az alábbi parancsokkal orvosolható:

```
Bash$ gcc -c program.c
Bash$ gcc -c zh.c
Bash$ gcc program.o zh.o -lm -o program
```

Az első parancs lefordítja a `program.c`-t de nem eredményez futtatható állományt. Ehelyett, egy gépi kódú utasításokat tartalmazó, ám nem teljes programot állít elő. Ezt a `gcc`-nek a `-c` kapcsolójával érjük el, aminek hatására a fordító csak a fordítást végzi el, így eredményül, egy úgynevezett tárgykódú (object code) programot kapunk, mely esetünkben most `program.o` névre hallgat. E program még nem futtatható, de már tartalmazza a majdani futtatható állomány egy részét. A második parancs célja ugyanez, csak a `zh.c`-re kihegyezve. Az "összefűzést" az utolsó parancs végzi el, mely létrehozza a `program` nevű futtatható állományt. (Mint már tudjuk, az `-lm` a programban használt `math.h` miatt kell csak, míg az `-o` a "névadó".) Ki lehet hát próbálni a `./program` parancs bepötyögését, s büszkék lehetünk magunkra, mert újra csak közelebb kerültünk egy lépéssel az "éles" programozáshoz.

Jó, jó, van még addig néhány egyéb elsajátítandó dolog is, de az biztos, hogy sokkal életszerűbb, életközelibb a most megtanult forma, mint az, ahogyan eddig programozgattunk, mert így már valóban többen is dolgozhatunk együtt egy nagyobb projekten, ahol mindenkinek csak a saját részfeladatára kell összpontosítani.

7. fejezet

Összetett típusok

Nézzük elöljáróban, hogy kikkel is találkozhatunk e fejezetben! Újdonság gyanánt, először is a *tömbök* fogalmával ismerkedünk meg, ahonnan utunk irányát a *mutatók* jelzik majd tovább. Ha túléltük a velük való barátkozást és maradt bennünk még némi életerő is, akkor visszatekintve látni fogjuk, hogy ők voltaképp szoros rokonságban állnak, olyannyira, hogy közös gyermekük, a karakterláncok meg sem szelídíthető nélkülük.

Annyit mindenképpen illik tudnunk a jelen fejezetcím tárgyát képező típusok közé tartozó változókról, hogy azokat, valamilyen feladat megoldása érdekében – mintegy ahhoz idomítva – elemi adattípusok felhasználásával mi hozzuk létre. Ezek közül a legegyszerűbb típus:

7.1. A tömb

, amit – amennyiben egydimenziós – vektorként is emlegetnek, s ami voltaképp nem más, mint egy azonos típushoz tartozó változókból összefűzött ”csokor”. Ebből egyenesen következik a tömb létrehozásának módja is, melyet az alábbiakban demonstrálok.

```
main(){
  int  n=10;
  int  tomb[n];
}
```

Fentiek szerint tehát, egy tömb létrehozásakor, először is meg kell neveznünk azt

az adattípust, melyhez a tömböt képező elemek tartoznak! Esetünkben ez a típus az `int` volt. Rendszerint, a ”gyermek nemének” ilyetén módon való meghatározását a ”keresztelő” követi, magyarán nevet adunk a szóbanforgó tömbnek, ami fent `tomb` lett.

Ha ezen is túl vagyunk, akkor már csak annyi teendőnk akad, hogy egy szögletes zárójelbe írt számmal (vagy mint az fent látható, egy – *már a fordításkor értékkel rendelkező* – változóval[1]) jelezzük, hogy a létrehozott tömbnek mennyi eleme lesz.

Fontos, hogy a szögletes zárójelbe írt érték, már a fordításkor ismert legyen, ugyanis a fordító innen tudja, hogy mekkora helyet kell a memóriában a tömbnek szorítani!

Ha esetleg az eddigiekből nem világlott volna ki, hogy miért kell a tömbökkel ”bonyolítani” az életet, gondoljunk csak bele, hogy mit tennénk akkor, ha például sok egész számmal (vagy bármi más, egyazon típushoz tartozó adattal) kellene dolgoznunk úgy, hogy azokat – legalább a program futásának az idejére – el is

[1] Igazából itt csaltunk egy kicsit, mert szemérmesen elhallgattuk, hogy van olyan C szabvány (például az `ISO C90`), ami tiltja azt, hogy egy statikus tömb hosszát egy változóban tároljuk, illetve azáltal adjuk meg (`int tomb[n]`). Valószínűleg az motiválta e rigorózus hozzáállást, hogy ezúton is biztosítva legyen a statikus tömb hosszának futás közbeni megváltoztathatatlansága. A `C90` óta azonban eltelt csaknem 30 esztendő, s ma már gyakorlatilag csak akkor találkozik a programozó e figyelmeztetéssel, ha készakarva úgy állítja be a `gcc` parancs vonatkozó opcióit / kapcsolóit a parancssorban, hogy csak az említett dialektust "beszélje" a fordító (illetve azt is).

Valójában elég ritka az ilyesfajta korlátozás a C nyelvben, s a `C90` e rigolyája sem véletlenül lett már rég "felülírva" azóta, ugyanis a C-t okos embereknek találták ki, akikről a nyelv feltételezi, hogy tudják, mit csinálnak, s épp ezért elég nagy szabadságot is biztosít nekik. Igenis jogos az az igény a programozó részéről, hogy - például esetünket tekintve - egy helyen, *egyszer* legyen csak megadva a tömb elemszámának / hosszának az értéke, s így - futásidőn kívüli(!) - módosítás esetén elég legyen csak egy helyen átírni azt, s ne mindenhol, ahol csak előfordul. Az pedig nyilvánvaló, hogy ez esetben nem adunk lehetőséget a felhasználónak arra, hogy futásidőben "belenyúlhasson" e változó értékébe, így az futás közben változatlan marad, s nem fenyegeti veszély a statikus tömb hosszát.

Már csak azért sem gond, ha "rászoknunk" a tömb hosszának ily módon történő tárolására, mert később, amikor már dinamikus helyfoglalású tömbökkel dolghozunk, nem is igen lesz más lehetőségünk, mint az, hogy tömbjeink hosszát egy-egy változó által tároljuk, hiszen amikor futásidőben kiderül majd, hogy pontosan mekkora tömbnek kell "helyet csinálnunk", a szóban forgó értéket nyilván egy változóban tudjuk a legegyszerűbben megőrizni.

Mindenesetre, ha akadnak köztünk olyanok, akiket nagyon zavar, hogy létezik egy szabvány, ami fájlalja, ha a statikus tömb hosszát egy változóban tároljuk, azok számára a kompromisszumos javaslatom a következő: `int n = 5, tombNev[5];` , mely esetben, minden olyan függvény, aminek a tömbbel van dolga, az `n` változót kaphatja meg paraméter gyanánt, miközben a tömb deklarálásakor az `n` értékét (itt épp `5`-öt) beírjuk a `[]` jelek közé. Így nem sértünk egy szabványt sem, csak arra kell vigyáznunk, hogy az `n` értéke mindig ugyanakkora legyen, mint a `[]` jelek közé írt szám!

kéne tárolnunk!

Gyártanánk minden szám tárolására egy-egy `int` típusú elemi változót, külön nevet adván mindegyiknek? Néhány adat esetén még működőképes is lehetne ez a stratégia, na de 100, 1000 vagy sokkal több adatot hogyan kezelhetnénk ilyen módon hatékonyan?

Felteszem, könnyen belátható, hogy a tömbök alkalmazása által elképesztő mértékben leegyszerűsödik a feladat ..... s akkor az ily módon megnyíló egyéb lehetőségekről (pl.: az elemek könnyed rendezése, stb) még szót sem ejtettünk.

Namármost, a fenti példában létrehozott tömb `int` típusú, tehát minden eleme képes egy egész szám tárolására.

Igen, zseniális az észrevétel, miszerint a tároláshoz előbb fel is kéne tölteni a létrehozott tömböt, éppen ezért alább meg is mutatom annak módjait. Eljárhatunk például így:

```
main(){
int i, n=10;
int tomb[n]; /*vagy akar: int i, n=10, tomb[n]; */
for(i=0;i<n;i++)
{
 tomb[i]=i*i;
}/*Persze, mashonnan is johetnenek az ertekek.*/
}
```

Látható, s egyben igen igen *fontos megjegyzendő tény, hogy a tömb elemeinek indexelése 0-tól kell, hogy induljon!* Következésképp, egy `n` elemű tömb, *első elemének indexe 0, míg az utolsóé n-1 kell, hogy legyen!* Nyilván máshonnan – billentyűzetről, fájlból, stb – beolvasva is feltölthető egy tömb, csak arra kell figyelnünk, hogy *ne lépjünk túl az előre megadott méreten!* Ebben az esetben ugyanis a „jutalmunk” **memóriafelosztási hiba**, s akár az épp futó folyamat kivégzése is lehet.

Természetesen van lehetőség arra is, hogy egy tömb a lent bemutatottak szerint „töltekezzen”, s úgymond, kézzel tegyük bele a forrásba azt.

```
main(){
int tomb[]={1,-88,4,5,7,9,10,100,1000000,9};
/*Ilyenkor nincs szukseg az elemek
darabszamara sem!*/
}
```

Most pedig, hogy már egész jól állunk a tömbök létrehozásának terén, ideje, hogy kezdjünk is velük valamit! Bemelegítésképp írassuk ki a legutóbbi tömb elemeit fordított, tehát – az indexek sorszámát tekintve – csökkenő sorrendben! Valahogy így:

```
#include<stdio.h>

main(){
int i, n;
int tomb[]={1,-88,4,5,7,9,10,100,1000000,9};
n=sizeof(tomb)/sizeof(int);
for(i=n-1;i>=0;i--)
 {
  printf("A␣tomb␣%d.␣eleme:%d.\n", i+1, tomb[i]);
 }
}
```

Láthatóan, a `for` ciklusban szükség volt a tömb elemeinek darabszámára, viszont azt most a tömb létrehozásakor, a `{}` jelek közé írt számok darabszáma határozta meg, s nem mi adtuk meg, mint például az első alkalommal, amikor beírtuk a kódba, hogy `int n=10, tomb[n];`, stb. Ilyen esetben, hacsak nem szeretnénk minden alkalommal ”kézzel” megszámolni, hogy mennyi eleme van a tömbnek, jól jöhet a `n=sizeof(tomb)/sizeof(int);` típusú huncutkodás, hiszen az az utasítás az egész tömb méretének, és a tömb *egy* eleme méretének a hányadosán keresztül adja meg a tömb elemeinek a darabszámát, tetszőleges esetekben is.

Most pedig, példának okáért, írhatnánk egy programot amely bekéri egy 10 fős csoport zh jegyeit, majd kiszámítja és kiírja a csoport tanulmányi átlagát. Fogjunk hát neki!

```
#include<stdio.h>

main(){
 int i, j, n=10;
 float jegyek[n], osszeg=0;

 for(i=0;i<n;i++)
 {
  printf("%d.␣jegy:␣", i+1);
  scanf("%f", &jegyek[i]);
  /*A tomb elemei is ugy kapnak erteket, mint az
  elemi valtozok.*/
 }
```

```
 for(i=0;i<n;i++)
 {
  osszeg+=jegyek[i];
  /*A tomb elemeinek osszeadasa.*/
 }

 printf("A␣csoport␣jegyeinek␣atlaga:␣%1.2f.\n", osszeg/n);
}
```

Természetesen, az előbbi feladatot, a kód „tömörítéséből” végképp sportot űzvén, akár így is megoldhattuk volna:

```
#include<stdio.h>

main(){
 int i,n=10; float jegyek[n], osszeg=0;

 for(i=0;i<n;osszeg+=jegyek[i++])
 {
  printf("%d.␣jegy:␣", i+1); scanf("%f", &jegyek[i]);
 }

 printf("A␣csoport␣jegyeinek␣atlaga:␣%1.2f.\n", osszeg/n);
}
```

Sőt, akár némileg csalafintább módon ugyan, de a tömböt nélkülözve is zöld ágra vergődhettünk volna, ahogy azt alább meg is mutatom, bár gondolom, sokan maguktól is rájöttek már, mi a teendő:

```
#include<stdio.h>

main(){
 float jegy, osszeg=0; int i, n;

 printf("Mennyi␣szam␣lesz?␣"); scanf("%d", &n);

 for(i=1;i<=n;osszeg+=jegy, i++)
 {
  printf("%d.␣jegy:␣", i); scanf("%f", &jegy);
 }

 printf("A␣csoport␣jegyeinek␣atlaga:␣%1.2f.\n", osszeg/n);
}
```

Így még a tömb terjedelmi korlátai alól is felszabadulhattunk volna, viszont tovább haladván a tananyagban, egyre kevesebb ilyen kiskapunk lesz! Ráadásul, a lokális célunk itt most épp az volt, hogy minél jobban kiismerjük magunkat a tömbök lelkivilágában.

7.2. Mutatók

A mutató (pointer) a C nyelvben központi szereppel bír, s meglehetősen jól kell bánnunk vele, hogy háláját kimutassa, viszont ha egyszer megértettük a motivációit, legott hálásan doromboló kedvenccé szelídül. Mielőtt azonban felvennénk a kesztyűt, tisztázzuk, mi is a mutató!

A mutató egy olyan változó, melynek értéke egy memóriacím. Tehát, egy hely a memóriában. Épp innen származik a neve, hiszen **megmutat** egy címet, rámutat egy helyre. Ez – legalábbis eddig – ennyire egyszerű.

A kérdés, hogy hát mégis mi szükség van az életünk – mutatók általi – még további bonyolítására, természetesen bárkiben felmerülhet, s ígérem a válasz sem marad el, viszont a megértéséhez, előbb az ahhoz nélkülözhetetlen ismeretekkel kell megbarátkoznunk. Haladjunk apránként, s első lépés gyanánt nézzük meg alább, a mutató létrehozásának (deklarálásának) a módját!

```
float *mutato;
```

Fenti példánkban a `float`-tal azt jeleztük a fordítónak, hogy a mutató által célba vett helyen milyen típusú adatot talál. A * pedig arra volt hivatott felhívni a figyelmét, hogy az általunk `mutato`-nak elnevezett változó egy MUTATÓ.

Mivel feltehetőleg nem azért hoztunk létre egy mutatót, hogy aztán használatát mellőzve küldjük nyugdíjba, a következő megtanulandó lépés az értékadás. Mutassunk rá vele a fentebb már használt `osszeg` nevű változó helyének memóriacímére! Ez esetben a következőt kell tennünk!

```
mutato=&osszeg;
```

A & jelet – ugye ismerős(?) – a változó neve elé írva jeleztük, hogy annak nem az értékére, hanem a címére van ezúttal szükségünk (jelen esetben, a mutatónknak való értékadás végett). Magyarán, a `mutato` nevű mutató *értéke*, mostantól az `osszeg` nevű változó *lakcíme* lesz.

A fentieket egészen nyugodtan elképzelhetjük úgy, mintha a mutató egy nyíl lenne, amit az értékadással „ráállítottunk” a memória azon helyére, ahová az `osszeg`

nevű változót elmentette a vezérlés.

Most pedig, hogy végre tudjuk, mik is azok a mutatók, ismerjük létrehozásuk módját, valamint értékadásuk fortélyait ideje, hogy használatukat is elsajátítsuk, s – kicsivel később – megértsük létezésük okát is!

Haladjunk azonban apránként, lépésről lépésre!

Első körben azt fogjuk megnézni, hogyan lehet egy mutató használatával felülírni az általa kijelölt változó értékét. (Később megértjük majd azt is, hogy miért nagyon fontos birtokában lennünk ennek az ismeretnek és tökéletesen megérteni a mögötte meghúzódó hátteret.)

```
#include<stdio.h>

main(){
int valtozo=1; /*Itt foglalunk helyet, s adunk erteket a
valtozo nevu valtozonknak.*/

int *mutato; /*Letrehozzunk egy egesz tipusu valtozo
helyet jelolo mutatot.*/

mutato=&valtozo; /*"Raforditjuk" a mutato "nyilat" a
valtozo "lakhelyere".*/

printf("A valtozo erteke: %d\n", valtozo);
/*Kiiratjuk a valtozo erteket.*/

*mutato=2; /*Ez a "mutato kovetese" nevu muvelet.
Tovabbiakert lasd a szoveget!*/

printf("A valtozo uj erteke: %d\n", valtozo);
/*Kiiratjuk a valtozo uj erteket.*/
}
```

A fenti programocska újdonság gyanánt a `*mutato=2;` műveletet foglalja magába, melyről egyelőre csak annyit tudunk, hogy a „*mutató követése*”-ként emlegettük és hatására a `valtozo` nevű változónk értéke átíródott 1–ről 2–re.

Hogyan történt ez és miről is van itt szó egyáltalán?

Nos a lényeg az, illetve ott kezdődik, hogy a `*mutato=2;` sor csillagának semmi köze az `int *mutato;` sorba beírt csillag jelhez, mármint teljesen mást jelent a fordító számára. A `*mutato=2;` mutatványt ugyanis csak abban az esetben

vethetjük be, ha egy mutató már „él”, tehát deklaráltuk és már „rá is forgattuk” valamire, tehát határozott értékkel is rendelkezik.

Ha ilyenkor, egy ilyen – már „élő” – mutató neve elé betesszük a csillag jelet (tehát úgy teszünk, mint itt: `*mutato=2;`), akkor azzal nagyjából a következőt „mondjuk” a vezérlésnek:

„Ne a mutatóval foglalkozz, hanem kizárólag azzal, amit az általa megmutatott helyen találsz! Magyarán, azzal tedd meg azt, amire utasítalak, s ne a mutatóval!"

Nyilván ez így nem feltétlenül világos, ezért – érthetőbbé teendő a dolgot – maradjunk a fenti példánál, s vegyük át az ott történteket!

Tehát: `*mutato=2;`

A csillag jel hatására a vezérlés „fogja” a mutatót, megnézi, hogy hová mutat a „nyila”, s azt követve („megy”, ahová az mutat → mutató követése) eljut a mutató által kijelölt memóriaterületre, ahol megleli a `valtozo` névre hallgató változót, majd módosítja annak értékét.

Tehát: a „mutató követése”-kor nem magát a mutatót „maceráljuk” (nem a mutató nyilát „forgatjuk át” más memóriaterületre), hanem az általa mutatott területet (az ott „lakó” változó értékét) módosítjuk.

Érthető?

Természetesen van mód arra is, hogy magát a mutató „nyilát” forgassuk át egy másik változó „lakhelyére”. (Szebben szólva: a mutató, mint memóriacím-hordozó változó kapjon új értéket, egy másik változó memóriacímét.) Legyen a másik változó neve mondjuk `osszeg2`! A megoldás nyilvánvaló: `mutato=&osszeg2;`. Világos?

A lényeg az, hogy ha már van egy „élő” mutatónk, akkor az kétféleképpen is szerepelhet egy értékadás bal oldalán.

- Az egyik eset az, amikor nincs előtte csillag. Pl.: `mutato=&osszeg2;`. Ez esetben maga a mutató kapott új értéket, egy új memóriacímet, méghozzá az `osszeg2` nevű változóét. („Átforgattuk” a mutatónk „nyilát” az osszeg2–re.)

- A másik esetben van előtte csillag. Pl.: `*mutato=8;`. Ilyenkor maga a mutató értéke – a memóriacím – nem változik, tehát a mutató „nyila” marad

ott, ahol volt – itt most az `osszeg2` nevű változó címére szegezve – viszont az `osszeg2` értéke mostantól 8–cal lesz egyenlő.

Remélem, sikerült érthetően megvilágítanom a mutató követésének a „műveleti hátterét”, kiváltképp és különösen azért, mert a továbbiakban ez a fogás nélkülözhetetlen részét képezi majd C-beli eszköztárunknak, ahogy értő, gondos alkalmazása nemkülönben.

Voltaképpen elegendő, ha már ennyit értünk belőle (**de ezt aztán nagyon(!)**), viszont – nem kötelező ugyan, de – áshatunk kicsit tovább is...

Érdekesség gyanánt – minden magyarázatot nélkülözve – álljon itt egy kód, aminek futtatása és önálló megértése az elvártnál mélyebb bepillantást enged az említettekbe! Mielőtt futtatnánk, mindenképpen próbáljuk meg kitalálni, hogy mit láthatunk majd a képernyőn! Ha sikerül előre kitalálnunk az eredményt, az azt jelenti, hogy kezdjük egész jól megérteni az eddig – a mutatókkal kapcsolatban – leírtakat.

Íme:

```
#include<stdio.h>

main()
{
 int a=1, b=2, *p;

 p=&a; *p+=a+b; printf("%d\n",a);

 p=&b; *p+=a+b; printf("%d\n",b);

 printf("%d=%d, %d, %d\n", b, *p, *p+*p, *p**p);

 printf("'b' mennyi legyen?"); scanf("%d", &b);
 printf("'b' %d lett.\n", b);

 printf("'b' mennyi legyen?"); scanf("%d", p);
 printf("'b' %d lett.\n", b);
}
```

Most pedig lássuk a választ a mutatók létének értelmét firtató kérdésre! (Igazából nem csak az alább bemutatott műveletekre használjuk majd a mutatókat, hanem fontosabb dolgokra is, csak igyekszem nem mindent egyszerre a nyakatokba zúdítani. Majd elmondom az itt egyelőre le nem írtakat akkor, amikor szükségünk

lesz rájuk.)

Vegyük az alábbi egyszerű esetet, melynek kapcsán egy függvény igyekszik eggyel megnövelni a neki átadott változó értékét!

```
#include<stdio.h>

int egyremegy(int b);

main()
{
 int a=1;
 printf("'a' erteke: %d\n", a);
 egyremegy(a);
 printf("'a' uj erteke: %d\n", a);
}

int egyremegy(int b)
{
 b+=1;
 return b;
}
```

Bárki, aki nem értette meg teljes mélységében a szemeszter folyamán a függvényekről eddig leírtakat, az állományt futtatva azt várná, hogy miután a függvény kezelésbe vette a neki átadott a változót, a második kiíratáskor annak eggyel növelt értéke jelenik meg. Sajnos nem ezt tapasztaljuk.

Persze kicselezhetjük a végzetet úgy is, hogy `egyremegy(a);` helyett `a=egyremegy(a);`-t írunk vagy esetleg betesszük az `egyremegy()` függvény hívását a `printf`-be, így valahogy:

```
printf("'a' uj erteke: %d\n", egyremegy(a));
```

Noha miénk a győzelem, mégis, nyugtalanító szorongás fészkelte magát gondolataink közé, mely kétségbeejtő érzésnek rendszerint oly módon adunk hangot, hogy azt mondjuk: "Valamit nem értek...". Mert hát mivel is szembesültünk odafent? Azzal, hogy miután értékadásban hívtuk a függvényt, így: `a=egyremegy(a);` vagy a `printf`-ben így:

```
printf("'a' uj erteke: \%d\n", egyremegy(a));
```

, ahogy vártuk, annak visszatérési értéke jelent meg a képernyőn, míg más esetben,

amikor csak úgy, magányosan számolgatott, semmit sem változtatott a neki adott - a - változó értékén jóllehet, világosan utasítottuk erre, midőn azt írtuk volt, hogy

```
int egyremegy(int b)
{
 b+=1;
 return b;
}
```

Mi lehetett a gond? Próbáljunk meg rájönni! (Aki jól figyelt a függvényekkel kapcsolatban régebben elmondottakra, az feltehetőleg már érti, hogy mi a probléma.)

Első körben képezze kiindulásunk alapját az általunk már eddig is biztosan tudott tény, miszerint, ha egy függvény hívására egy értékadás jobb oldalán kerül sor, akkor annak visszatérési értékét megkapja az egyenlőségjel bal oldalán álló változó.

Nézzük, hogy miben különbözött ezektől az az eset, amikor nem a várt értéket írta ki a program!

Egyetlen dologban: A hívott függvény nem tudta minek átadni a visszatérési értékét! Ezzel a mondattal el is érkeztünk a megvilágosodás kapujába, azért mégpedig, mert innen már csak egy lépés, hogy megértsük, mi is volt a gond! A válasz egyszerű: *C nyelvben, függvényhívás esetén, a paraméterátadás érték szerinti*. Nem világos? Kifejtem alább részletesebben.

Onnan kell indulnunk, hogy az általunk eddig használt változók csak lokális hatókörrel bírtak, vagyis mind a hívó, mind a hívott függvény csak a sajátjait használhatta, a másikét el nem érhette (még csak nem is „láthatta”). Mi viszont ennek ellenére azt szeretnénk, hogy a hívott függvény feldolgozza a számára fogyasztásra felkínált helyi változóinkat, a saját helyi változói segítségével. . . . . .

Na de hogyan, ha nem férhetnek hozzá egymás helyi változóihoz?

Nos, a dolog áthidalható, mégpedig a már említett érték szerinti paraméterátadással, magyarán, a hívott függvény nem kapja meg, úgymond szőröstül-bőröstül a helyi változóinkat, hogy aztán, miután eljátszogatott velük, azokat új értékkel felruházva visszadobja a mi játszóterünkre, hanem csak azok értékei másolódnak át, majd rendelődnek hozzá a hívott függvény helyi változóihoz.

Ezután a hívott függvény helyi változóinak értéke – a műveletek folyamán – meg is változik.

A gond itt kezdődik, ugyanis miután (a hívott függvény) dolga végeztével vissza-

tér saját létsíkjára, a helyi változói megsemmisülnek, így velük együtt sírba szállnak azok értékei, ahogy a bennük beállt változások is.

A mi helyi változónkat a hívott függvény számításai csak akkor érintenék, ha a visszatérési érték valahogy átadódna neki. (Erre szolgált például az egyenlőségjel jobb oldalán elkövetett függvényhívás, az `a=egyremegy(a);`)

Sajnos, ha valaki nincs tisztában azzal, hogy a C nyelvben érték szerinti paraméterátadás zajlik ilyen esetben, az könnyen beleszaladhat hasonló csapdákba.

Természetesen van rá mód, hogy biztosra menjünk ilyenkor is, mégpedig a mutatók használata. S bár mutogatni illetlenség, időnként mégis hasznos lehet ... kiváltképp a C nyelvben :).

Az alábbiakban a kezdeti példából kiindulva, annak átírása által vázolom, hogyan lehet mutató segítségével – „szőröstül–bőröstül” – elérhetővé tenni egyik függvény (a hívó) helyi változóját egy másik függvény (a hívott) számára!

Feltehetőleg – mielőtt egyáltalán belekezdene a kód alatti magyarázat elolvasásába – mindenki számára világos lesz, hogy miért kellett annyira alaposan átrágnunk a „mutató követése” névre hallgató műveletet.

```
#include<stdio.h>

void egyremegy(int *b);

main()
{
 int a=1;
 int *p;
 p=&a;
 printf("'a' erteke: %d\n", a);
 egyremegy(p);
 printf("'a' uj erteke: %d\n", a);
}

void egyremegy(int *b)
{
 *b+=1;
}
```

Látható, hogy a függvényből eljárás lett, mert már nincs is szükség rá, hogy rendelkezzen visszatérési értékkel (ettől még persze lehetne neki, csak épp mondjuk hibajelzésre is használhatnánk azt).

Az eljárás írásakor közöltük a fordítóval, hogy bemenő paraméterként egy egész típusú adat helyét jelölő mutatót kap majd az eljárás (`void egyremegy(int *b)`). Ez a paraméter, a híváskor átadott `p` mutató lett (`egyremegy(p);`), ami az `a` nevű változó memóriacímét hordozta magában, tehát valóban egy mutató, ahogy azt az eljárás is várta.

A *lényeg*et a függvény magjában látható művelet rejti!

A fenti programban az `egyremegy` nevű eljárás a kapott `p` mutató által képviselt memóriacím értékét – tehát az `a` nevű változó címét – hozzárendeli a saját helyi változójához, a `b` nevű mutatóhoz.

Ezután következik a `*b+=1;` utasítás, ami pontosan az, aminek gondoljuk: a **mutató követése**.

A látottakat kicsit önállóan átgondolva, most már egészen biztosan magunk is meg tudjuk válaszolni az oldalakkal korábban felvetett – a mutató követésének értelmét firtató – kérdést, mely nagyjából így hangzik: mégis mire jó a mutató követése?

(Hát) a *függvényeken beüli változóelérésre*, magyarán az egyik függvényből közvetlenül elérhetjük, következésképp felül is írhatjuk egy másik függvény lokális változóit (amikhez egyébként csak ő férne hozzá), hiszen a mutatók értékéből tudjuk a címüket, tehát tudjuk, hogy hol keressük őket a memória határtalan vadonában, s ha egyszer rájuk leltünk .....

7.2.1. Apró kiegészítés

Természetesen egy sokatlátott hallgatónak szemet szúrhat egy apróság ... Hogy mégis mi az?

Nos, csak az a tény, hogy maga a paraméterátadás továbbra is érték szerint zajlott, hiszen egy memóriacím – a `p` mutató értéke – lett átmásolva a hívott függvény – az `egyremegy()` – helyi mutatójába (a `b` mutatóba)!

A különbség a „mezítlábas” változók átadásához képest a következő: ha egy „sima” változót adunk meg paraméter gyanánt a hívott függvénynek, akkor annak értéke a hívott függvény helyi változójába másolódik, s onnantól kezdve minden értékváltozás csak ezt a helyi változót érinti, így az eredeti változó értéke megőrződik. (Kivéve persze azt a jólismert esetet, melyben a visszatérési értékkel rendelkező függvényt egy értékadás jobb oldalán hívjuk, míg az egyenlőségjel bal

oldalát az eredeti változónk támasztja.)

Mutatók használatakor is csak másolással (érték szerint) adódik át a beadott paraméter (a mutató értéke), viszont ez az érték a megváltoztatni kívánt eredeti változó memóriacíme, mely által a hívott függvény „ki tudja szúrni" a változó „tartózkodási helyét", s ily módon már, a 'mutató követése' művelet által meg tudja találni és képes közvetlenül meg is változtatni azt.

(A függvény hívása előtti állapotot elképzelhetjük úgy, hogy egyelőre csak a hívó függvény mutatója van „ráállítva" a – módosítandó – változóra, a függvény hívása utáni állapotot pedig úgy, hogy – a memóriacím értékének átmásolása miatt – most már a hívott függvény mutatójának a „nyila" is „rá lett fordítva" a kérdéses változóra. Ennélfogva, a hívott függvénynek is „meg lett mutatva" a változó lakhelye, így az már tudja, „hol keresse" azt, tehát innentől fogva közvetlen hozzáféréssel rendelkezik a változóhoz.)

Azért hangsúlyozom ennyire a 'mutató követésének' műveletét, mert – ha esetleg még nem vettük volna észre – nagymértékben kiszélesíti az általunk írt függvények lehetőségeit.

Vegyük észre, hogy abból fakadóan, hogy függvényeink – legyenek bármily kifinomultak is – csupán egy és csakis egy értéket adhatnak vissza az őket hívó függvénynek, mutatók híján – csak a `return` adta lehetőséggel élve – mindössze egyetlen változó értékét képesek módosítani a hívó függvény változói közül.

Mostantól viszont, kezünkben a 'mutató követésének' eszközével gyakorlatilag akármennyi változó értékét felülírhatjuk egyetlen függvény egyszeri hívása által is, hiszen annyi változó memóriacímét adhatjuk meg a hívott függvénynek, amennyiét csak szeretnénk. A `return` pedig felszabadul végre, s használhatjuk másra is.

7.2.2. Némi egyszerűsítés

A

```
void egyremegy(int *b)
{
 *b+=1;
}
```

eljárás megfogalmazásakor láthatóan úgy jártunk el, hogy az `(int *b)` paraméterlistában arra készítettük fel, hogy a híváskor egy mutatót fogunk vele „meg-

etetni”.

Jussanak eszünkbe az „apró kiegészítés” cím alatt tárgyaltak, s emlékezzünk vissza arra, hogy ami „átmegy” a hívott függvénybe/eljárásba, az kizárólag csak a paraméterként átadott `p` mutató értéke (ez másolódik át az `egyremegy()` saját, `b` nevű mutatójába), tehát mindössze egy memóriacím.

Ha megértettük, hogy ez mit jelent, akkor rájövünk arra is, hogy a `p` mutató akár ki is iktatható a játékból, mivel nekünk – mint azt az előbb leírtam – csak az értékére, tehát magára a mutatott változó memóriacímére van csupán szükségünk, amit viszont az ismert `&` operátorral is életre hívhatunk (ahogy tettük is volt azt, midőn a `p` értéket kapott, a `p=&a;` utasítás által).

Tehát, akkor mi is a teendő? Íme:

```
#include<stdio.h>

void egyremegy(int *b);

main()
{
 int a=1;
 printf("'a' erteke: %d\n", a);
 egyremegy(&a);
 printf("'a' uj erteke: %d\n", a);
}

void egyremegy(int *b){*b+=1;}
```

Látható, hogy nem a – már nem is létező – `p`-be „tettük bele” az `a` változó memóriacímét, hanem közvetlenül a függvénynek adtuk át azt, annak hívásakor, az `egyremegy(&a);` utasítás segítségével. Ez a manőver most ugyan nem „dobott” sokat a kódon, viszont igen hasznos lehet, ha több változóval dolgozunk, hiszen általa fölösleges mutatók deklarálásától tekinthetünk el.

Nyilván játszhattunk volna általános (globális) változóval is, melyhez minden függvény közvetlen hozzáféréssel bír, így nem kellett volna mutatót használnunk. Ennek módjára viszont most itt legfeljebb egy **elrettentő példa** erejéig térek ki, mivel az ilyen változók használatának megvan a maga veszélye (hiszen minden függvény számára

elérhetőek)! (Ettől még persze, ha ritkán is, de időnként hasznosak lehetnek.) Tehát, az előbbi feladat megoldásának alábbi módja, maga az **elrettentő példa**:

```
#include<stdio.h>

int a;
/*Ez itt a globalis - mindenki altal elerheto - valtozo,
illetve annak - minden mast megelozo modon valo -
deklaralasa.*/

void egyremegy();

main()
{
 a=1;
 printf("'a' erteke: %d\n", a);
 egyremegy(a);
 printf("'a' uj erteke: %d\n", a);
}

void egyremegy()
{
 a+=1;
}
```

7.2.3. Néhány mutató-s dolog

Néhány szó erejéig visszatérek a fejezet elején tárgyaltakhoz, kiegészítendő azokat egy-két aprósággal.

Típuskényszerítés

Említettem volt a mutatók létrehozásával kapcsolatban, hogy még mielőtt egyáltalán elneveznénk az újdonsült pajtást, meg kell adnunk annak a változónak a típusát, amit a mutató által kijelölt memóriacímen talál majd a vezérlés.

Erre azért van szükség, mert a fordító ellenőrzi, hogy tényleg a megnevezett típusú adat van-e tárolva a kijelölt területen, s ezt csak úgy tudja megtenni, ha az azt kijelölő mutatóhoz típust rendelünk.

A fordító – na persze nem mindegyik(!) – egyébként akkor is lefordítja a forrást, ha a mutatóhoz rendelt típus nem egyezik a mutató által kijelölt területen lévő

adat típusával, mivel a különböző típusú adatokat jelölő mutatók mérete azonos, viszont figyelmeztetést küld számunkra a fordítás során. Alább egy ilyen esetet eredményező program látható.

```
main(){
  float a; int *mutato;

  mutato=&a;
}
```

... és persze a figyelmeztetés: „*[Warning] assignment from incompatible pointer type*"

Ha a figyelmeztetéstől meg akarunk szabadulni, akkor természetesen az a legjobb, ha „összehangoljuk" a mutatót, a mutatottal, s a megfelelő típust írjuk be annak létrehozásakor, de ha valamiért az jobb (lesz majd később ilyen bőven(!)), akkor megtehetjük azt is, hogy a típuskényszerítés eszközéhez folyamodunk, melyet alább demonstrálok:

```
main(){
  float a; int *mutato;

  mutato=(int*)&a;
}
```

A fenti programban az `a` nevű lebegőpontos változót kijelölő `&a` „mutatót" (címet) úgy tekintjük, mintha `int` típusú változóra mutatna. *Ez a típuskényszerítés*, (amit majd gyakran fogunk alkalmazni e szemeszter folyamán, például a dinamikus helyfoglalású tömbökkel való munkánk során is).

7.3. Mutatók és tömbök

Mint arra már utaltam, a mutatók és a tömbök szegről végről bizony rokonok. Eme állítás belátásához, vegyük szemügyre az alábbi programot, mely futtatáskor egy függvény segítségével feltölt egy tömböt zh–eredményekkel, majd hív egy másik függvényt, ami kiírja a tömb elemeit!

```
#include<stdio.h>

void kiir(int *tomb, int meret);
void feltolt(int *tomb, int meret);
```

```
main()
{
 int i, n=10, jegyek[n];
 feltolt(jegyek, n);
 kiir(jegyek, n);
}

void kiir(int *tomb, int meret)
{
 int i;
 for(i=0;i<meret;i++)
 {
  printf("%d. jegy: %d\n", i+1, tomb[i]);
 }
}

void feltolt(int *tomb, int meret)
{
 int i;
 for(i=0;i<meret;i++)
 {
  printf("%d. jegy: ", i+1); scanf("%d", &tomb[i]);
 }
}
```

Vegyük sorra a tanulságokat!

Az általunk írt `kiir` nevű függvények (`feltolt()`, `kiir()`), első paraméter gyanánt ugyan egy mutatót vártak, mégis, zokszó nélkül vették tudomásul, hogy mi egy tömb (`jegyek`) nevét írtuk be az adott paraméter helyére, s el is végezték a feladatukat.

Ez azért volt lehetséges, mert egy *tömb neve* nem más, mint a tömb első elemének a memóriacímét hordozó *mutató*. Ezt tudva már nem lepődünk meg azon, hogy minden „flottúl" működött, hiszen függvényeink pontosan azt kapták, amit vártak: egy mutatót (illetve annak értékét, egy memóriacímet)[2].

[2]Ez ugyan mind igaz, de - kiegészítendő a fentieket - meg kell jegyeznünk, hogy a tömb neve egy olyan mutató, amelyet nem módosíthatunk, egy konstans pointer. Szemben ugyanis az eddig használt mutatóinkkal, melyeket akár rá is forgathatunk egy tömb elejére, s például fel is tölthetjük segítségükkel az adott tömböt, a statikus tömbök neve olyan mutató, melynek "nyila" / értéke (maga a memóriacím, amit tárol) nem változtatható meg. Míg tehát azt megtehetjük, hogy beírjuk a kódba következőt: `int egyTomb[5], *egyMutato; egyMutato = egytomb;` , vagyis egy mutatót (az `egyMutato`-t) ráál-

Vegyük észre továbbá azt a nem kevésbé fontos tényt is, hogy a kapott mutatót aztán tömbszerűen használták (hiszen indexelgettük rendesen), s vonjuk le a tanulságos következtetést, miszerint a mutatók – ilyenkor, amikor egy már számunkra lefoglalt, tehát „legálisan" igénybe vehető terület elejére vannak „ráállítva" – tömbszerűen is használhatók.

Egyébként *ezért is fontos* a fordítónak tudnia, hogy milyen típusú adat helyét jelöli a mutató, ugyanis – fenti esetet alapul véve – miután megtalálta a `jegyek` tömb első elemét, a továbbiakat úgy leli meg, hogy az első elem helyétől, annyit „lép előre", amennyit a `[]` jelek közötti index értéke jelez neki.

És itt a bökkenő! Hiszen az index értékéből megtudja ugyan, hogy mennyit lépjen az első elemtől a keresettig, ám azt, hogy ezt *mekkora lépésekkel* tegye, kizárólag a tömböt alkotó elemek méretéből képes kikalkulálni, mely méret viszont nem más, mint az adott típus számára igénybevett memóriaterület nagysága.

Ezért kell tudomására hozni a mutatott adat típusát, hisz az egyes elemek mérete is annak a függvénye. Ezért (is) kell egyeznie a mutatóhoz rendelt típusnak, a kijelölt területet elfoglaló adat típusával.

Látható még a fenti programban az is, hogy a függvény hívásakor a tömb címével együtt, második paraméterként át lehet küldeni annak méretét is. Ez bevett gyakorlat a C nyelvben.

A mutatókkal kapcsolatban fontos még szót ejtenünk azok – úgynevezett– **NULL** értékéről!

Mindenekelőtt meg kell említeni, hogy a C nyelvben, egy létrehozott, de értéket még nem kapott változó értéke bármi(!) lehet. Lévén maguk is változók, a mutatók sem kivételek az említett szabály alól. Ez viszont azt jelenti, hogy amíg egy mutató nem kap értéket, addig bárhová mutathat.

Történetesen ez a „bárhová" akár a vezérlés számára nem hozzáférhető helyet is kijelölhet, ami viszont halálos ítéletet jelent az épp futó folyamatra nézve.

Elkerülendő az efféle afférokat, nyilván meg kell valahogy akadályoznunk az ilyen „fertőzött" mutatók programunkban való használatát, amit oly módon tehetünk meg, hogy megkülönböztetjük az értéket még nem kapott – felhasználásra

---

lítunk egy tömb elejére, addig ez "visszafelé" nem igaz. Nem tehetjük meg, hogy egy statikus tomb nevét, mint mutatót "elfordítjuk" a tömb számára lefoglalt memóriaterületről, például valahogy így: `int egyValtozo, egyTomb[5], *egyMutato; egyMutato = &egyValtozo; egyTomb = egyMutato;` . Ez így *szintaktikai* hibát eredményez, vagyis még csak le sem fordul a kód.

még csak véletlenül sem javasolt – mutatókat, a „legális” helyet kijelölő társaiktól.

E nemes szándékot szolgálja a mutatók NULL értékre való beállítása, ugyanis az ilyen értékkel ellátott mutatókról biztosan tudjuk, hogy nem mutatnak olyan helyre, ahonnan olvashatnánk vagy ahová írhatnánk, így még csak véletlenül sem jut eszünkbe azokat felhasználni.

Az alábbi program azt mutatja be, hogyan vehetjük hasznát a NULL érték beállításának, a mutatók használhatóságának ellenőrzésekor.

```
#include<stdio.h>

int osszead(int *tomb, int meret, int *ossz);
int feltolt(int *tomb, int meret);

main()
{
 int i, n=5, osszeg=0, szamok[n];
 int *sz=NULL;

 if(feltolt(sz, n)){printf("Gond van a mutatoval.\n");}

 sz=szamok;

 if(feltolt(sz, n)){printf("Gond van a mutatoval.\n");}
 if(osszead(sz, n, &osszeg))
 {
  printf("Gond van a mutatoval.\n");
 }

 printf("Az elemek osszege: %d.\n", osszeg);
}


int feltolt(int *tomb, int meret)
{
 int i, osszeg=0;
 if(tomb==NULL){return (1);}
 for(i=0;i<meret;i++)
 {printf("%d. elem: ", i+1); scanf("%d", &tomb[i]);}
 return (0);
}
```

```
int osszead(int *tomb, int meret, int *ossz)
{
 int i;
 if(tomb==NULL){return (1);}
 for(i=0;i<meret;i++){ *ossz+=tomb[i]; }
 return (0);
}
```

A fenti program lényege az, hogy megmutassa mire jó a NULL érték. Látható, hogy az `sz` nevű mutatót erre az értékre állítottuk be a létrehozásakor. A `main` első `if`-jében, a `feltolt()` függvény első hívásakor ezt a mutatót adtuk meg az első paraméter helyére (`feltolt(sz, n)`). A hívott függvény viszont – lévén óvatosabb fajta – először ellenőrizte, hogy a kapott mutató biztonsággal felhasználható-e, s mivel az első híváskor azt tapasztalta, hogy annak értéke NULL, legott kiléptette magából a vezérlést a `return` által, s rögtön tudomásunkra hozta a hibát az 1–es szám visszaadása által[3] (ugye szintén csak a `return` segítségével), amit az `if` szépen le is kezelt.

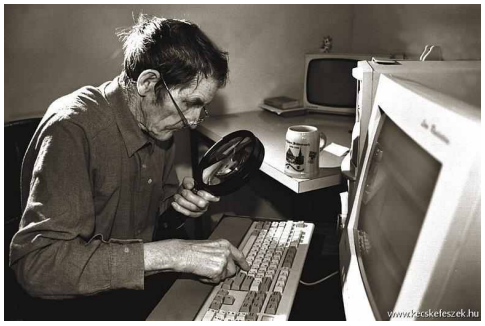
Ezután a programban az `sz` mutató, új értékként megkapta a `szamok` tömb 0 indexű – tehát első – elemének a címét (a tömb neve által), így a `feltolt()` függvény ezt követő, második meghívásakor, az már használható mutatóval lett „ellátva”, amit a `szamok` tömb elemeinek feltöltésével, honorált. Végül e tömb elemeit az `összead()` függvény összegezte.

Egyébként a NULL nem alapvető eleme a C nyelvnek, mint oly sok egyebet, ezt is utóbb „írták hozzá”, de mivel gyakorlatilag a C könyvtár minden eleme használja azt, ezért igénybevételéhez szükségtelen külön fejállományt írnunk programunk előfeldolgozó részébe, ha az már legalább egyet amúgy is tartalmaz.

[3] C nyelvben egy programnak a hibamentes futást rendszerint 0 érték visszaadásával illik jelezni, így hát én itt csak azért adtam 1–et, hogy nullától eltérve, ily módon is utaljak a probléma jelenlétére.

Az alábbiakban megkísérlünk megoldani néhány mutatós-tömbös feladatot. Következzék hát újfent egy kis

7.4. Gyakorlás



Feladat: Írjunk egy programot, mely hív egy függvényt, ami feltölt egy `n` elemű tömböt `m` darab lebegőpontos számmal (`m`-et a függvény kéri be, méghozzá vigyázva, hogy az ne haladja meg az `n` értékét), majd hív egy másik függvényt, ami bekér egy számot, s megvizsgálja, hogy a beküldött szám szerepel e a tömbben, majd értesít minket az eredményről!
Lehetséges megoldás:

```
#include<stdio.h>

void beker(float *sz, int maxmeret, int *meret);
void kereso(float *t, int elemszam);


main()
{
 int n=10, m; float szamok[n];
 beker(szamok, n, &m);
 kereso(szamok, m);
}


void beker(float *sz, int maxmeret, int *meret)
{
  int i;
  do{
      printf("Mennyi szam lesz? (Max. 10 lehet.)\n");
      scanf("%d", meret); /*Nem kell '&' jel a scanf-nek,
      hiszen a 'meret' erteke a main-beli 'm' cime.*/
      }while(*meret>maxmeret);
      /*A *meret erteke a main-beli 'm' erteke, a
      mutato kovetes miatt*/
```

```
  for(i=0;i<*meret;i++)
  {
  printf("%d. szam: ", i+1); scanf("%f", &sz[i]);
  } /*Mivel az 'sz' is, ahogy a 'meret' is mutato,
  azt gondolnank, hogy itt sem kell
  az '&' jel, de jegyezzuk meg, hogy tomb-elemek
  eseteben ki kell tennunk!*/
}

void kereso(float *t, int elemszam)
{
  int i; float szam;
  printf("Kit keresunk? "); scanf("%f", &szam);
  for(i=0;i<elemszam;i++)
  {
   if(szam==t[i]){break;}
  }
  if(i==elemszam)
  {printf("%1.1f nem eleme a tombnek.\n", szam);}
  else{printf("%1.1f eleme a tombnek.\n", szam);}
}
```

Feladat:
Írjunk egy programot, melyben egy függvény feltölt egy `n` elemű tömböt egész számokkal, egy másik függvény kiírja a tömb elemeit, a következő függvény pedig növekvő sorrendbe rendezi a tömb által tartalmazott számokat, majd ezután, a már elkészített „kiíró” fügvény megjeleníti a rendezett tömböt! **Kikötés:** csak egyetlen tömböt használhatunk a rendezéshez, magát a rendezendőt!

Lehetséges megoldás:

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

void beker(int *sz, int elemszam);
void kiir(int *tomb, int meret);
void rendezo(int *t, int hossz);

main()
{
 int n=10, szamok[n];
 beker(szamok, n);
 printf("\nA rendezendo tomb:\n");
```

```
 kiir(szamok, n);
 rendezo(szamok, n);
 printf("\nA rendezett tomb:\n");
 kiir(szamok, n);
}


void beker(int *sz, int elemszam)
{
 int i;
 for(i=0;i<elemszam;i++)
  {
   printf("%d. szam: ", i+1); scanf("%d", &sz[i]);
   system("clear");
  }
}


void rendezo(int *t, int hossz)
{
 int i, j, segedv;
 for(i=0;i<hossz-1;i++)
 {
  for(j=i+1;j<hossz;j++)
  {
   if(t[i]>t[j]){segedv=t[i];t[i]=t[j];t[j]=segedv;}
  }
 }
}


void kiir(int *tomb, int meret)
{
 int i;
 for(i=0;i<meret;i++){printf("%d ", tomb[i]);}
 printf("\n\n");
}
```

Kicsit **gyorsíthatunk** a rendezésen, ha nem cseréljük unos-untalan a tömb elemeit, valahányszor egy kisebb értékkel bíró elemet talál a belső ciklus a külső által indexeltnél.

Ehhez némileg át kell alakítanunk a `rendezo` szerkezetét, úgy mégpedig, hogy a

belső ciklus először keresse meg a tömb – általa éppen vizsgált részének – legkisebb értékkel rendelkező elemét (minimumkeresés), s csak miután megtalálta azt, akkor cserélje ki a vizsgált tartomány elején lévővel, de ezt a cserét is csak abban az esetben tegye meg ha indokolt! (Ha például nem volt már eleve a vizsgált tartomány elején a legkisebb elem.)

Fentiek végiggondolása, valami lentihez hasonlatos függvényt eredményezhet:

```
void rendezo(int *t, int hossz)
{
 int i, j, segedv, min, minindex;
 for(i=0;i<hossz-1;i++)
 {
  for(j=i;j<hossz;j++)
  {
   if(j==i || min>t[j]){min=t[j]; minindex=j;}
  }
  if(i!=minindex)
  {segedv=t[i];t[i]=t[minindex];t[minindex]=segedv;}
 }
}
```

Természetesen összehozhatunk más rendezési eljárást is, elég csak az „algoritmusok” tantárgy keretében látottakra gondolni. Amennyiben a rendezett tömbnél szeretnénk eltekinteni az ismétlődő elemek kiíratásától, készíthetünk egy `kiir2` függvényt is, amit meghívhatunk a rendezés után:

```
void kiir2(int *tomb, int meret)
{
 int i=0;
 while(i<meret)
 {
  if(i<meret-1 && tomb[i]==tomb[i+1]){i++;}
  else{printf("%d ", tomb[i++]);}
 }
 printf("\n\n");
}
```

Feladat:
Írjunk programot, melyben egy eljárás bekér egy értéket, ami egy egész számokból álló négyzetes mátrix oszlopainak (s egyben sorainak is) a száma, ezután bekéri és egy egydimenziós – legfeljebb 25 elemű – tömbben el is tárolja a mátrix elmeit, majd egy másik eljárás meg is jeleníti a mátrixot. Ezután egy újabb eljárás

a megadott mátrix nyomát is kiszámolja, amit mi is megtekinthetünk!

Lehetséges megoldás:

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

void kiir(int *tomb, int oszl);
void beker(int *mat, int m, int *o);
void nyomozo(int *matr, int oszam);

main(){
 int oszlopszam, max=5, matrix[max*max];
 beker(matrix, max, &oszlopszam);
 kiir(matrix, oszlopszam);
 nyomozo(matrix, oszlopszam);
}


void beker(int *mat, int m, int *o)
{
 int i, j;

 do
 {printf("Mennyi␣oszlopa␣(illetve␣sora)␣van␣a␣matrixnak?␣");
  printf("(Max.␣5␣lehet!)\n");
  scanf("%d", o);
  }while(*o>m);

 for(i=0;i<*o**o;i++)
 {
  printf("Kerem␣a␣matrix␣%d.␣elemet!␣",i+1);
  scanf("%d",&mat[i]);
 }
}


void kiir(int *tomb, int oszl)
{
 int i, j; system("clear"); printf("A␣matrix:\n\n");
 for(i=0;i<oszl;i++)
 {
  for(j=0;j<oszl;j++)
```

```
   {
    printf("%d\t",tomb[j+i*oszl]);
   }
   printf("\n\n");
  }
}


void nyomozo(int *matr, int oszam)
{
 int i=0, nyom=0;

 while(i<oszam*oszam){nyom+=matr[i]; i+=oszam+1;}

 printf("A matrix nyoma: %d.\n\n",nyom);
 }
```

Feladat:
Írjunk programot, mely bekéri egy egész számokból álló mátrix oszlopainak és sorainak a számát, ezután bekéri és egy egydimenziós – legfeljebb 25 elemű – tömbben eltárolja a mátrix elmeit, amit meg is jelenít, majd transzponálja – oszlopait soraival felcseréli – a megadott mátrixot, s „szembesít” is a transzponálás eredményével! „Függvényesítsük” a programot belátásunk szerint!

Lehetséges megoldás:

```
#include<stdio.h>

void kiir(int *mat, int sor, int osz);
void beker(int *t, int *sor, int *osz, int max);
void transzponal(int *mat, int *sor, int *osz, int meret);


main()
{
 int sorok, oszlopok, l=25, matrix[l];

 beker(matrix, &sorok, &oszlopok, l);
 transzponal(matrix, &sorok, &oszlopok, l);
 printf("\nA matrix transzponaltja: \n\n");
 kiir(matrix, sorok, oszlopok);
}
```

```
void beker(int *t, int *sor, int *osz, int max)
{
 int i, j;

 printf("Mennyi sora es oszlopa van a matrixnak? ");
 printf("(Max 25 elem!)\n");


 do
 {
  printf("Sor: "); scanf("%d", sor);
  printf("Oszlop: "); scanf("%d", osz);
  }while(*sor**osz>max);

 for(i=0;i<*sor**osz;i++)
 {printf("A matrix %d. eleme: ",i+1);scanf("%d",&t[i]);}

 printf("A megadott matrix alabb tekintheto meg:\n\n");

 kiir(t, *sor, *osz);
}



void transzponal(int *mat, int *sor, int *osz, int meret)
{
 int i, j, segedv, tomb[meret];

 for(i=0;i<*osz;i++)
 {
  for(j=0;j<*sor;j++)
  {
   tomb[j+i**sor]=mat[j**osz+i];
  }
 }

 for(i=0;i<*sor**osz;i++){mat[i]=tomb[i];}

 segedv=*sor; *sor=*osz; *osz=segedv;
}
```

```
void kiir(int *mat, int sor, int osz)
{
 int i, j;
 for(i=0;i<sor;i++)
 {
  for(j=0;j<osz;j++)
  {
   printf("%d\t",mat[j+i*osz]);
  }
  printf("\n\n");
 }
}
```

Feladat:
Írjunk programot, ami bekéri, s egy legfeljebb 10 elemből álló tömbben eltárolja egy `scanf`-fel megadott számú sorral, valamint 1 oszloppal bíró mátrix elemeit, majd a megadott mátrixot, s annak transzponáltját is megjeleníti a képernyőn! A mátrix elemei legyenek kettő tizedesjegyet tartalmazó, lebegőpontos számok! A program, az eredeti és a transzponált mátrix ebben a sorrendben való összeszorzásának eredménymátrixát és annak nyomát is meg kell, hogy jelenítse!

Én itt most csak egy darab függvényt írok meg, a program többi részét a `main`-be teszem és ki-ki kedvére függvényekre bonthatja a folyamatot.

Lehetséges megoldás:

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

void transzor(float *tomb, int sor)
{
 int i,j; float nyom=0;
 for(i=0;i<sor;i++)
 {
  for(j=0;j<sor;j++)
  {
   printf("%1.1f\t",tomb[i]*tomb[j]);
   if(i==j){nyom+=tomb[i]*tomb[j];}
  }
  printf("\n\n");
 }
 printf("\nA szorzatmatrix nyoma: %1.1f\n",nyom);
}
```

```
main()
{
 int i, n, x=10; float matrix[x];

 do
 {
  printf("Mennyi␣sora␣legyen␣a␣matrixnak?␣");
  printf("(Max.␣10␣lehet!)\n");
  scanf("%d",&n);} while(n>x);

 for(i=0;i<n;i++)
 {
  printf("Kerem␣a␣matrix␣%d.␣elemet!␣",i+1);
  scanf("%f",&matrix[i]);
 }

 system("clear");

 printf("A␣megadott␣matrix:\n\n");

 for(i=0;i<n;i++){printf("%1.1f\n\n",matrix[i]);}

 printf("\nA␣megadott␣matrix␣transzponaltja:\n\n");

 for(i=0;i<n;i++){printf("%1.1f\t",matrix[i]);}

 printf("\n\nA␣matrix␣es␣transzponaltjanak␣szorzata:\n\n");

 transzor(matrix, n);
}
```

Végül pedig, a bónusz feladat:
A program kérje be két összeszorzandó mátrix sorainak és oszlopainak számát, a mátrixok szorzásának sorrendjét, döntse el ezek alapján, hogy összeszorozhatóake, s ha igen, a mátrixelemek bekérése után végezze el a műveletet, s jelenítse meg az eredményt! Kikötés: a mátrixok elemeit egydimenziós tömbökben szabad csak tárolni (elég, ha a mátrixok legfeljebb 20 elemmel bírnak)! Elkötelezettebb hallgatók, feltölthetik a két összeszorzandó mátrix elemeit egyetlen tömbbe is. Alább, ez utóbbi utat követem majd.

Mint az látható, a lenti program nincs „függvényesítve”, hanem **elrettentő példaként** minden a `main`-be van ömlesztve benne.

Ha tekintetbe vesszük, hogy ez csupán egy egyszerű – két, limitált mátrixot összeszorzó – programocska, elképzelhetjük, hogy micsoda átláthatatlan dzsungelbe keverednénk egy valós, életközeli program megfogalmazásakor, ha mindent a `main`-be gyömöszölnénk, hiszen egy olyan programnak, egy ilyen kis algoritmus csupán egy ici-pici részét képezné ...

Lehetséges megoldás, vagyis inkább: **elrettentő** megoldás:

```
#include<stdio.h>

main(){
int n, m, k, l, i, j, a, x, z, v, y=40;
float matrixelem, sv, matrixok[y];


printf("Kerem␣A,␣B␣matrix␣sorainak␣es␣oszlopainak␣szamat");
printf(",␣ebben␣a␣sorrendben,␣ENTER-rel␣elvalasztva!");
printf("␣Szorzatuk␣erteke␣mindket␣matrix␣eseten␣");
printf("legfeljebb␣%d␣lehet!\n", y/2);

scanf("%d%d%d%d", &n,&m,&k,&l);

if((m!=k && l!=n) && (n*m>y/2 || k*l>y/2)){z=1;}
else{if(m!=k && l!=n){z=2;}
else{if(n*m>y/2 || k*l>y/2){z=3;}
else{z=4;}}}

switch(z)
{
case 1:
printf("A␣ket␣matrix␣semmilyen␣sorrendben␣nem␣szorozhato");
printf("␣ossze␣es␣az␣elemek␣szama␣is␣nagyobb␣a␣");
printf("megengedettnel.\n");
break;

case 2:
printf("A␣ket␣matrix␣semmilyen␣sorrendben␣nem");
printf("␣szorozhato␣ossze.\n");
break;

case 3:
printf("A␣matrixelemek␣szama␣nagyobb␣a␣megengedettnel.\n");
break;
```

```
case 4:
if(m!=k)
{printf("A␣ket␣matrix␣AxB␣sorrendben␣NEM␣szorozhato!\n");}
else
{if(n!=1)
 {
  printf("A␣ket␣matrix␣BxA␣sorrendben␣NEM␣szorozhato!\n");
 }
}

printf("Kerem␣az␣A␣matrix␣elemeit,␣balrol␣jobbra,");
printf("␣sorfolytonosan!\n");
for(i=0;i<n*m;i++){scanf("%f", &matrixok[i]);}

printf("Kerem␣az␣B␣matrix␣elemeit,␣balrol␣jobbra,");
printf("␣sorfolytonosan!\n");
for(i=0;i<k*l;i++){scanf("%f", &matrixok[i+y/2+1]);}

printf("Ha␣AxB␣sorrendben␣szorozzak␣irj␣be␣egy␣1-est,␣");
printf("ha␣BxA-ban,␣akkor␣egy␣2-est,␣");
printf("majd␣nyomj␣egy␣ENTER-t!\n");
scanf("%d", &x);

if((x==1 && m!=k) || (x==2 && l!=n))
{
printf("Mar␣szoltam,␣hogy␣ebben␣a␣sorrendben␣");
printf("nem␣szorozhatoak␣ossze.\n");
break;
}
else
 {
 if(x==1)
 {
  printf("A␣matrixok␣-␣adott␣sorrendu␣-␣szorzata:\n\n");
  for(a=0;a<n;a++)
  {
   for(i=0;i<l;i++)
   {
   for(matrixelem=0,j=0;j<m;j++)
   {
    matrixelem+=matrixok[j+a*m]*matrixok[y/2+i+1+j*l];
   }
```

```
    printf("%1.2f\t", matrixelem);
    }
   printf("\n\n");
   }
  }
   else
   {
   if(x==2)
    {
     printf("A matrixok - adott sorrendu - szorzata:\n\n");
    for(i=0,sv=0;i<y/2;i++)
    {
     sv=matrixok[y/2+i+1];
     matrixok[y/2+i+1]=matrixok[i];
     matrixok[i]=sv;
    }
    for(a=0;a<k;a++)
    {
     for(i=0;i<m;i++)
     {
      for(matrixelem=0,j=0;j<l;j++)
      {
       matrixelem+=matrixok[j+a*l]*matrixok[y/2+i+1+j*m];
      }
      printf("%1.2f\t", matrixelem);
     }
     printf("\n\n");
    }
    }
    }
   }
}
}
```

Szorgalmi feladat:

Tegyük a fenti programot kulturáltabbá!

Mindenekelőtt különítsük el a részfeladatokat függvényekre, tegyük felhasználóbarátabbá is (pl.: lehessen „újrázni”, ha mondjuk nem megengedett sorrendben szerettünk volna szoroztatni, stb).

Ha mindezzel megvagyunk, érdemes lehet a függvényeket elmenteni egy `linearalgebra.c` állományba (akár a transzponáló, stb függvényekkel együtt), hogy aztán a `main`-ben már csak hívni kelljen őket ... persze, csak miután beírtuk az előfeldolgozónak szóló részbe, hogy `#include"linearalgebra.h"`.

Egyelőre ennyi. :)

8. fejezet

Karakterláncok (string-ek)

Remélem, mindenkinek sikerült kellő mélységig megértenie az előző fejezet anyagát, mert a továbbiakban, kizárólag a mutatók és a tömbök értő kézzel való használata által boldogulhatunk. Rendszerint a szövegeket – meg persze nem csak azokat – a C nyelvben, a `char` változótípusból készített *tömbökkel* tudjuk kezelni, melyeket *karakterláncoknak* hívunk.

A tananyag könnyebb megértéséhez kicsit vissza kell kanyarodnunk, az eddig méltatlanul hanyagolt `char` típushoz és annak használatához. Ráérünk majd akkor "láncra verni" őket, ha relatíve pontosan értjük az egyes láncszemeket is.

A `char` típus ugye egy 8 bites (1 byte-os) szám *és* egyben karakter. De mit jelenthet ez az „és egyben"? A választ magunk is megadhatjuk, ha áttanulmányozzuk, majd futtatjuk az alábbi kódot, s átrágjuk a futtatás folyamán képernyőre írt dolgokat!

```
#include<stdio.h>

main()
{
char a;
printf("Kerem␣a␣karaktert:␣"); scanf("%c", &a);
printf("A␣%c␣karakter␣ASCII␣kodja:␣%d.\n", a, a);
}
```

Látható, hogy egyfelől az `a`-ra tekinthetünk egy „mezítlábas" egész számként (ilyenkor `%d`-t használunk a kiíratás során), másfelől viszont tekinthetünk rá úgy is, mint ASCII–kódra. Ha az utóbbit választjuk és `%c`-t használunk a kiíratásakor, akkor ama karakterként hivatkozunk rá, melynek az ASCII–kódja nem más, mint

az a változó számértéke.

Természetesen ekkor nem az a változó számértékét írja ki a vezérlés, hanem az a változó számértéke által (ASCII) kódolt karaktert.

Vajon hivatkozhatunk-e a billentyűzeten szereplő karakterek ASCII kódjára anélkül, hogy előbb char típusba „ágyaznánk" őket? Nyilván igen, úgy mégpedig, hogy a karaktert ' ' -jelek közé szorítjuk, ugyanis ezzel az adott karakter ASCII kódját csalogatjuk elő. Nézzünk egy példát, s legott világos lesz!

```
printf("%d a karakter ASCII kodja. \n", 'b');
```

Az utasítás hatására megjelenik a képernyőn a 98-as szám, mint a b betű ASCII kódja. Nyilvánvaló, hogy bármely más karakter esetén is működik ez a módszer, ahogyan az is, hogy az ilyen printf-es játékoknál épületesebb célokat hivatott szolgálni, mivel akár kifejezésekbe, feltételekbe is beépíthetjük így a karaktereket, illetve kódjuk értékeit. (Ahogy tettük is volt azt már egyszer-egyszer, az eddigiek során is.)

Erről később még beszélünk, addig is – némi ízelítőként – próbáljuk meg eldönteni egy adott karakterről, hogy számjegy-e! Továbbá: a használatukon keresztül ismerkedjünk meg a getchar és putchar függvényekkel!

Itt ugyan erőltetettnek tűnhet, hogy helyenként a printf és a scanf helyett alkalmazzuk őket, mégis meglépem ezt, mert:

1. jól megérthető így a működésük,
2. később még jól fognak jönni (ahogy a rokonaik is)!

Nosza!

```
#include<stdio.h>

main()
{
 char a, ujsor='\n';
 /*igen, a soremelest lehet igy - '\n' - is jelezni*/
 printf("Kerem a karaktert! ");
 a=getchar();

 if(a>='0' && a<='9')
 {putchar(a); putchar(ujsor); printf("Szamjegy.\n");}
 else
 {putchar(a); putchar(ujsor); printf("Nem szamjegy.\n");}
}
```

(A két új függvényről egyelőre nem is nagyon írnék, ha valakit több érdekel róluk, az nézzen körül a C könyvtár függvényei között vagy egyéni „próbálgatás” útján tapasztalja ki a működésüket!)

A fenti kis programból leszűrhető az is, hogy a 0 és a 9 közötti számjegyek, az ASCII kódtáblán egymást követik, amennyiben a kódértékeik 0-tól „jobbra” haladván eggyel nagyobbak a tőlük „balra” találhatóaknál.

Nincs ez másképp az angol abc esetén sem. A betűk előbb **A-Z** -ig, majd rögtön a Z után, **a-z** -ig követik egymást, midőn a kódjuk értéke minden egyes – *abc* szerint – növekvő irányba tett lépés esetén 1-gyel nő.

Ennyi ismeret már elegendő ahhoz, hogy végre rátérhessünk a karakterláncokra.

C-ben tehát a karakterláncok **mindig** *tömbök*. A karakterláncok **vég**ét a **0 kódú** karakter jelzi. Ez *nem* a 0 *számjegy*, hanem a 0 értékű bájt!
Nézzünk egy példát!

```
#include<stdio.h>

int szoveghossz(char *tomb);

main()
{
 char *szoveg;
 szoveg="Hello␣Kitty!";
 printf("A␣'%s'␣szoveg␣hossza:␣", szoveg);
 printf("%d␣karakter\n", szoveghossz(szoveg));
}

int szoveghossz(char *tomb)
{
 int n=0;
 while(tomb[n]!=0){n++;}
 return(n);
}
```

Összpontosítsunk először kizárólag a `main`-re! Létrehoztunk egy `szoveg` nevű mutatót, mely `char` típus helyét jelöli a memóriában. Ebben eddig semmi újdonság nincs, nem úgy az *értékadás módjában*! Ilyet még nem láttunk. A C nyelvben a *kettős idézőjel a karakterlánc típusú* **állandókat** jelzi.

Ilyen esetben a következő történik:
Az idézőjelek közötti szöveget a fordító elhelyezi a futtatható állományban, de úgy,

hogy jelzi annak a végét is a 0 kódú karakter hozzáfűzése által (ez tehát nem a mi dolgunk, nem kell vele külön bajlódnunk, a fordító megteszi helyettünk). Ezután pedig, magába a programba, oda, ahol eddig a karakterlánc (a szöveg) volt, beteszi azt a memóriacímet, ami a szöveg helyét jelöli a memóriában, ezáltal értéket adva a mutatónak, melyet az egyenlőségjel másik oldalán hagytunk sorsára.

Lépjünk tovább a `printf`-ben foglaltakra! A `%s` jelzi a függvénynek, hogy a kapcsolódó paraméter (itt most a `szoveg`) egy mutató, ami egy karaktertömböt jelöl. Az `%s` által motiválva a függvény egymás után kiírja az itt levő karaktereket, egészen az első 0 kódú karakterig! (Látható, hogy a `szoveg` mutató típusa is megfelelő!)

A 2. `printf` második paraméterében hívjuk az általunk már megírt függvényt, mely a beírt szöveg hosszát hivatott lemérni. Látható, hogy egy elöltesztelő ciklussal végigfut a kapott karakterláncon, egészen annak 0 kódú karakteréig, de azt már nem számolja bele, hiszen elöltesztelő!

Ezután – munkája végeztével – a ciklusváltozó aktuális értékét adja vissza a hívónak. Legyünk résen és vegyük észre, hogy az így visszaadott érték eggyel kevesebb, mint a szöveget tároló bájtok száma, hiszen a nulla kódú (a lezáró) karakter már nem lett beszámítva, s jól is van ez így, mert mi a szöveg hosszára voltunk kíváncsiak, nem pedig a tárolásához szükséges karaktertömb elemeinek a számára!

Feladat: Írjuk át `for`-ciklusba a fenti program `szoveghossz` függvényét!

Lehetséges megoldás:

```
int szoveghossz(char *tomb)
{
 int n;
 for(n=0;tomb[n]!=0;n++);
 return(n);
}
```

Feladat: Írjuk át az előző program szöveghosszt mérő függvényét olyan `for`-ciklusúvá, mely ciklus fejében csak a középső rész jelenik meg!
Lehetséges megoldás:

```
int szoveghossz(char *tomb)
{
 int n=0;
 for(;tomb[++n]!=0;);
 return(n);
}
```

Látható, hogy `++n` -t írtunk `n++` helyett! Azért volt erre szükség, mert utóbbi esetben – elérvén a lezárókaraktert – annak megvizsgálása után már nem engedné ugyan a vezérlés érvényesülni a `for`-ciklus fejének harmadik részét, (mely a ciklusváltozó értékét hivatott növelni (bár most az úgyis üres)), mégis megnőne az `n` értéke, hiszen elhagyván a második részt – a `tomb[n++]!=0` -t – az `n++` utóhatásaként, a program egyet még hozzáadna ahhoz. Lássuk meg azonban azt is, hogy az általunk adott megoldás viszont ebben az esetben kvázi „hátultesztelővé" tette az eredetileg elöltesztelő `for`-ciklust! Bizony, bizony, ugyanis a *karakterlánc első elemét, a nulla indexűt, úgy ugorja át, hogy tudomást sem vesz annak értékéről, hiszen rögtön az egyes indexnél, a második elemnél kezdi a vizsgálódást!* Magyarán **hibásan működik**, ugyanis ha a szöveg egy elemet sem tartalmaz (ha közvetlenül egymás mellé tesszük a kettős idézőjeleket, így: `szoveg="";`) a fordító akkor is „lezárja" azt, tehát a karakterláncnak egy eleme mindenképp lesz, mégpedig a nulla kódú lezárókarakter!

Ebben az esetben pont ennek az elemnek a vizsgálatát mulasztja el a függvény, mert hát az első elemet átugorja, hiszen a feltételbe mindjárt az 1-es indexet, a második elemet passzírozza bele, így akkor is 1-es szöveghosszt jelez, ha nem is írtunk be szöveget.

(Arról pedig már ne is beszéljünk, hogy az adott karakterláncnak (tömbnek) nincs is 1-es indexű eleme, mert összesen 1 elemből áll (a nulla kódú lezárókarakterből), aminek indexe esetünkben 0. Magyarán, szerencsénk van, ha nem hal bele a program egy memóriafelosztási hibába, ugyanis egy 1 elemet tartalmazó tömbnél, annak második (tehát 1-es indexű) elemére is hivatkoztunk!)

Feladat:
Írjuk át az előző program szöveghosszt mérő függvényét oly módon, hogy az továbbra is olyan `for` ciklussal bírjon mely ciklus fejében csak a középső rész jelenik meg, mégis működjön megfelelően!

Lehetséges – elég butácska – megoldás:

```
int szoveghossz(char *tomb)
{
  int n=0;
  if(tomb[0]==0){return(n);}
        else
        {
         for(;tomb[++n]!=0;);
        }
  return(n);
}
```

Másik lehetséges – egy fokkal elegánsabb – megoldás:

```
int szoveghossz(char *tomb)
{
 int n=-1;
 for(;tomb[++n]!=0;);
 return(n);
}
```

Próbáljuk csak ki az utóbbi három függvényt úgy, hogy csak `szoveg="";` -t írunk be! Láthatjuk, hogy az első 1-et ír ki, míg az utóbbi kettő már helyesen, 0-t.

Feladat: Írjuk át hátultesztelő ciklust tartalmazóvá a `szoveghossz` függvényt!
Lehetséges megoldás:

```
int szoveghossz(char *tomb)
{
 int n=-1;
 do{n++;}while(tomb[n]!=0);
 return(n);
}
```

Azt hiszem, ezt a függvényt már eléggé körülrajongtuk.... ideje továbblépnünk:

Feladat: Írjunk programot, mely egy általunk írt függvény segítségével, átmásol egy karakterláncot egy üres tömbbe!
Lehetséges megoldás:

```
#include<stdio.h>

void masol(char *ide, char *innen);

main()
{
 char tomb[20];
 masol(tomb, "Hello␣Kitty!:)");
 printf("Az␣atmasolt␣szoveg:␣%s\n", tomb);
}

void masol(char *ide, char *innen)
{
 int i;
 for(i=0;innen[i]!=0;i++){ide[i]=innen[i];}
 ide[i]=0;
}
```

A függvény a *második* paraméter területé*ről teszi* a karaktereket az *első* paraméterként kapott hely*re*, a *0* karakter*ig*. A *másolás*t – tömbökről lévén szó – elemről elemre, az *értékadás* végzi. Vegyük észre, hogy ilyen esetben – amikor tömböket masszírozunk – nem kell külön figyelnünk a 'mutató követésére', *-ozás nélkül is maradandó változást idéz elő a tömb elemeiben a hívott függvény!
Az `ide[i]=0;` sorra azért van szükségünk, hogy a másolást végző ciklus munkája után le is zárjuk a karakterláncot. Próbáljuk ki a programot nélküle, hogy lássuk miért is fontos odabiggyesztenünk ezt az utasítást! (Nem feltétlenül fog felszínre bukkanni a hiba, de attól még ott van!)

Feladat: Írjunk egy programot, melyben elhelyezünk egy függvényt, aminek hatására az alábbi szöveg kisbetűi naggyá lesznek! :)

> "Up he rose and donn'd his clothes
> Dopp'd the chamber door
> Let in the maid, that out a maid
> Never departed more."
> Shakespeare[1]

Háttérismeret: mint már írtam, az ASCII kódtáblán az angol nyelvű ABC nagybetűi A-tól Z-ig egymást követik, növekvő értékű kódokkal, s ugyanez igaz a kisbetűkre is, ahogy az is, hogy rögtön a nagybetűk „után" következnek. Emlékezzünk arra is, hogy egy adott karakter kódjának értékére úgy hivatkozhatunk, hogy azt ' jelek közé szendvicseljük, miáltal egy `int` típusú állandót kapunk, ami a karakter ASCII kódjának az értéke!

Lehetséges megoldás:

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

void nagybeture(char *t);
```

[1] „Kelt a legény, felöltözött
Ajtót nyitott neki
Bement a lány, de mint leány,
Többé nem jöve ki."
Arany János

```
main()
{
 char tomb[120];
 strcpy(tomb, "Up␣he␣rose␣and␣donn'd␣his␣clothes/␣Dopp'd
␣the␣chamber␣door/Let␣in␣the␣maid,␣that␣out␣a␣maid/
␣Never␣departed␣more.");
 /*Ezt a szoveget, az egesz strcpy-s reszt,
 IRJUK EGY SORBA, ha kiprobaljuk a kodot!*/
 nagybeture(tomb);
 printf("%s\n", tomb);
}

void nagybeture(char *t)
{
 int n, N;
 N=strlen(t);
 for(n=0;n<N;++n)
 {if(t[n]>='a' && t[n]<='z'){t[n]=t[n]-('a'-'A');}}
}
```

A megoldás lényegi részét az utolsó előtti sorban találjuk, s nincs hozzáfűznivalóm, mert az magáért beszél...
Csak vicceltem.....lásd lentebb! :)

Ami újdonság gyanánt feltűnhet az az `strlen` függvény, ami a karakterlánc hosszát mondja meg és az `strcpy` (szintén függvény) mely a másolást végzi, mégpedig a második paraméteréből (ami ugye itt maga a karakterláncot jelölő mutató) az elsőbe, ami pedig az üres tömbünk. Ők ketten a `string.h` lakói. Egyébiránt, igen hasonló módon működnek, mint az általunk e céllal már önállóan megírt, szöveghosszat mérő és szöveget másoló függvények!

Veregessük magunkat vállon egészen nyugodtan! ;)

`N` a tömb hosszát tartalmazza, `n`-nel pedig végigzongorázzuk a tömböt. Az `if`-es rész alakítja naggyá a betűket.

Ha egy betű kódjának értéke az **a** és **z** közé esik az ASCII kódtáblában (tehát kisbetű), akkor a vezérlés az **a** és **A** kódja közti különbséget kivonja a betű kódjából, magyarán a neki megfelelő nagybetű kódjának értékét adja meg az adott karaktertömb elemnek, mint új ASCII kódot. (Gondolhatunk erre a műveletre úgy is, hogy „eltolja” a betűt egy *abc*-nyivel, így az pont a megfelelő nagybetű helyére kerül.)

(ASCII-ben egyébként az angol betűk vannak sorrendben, ezért az ékezeteseket –

amik elszórtan „lézengenek” a kódtáblán – elrontaná ez a módszer.)

Feladat: Írjunk át a fenti program függvényét úgy, hogy most cserélje ki a kisbetűket nagyra és viszont!

Lehetséges megoldás:

```
/* Minden ugyanaz, csak valtoztassuk meg a fuggveny
nevet a tipusdef.-nel es a hivasnal is!*/

void kisnagybeture(char *t)
{
 int n, N;
 N=strlen(t);

 for(n=0;n<N;++n)
 {
  if(t[n]>='a' && t[n]<='z'){ t[n]=t[n]-('a'-'A'); }
  else{ if(t[n]>='A' && t[n]<='Z'){t[n]=t[n]+('a'-'A');} }
 }
}
```

Egy fontos tanulság, amiről mindenképp meg kell még emlékeznünk e helyütt! Tegyük fel, hogy egy mutató megkapja az általunk idézőjelbe írt szöveg memóriacímét, s mi, miután használtuk, kiírattuk, másoltuk, stb, úgy döntünk, hogy ideje módosítani rajta. Nézzünk erre is egy példát, mellyel kapcsolatban már most előrebocsátható, hogy (direkt) hibás:

```
#include <stdio.h>

main()
{
 char *tomb="Hello Kitty!";
 tomb[0]='h';
 printf("%s\n", tomb);
}
```

Próbáljuk futtatni!.....hoppsz! Mi lehet a hiba?
A gondot az okozta, hogy egy karakterlánc típusú *állandót* próbáltunk módosítani! Ugyanis a programban egy karakterlánc típusú állandót hoztunk létre, aminek címét a `tomb` nevű mutatóval jelöltük. A fordító azonban a *karakterlánc típusú állandókat olyan helyre teszi, amit a programnak nincs joga bolygatni*! Mi viszont pont erre akartuk rávenni a `tomb[0]='h';` paranccsal... naná, hogy visszaszólt

(bár lefordította(!), az igaz)!
Hogy használhatunk módosítható karakterláncot? Valahogy így:

```
#include <stdio.h>

main()
{
 char tomb[]="Hello␣Kitty!";
 tomb[0]='h';
 printf("%s\n", tomb);
}
```

Ez már döfi(!) :), hisz itt nem mutatót, hanem egy tömböt hoztunk létre, ami már módosítható, a program számára is hozzáférhető memóriaterületen van. Arra az esetre, ha a mutatós-tömbös-karakterláncos témakörben valaki vágyik még némi szellemi kihívásra, javaslom, hogy próbálja meg önállóan megoldani az alább következő néhány feladatot!

8.1. Gyakorlás

Feladat: Írjunk programot, mely – miután meghívja az általunk írt függvényt, annak segítségével – egy szöveget annyiszor ír ki a képernyőre, ahány szóból áll az!

Lehetséges megoldás:

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

void szoszo(char *tomb);

main()
{
 char mondat[]="Beam␣me␣up,␣Scotty!:)";
 szoszo(mondat);
}

void szoszo(char *tomb)
{
 int i, hossz, szoszam=1;
 hossz=strlen(tomb);
 for(i=0;i<hossz;i++)
```

```
 {
  if(i==0 || tomb[i]==' ')
  {printf("%d.: %s\n", szoszam, tomb); szoszam++;}
 }
}
```

Feladat:
Írjunk programot, s benne egy függvényt, ami két – könnyítés gyanánt – egyforma hosszú karakterlánc tartalmát kicseréli, s ehhez a karakterláncokat tartalmazó tömbökön kívül nem használ további tömböt segítség gyanánt!

Lehetséges megoldás:

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

void cserebere(char *tomb1, char *tomb2);

main()
{
 char mondat1[]="Beam me up Scotty!";
 char mondat2[]="Hello Kitty!!!!!!!";
 printf("Az elso mondat: %s\n", mondat1);
 printf("A masodik mondat: %s\n\n", mondat2);
 cserebere(mondat1, mondat2);
 printf("Az elso mondat: %s\n", mondat1);
 printf("A masodik mondat: %s\n\n", mondat2);
}

void cserebere(char *tomb1, char *tomb2)
{
 int s, hossz, i;
 hossz=strlen(tomb1);
 for(i=0;i<hossz;i++)
 { s=tomb1[i]; tomb1[i]=tomb2[i]; tomb2[i]=s; }
}
```

Feladat: Írjunk programot, mely meghív egy általunk készített függvényt, ami a *„Reszkess, mert Pampalini a nyomodban van!”* mondat minden páratlan sorszámú („első”) szavát csupa nagybetűből állóvá alakítja!

Lehetséges megoldás:

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

```
#include<string.h>

void mindenELSO(char *tomb);

main()
{
 char mondat[]="Reszkess, mert Pampalini a nyomodban van!";
 mindenELSO(mondat);
 printf("%s\n", mondat);
}

void mindenELSO(char *tomb)
{
 int i=0, hossz=strlen(tomb);
 while(i<hossz)
 {
  while(tomb[i]!=' ')
  {
   if('a'<=tomb[i] && tomb[i]<='z')
   {tomb[i]-='a'-'A'; } i++;
  }
  i++;
  while(tomb[i]!=' '){ i++; }
  i++;
 }
}
```

Feladat: Programunk tegye *abc* -sorrendbe a „*workaholic*” szó betűit, egy erre a célra írt, majd hívott függvény segítségével! (Nem baj ha több azonos betű szerepel egymás mellett.) *Lehetséges megoldás*:

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

void abc(char *tomb);

main()
{
 char szo[]="workaholic";
 printf("A '%s' ABC sorrendben: ",szo);
 abc(szo);
 printf("%s\n",szo);
}
```

```
void abc(char *tomb)
{
 int i, j, min, minindex, hossz=strlen(tomb); char s;
 for(i=0;i<hossz-1;i++)
 {
  for(j=i;j<hossz;j++)
  {
   if(j==i || min>tomb[j]){min=tomb[j]; minindex=j;}
  }
  if(i!=minindex)
  {s=tomb[i];tomb[i]=tomb[minindex];tomb[minindex]=s;}
 }
}
```

Ha szeretnénk elkerülni a betűismétlést, így is eljárhatunk:

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

void abc(char *tomb);
void kiir(char *t);

main()
{
 char szo[]="workaholic";
 printf("A␣'%s'␣ABC␣sorrendben:\n",szo);
 abc(szo);
 kiir(szo);
}

void abc(char *tomb)
{
 int i, j, min, minindex, hossz=strlen(tomb); char s;
 for(i=0;i<hossz-1;i++)
 {
  for(j=i;j<hossz;j++)
  {
   if(j==i || min>tomb[j]){min=tomb[j]; minindex=j;}
  }
  if(i!=minindex)
  {s=tomb[i];tomb[i]=tomb[minindex];tomb[minindex]=s;}
 }
}
```

```
void kiir(char *t)
{
 int i, meret=strlen(t);
 for(i=0;t[i]!=0;i++)
  {
    while(i<meret-1 && t[i]==t[i+1]){ i++; }
    putchar(t[i]);
  }
 printf("\n");
}
```

Feladat („nyers" változat): Az *„en ezt mar mondtam talan egyszer neked"* karakterláncban látható szavak tetszőleges sorrendben elrendezve újra és újra változatlan jelentésű mondatot szülnek. Fogalmazzunk meg egy programot, mely a felhasználó által megadott sorrendben írja ki a karakterláncba foglalt szavakat, ezáltal is szemléltetve a feladat alapvetésében kifejtett állítás igazságát!

Lehetséges megoldás:

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

void sorrend(int *sor, int n);
void szemleltet(char *klanc, int *sor, int n);

main()
{
 int n=7, sorozat[n];
 char mondat[]="en␣ezt␣egyszer␣talan␣mondtam␣mar␣neked";
 sorrend(sorozat, n);
 szemleltet(mondat, sorozat, n);
}

void sorrend(int *sor, int n)
{
int i;
printf("Kerem␣a␣kiirasi␣sorrendet!␣(pl.:3452671)␣");
printf("Minden␣szam␣utan␣nyomjunk␣ENTER-t!\n");
for(i=0;i<n;i++){scanf("%d", &sor[i]);}
}


void szemleltet(char *klanc, int *sor, int n)
```

```
{
 int i, j, szoszam, l=strlen(klanc);
 for(i=0;i<n;i++)
 {
  for(j=0, szoszam=1;j<l && szoszam<=sor[i];j++)
   {
    if(szoszam==sor[i]){ putchar(klanc[j]); }
    if(klanc[j]=='␣'){ szoszam++; }
   }
  }
 printf("\n");
}
```

Amennyiben finomítani szeretnénk a programot, átépíthetjük olyanra is, ami már képes kiküszöbölni a szavak esetleges egybeírását, ügyel a mondat nagy betűvel való kezdésére, a végén a pontra, stb, így valahogy:

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

void sorrend(int *sor, int n);
void szemleltet(char *klanc, int *sor, int n);


main()
{
 int n=7, sorozat[n];
 char mondat[]="en␣ezt␣egyszer␣talan␣mondtam␣mar␣neked";
 sorrend(sorozat, n);
 szemleltet(mondat, sorozat, n);
}


void sorrend(int *sor, int n)
{
int i;
printf("Kerem␣a␣kiirasi␣sorrendet!␣(pl.:3452671)␣");
printf("Minden␣szam␣utan␣nyomjunk␣ENTER-t!\n");
for(i=0;i<n;i++){scanf("%d", &sor[i]);}
}


void szemleltet(char *klanc, int *sor, int n)
```

```
{
 int i, j, szoszam, l=strlen(klanc);

 for(i=0;i<n;i++)
 {
  for(j=0, szoszam=1;j<l;j++)
  {
   if(klanc[j]=='␣'){ szoszam++; j++; }
   if(szoszam==sor[i] && i==0 && (klanc[j-1]=='␣' || j==0))
    {
     klanc[j]-='a'-'A'; putchar(klanc[j]);
    }
    else{ if(szoszam==sor[i]){ putchar(klanc[j]); } }

    if(i<n-1 && j==l-1){printf("␣");}
    else{ if(i==n-1 && j==l-1){ printf(".\n"); } }
   }
 }
 printf("\n");
}
```

A fenti programot csinosítandó – egyebek mellett –, érdemes feloldani azt a problémát vagy inkább nehézséget, mellyel a felhasználó a sorrend megadásakor szembesül, jelesül: minden egyes számjegy után le kell nyomnia az `enter` -t. Egy felhasználó ugyanis – jogosan – arra számít, hogy be kell írnia egy számsort, majd az utolsó számjegy után le kell nyomnia az `enter` billentyűt. Egyszer. Ehhez képest viszont .....

Mit tehetünk mi, fejlesztők ez ügyben? Az egyik megoldás, ami eszünkbe juthat, az az, hogy bekérjük a számsort egy darab egész számként, egy darab változóba, viszont ez esetben a 10 hatványai szerinti (a helyiértékek szerinti) tényezőkre kellene bontani a beadott számot, s a 10 megfelelő – egész kitevős – hatványainak szórzói adnák ki végül a beadott szám számjegyeit.

Megoldható így is a feladat, de talán (biztosan) egyszerűbb, könnyebben járható egy másik út: olvassuk be a számsort egy karakterláncba, majd a karakterláncot vizsgálva derítsük ki, hogy annak egyes elemei melyik számjegyet kódolják az ASCII táblán, s ezeket a számjegyeket írjuk ki abba a tömbbe, amelyik tartalmazza a szavak kiíratásának a sorrendjét (esetünkben ez a tömb a `sorozat[n]`).

Lehetséges megoldás:

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

void sorrend(int *sor);
void szemleltet(char *klanc, int *sor, int n);


main()
{
 int n=7, sorozat[n];
 char mondat[]="en␣ezt␣egyszer␣talan␣mondtam␣mar␣neked";
 sorrend(sorozat);
 szemleltet(mondat, sorozat, n);
}


void sorrend(int *sor)
{
 int i=0, n=8, j, k; char sorrend[n];
 printf("Kerem␣a␣kiirasi␣sorrendet!␣(pl.:3452671)␣");
 scanf("%s", sorrend);
 while(sorrend[i]!=0)
 {
  for(j='0', k=0; j<='9'; j++, k++)
   {
    if(j==sorrend[i]){break;}
   }
  sor[i]=k; i++;
 }
}


void szemleltet(char *klanc, int *sor, int n)
{
/*Ez a resz nem valtozott.*/
}
```

A `sorrend` eljárásban látható ”számvisszafejtéssel” kapcsolatban érdemes megjegyezni, hogy innen már csak egy lépés, hogy karakterláncként megadott – egész – számokat, számmá konvertáljunk, hiszen az egyes helyiértékeket szorzó számjegyek már megvannak, így a tennivaló már csupán annyi, hogy megszorozzuk őket a 10-es számrendszer megfelelő helyiértékeivel (a 10 megfelelő hatványaival), majd az így kapott szorzatok eredményeit összeadjuk. Az így kapott összeg

nem más, mint a szám, amit kezdetben karakterláncként adtak be a programba. Ha valaki szeretné, csámcsogjon el nyugodtan ezen a feladaton, s ha úgy érzi, hogy nagyon penge, oldja meg a dolgot a lebegőpontos számok esetére is!

9. fejezet

Struktúrák, struktúratömbök, egyebek

9.1. Struktúrák

E fejezettel érkeztünk el ama határhoz, melyen átlépve már nem csupán a C nyelv beépített – egyszerű vagy összetett – típusait vehetjük igénybe a különböző feladatok kezelésére írt programjainkban, hanem mi magunk hozhatunk létre – az adott feladat lényegének megragadását leginkább segítő – belső szerkezettel (*struktúrával*) bíró adatszerkezeteket.

Egyszerűen arról van szó csupán, hogy az emberi gondolkodás „szereti egy helyen tudni” a valamilyen értelemben összekapcsolható dolgokat azok összefűzése által, s nem vágyik rá túlzottan, hogy innen-onnan kelljen „összecsipegetni” az összetartozó információkat. A programnyelv pedig – lehetőségeihez mérten – igyekszik ehhez a gondolkodásmódhoz alkalmazkodni. Amíg az eddig tanultak folyamán mindössze abban merült ki az ilyen feladat, hogy – teszem azt – sok szám tartozott össze valamilyen értelemben (például a zh jegyek az átlagszámoláshoz), melyeket ésszerűbb volt külön változók helyett egyetlen változóban összefogva tárolni, addig a *tömbök* felkínálásával támogatott minket a C nyelv.

A valós esetek zömében azonban nem úszhatjuk meg a dolgokat ilyen egyszerűen, s ezzel el is érkeztünk az általam csak „**kis zöld fakocka**” illetve „**NAGY PIROS VASGOLYÓ**” problémájának nevezett esethez. Miről van itt szó egyáltalán? Adott **négy tulajdonság**, s mindegyik **ugyanazt** az objektumot jellemzi különböző szempontok szerint.

Egyfelől látunk egy méretet, ami egy szám (pl.: `int` vagy `float` típusú), másfelől egy színt, ami karakterláncként tárolható, aztán feltűnik a tárgy anyaga (ami lehet akár egy anyagbesorolási katalógusból való szám is), majd végül az alakot jellemző szó. Világos, hogy logikailag összetartozó információkról van szó (hiszen mind ugyanazon objektum, más-más szempontok szerinti leírását nyújtják számunkra), tehát valahogy „összefűzve" kéne őket tárolni, de az is nyilvánvaló, hogy erre a tömb típus alkalmatlan, mivel nem egyezik a tárolandó adatok típusa.

Nézzünk egy példát! Tegyük fel, hogy az informatikus szak bulit rendez! A szaknapok szervezői szeretnék nyilvántartani, hogy ki az aki eljön, hányad éves az illető, továbbá azt is, hogy mennyivel száll be a buli büdzséjébe.

Tekintve, hogy itt egy személyhez több adat is kapcsolódik, ám azok típusa nem egyezik – hiszen felbukkan *szöveg* éppúgy, mint *egész szám*, ahogy annak *lebegőpontos* változata is (már ha Euro–ban számolunk) – , logikailag összefogott módon való együttes tárolásuk túlmutat a tömb típus műfaji keretein.

Ilyen esetekben vesszük nagy hasznát a **struktúrák**nak, ugyanis a struktúrák olyan összetett típust testesítenek meg, melyek több (akár különböző típusú) változó összekapcsolása által jönnek létre. Nekünk most épp ilyesmire van szükségünk! *Hozzuk hát létre a buli struktúráját, avagy mondhatnám azt is, hogy csináljunk egy buli-t!*

A lényeg:

```
#include<stdio.h>

struct buli{
            char nev[10];
            int evf;
            float bef;
            };

main(){
struct buli hallgato;
printf("Ki␣jon␣el?␣"); scanf("%s", hallgato.nev );
printf("Hanyad␣eves?␣"); scanf("%d", &hallgato.evf );
printf("Mennyit␣ad␣bele?␣"); scanf("%f", &hallgato.bef );

printf("%s,␣%d.␣", hallgato.nev, hallgato.evf);
printf("evfolyamos␣hallgato␣eljon␣csinalni␣a␣fesztivalt");
printf(",␣%1.1f␣Eurobol.\n", hallgato.bef);
}
```

Látható, hogy a `main` előtt hoztuk létre a *buli struktúra* – új(!) – változó*típust*, ami egy struktúra. (Nem muszáj persze a struktúrákat feltétlenül a `main` előtt megkoreografálnunk, de azért tanácsos, hiszen így érhetjük el azt, hogy más függvények is „láthassák" és használhassák újsütetű típusunkat.) Új típusunk képes a partin résztvevő diákok egyes adatainak, személy szerinti csoportosításban való elraktározására. Az így létrehozott struktúrát később a programban, az egyes hallgatók vonatkozó adatainak tárolására használhatjuk. *Figyeljünk nagyon a létrehozás során arra, hogy a* `}` *után kitegyük a pontosvesszőt!*

A `main` -ben válik nyilvánvalóvá, hogyan lehet a struktúrát konkrét változók készítésére igénybe venni. A már `main` -en belül létrehozott (deklarált) `hallgato` nevű változó típusa `struct buli`, vagyis buli struktúra :).

Nem kell ettől az egésztől megijedni! Ha nem teljesen világos, hogy miről is van szó, akkor idézzük fel, hogyan deklaráltunk mondjuk egy egész típusú változót! Ott ugye az volt a mondás például, hogy `int i`. Ami ott `int` volt, az most itt `struct buli`, s ami ott `i` volt, az itt most `hallgato`. Magyarán, ami akkor `int i` volt, az most `struct buli hallgato`.

A különbség csak az, hogy míg azt `int` beépített típus, addig a struktúra nem az, tehát mielőtt egyáltalán deklarálhatnánk (lefoglalhatnánk a helyét a memóriában), ténylegesen meg is kell „alkotnunk" azt, aminek során mi magunk szab(hat)juk meg, hogy milyen is legyen.

Vegyük észre, hogy a program elején **nem** változót hoztunk létre (illetve **nem** hoztunk létre, nem **deklaráltunk** változót), hanem csak egy új típust, ami önmagában nem foglal még memóriát sem..... nem úgy a `hallgato`, ami már egy változó (, melynek típusa a `struct buli`)!

Ezt a `hallgato` nevű, buli struktúra típusú változót használjuk majd a továbbiakban. A **struktúra mezői**re (elemeire, a benne tárolt különböző (vagy épp nem különböző) típusú változókra) úgy **hivatkoz**hatunk, hogy leírjuk a szóbanforgó struktúratípushoz tartozó **változó nevét**, majd egy **pont** után az általunk hivatkozni kívánt **mezőét** is.

Ezt láthatjuk például a `scanf("%s", hallgato.nev );` -ben is, ahol a `nev` mezőre utalunk (megjegyzés: nem akarván bonyolítani a helyzetet, a programban alkalmazott karakterlánc és annak feltöltési módja csak egy egyszavas név megadását, illetve kiíratását támogatja, s a lefoglalt tárhely mérete is előre behatárolt).

Feltűnhet persze az is, hogy egy struktúramező memóriacímét is épp úgy `&` jellel adtuk meg, mint minden más változóét is eddig (kivéve a karakterláncét, melynek

neve már eleve egy pointer – felteszem, emlékszünk még rá, hogy miért. (Igen, mert a tömbök neve az első elemüket címző mutató.)) Igazából a struktúrák, mint olyanok, nem teljes értékű változói a C nyelvnek, ugyanis nem jelenhetnek meg például értékadásban (leszámítva egy-egy majd általunk is alkalmazott igen korlátozott esetet), ahogy bizonyos kifejezésekben sem. A fordító nemigen tud mit kezdeni a struktúrákkal. Nem tudna például összeadni sem két struktúrát, s erre még a legjobb akarattal sem tudnánk megtanítani.

Egyvalamit azért el *tud* végezni a struktúrákkal, s ez a művelet nem más, mint a *másolás*. (Ugyanis ha a létrehozott struktúra belső szerkezetét nem is, a méretét azért ismeri, ami viszont már elég a másoláshoz.) Első látásra ez ugyan sovány vigasz, viszont hamarosan látni fogjuk, hogy még ez esetben is számos lehetőség áll majd a rendelkezésünkre.

Haladjunk azonban „lépésben”! Egészítsük ki a bulis programot egy második hallgatóval, majd miután „feltöltöttük” az első mezőit, másoljunk át belőle mindent az új hallgató mezőibe!

```
#include<stdio.h>

struct buli{char nev[10]; int evf; float penz;};

main(){
struct buli hallgato1, hallgato2;

printf("Nev:␣"); scanf("%s", hallgato1.nev);
printf("Evfolyam:␣"); scanf("%d", &hallgato1.evf);
printf("Mennyire␣adja?␣"); scanf("%f", &hallgato1.penz);

hallgato2=hallgato1; /*masolas*/
/*Alabb a masolas tesztje:*/
printf("Nev:␣%s,␣", hallgato2.nev);
printf("evf.:␣%d,␣", hallgato2.evf);
printf("penz:␣%f\n", hallgato2.penz);
}
```

A fenti programban két változóval (`hallgato1`, `hallgato2`) dolgoztunk, melyek azonos típusúak (belső szerkezetűek, struktúrájúak (`struct buli`)), ezért minden további nélkül át lehetett másolni egyik értékét/értékeit, a másikba.

Mióta képesek vagyunk alkalmazni a „mutató követése” műveletet, nem vagyunk rászorulva arra, hogy értékadásra „pazaroljuk el” a függvények egyetlen visszatérési értékét, alább viszont mégis „elkövetem” az említett dolgot, mert nem árt

látni, hogy a struktúrák másolhatósága alkalmassá teszi őket arra (is), hogy a hívott függvény akár vissza is adjon egy struktúrát a `return` utasítással (ami a C nyelv érték szerinti paraméterátadása alapján szintén másolást jelent).

```
#include<stdio.h>

struct buli{char nev[10]; int evf; float penz;};

struct buli valtoztat();

main()
{
 struct buli hallgato;

 hallgato=valtoztat();
 /*Kiiratas (teszt):*/
 printf("Nev:␣%s,␣", hallgato.nev);
 printf("evf.:␣%d,␣", hallgato.evf);
 printf("penz:␣%f\n", hallgato.penz);
}

struct buli valtoztat()
{
 struct buli a;
 printf("Nev:␣"); scanf("%s", a.nev);
 printf("Evfolyam:␣"); scanf("%d", &a.evf);
 printf("Penz:␣"); scanf("%f", &a.penz);
 return (a);
}
```

Hagyjuk azonban most az „őskort”, s nézzük, hogyan lehetne megoldani például a másolást mutató követéssel!

```
#include<stdio.h>

struct buli{char nev[10]; int evf; float penz;};

void masolo(struct buli *innen, struct buli *ide)
{
 *ide=*innen;
 /*A mutato kovetes muvelete altali masolas.*/
}
```

```
main()
{
 struct buli hallgato1, hallgato2;

 printf("Nev: "); scanf("%s", hallgato1.nev);
 printf("Evfolyam: "); scanf("%d", &hallgato1.evf);
 printf("Mennyire adja? "); scanf("%f", &hallgato1.penz);

 masolo(&hallgato1, &hallgato2);

 /*Kiiratas (teszt):*/
 printf("Nev: %s, ", hallgato2.nev);
 printf("evf.: %d, ", hallgato2.evf);
 printf("penz: %f\n", hallgato2.penz);
}
```

Látszik, hogy a `masolo` függvény megkapja mindkét struktúra címét, majd – mintha csak két egyszerű változóval lenne dolga – simán „követi” a mutatókat és kész is a másolás. Van lehetőségünk persze arra is, hogy a hívott `masolo` függvénynek csak a struktúra bizonyos mezőit „mutassuk meg” a mutatóval, például így:

```
#include<stdio.h>

struct buli{char nev[10]; int evf; float penz;};

void masolo(int *innen, int *ide){
 *ide=*innen;
 /*A mutato kovetes muvelete altali masolas.*/
}

main(){
 struct buli hallgato1, hallgato2;

 printf("Nev: "); scanf("%s", hallgato1.nev);
 printf("Evfolyam: "); scanf("%d", &hallgato1.evf);
 printf("Mennyire adja? "); scanf("%f", &hallgato1.penz);

 masolo(&hallgato1.evf, &hallgato2.evf);
 /*Kiiratas (teszt):*/
 printf("Evf.: %d.\n", hallgato2.evf);
 /*A tobbi mezot most hatarozatlanul hagytuk.*/
 }
```

Felteszem, ezen a ponton kezd világossá válni, hogy az, hogy a struktúrákat csak másolni lehet, mindössze annyit jelent, „amennyit”. Ugyanis *attól, hogy magukat a struktúrákat csak másolni tudjuk, a mezőikkel/elemeikkel még megtehetjük mindazt, amit egy velük egyező típusú, egyszerű változóval is!* Alább, ugyan mindez még nem lesz több, mint egy egyszerű értékadás, viszont a későbbiekben egyre inkább bele fogunk mélyedni a mezőkkel való munkába.

Van azonban valami **nagyon fontos** dolog, amire valószínűleg nem gondolnánk, az mégpedig, hogy *amennyiben nem csupán egy-egy mezőjét, hanem magát az egész struktúrát jelöljük ki a memóriában egy mutató segítségével, akkor egy azt megkapó függvény – esetünkben a* `masolo` *– „belsejében” az így megadott struktúra mezőire már nem ponttal, hanem a - jelből és > jelből összeállított -> jellel kell hivatkoznunk!*

Lássuk hát a programot, mely ugyanazt teszi mint az előző, de úgy, hogy – mint arról fentebb írtam – az egész struktúrát kapja meg a mutató segítségével, s nem csak a másolandó mezőt!

```
#include<stdio.h>

struct buli{char nev[10]; int evf; float penz;};

void masolo(struct buli *innen, struct buli *ide)
{
 ide->evf=innen->evf;
}

main()
{
 struct buli hallgato1, hallgato2;

 printf("Nev:␣"); scanf("%s", hallgato1.nev);
 printf("Evfolyam:␣"); scanf("%d", &hallgato1.evf);
 printf("Mennyire␣adja?␣"); scanf("%f", &hallgato1.penz);

 masolo(&hallgato1, &hallgato2);
 /*Kiiratas (teszt):*/
 printf("Evf.:␣%d.\n", hallgato2.evf);
 /*A tobbi mezot most hatarozatlanul hagytuk.*/
}
```

Nagyon fontos továbbá az is, ami ezúttal *nem szerepelt* a programban (azaz, kimaradt belőle): a *mutató követése*. Ilyen esetben ugyanis, amikor az egész struktúrát

„mutatjuk” meg egy függvénynek a mutató követése művelet (a „csillagozás”) nélkül is átíródik a hívó (itt most a `main`) által átadott címmel jelölt terület mezője, amennyiben arra külön hivatkoztunk, a `->` jellel. Sőt, a csillagozást ilyenkor el sem fogadná a fordító!

Nem vizsgáltuk még meg azt az esetet – sok másikkal egyetemben – amikor kizárólag csak a karakterláncot képviselő mezőt szeretnénk másolni! Ilyenkor nyilván nem működne az `ide->nev=innen->nev;` módon való értékadás, hiszen ezek tömbök, s mint ilyenek, több elemet tartalmaznak, melyeket nem lehet „letudni” egyetlen értékadással. Egyetlen értékadással akkor boldogulnánk, ha a *karakterláncot tartalmazó* karaktertömböt, lecserélnénk egy *karakterláncot jelölő* mutatóra, melyet `char *nev` -re keresztelnénk. Ez esetben viszont már a forrásban meg kellene ejtenünk az értékadást, mégpedig így valahogy:
`hallgato1.nev="Gizike";` ami – lássuk be – nem valami szép, ráadásul módosítani sem tudjuk utólag, tehát a másolás lehetősége sem adatik meg általa. Ezúttal jobban tesszük, ha visszatérünk a karakterláncot tartalmazó tömbhöz, s azzal próbálunk meg zöld ágra vergődni. Nézzük hogyan! Rögtön eszünkbe juthat az `strcpy` függvény, csak arra figyeljünk, hogy az `#include<string.h>` -t is betöltsük! Csökkentendő a progi terjedelmét, ezúttal ne is töltsük fel a struktúra egyéb területeit! Az eredmény:

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

struct buli{char nev[10]; int evf; float penz;};

void masolo(struct buli *innen, struct buli *ide)
{
 strcpy(ide->nev, innen->nev);
}

main()
{
 struct buli hallgato1, hallgato2;
 printf("Nev: "); scanf("%s", hallgato1.nev);
 masolo(&hallgato1, &hallgato2);
 /*Kiiratas (teszt):*/
 printf("Nev: %s\n", hallgato2.nev);
}
```

Persze, ha nem bízunk mások függvényeiben, akár saját eljárást is írhatunk a másolásra, például így:

```
void masolo(struct buli *innen, struct buli *ide)
{
 int n=0;
 while(innen->nev[n]!=0){ide->nev[n]=innen->nev[n]; n++;}
 ide->nev[n]=0;
}
```

Az `ide->nev[n]=0;` utasítás a karakterlánc lezárásáról hivatott gondoskodni.

Feladat:
Írjunk programot, melyben egy függvény bekéri, s egy-egy `main` -beli struktúrában eltárolja a kétdimenziós tér két pontjának koordinátáit, majd – ezeket megkapva – egy másik függvény kiszámolja a pontok távolságát, amit el is tárol egy `main` -ben lévő lebegőpontos változóba, amit a `main` ki is írat ezután!
Lehetséges megoldás:

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

struct pont{ int x; int y;};

void beker(struct pont *a, struct pont *b);
void tav(struct pont u, struct pont v, float *t);

main(){
 struct pont p1, p2;
 float tavolsag;
 beker(&p1, &p2);  /*a strukturak mutatokkal atdobva*/
 tav(p1, p2, &tavolsag);  /*es mutatok nelkul atdobva*/
 printf("A␣pontok␣tavolsaga:␣%1.1f␣egyseg.\n", tavolsag);
}

void beker(struct pont *a, struct pont *b)
{
 printf("Kerem␣az␣egyik␣pont␣x␣es␣y␣koordinatajat!\n");
 scanf("%d%d", &a->x, &a->y);
 printf("Kerem␣a␣masik␣pont␣x␣es␣y␣koordinatajat!\n");
 scanf("%d%d", &b->x, &b->y);
}

void tav(struct pont u, struct pont v, float *t)
{
 *t=sqrtf(powf(u.x-v.x, 2)+powf(u.y-v.y, 2));
}
```

Nyilván megoldhattuk volna még sokféleképp a feladatot, ahogy az is, hogy mutatókat sem lett volna feltétlenül muszáj igénybe vennünk (vagy épp tele is hinthettük volna velük a kódot), de itt épp az volt a cél, hogy vázoljam mire kell figyelnünk ha azokkal struktúrákra mutogatunk, s mire, ha nélkülük küldjük át az említett adattípust.

Látható, hogy a `beker` függvényen belül, ahová mutatóval küldtük át a `struct pont` típusú változóinkat (`p1`-et és `p2`–t) végig a `->` jellel kellett hivatkoznunk a struktúrák mezőire (`scanf("%d%d", &a->x, &a->y);`), viszont a `tav` függvényben, ahová már mutató nélkül kerültek át (tehát az értékeik másolásával) elég volt a „normál” esetben használt pont. A továbbiakban rátérünk arra, mit tehetünk, ha szándékunkban állna új típusokat létrehozni.

9.2. Új típusok létrehozása

... illetve a már meglévőek átnevezése. Ez esetben a kulcsszó a `typedef`. Idézzük fel (lapozzunk vissza) a hallgatókat ”nyilvántartó” `struct buli` típussal dolgozó programot, s legott írjuk is át egy kissé!

```
#include<stdio.h>

struct buli{char nev[10]; int evf; float penz;};

main()
{
 typedef struct buli szakest; szakest hallgato;

 printf("Ki␣jon␣el?␣"); scanf("%s", hallgato.nev );
 printf("Hanyad␣eves?␣"); scanf("%d", &hallgato.evf );
 printf("Mennyit␣ad␣bele?␣"); scanf("%f", &hallgato.penz );

 printf("%s,␣%d.␣", hallgato.nev, hallgato.evf);
 printf("evfolyamos␣hallgato␣eljon␣csinalni␣a␣fesztivalt");
 printf(",␣%1.1f␣Eurobol.\n", hallgato.penz);
}
```

Fentebb, a `main` első sorában látható a lényeg. Arról van szó csupán, hogy ha megunnánk azt, hogy minden egyes alkalommal, ahányszor csak `struct buli` típusú változót deklarálunk tetszőleges névvel, (például `hallgato`, `h0` vagy `h1`, stb) be kell írnunk a `struct`-ot és a `buli`-t is, megtehetjük azt is, hogy a `struct buli` típusnak valami más, könnyebben kezelhető nevet adunk.

Fenti esetben a `szakest`-re esett a választásunk.

Természetesen az adott struktúra megalkotása és a `typedef` alkalmazása össze is vonható egy közös szerkezetbe, valahogy így:

```
typedef struct buli{
                char nev[10];
                int evf;
                float penz;}szakest;

main(){ szakest hallgato; /*a tobbi ugyanaz*/}
```

Az sem okoz gondot, ha a struktúra és a típus neve megegyezik! (Sőt, *ez a bevett gyakorlat*!) Ez esetben így fogalmazhatunk[1]

```
typedef struct buli{
                char nev[10];
                int evf;
                float penz;}buli;

main(){ buli hallgato; /*a tobbi ugyanaz*/}
```

Hisszük vagy sem, az eddig tárgyalt változótípusok messze nem a legösszetettebb elemei a C nyelv arzenáljának. A továbbiakban szót ejtek néhány olyan típusról, melyek összefoglaló besorolásának neve:

9.3. Összetett típusokból készült típusok

Jó ha tudjuk, hogy az alfejezetcím által jelölt összetett típusokból is készíthetünk további összetett típusokat és a sort folytathatjuk ameddig csak kedvünk tartja, ezért e helyütt – nem bocsátkozván bele túl mélyen a kapcsolódó részletek ágas-bogas erdejébe – csak az összetett típusok „iskolapéldáit” fogom bemutatni, azokon belül is csak a számunkra leghasznosabbakat, melyek már önmagukban is szerfelett érdekes, **gyakorlatban is jól használható adatszerkezetek**. (Számos fontos és hasznos típus tárgyalásáról viszont – az idő rövidsége miatt – kénytelenek vagyunk most lemondani. Sajnos.) Nosza...

[1] Viccesebb kedvű egyének akár a C nyelv beépített változóit is átkeresztelhetik a `typedef` segítségével, például akár így is: `typedef float tizedes;` . Innentől fogva, ha pl egy `a` nevű, lebegőpontos változót deklarál az illető, akkor azt így teheti meg: `tizedes a;`

9.3.1. Struktúratömbök

Elérkeztünk végre ahhoz a típushoz, melynek alkalmazása esetén *praktikusabb adatszerkezethez jutunk, mint eddig bármikor*.

Emlékezzünk csak a struktúrákra, ahol az volt a „mondás”, hogy az általunk megformálható belső szerkezetből fakadóan, végre képesek vagyunk az egymással összetartozó, különféle adattípusokat megtestesítő változókat egyetlen egybe „belegyüszmékelve” csokorba fogni, s ily módon egy helyen tárolni.

Felmerülhet persze a kérdés: mi a teendő akkor, ha több ilyen összetett változóval szeretnénk dolgozni? Például ha egy ügyfeleket nyilvántartó programot kívánnánk megalkotni vagy épp csak bulizni támadna kedvünk. :)

Nos, a válasz az, hogy ilyen esetben igen jól járunk ha struktúratömbök igénybevételével oldjuk meg a feladatot. Az egésznek az a lényege, hogy miután a struktúrák lehetővé tették számunkra, hogy a gondolkodásmódunk szerint összetartozó, az egyes objektumokat különböző szempontok szerint jellemző adatokat összefűzzük egy struktúrába, azután már a *többi* – hasonló objektumot leíró – *struktúrát tömbbe* foglalhatjuk, hiszen ezek a változók már egyazon típushoz tartoznak.

Ily módon már sok-sok hasonló objektum leírását képesek vagyunk egyetlen változóban, egy tömbben eltárolni, melynek minden eleme egy olyan struktúra, amely egy-egy objektumot jellemez a sok közül. Megkötés persze, hogy a tömb elemeit csak azonos „belsejű” struktúrák képezhetik.

Maga a struktúratömb egy adatbázisban lévő táblához hasonlítható, melyet sorok és oszlopok alkotnak.

A **sorok** a **tömb elemei** (ezek az egyes struktúrák), az **oszlopok** pedig az egyes tömbelemek **mezői**.

Nézzünk néhány példát!

Feladat: Írjunk programot, melyben egy függvény futásidőben a billentyűzetről feltölt egy banki ügyfeleket nyilvántartó struktúratömböt (ezúttal beérjük egy 3 elemű tömbbel), úgy mégpedig, hogy minden eleméhez bekéri annak három mezőjét (betétes neve, betett összeg, kamatláb)!

Lehetséges megoldás:

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

```
typedef struct ugyfel
{char nev[10]; float penz; float klab;} ugyfel;

void feltolto(ugyfel *u, int meret);

main()
{
  int n=3; ugyfel ugyfelek[n];

  feltolto(ugyfelek, n);
}

void feltolto(ugyfel *u, int meret)
{
 int i;

 for(i=0;i<meret;i++)
 {
  printf("%d. nev: ", i+1); scanf("%s", u[i].nev);
  printf("Betett penz: "); scanf("%f", &u[i].penz);
  printf("Eves kamatlab: "); scanf("%f", &u[i].klab);
 }
}
```

A fenti kis programban a struktúratömb létrehozásának módja látható, ami pontosan megegyezik a tömbök alkotásának már ismert technikájával, jelesül; megadjuk, hogy milyen típusú elemek alkotják a tömböt, aztán a tömb nevét, s végül annak méretét.

Az `ugyfel ugyfelek[n];` -t tartalmazó sorban is épp így jártunk el. Fontos megjegyeznünk továbbá, hogy *noha mutatóval – a tömb nevével – küldtük át a struktúratömböt (s benne az egyes struktúrákat), mégsem kellett a pontokat nyilakra cserélnünk a mezőkre való hivatkozásnál, mivel itt nem egy egyedi struktúrát, hanem egy tömböt kezeltünk* (ami történetesen struktúrákból állt).

A feltöltés módja pedig hasonló volt az egyszerű változókból álló tömböknél látotthoz, ahol szintén `&` jelet kellett használnunk a `scanf` függvény alkalmazásakor (kivéve a nevet tároló karakterláncnál, de ennek okát már megbeszéltük fentebb).

Egészítsük ki a fenti programot egy függvénnyel, mely minden betét értékét megnöveli az általunk futásidőben beírt értékkel!

Lehetséges megoldás:

```
#include<stdio.h>

typedef struct ugyfel
{char nev[10]; float penz; float klab;}ugyfel;

void feltolto(ugyfel *u, int meret);
void novelo(ugyfel *v, int hossz);

main()
{
  int n=3; ugyfel ugyfelek[n];
  feltolto(ugyfelek,n);
  novelo(ugyfelek,n);
}

void feltolto(ugyfel *u, int meret)
{
 int i;
 for(i=0;i<meret;i++)
 {
  printf("%d. nev: ", i+1); scanf("%s", u[i].nev);
  printf("Betett penz: "); scanf("%f", &u[i].penz);
  printf("Eves kamatlab: "); scanf("%f", &u[i].klab);
 }
}

void novelo(ugyfel *v, int hossz)
{
  int i; float p;
  for(i=0;i<hossz;i++)
  {
   printf("%d. betet mennyivel nojon? ", i+1);
   scanf("%f", &p);
   v[i].penz+=p;
   }
}
```

A függvényt alaposabban átböngészve hamar észrevehetjük a legújabb **fontos** fejleményt, miszerint, jóllehet átírtuk a tömböt alkotó struktúrák mezőit, mégsem volt szükség a mutató követését jelző csillagozásra[2].

[2]Persze, ez a `feltolto`-ben is szemet szúrhatott volna már.

Ez épp az helyzet, mint amivel akkor szembesültünk, amikor annak idején egy „mezei” változót egy hívott függvényben írtunk át, ahol kellett a mutató követését jelző csillag, majd azt tapasztaltuk, hogy ha egy ugyanolyan változókból álló tömb elemeit írjuk felül a hívott függvényben, akkor nincs szükség a csillagozásra. . . ..

Na igen...

Azt hiszem most jött el az ideje, hogy összefoglaljuk azt, amit a mutató követéséről az eddigiekben, itt-ott elhintve már megpendítettünk.

Tehát: az alapfelállás mindig a következő: a hívott függvény mutatóval megkapja az átírandó paraméter címét.

- Egyszerű változó esetén kell a „csillagozás”!
  Példa:

```
void fgv(float *a){*a+=1;}
```

  vagy akár:

```
void fgv(float *a, float *b){*a+=*b;}
```

- Egyszerű változókból álló tömb elemének átírásakor nem kell a $\star$!
  Példa:

```
void fgv(float *tomb){tomb[2]+=1;}
```

- Egész struktúra – mint „egyszerű” változó – átírása (másolása) esetén, ha mutatóval küldjük át: kell csillag!
  Példa:

```
void fgv(struct valami *a, struct valami *b ){*a=*b;}
```

- Ha csak egy mezőt küldünk és írunk át, akkor ugyanúgy kell eljárni, mint egy egyszerű változó esetében: kell a csillag! A függvény is ugyanúgy néz ki, csak a paraméterátadásnál látszik, hogy egy struktúrának a mezőjét küldjük át, s nem csak egy egyszerű változót.
  Példa:

```
typedef struct bank{... float betett;...}bank;
void fgv(float *a){*a=10;}
main(){bank otp;... fgv(&otp.betett); ... }
```

- Ha mutatóval küldjük át az egész struktúrát és a hívott függvényben hivatkozunk a mezőjére, amit át is írunk, nem kell a csillag, viszont a pontokat nyilakra kell cserélnünk a mezők neve előtt!
  Példa:

```
typedef struct bank{... float betett;...}bank;
void fgv(bank *a){a->betett=10;}
main(){bank otp;...fgv(&otp);...}
```

- **A számunkra legfontosabb – majdan leginkább alkalmazott – pont:** Struktúratömb átküldése esetén sem a csillag, sem a nyilak nem szükségesek. Ilyet legutóbb épp az ügyfeleket kezelő példaprogramban láthattunk.

- Amennyiben valaki további ínyencségekre áhítozik – olyasmire például, hogy egy struktúratömb egyik elemének egyik mezőjét küldjük csak át mutatóval, majd azt módosítjuk, stb, stb – az már eleget tud ahhoz, hogy némi gondolkodás és esetleg egy-két próba után maga is megtalálja a megoldást.

Feladat gyanánt, gyakorlásképpen vegyük elő a bankos programot, melynek tennivalója eleinte ugyanaz marad, vagyis: legyen benne egy függvény, mely futásidőben a billentyűzetről feltölt egy banki ügyfeleket nyilvántartó struktúratömböt (ezúttal is beérjük egy 3 elemű tömbbel), melynek az eddig használt három mezőjén kívül (betétes neve, a betett összeg, és az éves kamatláb), legyen most egy újabb mezője is, amit elsődleges kulcsnak[3] fogunk hívni! Azért van szükség erre a mezőre, mert egy adatbázisban egy személy neve sosem lehet ”perdöntő” azonosító, hiszen több, azonos nevű ember is szerepelhet egy nyilvántartásban, így az elsődleges kulcsra hárul az ”azonosító” szerepe[4]. Természetesen ez nem lesz "igazi" elsődleges kulcs, mert nem pont úgy viselkedik (látnánk is, ha például törölnénk a listából valakit), arra viszont jó, hogy rögzüljön bennünk: ez is fontos szempont egy nyilvántartásban[5].

Az újdonság, ezúttal az, hogy meg kell alkotnunk egy újabb függvényt, mely megkérdezi, hogy kit keresünk, s miután beírtuk a nevet, megkeresi és kilistázza a találato(ka)t. Ezek után megkérdi, melyikkel lesz dolga, majd azt is, hogy mit szeretnénk választani az általa kínált két lehetőségből, melyek a következők:
- kiírja az illető jelenlegi egyenlegét és az összes adatát (ezt egy hívott függvénnyel hajtatja végre)

[3] A programban `pk` jelöli majd ezt a mezőt, az ”elsődleges kulcs”, adatbázisokban használatos ”primary key” elnevezésének megfelelően.

[4] Képzeljük csak el, mekkora kalamajka adódna, ha például név alapján akarnánk valakit törölni egy adatbázisból, ahol akár több, azonos nevű személy adatai is dekkolhatnak. Ha ilyenkor, mondjuk `SQL`-ben valami olyasmit tennénk, hogy
`DELETE FROM ugyfelek WHERE nev='Törlendő Név';`, valószínűleg nézhetnénk új állás után, s egy két - törölt számlájú ügyfél által nyakunkba akasztott - per elé .... Persze, hogy **elsődleges kulcs** alapján szabad csak ilyesmit elkövetni.

[5] Minderről jóval több szót ejtek majd a "Webprogramozás" nevű tárgyban, illetve nyilván az "Adatbázisok"-ban is megemlékszik majd róla a tárgyat oktató kolléga.

- (szintén egy hívott függvény által) kiírja, hogy mennyi lesz az illető egyenlege az általunk beírt évben.

Befejezésül pedig, esetleg megtoldhatjuk programunkat egy eljárással, mely kilistázza az összes ügyfelet. Essünk hát neki!

Lehetséges megoldás:

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

typedef struct ugyfel
{
 int pk; char nev[10];
 float penz; float klab;
}ugyfel;


void feltolt(ugyfel *u, int elemszam);
void keres(ugyfel *u, int elemszam);
void kiir(ugyfel *u, int a);
void kamatozo(ugyfel *u, int a);
void lista(ugyfel *u, int elemszam);


main()
{
  int n=3, ujra, listazz; ugyfel ugyfelek[n];

  feltolt(ugyfelek, n);

  do{
    keres(ugyfelek, n);
    printf("Meg␣egy␣kereses?␣(i/n)␣");
    while((ujra=getchar())=='\n');
    }while(ujra=='i');

  printf("Kivancsi␣a␣teljes␣listara?␣(i/n)␣");
  while((listazz=getchar())=='\n');
  if(listazz=='i'){ lista(ugyfelek, n); }
}
```

```
void feltolt(ugyfel *u, int elemszam)
{
 int i;
 for(i=0;i<elemszam;i++)
 {
  system("clear");
  u[i].pk=i+1; printf("%d. nev: ", i+1);
  scanf("%s", u[i].nev);
  printf("Penz: "); scanf("%f", &u[i].penz);
  printf("Kamatlab: "); scanf("%f", &u[i].klab);
 }
}



void keres(ugyfel *u, int elemszam)
{
 char kit[10];  int i, j, kithossz=0, nevhossz, van_e=0;
 int azonosito, opcio; system("clear");
 printf("Kit keresunk? "); scanf("%s", kit);

 while(kit[kithossz]!=0){kithossz++;}

 for(i=0;i<elemszam;i++)
 {
   nevhossz=0;
   while(u[i].nev[nevhossz]!=0){nevhossz++;}

   if(kithossz==nevhossz)
   {
     j=0;
     while(j<nevhossz)
     {
      if(u[i].nev[j]!=kit[j]){break;} j++;
     }
     if(j==nevhossz)
     {
     van_e++;
     printf("Azonosito: %d, nev: %s\n", u[i].pk, u[i].nev);
     }
   }
  }/*A fuggveny alabb folytatodik!*/
```

```
 if(van_e==0)
 { printf("Nincs␣ilyen␣nev␣a␣nyilvantartasban.\n"); }
 else
 {
  printf("Irja␣be␣az␣ont␣erdeklo␣azonosito␣erteket:␣");
  scanf("%d", &azonosito);
  printf("Opciok:\n−␣1:␣szamlainformacio\n");
  printf("−␣2:␣varhato␣egyenleg\n"); scanf("%d", &opcio);
 }

 switch(opcio)
 {
  case 1: kiir(u, azonosito); break;
  case 2: kamatozo(u, azonosito); break;
 }
}



void kiir(ugyfel *u, int a)
{
 int i=0;
 while(u[i].pk!=a){++i;}
 printf("Azonosito␣szam:␣%d\n", u[i].pk);
 printf("Nev:␣%s\n", u[i].nev);
 printf("Egyenleg:␣%f\n", u[i].penz);
 printf("Eves␣kamat:␣%f␣%%\n", u[i].klab);
}



void kamatozo(ugyfel *u, int a)
{
 int i=0, ev, aktev; float egyenleg;
 while(u[i].pk!=a){++i;}
 printf("Aktualis␣ev?␣"); scanf("%d", &aktev);
 printf("Melyik␣ezutani␣ev␣egyenlege␣erdekli?␣");
 scanf("%d", &ev);
 egyenleg=u[i].penz*powf(1+u[i].klab/100,ev−aktev);
 printf("%1.1f␣lesz␣az␣egyenlege.\n", egyenleg);
}
```

```
void lista(ugyfel *u, int elemszam)
{
 int i; system("clear");
 for(i=0;i<elemszam;i++)
 {
  printf("azonosito: %d\t", u[i].pk);
  printf("nev: %s\t", u[i].nev);
  printf("egyenleg: %1.2f\t", u[i].penz);
  printf("kamatlab: %1.2f %%\n\n", u[i].klab);
 }
}
```

Láthatjuk fentebb, s elmondjuk újra, hogy a `scanf("%s", u[i].nev);` és a `scanf("%s", kit);` utasítások esetében, amikor a `scanf`-nek egy karakterláncot kell beolvasnia egy karaktertömbbe, akkor elboldogul a címoperátor nélkül is, mivel a tömb neve már eleve egy cím, az első elem címe, így kifejezetten szükségtelen is az `&` jel betoldása[6].

Tekintve, hogy használjuk a `powf` függvényt, tartsuk észben, hogy nem csupán a programban kell benne lennie az `#include<math.h>`-nak, hanem a fordításra kiadott parancsban is gondoskodnunk kell a szóbanforgó könyvtárról az `-lm` beszúrásával: `gcc prgnev -lm -o ujprgnev`.

A keresést végző függvényünket persze némileg áttekinthetőbbé tehetjük, ha a már ismert karakterlánckezelő függvényünknek is behívót küldünk. Ilyenkor természetesen be kell gyüszmékelnünk az `#include<string.h>` -t is az előfeldolgozónak szóló részbe. Ha rászánjuk magunkat minderre, valami ilyesmit kapunk:

```
void keres(ugyfel *u, int elemszam)
{
 char kit[10];  int i, j, kithossz=0, van_e=0;
 int azonosito, opcio; system("clear");
 printf("Kit keresunk? "); scanf("%s", kit);

 kithossz=strlen(kit);

 for(i=0;i<elemszam;i++)
 {
```

[6]Ettől függetlenül egyes fordítók igen megengedőek, s nem tapasztalunk zavart, ha mégis kitesszük az `&` jelet ilyenkor is, amiért az ilyen fordító alkotóját szívem szerint igen nagy összegű bírsággal sújtanám, mivel az efféle lazaság, ha másra nem is, de a logikusan gondolkodó és tanuló emberek összezavarására mindenképpen kiválóan alkalmas.

```
    if(kithossz==strlen(u[i].nev))
    {
      j=0;
      while(j<kithossz)
      {
       if(u[i].nev[j]!=kit[j]){break;} j++;
      }
      if(j==kithossz)
      {
      van_e++;
      printf("Azonosito: %d, nev: %s\n", u[i].pk, u[i].nev);
      }
    }
  }/*A fuggveny alabb folytatodik!*/

  if(van_e==0)
  { printf("Nincs ilyen nev a nyilvantartasban.\n"); }
  else
  {
   printf("Irja be az ont erdeklo azonosito erteket: ");
   scanf("%d", &azonosito);
   printf("Opciok:\n- 1: szamlainformacio\n");
   printf("- 2: varhato egyenleg\n"); scanf("%d", &opcio);
  }

  switch(opcio)
  {
   case 1: kiir(u, azonosito); break;
   case 2: kamatozo(u, azonosito); break;
  }
}
```

Mi persze nem érjük be ennyivel, s bevetünk egy eddig még sosem hívott függvényt, az `strcmp` -t, amiről most elég annyit tudnunk, hogy a visszatérési értéke 0, ha a bemenő paraméterekként megadott két karakterlánc megegyezik. Nézzük csak, mennyit karcsúsodik az eljárásunk!

```
void keres(ugyfel *u, int elemszam)
{
 char kit[10];  int i, van_e=0, azonosito, opcio;
 system("clear");
 printf("Kit keresunk? "); scanf("%s", kit);

 for(i=0;i<elemszam;i++)
```

```
 {
  if(strcmp(kit, u[i].nev)==0)
  {
   van_e++;
   printf("Azonosito: %d, nev: %s\n", u[i].pk, u[i].nev);
  }
 }

 if(van_e==0)
 { printf("Nincs ilyen nev a nyilvantartasban.\n"); }
 else
 {
  printf("Irja be az ont erdeklo azonosito erteket: ");
  scanf("%d", &azonosito);
  printf("Opciok:\n- 1: szamlainformacio\n");
  printf("- 2: varhato egyenleg\n"); scanf("%d", &opcio);
 }

 switch(opcio)
 {
  case 1: kiir(u, azonosito); break;
  case 2: kamatozo(u, azonosito); break;
 }
}
```

Láthatóan sokmindent sikerült összerántani a `keres` kódjában, viszont tartsuk szem előtt, hogy ez ”csak” számunkra jelent könnyítést, ugyanis a kód valóban áttekinthetőbb, ám a program mostantól egyfelől maga után kényszerül vonszolni az egész `string.h` ”csóváját”, másfelől a `strcmp` függvény, ahányszor csak hívjuk, megállapítja mind a listában szereplő, éppen vizsgált, mind pedig a keresendő név hosszát, jóllehet, a keresendő névét elég lenne egyszer lemérni, hiszen az nem változik a lista bejárása során. Magyarán: gyorsabban futna a programunk, ha ezt elkerülnénk (ahogy tettük azt egyébként a `strlen` -t használó kód esetén).

Lehet, hogy rosszul ítélem meg a dolgot, de ha csupán e két függvény használatáról van szó (sőt, igazából vagy csakaz egyiket vagy csak a másikat használjuk), úgy is átláthatóbbá tehető a kód, ha az ember megír két függvényt, melyek ugyanazt elvégzik, mint a fent hívott karakterlánc kezelő függvények, s ugyanott, ugyanúgy hívja őket.

Természetesen az ilyen – például tömböt is tartalmazó – programjaink csak akkor lesznek életszerűek, ha képesek takarékosan bánni a rendelkezésre álló memóriával, úgy mégpedig, hogy – szemben az eddigi gyakorlattal – csak annyit foglalnak

le belőle, amennyire valóban szükségünk lesz. De persze erről és még sok egyéb dologról is szól lesz majd a következő fejezetekben[7].

E fejezet további részében megismerhetünk néhány adatszerkezetet, *melyeknek később még nagy hasznát vesszük*. Önhivatkozó adatszerkezeteket ugyan még itt sem fogunk alkotni (ez majd a szemeszter végén várható), de azért, néhány könnyebben emészthető dologgal így is bővíthetjük a repertoárunkat. Ilyen például a

9.3.2. mutatót jelölő mutató

A cím által fémjelzett változótípushoz azok a mutatók tartoznak, amelyek mutatókat jelölnek a memóriában. Gondolhatunk erre a típusra úgy is, mint mutatóra mutató mutatóra. Másként: ez a mutató mutató mutató. :)

Megvilágítandó eme al-alfejezet lényegét vessünk egy – jó hosszan kitartott – pillantást az alábbi, első körben még nem egészen a szándékunk szerint üzemelő programocskára!

```
#include<stdio.h>

void  szovegre(int e, char *szoveg);

main()
{
 int egesz;
 char *string="semmi";
 printf("Kerek␣egy␣egsz␣szamot!␣");
 scanf("%d", &egesz);
 szovegre(egesz, string);
 printf("%s\n", string);
}

void  szovegre(int e, char *szoveg)
{
 switch(e)
 {
  case 1: szoveg="nem␣semmi"; break;
```

[7] Például a már említett, dinamikus tömbökön kívül arról is, hogy pötyögés helyett fájlból olvastassanak be, s írassanak ki az adatok, de persze úgy, hogy már a program indításakor megadjuk – a `main` parancssori paraméterezése által – a felhasználandó fájlokat, továbbá a `main` -ben hívott függvények kezeljék/jelezzék az esetleges hibákat is, stb.

```
    case 2: szoveg="valami";   break;
    default: szoveg="akarmi...";
  }
}
```

Fenti próbálkozásunkon keresztül a következőt kíséreltük meg véghezvinni: A `main` -ben létrehoztunk egy karakterlánc típusú állandót (a `"semmi"` -t) – ami ugye egy memóriacímet „hagy maga után” – , melyet érték gyanánt megadtunk a string nevű mutatónak. Ezt a mutatót igyekeztünk aztán a `szovegre` nevű függvény által módosítani oly módon, hogy a beadott számtól függően az a `"nem semmi"` -re, a `"valami"` re vagy az `"akarmi..."` -re mutasson. Rá kellett azonban döbbennünk, hogy nem jártunk sikerrel. Nem történt `"semmi"`. :) Természetesen felmerül a kérdés: Miért? A válasz megértéséhez fel kell idéznünk a mutatókkal való ismerkedésünk korai fázisát! Ott, akkor azt írtam, hogy a C nyelvet jellemző érték szerinti paraméterátadás miatt a hívott függvény nem tudja közvetlenül elérni a hívó függvény helyi változóit, ennélfogva csak azok átmásolt értékeivel tud dolgozni, ami nem mindig elegendő.

Azt is írtam, hogy a mutatók alkalmazása által tehetjük a hívó helyi változóit közvetlenül elérhetővé a hívott függvény számára, hiszen ebben az esetben a változók címeit adjuk át neki, s ezáltal tesszük képessé arra, hogy magukkal a változókkal dolgozzon, s ne csak azok értékeivel (ha jól emlékszem a vázolt példával kapcsolatban jegyeztem meg, hogy az nem más, mint a mutató követése).

Itt most a változó, melynek szerettük volna megváltoztatni az értékét, már *eleve mutató* volt! (Tehát nem az általa kijelölt memóriacímen tárolt adatokon akartunk módosítani valamit, hanem egy új címet – új értéket – akartunk adni a mutatónak, hogy mostantól az ott található területet jelölje.) Nyilván ahhoz, hogy a hívott függvénynek közvetlen hozzáférést biztosítsunk a módosítani kívánt változónkhoz (a `string` nevű mutatóhoz) meg kell adnunk annak memóriacímét.

Márpedig ezt elmulasztottuk megtenni!

Ezért aztán nem is csoda, hogy a program nem tette meg azt, aminek végrehajtására létrehoztuk. (Egészen pontosan: a saját, lokális, `szoveg` nevű mutatóját „ráforgatta” az adott címre, de ez nyilván a `string` -et nem érint(h)ette.) Alább, már figyelünk erre is, s megadjuk az átalakítani kívánt változó memóriacímét is! Tekintve egyébként, hogy ez a memóriacím egy mutató, továbbá szem előtt tartva, hogy az, aminek a címét kijelöli szintén mutató, világos, hogy ez a cím egy mutatót jelölő mutató értéke.

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

```
void  szovegre(int e, char **szoveg);


main()
{
 int egesz;
 char *string="semmi";
 printf("Kerek egy egsz szamot! ");
 scanf("%d", &egesz);
 szovegre(egesz, &string);
 printf("%s\n", string);
}

void  szovegre(int e, char **szoveg)
{
 switch(e)
 {
  case 1: *szoveg="nem semmi"; break;
  case 2: *szoveg="valami";  break;
  default: *szoveg="akarmi...";
 }
}
```

Fenti programunkban, a `switch` mindhárom esetében, újfent szép példáját láthatjuk a *mutató követésének*. **Jegyezzük meg jól** a mutatót jelölő mutatóval való bánásmódot, mert a következő fejezetekben folyton folyvást használjuk majd!

9.3.3. Mutatótömbök

A jelen alfejezet tárgyát képező ismeretek szintén **alapvető fontossággal bírnak** majd, amikor a `main` paramétereit kell használnunk, fordítsunk hát rá illő figyelmet!

Emlékszünk még arra a feladatra, ahol az „*en ezt mar mondtam talan egyszer neked*” karakterláncban látható szavak tetszőleges sorrendbe való rendezése által kellett bemutatnunk azt, hogy az ily módon alkotott „szósorozatok” újra és újra változatlan jelentésű mondattá állnak össze?

Akkoriban megfogalmaztunk egy programot, mely a felhasználó által megadott

sorrendben írta ki a karakterláncba foglalt szavakat, ezáltal is szemléltetve a feladat alapvetésében kifejtett állítás igazságát. Feltehetőleg emlékszünk arra is, milyen „macerás meló” volt karakterláncból kiindulva kezelni a problémát.

Alább kifejtem, hogyan lehetett volna kissé áttekinthetőbb kóddal dolgozni, példának okáért *mutatótömb* segítségével, mely *típus* – ahogy az nevéből is sejthető – azokat a *tömbök*et jelöli, amelyeknek az *elemei mutatók*. Nézzük, hogyan járhattunk volna el, ha ismerjük a tárgyalt típust!

```
#include<stdio.h>

main()
{
 int i, n=7, sorrend[n];
 char *szavak[n];

 szavak[0]="en";
 szavak[1]="ezt";
 szavak[2]="mar";
 szavak[3]="mondtam";
 szavak[4]="talan";
 szavak[5]="egyszer";
 szavak[6]="neked";

/*vagy:
szavak[]={"en","ezt","mar","mondtam","talan",
"egyszer","neked"};
*/

 printf("Szavak␣kiirasanak␣sorrendje␣(pl.:3452671):\n");
 printf("(Minden␣szam␣utan␣nyomjunk␣ENTER-t!)\n");

 for(i=0;i<n;i++){ scanf("%d", &sorrend[i]); }

 for(i=0;i<n;i++){ printf("%s␣", szavak[sorrend[i]-1]); }

 printf("\n");
}
```

Miről is van szó odafent?

Létrehoztunk egy tömböt, melynek a `szavak` nevet adtuk, ennek során nyilván közöltük a méretét, továbbá azt, hogy az elemei karakter típusú adatok memóriacímét jelölő mutatók. Ezt mind, mind – úgymond – egy szuszra elintéztük a `char`

`*szavak[n];` utasítás begépelése által.

Ezután már csak értéket adtunk a tömb elemeinek, mely értékek – mutatókról lévén szó – memóriacímek, mégpedig a beírt karakterlánc típusú állandók memóriában elfoglalt helyét kijelölő adatok.

Feladat:
Gyakorlásképpen írjunk a fenti programhoz egy függvényt, mely a kiíratás során némileg finomít az eredményen, úgy mégpedig, hogy a mondat végét ponttal zárja le, továbbá ügyel arra is, hogy az első szó első betűjét „naggyá tegye”!

Lehetséges megoldás:

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

void finomito(char **sz, int *s, int elemszam);

main()
{
int i, n=7, sorrend[n];
char *szavak[]={"en","ezt","mar","mondtam","talan",
"egyszer","neked"};

printf("Szavak␣kiirasanak␣sorrendje␣(pl.:3452671)\n");
printf("(Minden␣szam␣utan␣nyomjunk␣ENTER-t!)\n");

for(i=0;i<n;i++){ scanf("%d", &sorrend[i]); }

finomito(szavak,sorrend,sizeof(szavak)/sizeof(szavak[0]));
}


void finomito(char **sz, int *s, int elemszam)
{
 int i; char betuk[10];

 for(i=0;i<elemszam;i++)
 {
  if (i==0)
  {
   strcpy(betuk, sz[s[i]-1]);
   betuk[0]-='a'-'A'; printf("%s␣", betuk);
  }else if (i<elemszam-1) printf("%s␣", sz[s[i]-1]);
```

```
    else printf("%s.\n", sz[s[i]-1]);
  }
}
```

Felteszem világos! :) Na jó, a lényeg, hogy nem szabad elveszni a részletekben!

Először is azt kell végiggondolnunk, hogy miként kell megoldani a felvetődő problémát! Abban maradtunk, hogy az első szó, első betűjének nagynak kell lennie. Ehhez – az adott feltétel teljesülése esetén – módosítani kellene egy karakterlánc típusú állandót, oly módon, hogy az első karakterének ASCII-kód értékét eltoljuk a megfelelő „nagybetűs" helyre (lásd: a karakterláncos fejezet „nagybetűsre alakító" feladatát). Igen ám, de a karakterlánc típusú állandók olyan helyen tárolódnak a memóriában, amit nincs jogunk módosítani, ezért előbb átmásoljuk annak tartalmát egy karaktereket tartalmazó tömbbe, amit már kedvünkre „apríthatunk". Ezt a feladatot végzi el a `finomito` függvény

```
if (i==0)
{strcpy(betuk, sz[s[i]-1]);
betuk[0]-='a'-'A'; printf("%s ", betuk);}
```

része. (Arról már talán ejtettem szót, hogy az `strcpy` miatt kellett betoldanunk az `#include<string.h>` -t az előfeldolgozóba.) A pontot nyilván az utolsó elemhez érve tesszük ki, aminek megtörténtét szintén egy feltétellel teszteljük. Fontos látnunk, hogy a `finomito` függvény, a hívásakor három paramétert kap, melyek közül az első, egy *mutatót jelölő mutató*, mely a `szavak` mutatótömb neve. Ilyen minőségében pedig nem más, mint a tömb első elemének memóriacímét kijelölő mutató, s mivel e tömb első eleme maga is egy mutató (melynek értéke egy karakterlánc típusú állandó memóriacíme), a tömb neve nem más, mint egy mutatót kijelölő mutató. *Ezért* használunk két $\star$ jelet. A második paraméter – lévén tömbnév – már csak egy „mezítlábas" mutató, egy tömbnév, míg a harmadik paraméterben a szavak mutatótömb elemeinek a számát adjuk meg egy kifejezésen keresztül.

További önálló feladat gyanánt, érdemes lehet elgondolkodni rajta, hogyan lehetne megoldani azt, hogy a szavak kiírási sorrendjének megadásakor ne kelljen minden szám után ENTER-t nyomkodni, hanem csak egyszer, mégpedig a számsor végén!

Megjegyzés: mint már arra utaltam, természetesen jó okkal hallottunk a kétszeres indirektségű mutatókról (a mutatót jelölő pointerekről), s a mutatótömbökről e helyütt. Egyfelől, nemsokára hasznát vesszük majd függvényeinkben a kétszeres indirektségű mutatóknak, másfelől, a `main()` függvény, szintén hamarosan tárgyalandó paraméterei közül az egyik maga is egy olyan mutató, mely a `main()`-

nek átadott paramétereket jelölő mutatókat tartalmazó tömb első elemére irányul.

10. fejezet

Dinamikus tömbök

A előző fejezetben – a struktúratömbök (más néven: listák) által – olyan adatszerkezettel bővült az eszközkészletünk, melynek alkalmazásán keresztül igen közel jutottunk a gyakorlatban felmerülő problémák hathatós kezelési módjához ... persze, mehetünk még ennél közelebb is.

Jóllehet sikerült egyetlen „változó” (egy struktúratömb) segítségével megoldanunk egy képzeletbeli bank ügyfeleinek nyilvántartását, s örültünk is ennek nagyon, mégis némileg gúzsba kötve éreztük a kezünket, hiszen a tömbméret előre történő megadásának kényszere alól továbbra sem tudtunk kibújni, ami szerfelett kellemetlen, hiszen egyszerűen képtelenség előre megmondani, hogy mennyi ügyfélnek kell majd helyet szorítanunk a szóbanforgó tömbben.

Játszhatnánk persze akkora tömbbel is, ami – úgymond – „csak elég nagy lesz már”, viszont ezzel az eljárással meg a memóriával való pazarlás bűnét vennénk magunkra, melyért esetleg tárhelyszűkével kellene később vezekelnünk.

Ideje, hogy jó hírt is halljunk! Íme: van megoldás a fentebb taglalt problémára. A C nyelv ismer olyan függvényeket, melyek segítségével – már futásidőben – foglalhatunk memóriát, méghozzá pontosan akkorát, amekkorára szükségünk van, illetve létezik olyan függvény is, melyet akkor hívhatunk segítségül, ha a lefoglalt helyet bővíteni vagy esetleg karcsúsítani támadna kedvünk.

Foglaljunk hát helyet! *... mutatósat...*

Arról van szó, hogy fontos és nagyon hasznos dolog ugyan a mutatók használatán keresztül biztosítani a függvények számára pajtásaik lokális változóinak elérését,

(ily módon elkerülve a globális változók – szerencsésnek éppen nem mondható – igénybevételét), mégis jó ha tudjuk, hogy *általában nem ezért alkalmazunk mutatókat.*

Hogy akkor mégis mire jók még azok?

Nos, például a **dinamikus memóriahasználatban** segítenek minket, melynek során memóriablokkot foglalhatunk le, amit aztán a lefoglalt blokk elejét jelölő mutató segítségével tudunk elérni. Ha pedig már nincs szükségünk az adott területre, rögtön fel is szabadíthatjuk azt (ahogy azt illik is jólnevelt berkekben), még akkor is, ha erről a folyamat kilépése után az operációs rendszer egyébként gondoskodik.

Fontos: minden memóriafoglalás után ellenőrizzük le, hogy sikeres volt-e a kísérlet! (Ennek módját később taglalom.) Ha ezt elmulasztjuk meglépni, illetve ha nincs kezdőértéke a mutatónknak, akkor minden esélyünk megvan rá, hogy a folyamat sírba szálljon. Hiába, a mutatók használata „veszélyes üzem".

Az alábbiakban megismerünk néhány olyan függvényt, melyek a memóriafoglalásban hivatottak minket segíteni, s egyébiránt az `stdlib.h` lakói.

Az egyik ilyen függvény a `malloc`. Róla az alábbiakat kell tudni:

- *bemenő* paraméter gyanánt a lefoglalandó memóriablokk *méretét* éhezi
- *visszatérési* értékként pedig egy *mutatót* kapunk tőle, mely a *lefoglalt terület elejére* mutat, s ami még **nagyon fontos**: ez a mutató típus nélküli – `void *mutato` –, hiszen előre az még nem tudható, hogy a lefoglalt területen milyen típusú változót szeretne tárolni a `malloc` megidézője
- ha valamiért nem volt sikeres a foglalás (pl nem volt már annyi hely, mint amennyit igényeltünk), akkor a visszaadott mutató-érték NULL lesz
- fontos észben tartanunk, hogy a *lefoglalt* memóriarész esetleges *tartalmát* a `malloc` *nem törli*, így bármi lehet ott!

Tekintsük az alábbi egyszerű példát, ahol egy lebegőpontos elemek számára otthont adó tömb részére foglalunk helyet, melynek a méretét a forrás írásakor még nem ismerjük!

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

int main()
```

```
{
 int n; float *tombeleje;

 printf("Mennyi␣szam␣lesz␣a␣tombben?\n"); scanf("%d", &n);

 tombeleje=(float *)malloc(n*sizeof(float));

 if(tombeleje==NULL)
 {
  printf("Sikertelen␣memoriafoglalas.␣Program␣kilep.\n");
  return(0);
 }

 printf("<ENTER>␣es␣felszabaditom␣a␣teruletet.\n");
 if(getchar()=='\n'){getchar();}

 free(tombeleje);
 return(0);
}
```

A fenti programban létrehoztunk egy `tombeleje` nevű mutatót, mellyel kapcsolatban csak annyit jeleztünk, hogy az általa kijelölt helyen lebegőpontos változó (lesz) található. Értéket csak a `tombeleje=(float *)malloc(n*sizeof(float));` sorban kap, mégpedig a `malloc` által visszaadott memóriacímet. Nézzük csak meg ezt az értékadást kicsit alaposabban!

Először is a `malloc` megkapja a lefoglalni kívánt terület méretét az `n*sizeof(float)` képében. Ezután lefoglalja az adott méretű blokkot, majd megadja a `tombeleje` mutatónak érték gyanánt a **lefoglalt terület elejének a címét** (ugyanis az a `malloc` visszatérési értéke).

A `(float *)` típuskényszerítésre (ugye emlékszünk még, hogy mi az?) azért van szükség, mert a `malloc` által visszaadott mutató, típusát tekintve, `void *` (azaz, típus nélküli), míg az általunk deklarált `tombeleje` mutató `float`-ot jelöl.

Mivel nem minden fordító fogadja el, ha egy adott típusú mutatónak az értékadáskor olyan memóriacímet igyekszünk adni, amely valamilyen más típusú adat helyét jelöli, kénytelenek vagyunk a típuskényszerítés eszközével élve rábírni az „üzletre”.

Ez átalakítást nem igényel, mivel minden mutató mérete egyforma (hiszen a cím

mérete független annak „lakójától”). A mutató típusa más miatt fontos, például azért, hogy a vezérlés – teszem azt – egy mutató tömbszerű használatakor (amikor tömbszerűen indexeljük azt) ki tudja kalkulálni a tömbelemek közötti „lépéshosszt”.

Az `if(tombeleje==NULL)..blablabla....` feltételben ejtjük szerét annak, hogy a foglalás sikeres vagy sikertelen mivoltát ellenőrizzük és utóbbi esetben a folyamatot kiléptessük. Azt, hogy meddig lesz sikeres a helyfoglalás, kipróbálhatja bárki kedvére, például oly módon, hogy egyre nagyobb elemszámok megadása által eljut addig a méretig, melynek már nem jut elég hely vagy egyszerűen csak -1-et ad meg az `n` változóba érték gyanánt. Ekkor maga is megtapasztalhatja a folyamat kilépésével jelzett NULL érték megjelenését.

Fontos, hogy a fenti esetben alkalmazott `tombeleje` névre keresztelt mutatóra ezúttal ne úgy tekintsünk, mint puszta „mezítlábas memóriacetlire”! Igazából, szerepét tekintve, a `tombeleje` itt egy **dinamikus memóriafoglalású változó** (egy dinamikus tömb), amiben mindenféle dolgot tárolhatunk (esetünkben épp `n` darab lebegőpontos típusú értéket)[1].

A program végi `free(tombeleje);` sor szolgál a lefoglalt rész felszabadítására, a `free` függvény segítségével a vázolt módon.

No de nézzük, *hogyan lesz ebből tömb*! A helyzet az, hogy sehogyan, mert már most is az, ... akkor is, ha nem. :)

Emlékezzünk csak arra a tényre, amit már oly sokszor emlegettem volt a gyakorlatok során, s nem győzöm elégszer ismételni:

Ha egy **tömb**nek csak a **nevét** említjük a forrásban, akkor az nem más, mint a tömb első elemét megcélzó **mutató**. Egy tömbnevet tehát mutatóként is használhatunk, s itt a kulcsa „mindennek”, ugyanis a fenti állítás „visszafelé” is igaz, amennyiben a **mutatók tömbszerűen** is *használhatóak*. Hogy hogyan? Egyszerűen úgy, hogy tömbnévként tekintünk rájuk, s elkezdjük „megindexelni” őket. (Fontos persze, hogy az indexelések által kijelölt helyekre írhasson, s onnan olvashasson is a vezérlés, ami viszont csak úgy lehetséges, ha az indexekkel hivatkozott memóriarész le van foglalva. **Ezt** (is) biztosítja a `malloc` függvény.)

Csak, hogy ez az egész „`malloc`-olás” világosabb legyen, nézzünk erre is egy példát, ahol az előző kódot oly módon bővítjük, hogy a lefoglalt helyet feltöltjük számokkal épp, mint egy tömböt (ezért is becézik *dinamikus tömb*-nek ezt a

[1] Igen, igen, persze, hogy végül is, továbbra is „csak” egy memóriacímről értekezünk, de azért ez mégsem ugyanaz, mint az eddigi értelemben vett „mutogatás”.

típust), majd ki is íratjuk a bevitt értékeket!

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

int main()
{
 int i,n; float *tombeleje;

 printf("Mennyi szam lesz a tombben?\n"); scanf("%d", &n);

 if(!(tombeleje=(float *)malloc(n*sizeof(float))))
 {
  printf("Sikertelen memoriafoglalas. Program kilep.\n");
  return(0);
 }

 for(i=0;i<n;i++)
 {
  printf("%d.szam: ", i+1);scanf("%f", &tombeleje[i]);
 }

 for(i=0;i<n;i++)
 {
  printf("\n%d.szam: %1.1f\n", i+1, tombeleje[i]);
 }

 printf("<ENTER> es felszabaditom a teruletet.\n");
 if(getchar()=='\n'){getchar();}

 free(tombeleje);
 return(0);
}
```

Lássuk csak, hogy mi is történt odafent! A fenti kódban, miután a vezérlés bekérte az `n` értékét, a `malloc` lefoglalta a kívánt méretű memóriablokkot (mint ahogy például mi kihúznánk egy sort egy könyvben a filcünkkel), majd a foglalás után a vezérlés az adott sorban látható = jel hatására „ráfordította” a `tombeleje` mutató „nyilát” a lefoglalt „sor” elejére.

Tette mindezt úgy, hogy az egész történet egy feltétel fejében zajlott le. Hogy ez mégis mire volt jó? Ily módon lehetőségünk nyílt összevonni a memóriafoglalás műveletét és annak ellenőrzését, hogy vajon sikeres volt-e az. A célunk az

volt, hogy sikertelen foglalás esetén „szálljon ki” a program a „buliból”. Bogarásszuk végig, mi „járt” az `if` fejében! Tudjuk, hogy ha bármi probléma lép fel, a `malloc` NULL-t ad vissza, ami bekerül az `if` feltétel fejébe. A NULL – logikai értelemben – hamis, tehát a vezérlés nem hajtja végre az `if` magjába csomagolt utasítás(oka)t. Mi viszont épp arra szeretnénk rávenni, hogy NULL esetén hajtódjanak végre a szóbanforgó műveletek. Ezt úgy értük el, hogy a NULL által képviselt, logikailag hamis értéket igazzá tettük a `!` jel betoldása által.

Természetesen, ha sikeres volt a fogalalás, a `malloc` által visszaadott érték nem NULL, hanem a lefoglalt blokk elejének a címe, s mint tudjuk, logikai értelemben minden, ami nem nulla, az igaz, tehát az `if` magja végrehajtódna, de szerencsére a `!` jel ekkor is teszi a dolgát, s „meghamisítja” az „igazságot”, így sikeres foglaláskor nem lép ki a program.

De vajon muszáj így tennünk, csak így végezhetünk „hibavizsgálatot? Nem, egyáltalán nem muszáj, maradhatunk az elsőként látott „szeparált” változatnál, de akár az alábbi kivitelezés is „ér”:

```
if(NULL==(tombeleje=(float *)malloc(n*sizeof(float))))
{
 blablablabla...
}
```

Mindezek után az első `for` ciklusban mindenféle számokat írogattunk be a billentyűzeten keresztül, melyekkel kapcsolatban az volt a feltett szándékunk, hogy azok szépen, egymást követve kerüljenek be a lefoglalt blokk – a ciklus haladtával egyre fogyatkozó mennyiségű – szabad részére, ahonnan a második ciklus segítségével ki is írattuk mindet a képernyőre.

Namármost, nézzük például a kiíratást! Minden általunk kiíratni óhajtott szám esetén, meg kellett „mutatnunk” a `printf` függvénynek, hogy hol találja azt. Maga a `tombeleje` mutató csak a lefoglalt „sor” elejét mutatta meg, így azt, hogy a soron belül hol található az épp kiírandó rész, az `i` értéke által jelöltük ki.

Miután a vezérlés – a tombeleje mutató „nyilát” követve – megtalálta annak a „sornak” az elejét, ahol a kiírandó érték „lakik”, onnan még `i` számú lépést kellett megtennie, hogy elérjen a releváns részhez. Azt, hogy mégis mekkorákat lépjen, a `tombeleje` mutató típusából tudta meg. Ezért fontos a mutatók létrehozásakor megadni azok típusát, hiszen abból tudható meg, hogy milyen típusú adatra mutat, amiből viszont, mint esetünkben is, tömbszerű használat esetén kikalkulálható a „lépések hossza”.

Nem véletlenül nyúltunk hát a típuskényszerítés eszközéhez a `malloc`-os sor-

ban, mert hiába egyezik meg – típusuktól függetlenül – a mutatók mérete (hiszen mind egy–egy memóriacím), egyáltalán nem mindegy, hogy mire mutatnak rá, hogy mekkora méretű a címzett adattípus! Egyébiránt pedig, láttunk már ilyesmit régebben is, mégpedig akkor, amikor tömböt adtunk át valamely hívott függvénynek, így nemigen van már mit magyarázni a fentebb látottakon.

Ha ez így most világos, akkor örülünk, s megyünk is tovább legott, hogy megvizsgáljuk, mit kell tennünk abban az esetben, ha függvényeken belül kell megoldanunk a memória dinamikus foglalásával, a „lestoppolt" terület feltöltésével, valamint a feltöltött értékek megjelenítésével kapcsolatos teendőket. Mint rendszerint, most is több lehetőség kínálkozik a feladat megoldására. Az egyik, kissé fapados, a másik már némileg elegánsabb. (Sőt, van még egyéb módja is a probléma kezelésének, de nem szeretném a végtelenségig bonyolítani ezt a szerény kis segédletet.)

Lássuk előbb az egyszerűbb változatot, mellyel kapcsoltban meg kell jegyeznem, hogy azért a hiba jelzésére való igényünkből már itt sem engedünk.

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

float *foglal_es_feltolt(int *meret);
void kiir(float *t, int m);

int main()
{
 int n; float *tombeleje=NULL;

 if(!(tombeleje=foglal_es_feltolt(&n)))
 {
  printf("Sikertelen␣foglalas.␣Progi␣kilep.\n");
  return(0);
 }

 system("clear");

 kiir(tombeleje, n);
 free(tombeleje);
 return(0);
}


float *foglal_es_feltolt(int *meret)
```

```
{
  int i, m; float *helyi=NULL;
  printf("Mennyi szam lesz a tombben? ");
  scanf("%d", &m); *meret=m;

  if(!(helyi=(float *)malloc(m*sizeof(float))))
  {
   return helyi;
  }

  for(i=0;i<m;i++)
  {
   printf("%d. szam: ", i+1); scanf("%f", &helyi[i]);
  }

  return helyi;
}

void kiir(float *t, int m)
{
  int i;
  for(i=0;i<m;i++)
  {
   printf("%d. szam: %1.2f\n", i+1, t[i]);
  }
}
```

Ami a fentieket illeti, itt a lényeg az, hogy a `foglal_es_feltolt` függvény dolga a kívánt méret megérdeklődése mellett a válaszként megadott méretű memóriaterület lefoglalása, egyben feltöltése is.

A bonyodalmak ott kezdődnek, hogy a tömb mérete – `n` – és maga a tömb elejét kijelölő memóriacím (a `tombeleje` mutató értéke) is számíthat a későbbiek folyamán hívandó egyéb függvények (esetünkben a `kiir`) érdeklődésére.

A `foglal_es_feltolt` függvénynek tehát a saját `m` változója értékét bele kell írnia a `main`-beli `n`-be is, továbbá a `malloc`-tól a `helyi` mutatójába csorgott – a lefoglalt terület elejét jelölő – címet is el kell juttatnia valahogy a `main`-beli `tombeleje` mutatóba! Tekintve, hogy egy függvénynek csak egy visszatérési értéke lehet, kézenfekvő a „mutató követése” művelet alkalmazása, ily módon „megkímélendő” a `return`-t, más szerep betöltésére „eltartalékolván” azt.

A feladat első fele könnyen kezelhető az `n` címének a `foglal_es_feltolt`

függvény hívásakor paraméterként való átküldése, majd a mutató követése művelet – `*meret=m;` – használata által, viszont a `helyi` / `tombeleje` mutatók értékátadásával kapcsolatban nehezebb a dolgunk, ugyanis ebben az esetben nem egy „mezítlábas” változót kellene módosítanunk egy másik függvényből – a címét hordozó mutató követése által – , hanem egy olyan változót, ami már egyébként is mutató.

Ezt a problémát – egyelőre – úgy kerültük meg, hogy a `helyi` mutató értékét adtuk meg a `foglal_es_feltolt` függvényben visszatérési érték gyanánt (ezért `float *` a függvény típusa(!)), s magát a függvényt egy értékadás jobb oldalán hívtuk, így a szóbanforgó visszatérési érték (a lefoglalt memóriablokk címe) egyenest az értékadás bal oldalán várakozó `tombeleje` mutatóban landolt.

E nélkül az „akció” nélkül is megtörtént volna a foglalás és a feltöltés, de a lefoglalt hely címe a feledés homályába veszett volna, hiszen azt csak a `foglal_es_feltolt` függvény `helyi` mutatója kapta volna meg ajándékba, ami a függvény „visszavonulásakor” megsemmisült volna, annak összes helyi változójával egyetemben[2].

A `kiir` függvényben (eljárásban) nincs újdonság. Módosítania nem kell semmit, mindössze kap egy tömböt, annak elemszámát, melyeket felhasználva kiírja a tömbbe tett számokat.

Most, hogy ezt átrágtuk, készítsük el a már említett elegánsabb megoldást is, mely már *mutatót jelölő mutatókkal* operál, illetve a `scanf` függvény, második paraméterével kapcsolatos elvárásait is kihasználja a tömörebb formalizmus érdekében.

Nézzük hát, mire jutnánk a mutatót mutató mutatóval!

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

int foglal_es_feltolt(float **t, int *meret);
void kiir(float *t, int m);

int main()
{
 int n; float *tombeleje=NULL;

 if(foglal_es_feltolt(&tombeleje, &n))
```

[2]Sic transit gloria mundi :) „Így múlik el a világ dicsősége”

```
 {
  printf("Sikertelen␣foglalas.␣Progi␣kilep.\n");
  return(0);
 }
 system("clear");

 kiir(tombeleje, n);
 free(tombeleje);
 return(0);
}


int foglal_es_feltolt(float **t, int *meret)
{
 int i;
 printf("Mennyi␣szam␣lesz␣a␣tombben?␣");
 scanf("%d", meret);

 if(!(*t=(float *)malloc(*meret*sizeof(float))))
 {
  return (1);
 }

 for(i=0;i<*meret;i++)
 {
  printf("%d.␣szam:␣", i+1);
  scanf("%f", &(*t)[i]);
 }

 return (0);
}


void kiir(float *t, int m)
{
  int i;
  for(i=0;i<m;i++)
  {
   printf("%d.␣szam:␣%1.2f\n", i+1, t[i]);
  }
}
```

Látható, hogy itt most átadtuk a `main`-beli mutató címét és aztán elvégeztük vele

a mutató követésének műveletét:

```
*t=(float *)malloc(*meret*sizeof(float))
```

Az is feltűnhet továbbá, hogy a `scanf("%d", meret);`-ben a második paraméternél nem volt szükség a címoperátor használatára, mivel a `meret` mutató már eleve egy címet hordozott, mégpedig az általunk értékkel ellátni kívánt `n` változó címét, mely címet a `meret`-ből meg is kapott a `scanf`.

A `malloc(*meret*sizeof(float))`-ban pedig épp eme `n` változó értékére hivatkoztunk a `*meret`-tel, ami újfent csak a már jól ismert mutató követése művelet volt.

Amit itt még érdemes észrevenni az az, hogy a `for` ciklus
`scanf("%f", &(*t)[i]);` utasításában a `*t`, ami – meglepő módon – megint csak egy mutató követés, pontosan a `main`-beli `tombeleje` behelyettesítésére szolgál. (Világos, hiszen ha a `main`-ben töltenénk fel a tömböt, akkor
`scanf("%f", &tombeleje[i]);`-t
írnánk a kérdéses helyre. (A plusz zárójel azért kell, hogy a fordító megtudja, pontosan mely résznek „szól” a `*` és az `&` jel.))

Még egyszer, a lényeg ez: a `&tombeleje` beírása által a `tombeleje` mutató címe belemásolódik a `t` mutatóba a függvényhíváskor, s a `*t=` -vel azt „mondjuk” a vezérlésnek, hogy menj oda, ahová ez a cím mutat (**kövesd** a `t` mutató „nyilát”), s az ott talált változóba (ami nem más, mint a `tombeleje` mutató) írd bele az egyenlőségjel jobb oldalán lévő – `malloc`-tól kapott – címértéket! Ennyi.

Aki átrágta, s megértette az eddigieket, az már eleget tud ahhoz, hogy minden további magyarázat nélkül átböngéssze, s meg is értse a következő kódot, mely a régebbi „bankos - struktúratömbös” programunk „dinamizált” változata.

Nézzük hát meg, hogy lehet egy függvény segítségével dinamikusan lefoglalni a struktúratömbünk helyét, s egyben fel is tölteni azt! No és persze jelenítsük is meg a tömb tartalmát is, ha már feltöltöttük!

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

typedef struct ugyfel
{
 char nev[10]; float penz; float klab;
 }ugyfel;

int foglal_es_feltolt(ugyfel **u, int *m);
```

```
void kiir(ugyfel *u, int n);


int main()
{
  int n; ugyfel *ugyfelek=NULL;

  if(foglal_es_feltolt(&ugyfelek, &n))
  {
   printf("Sikertelen␣foglalas.␣Progi␣kilep.\n");
   return(0);
  }

  system("clear");

  kiir(ugyfelek, n);

  free(ugyfelek);
  return(0);
 }


int foglal_es_feltolt(ugyfel **u, int *m)
{
 int i;
 printf("Mennyi␣ugyfel␣lesz?␣"); scanf("%d", m);
 if(!(*u=(ugyfel *)malloc(*m*sizeof(ugyfel))))
 {
  return (1);
 }

 for(i=0;i<*m;i++)
 {
  printf("%d.␣nev:␣", i+1);
  scanf("%s", (*u)[i].nev);
  printf("Betett␣penz:␣");
  scanf("%f", &(*u)[i].penz);
  printf("Eves␣kamatlab:␣(%%)␣");
  scanf("%f", &(*u)[i].klab);
 }
 return (0);
}
```

```
void kiir(ugyfel *u, int n)
{
int i;
for(i=0;i<n;i++)
{
 printf("%d. nev: %s\t", i+1, u[i].nev);
 printf("Betett penz: %1.2f\t", u[i].penz);
 printf("Eves kamatlab: %1.2f %%\n", u[i].klab);
 }
}
```

Nna.. :)

A calloc

a másik memóriafoglaló függvény, mely amúgy hasonlóan működik, mint a `malloc`, s jóllehet a kettejük által lefoglalt területek egyenértékűek, van néhány apró különbség a két függvény működése között. Az egyik ilyen különbség abban nyilvánul meg, hogy a `calloc` nem egy darab paramétert kér, hanem kettőt, s e két érték szorzatából számolja ki a lefoglalandó terület méretét. Kezdőpéldánknál maradva, amennyiben `malloc` helyett `calloc`-kal szeretnénk lefoglalni az `n` elemű, `float`-okból alkotott tömbünknek szükséges helyet, azt az alábbi módon tehetjük:

```
tombeleje=(float *)calloc(n, sizeof(float));
```

Láthatjuk, hogy az említett különbség mindössze kettő karakterben nyilvánul meg.

A másik fontos különbség már kevésbé nyilvánvaló, ugyanis a `calloc` – túl azon, hogy lefoglalja – még fel is tölti 0 értékű bájtokkal az adott területet. A 0 bájtokkal való feltöltöttséget karakterláncokkal való munka esetén kell különösen észben tartanunk, hiszen azok esetében a lezáró karakterek szerepét épp az említett elemek töltik be. Minden más tekintetben egyébként a `calloc`, ugyanazt teszi mint a `malloc`.

Szép, szép, mondhatná az eddigiekre valaki, de mi van akkor, ha mondjuk a már lefoglalt területet szeretnénk növelni (vagy épp csökkenteni)?

Nos, ebben az esetben a **realloc**-ot kell hívnunk. Ez a függvény is kettő paramétert vár, melyek közül az első a már lefoglalt terület elejét kijelölő mutató (az, amit a `malloc` vagy a `calloc` visszaadott), a második pedig az új méret.

Arra kell csak figyelnünk, hogy a memóriaterületre való további hivatkozásaink során azt a mutatót használjuk, amit már a `realloc` adott vissza, ugyanis megeshet az, hogy a `malloc`/`calloc` által lefoglalt terület mellett már nincs elég hely a `realloc` által lefoglalni kívánt nagyobb területnek, amit a függvény úgy orvosol, hogy keres egy megfelelő méretű helyet, ahová az összes adatot bemásolja az addig használt területről, s már ennek a helynek a címével tér vissza. Ha valamiért csorbát szenvedett a működése, akkor – akár csak az eddigi függvények – a `realloc` is NULL mutatót ad vissza.

Feladat:
Csapjunk egy informatikus bulit, például egy szakestet! (Gondolom, eddig még menni fog.. :)) Írjunk a rendezőknek egy programot, ami a résztvevőket hívatott nyilvántartani! Tartalmazzon a progi egy függvényt, ami feltölti a feladathoz deklarált dinamikus tömböt, egy másikat, ami kiírja az elmeit, valamint egy harmadikat is, amely az egyik – menet közben megadott – , tetszőleges évfolyamhoz tartozó hallgatók adatait kimásolja egy különálló dinamikus listába!

Lehetséges megoldás:

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

typedef struct szakest{char nev[10]; int evf;}szakest;


int foglal_es_feltolt(szakest **h, int *meret);
void valogat(szakest *h, int meret);
void kiir(szakest *h, int meret);

int main()
{
 int n; szakest *hallgatok=NULL;

 if(foglal_es_feltolt(&hallgatok, &n))
 {
  printf("Sikertelen foglalas, leptem!\n"); return(0);
 }

 kiir(hallgatok, n);
 valogat(hallgatok, n);
 free(hallgatok);
 return (0);
}
```

```
int foglal_es_feltolt(szakest **h, int *meret)
{
 int i;
 printf("Mennyien␣jottek␣el?␣"); scanf("%d", meret);
 if(!(*h=(szakest *)calloc(*meret, sizeof(szakest))))
 { return (1); }

 for(i=0;i<*meret;i++)
 {
  printf("\n%d.␣nev:␣", i+1); scanf("%s", (*h)[i].nev);
  printf("Evfolyam:␣"); scanf("%d", &(*h)[i].evf);
 }
 return (0);
}


void valogat(szakest *h, int meret)
{
 int i=0, j=0, ev;
 szakest *v=(szakest *)malloc(sizeof(szakest));
 printf("\nMelyik␣evfolyamot␣valogassam␣ki?␣");
 scanf("%d", &ev);

 while(i<meret)
 {
  if(h[i].evf==ev && j==0){v[j++]=h[i];}else
  {/*Vigyazat, else nelkul nem azt tenne, amit varnank!*/
   if(h[i].evf==ev && j>0)
   {
    v=(szakest *)realloc(v, ++j*sizeof(szakest));
    v[j-1]=h[i];
   }
  }
  i++;
 }

 if(j==0){
  printf("\n\nEgy␣hallgato␣sincs");
  printf("␣az␣adott␣evfolyambol.\n\n");
 }else{ kiir(v, j); }
}
```

```
void kiir(szakest *h, int meret)
{
 int i=0; system("clear");
 do
 {
  printf("\nNev: %s\tevfolyam: %d\n", h[i].nev, h[i].evf);
 }while(++i<meret);
}
```

Próbáljuk meg önszorgalomból átírni a fenti programot oly módon, hogy önállóan szétválogassa az évfolyamokat egy-egy különálló tömbbe, a szakesten zajló évfolyamok közötti vetélkedő végett!

Meglehet, hogy igénybe kell venni akár olyan dinamikus mutatótömböket is, melyeknek minden eleme egy olyan mutató, mely egy dinamikus struktúratömb elejét jelöli, de ez csak az én egyik megoldásomnak a vége, melyben az említett dinamikus struktúratömbök tartalmazzák a válogatott évfolyamok hallgatóinak az adatait. Nyilván – mint szinte mindegyik – ez a feladat is megoldható sokféleképpen. **Persze, csak óvatosan, mert egy informatikus bulin pillanatok alatt elszabadulhat a pokol! :)**

11. fejezet

Szöveges állományok kezelése

11.1. Az előző fejezet margójára

Mielőtt a lovak közé csapnánk, élnék egy fontos kiegészítéssel az előző fejezet kapcsán. Tekintve, hogy már ismerjük a dinamikus helyfoglalású tömbök lelkivilágát, könnyű lesz megértenünk a megoldást egy régi problémára.

Kezdjük rögtön a problémával! Emlékszünk rá, ugye, hogy amikor megalkottunk egy struktúrát, mely például egy nevet tartalmazó – majdan karakterláncot tároló – tömbbel is rendelkezett, kénytelenek voltunk az alábbi módon eljárni:

```
typedef struct blabla{char nev[32]; int vmi;}blabla;
```

Problémánk lényege az, hogy a fenti `blabla` névre keresztelt típusunknak van nevet tároló mezője, aminek rögzített a mérete. Ezzel nemigen tudunk mit kezdeni, ráadásul szerfelelett veszélyes is, hiszen amikor a `scanf` beolvas oda egy nevet a `%s`-sel, akkor – teszem azt – egy 40 karakterből álló név esetén vígan kifut a folyamat számára fenntartott memóriablokkból, adatokat akarván írni oda, ahová nincs jogosultsága bármit is menteni. S noha úgy tapasztaltam, hogy a `C` nyelv egy-két tömbelem erejéig még tűri az ilyen huncutságokat, ezeknél nagyobb hiba esetén már sírba száll a folyamat. Meg persze az az egy-két elemes ”ráhagyás” sem épp életbiztosítás, amit még esetenként elvisel. Szóval, ilyen struktúra-mező esetén elég körülményes kordában tartani e problémát.

A jó hír az, hogy van megoldás! Ha jól tudom, úgy 10 esztendővel ezelőtt – most épp 2018-at írunk – kijött egy bővítmény, ami egyszerűen kezelhetővé teszi az ilyen jellegű hasfájásokat. Windows alatt nem tudom, miként csalogatható elő,

de az alább taglalt eszköz – ami egyébként GNU szabvány – Ubuntu Linux alatt nálam csont nélkül működik, anélkül, hogy igyénybe kellene vennem a `gcc` valamelyik kapcsolóját.

Tehát: ha a `scanf`-be`%s` helyett `%ms`-t írunk, akkor nagyjából úgy fog viselkedni, mint mi, amikor `malloc`-olunk, majd feltöltjük a futásidőben lefoglalt területet. Ebből viszont az következik, hogy *nem* egy `char` típusú *tömbbel*, *hanem* egy `char` típust jelölő *mutatóval* kell dolgoznunk!

A `scanf`, az `%ms` hatására, dinamikusan lefoglal egy akkora területet, ahol elfér az általunk beírt karakterlánc, plusz az azt lezáró 0 értékű bájt, majd a lefoglalt helyet szépen feltölti a beírt karakterekkel, lezárja a karakterláncot, a lefoglalt memóriablokk címét pedig beírja `char` típusú mutatóba (vagy akár azt is mondhatnánk, hogy a mutatót ráfordítja a szóban forgó helyre).

Lelki finomságok e megoldáshoz: mindössze annyi, hogy ez esetben a `scanf`-nek meg kell adni a *mutatónk címét*, hiszen most magát a pointert kell felülírnia (a hely címét kell beletennie), s nem csupán egy mutató által már eleve kijelölt – statikusan lefoglalt – tömbbe kell beírnia mindenféle karaktert. Eddig, amíg statikus tömbökbe írattunk be a `scanf`-fel nagyjából így jártunk el:

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

Fent, mivel a tömb neve (`tombneve`) már eleve egy – a tömb első elemét kijelölő – mutató, a `scanf` megkapta a címet, ahová az elemeket beírhatja.

`%ms` *esetén viszont merőben más a helyzet!* Az igaz, hogy itt is beír egy karakterláncot valahová, viszont itt – a `scanf` hívásakor – még nincs is hová beírni a karakterlánc elemeit! A beírás előtt még le kell foglalnia (a megfelelő méretű) tömb helyét, s csak aztán elvégezni a beírást. Igen ám, de a lefoglalt hely címét el kell menteni valahová, hiszen később is szeretnénk használni azt! No ezért van szükség mutatóra, ami képes eltárolni azt (a címet), s nem pedig egy statikus tömbre. Az eljárás ilyenkor az alábbi módon fest:

```
char *tombneve;
scanf("%ms", &tombneve);
```

Ezután a `tombneve` pointert épp úgy használhatjuk, mintha statikusan foglalt tömbbe írt karakterlánccal dolgoznánk. A kiíratása például mindkét esetben így működik:

```
printf("%s", tombneve);
```

Az elmeit módosíthatjuk – például kisbetűkből nagybetűkre, stb, stb – megszám-

lálhatjuk, ”meg minden”. Annyi a különbség csupán, hogy a lezáróbájttal nem csak a karakterlánc, de a karakterláncot tartalmazó tömb is véget ér. Végül, nézzük, miként festene az ilyen dinamikus `scanf` a struktúrás esetben! Nem kell visszalapoznunk, így nézett ki a ”hagyományos” változat (ahol fix – előre lefoglalt – mérete volt a tömbnek (persze, itt most nagyon lecsupaszítom a kódot)):

```
typedef struct blabla{char nev[32]; int vmi;}blabla;
main(){
blabla valtozo;
scanf("%s", valtozo.nev);
}
```

.... és így fest az új változat:

```
typedef struct blabla{char *nev; int vmi;}blabla;
main(){
blabla valtozo;
scanf("%ms", &valtozo.nev);
}
```

Nem kell hát tovább bajlódnunk a korlátozott tárolókapacitású – karakterláncokat tároló – , struktúrabeli tömbök rákfenéje miatt, mostantól olyan hosszú karakterláncot adunk meg futásidőben, amekkora csak szükséges.

Ez persze még nem minden, hiszen a másik nagy bánatunk az szokott lenni, hogy `scanf`-et használva nem tudunk szóközöket szendvicselni a vezeték- és keresztnév közé, mivel alapértelmezetten – `%s`-sel hívva – a függvény csak a szóközig végzi el a beolvasást, figyelmen kívül hagyva mindent, ami utána következik.

Van persze erre is gyógyír! Amennyiben egy karakter típusú, statikus tömbbe szeretnénk beolvasni egy *teljes* nevet úgy, hogy valóban szóköz legyen benne, s ne például a "_" karakter helyettesítse azt, akkor fentihez hasonló példával élve, kódunk az alábbi módon festene:

```
typedef struct blabla{char nev[32]; int vmi;}blabla;
main(){
blabla valtozo;
scanf("%[^\n]s", valtozo.nev);
}
```

Ez ugyan szép, viszont olybá tűnhet, hogy e lépéssel lemondani kényszerülünk a kötetlenül hosszú név (szöveg, stb) bevitelének imént felfedezett lehetőségéről... pedig dehogy! Eme opciókat össze is házasíthatjuk, tahát visszatérhetünk a nem maximált hosszúságú bevitelhez úgy, hogy közben a legutóbbi vívmányunkat (a

szóköz beolvasását) is használhatjuk. Ilyesmire az alábbi kód teszi képessé programunkat:

```
typedef struct blabla{char *nev; int vmi;}blabla;
main(){
blabla valtozo;
scanf("%m[^\n]s", &valtozo.nev);
}
```

Hozzácsapva a fenti kódhoz egy kiíratást, majd - pl Ubuntu alatt - kipróbálva, szépen látható, hogy valóban működik a fenti módszer: beírhatunk egy tetszőlegesen hosszú vezetéknevet, amit – a szóköz lenyomása után – egy szintén tetszőleges hosszúságú keresztnév követhet. Ha a beolvasás után ki is íratjuk a nevet, látható, hogy az hiánytalanul megjelenik a karakteres képernyőn.

Hogy Windows, illetve a termekben használt devcpp IDE alatt mindez miként oldható meg, azt momentán sajnos nincs időm kiszimatolni viszont azok számára, akik azt a rendszert, illetve környezetet használják – mentőöv gyanánt – azt tudom mondani, hogy tetszőleges hosszúságú karakterláncot ugyan nem tudnak majd beolvastatni (, illetve nem lesz meg az a kényelmük, hogy épp csak akkora helyet foglaljanak a karakterláncaik, amekkorára szükségük van), viszont annyit azért megtehetnek, hogy a `scanf`-fel csak annyi karaktert olvastatnak be az erre szánt tömbbe, amennyinek jut benne hely.
Egy példa, s rögtön világos lesz:
Legyen egy 32 elemű karaktertömbünk! Ha nem akarunk a `scanf`-es beolvasásnál "kifutni" belőle, a következőt kell a kódba beírni:

```
char tomb[32];
scanf("%31s", tomb);
}
```

Ha az eddig használt `%s` helyett a fent látható `%31s` írjuk be a `scanf`-be, akkor – ha a begépelt karakterlánc hosszab, mint 31 elem – a `scanf` csak az első 31 elem ASCII kódját pakolja bele a tömbbe, a többit figyelmen kívül hagyja. S, hogy miért csak 31 és nem 32? Azért, mert kell hely a lezáróbájtnak is, amivel a `scanf` mindenképp megkínálja a tömböt még, az utolsó beolvasott elem után.

A segédlet további részében a kódokat úgy fogalmazom majd meg, hogy azok a laborokban használt környezetek alatt bíbelődve is működjenek. Mindez azt jelenti, hogy aki élni kíván a `scanf` fentebb ismertetett – Linuxhoz kötődő – lehetőségeivel (tetszőleges hosszúságú, illetve szóközt is tartalmazó karakterláncok beolvasása), annak magának kell kisakkozni a vonatkozó kódváltozatot. (Az imént bevezetett ismeretek birtokában mennie kell majd a dolognak!)

11.2. Szöveges állományok

Lassan – e kis, programozásba való bevezető kurzusunk végéhez közeledvén – könnyen lehet, hogy sokan kezdik (amúgy tejesen jogosan) szükségét érezni annak, hogy az általuk megírt programok eredményei túléljék az őket eredményező folyamat futását, s nemkülönben annak, hogy a program által feldolgozandó adatok bevitele – ha lehet – ne kötődjön kizárólag a billentyűzeten történő pötyögéshez. Erről és még néhány egyéb dologról szól ez a fejezet.

Kezdet gyanánt meg kell említenem, hogy a címben említett „állománykezelés”-nek, mi itt csak a töredékével tudunk foglalkozni, mégpedig a szakirodalom által rendszerint „csatornák használata”-ként ismert részével[1]. Ennek az az oka, hogy a krónikus időhiány miatt kénytelenek vagyunk lemondani az „*alacsonyszintű állománykezelés*”-ről (*low level file handling*), mivel az ide tartozó eszközök használatának elsajátítása jóval tovább tart, mint a csatornák alkalmazásának megismerése. Ez nagy kár, ugyanis ezek az eszközök jóval rugalmasabbak, mint a csatornák, s általuk olyan műveletek is végrehajthatók, amelyek csatornákon keresztül kivitelezhetetlenek. Egyáltalán, amit egy állománnyal meg lehet tenni, azt alacsonyszintű fájlkezeléssel mind megtehetjük[2].

11.2.1. Áttekintés

C-ben, kétfajta fájltípust tudunk kezelni. Az egyik az általános/ **bináris**, a másik a szöveges/ **soros** típus.

Ha a fájlokra adatfolyamként (stream) tekintünk, akkor talán könnyebben lesz értelmezhető a nemsokára tárgyalandó csatorna kifejezés is (, ami lehet adatforrás, illetve adatnyelő). A bináris fájlok kezelésében minket segítő függvényeket „nem érdekli”, hogy mi van a fájlokban, csupán beolvassák/kiírják a bájtokat, mely adatokkal aztán azt teszünk, amit akarunk. Kezelhetünk ily módon szöveget, adatbázist, filmet, stb–t, de(!) az aktuális fájlformátumot ismernünk kell!

[1] Igazából a program, az indításakor három csatornát már eleve megnyitott állapotban vesz át a rendszertől. Ezeket részben már ismerjük, az `stdin` és az `stdout` (szabványos bemenet és kimenet) „személyében” (a `printf`-fel és `scanf`-fel beléjük írtunk, belőlük olvastunk). Amiről nemigen esett szó, az az `stderr` (szabványos hibacsatorna). A fájlkezeléshez viszont a már miáltalunk megnyitott egyéb csatornákon keresztül vezet majd az út.

[2] Amennyiben valaki önképzés gyanánt utánamenne (tele van a web forrással), tartsa szem előtt, hogy az alacsonyszintű állománykezelés kizárólag UNIX szabvány! Tehát Linux alatt több, mint valószínű, hogy „csont nélkül” alkalmazható.

Jelen kurzusban, mi most megelégszünk a szöveges állományok írásával/ olvasásával. (Nyilván, ha később valaki szükségét érzi, a bőségesen fellelhető forrásokból könnyedén továbbképezheti magát az állománykezelés egyéb módjainak terén is.) A C nyelv – „beépítetten" – csupán a szöveges állományok feldolgozásához nyújt némi támogatást, de az sem mondható épp a legkielégítőbbnek. Legyünk azonban elnézőek, s gondoljunk bele, hogy a C nyelv megalkotásának idején még a képernyő sem volt elterjedten alkalmazott periféria, így hát nem is készülhetett támogatás médiafájlokhoz, stb-hez! (Az újabb nyelvek (pl.: Java, PHP) azonban rendszerint rendelkeznek grafikus fájlokat kezelő függvényekkel, osztályokkal.)

11.2.2. Kezdhetjük?

Tehát: első körben, a **lényeg** az, hogy szeretnénk, ha az általunk írt programok valahová elmentenék a futásuk eredményét, ahonnan előhalászhatjuk azokat bármikor, anélkül, hogy újrafuttatnánk az adott futtatható állományt a terminálról való leolvasás végett (mert ugye ez eléggé macerás, meg különben is...)

Kérdés: Hogyan írathatunk ki valamit a programmal egy fájlba?

Válasz: Először is meg kell mutatnunk (egy mutatóval), hogy hová legyenek kiírva a megőrzendő adatok, másodszor pedig, el kell végeznünk a kiíratást egy arra alkalmas függvénnyel! (Létezik még persze nulladik, meg többedik lépés is ezeknél, de most azokat övezze – egyelőre – szemérmes hallgatás!)

Haladjunk hát apránként, s kezdjük a mutatóval, amiről egyelőre elég, ha annyit tudunk, hogy a fajtáját tekintve, ez alkalommal egy `FILE` típusú pointerrel – `FILE` típusú struktúrát jelölő mutatóval – hozott össze minket a gondviselés.

Létrehozásakor (deklarálásakor) tehát a következő módon kell eljárnunk:
`FILE *fm;` (`fm`, mint fájlmutató).

Értéket úgy adhatunk neki, hogy munkába hívjuk a `fopen` névre keresztelt függvényt, mely megnyitja (ha pedig még nem létezett, de szeretnénk beleírni, akkor létre is hozza) a minket érdeklő fájlt. Mindez összefoglalva, az alábbi formába dermeszthető:

```
FILE *fm;
fm=fopen("idements", "w");
```

Itt számunkra már csak a `fopen` két paramétere tartogathat némi újdonságot, melyekkel kapcsolatban az a fontos, hogy az első a nyitandó/ létrehozandó fájl neve (esetleg, Windows alatt megadható a kiterjesztése is, pl.: `.txt` mert még a

végén nem tudna mit kezdeni vele a rendszer), míg a második azt hivatott jelezni a vezérlésnek, hogy mi a szándékunk a fájllal.

Esetünkben írni szeretnénk bele, tehát a write-tól ránk maradt `w`-t írtuk be az idézőjelek közé.

Van tehát egy nyitott fájlunk, ami csak arra vár, hogy megírjuk a történetét, így ezennel – in medias res – meg is tesszük azt, s utána mindent elmagyarázok.

Példa: Írjunk ki egy szöveges fájlba egy tetszőleges méretű szorzótáblát!

Lehetséges megoldás:

```
#include<stdio.h>

main()
{
 int s, o, i, j;
 FILE *fm;

 fm=fopen("szorzotabla", "w");
 printf("Sorok␣szama:␣"); scanf("%d", &s);
 printf("Oszlopok␣szama:␣"); scanf("%d", &o);

 for(i=1;i<=s;i++)
 {
  for(j=1;j<=o;j++)
  {
   fprintf(fm,"%d*%d=%d\t", i, j, i*j);
  }
  fprintf(fm,"\n\n");
 }
 fclose(fm);
}
```

A fenti – jelen állapotában még szerfelett hiányos(!) – programocska futtatása után, remélhetőleg egy újdonsült *szorzotabla* nevű állomány tűnik fel abban a mappában, ahová a futtatható állományt (tehát nem a forrásfájlt(!)) elmentettük.

Látható, hogy a `main` első sorában deklaráltuk a fájlmutatónkat, hogy aztán a „nyilát” a következő sorban rögtön rá is fordítsuk a `szorzotabla` fájlra az `fm=fopen("szorzotabla", "w");` értékadás által, melynek következményeként megnyílt/ létrejött a fájl, csak arra (illetve a `"w"` miatt, kifejezetten csak arra) várva, hogy beleírjunk valamit. Ez persze így nem pontos, ugyanis itt való-

jában egy csatornát nyitottunk meg, ami a `"w"` módban való nyitás miatt adatnyelőként működik, s a megnyitás után várja is azokat.

Miután bekértük a tábla méreteit, az azok alapján működő ciklusokba ágyazottan tűnik fel egy újdonsült jövevény, az `fprintf` függvény, ami pontosan úgy működtethető, mint a már jól ismert `printf`. A nem elhanyagolandó különbség közöttük abban rejlik, hogy az `fprintf` a szöveges fájlt tekinti a „képernyőnek”, s „arra dolgozik”, amihez persze – legelső paraméter gyanánt – a fájlmutató segítségével meg kell adnunk (mutatnunk) neki, hogy hol találja a kezelendő állományt!

Miután befejeztük a fájlba való írogatásunkat, be kell zárnunk a csatornát az `fclose` függvény hívása által, mely függvény csak annyit kérdez, hogy „merre van” a bezárandó csatorna, mely kérdésre az `fm`-től kap választ.

Túl vagyunk hát az első fájlkezelésünkön! :) Jupppíí!

Mielőtt azonban önfeledt ünneplésbe kezdenénk, sietve leszögezném, hogy nem csak, hogy nagyon keveset láttunk még az adott témakörből, de amit láttunk, az is erősen **hiányos**.

Nézzük csak, miről is van szó!

- *Hibajelzés:* ha valamiért nem sikerül a `fopen`-nek megnyitnia a fájlt (csatornát), akkor az általa visszaadott mutató értéke NULL. Ezt felhasználhatjuk tesztelésre, illetve sikertelenség esetén kiléptetési feltétel gyanánt is. További kérdésünkre, jelesül, hogy miért nem sikerült a nyitás (lehet úgy 8–9 oka) is választ kaphatunk, de az ilyen mélységű vizsgálat – jóllehet, az előadáson feltehetően említésre kerül – nem képezi tárgyát a gyakorlati kurzusnak (Ha valakit érdekel, nézzen utána a `errno.h`-nak a C könyvtárban.)

- Amennyiben írtunk egy fájlba, úgy az `fclose` hívása előtt rendszerint megejtjük az `fflush` hívását is, mivel – ha esetleg még nem történt volna meg – ez a függvény kiírja az átmeneti pufferek tartalmát az állományba.

- Nem szóltunk a `FILE` típusról sem, ami egy érdekes struktúra, de sajnos ennek taglalása már szintén szétfeszítené a gyakorlat adta kereteket.

- Az elérési útra viszont mindenképpen kitérünk még, ahogy a megnyitás módjaira is!

Addig is, a fentiek alapján alkossunk egy relatíve korrekt, fájlba író függvényt

tartalmazó programot, ahol a függvény ad hibajelzést is a visszatérési értéke által, amit ez alapján akár tesztelhetünk is!

Legyen ez a program a legutóbbi – szorzótáblás – progi továbbfejlesztése!

Lássuk!

```
#include<stdio.h>

int fajlbaszorzo();

main()
{
 if(fajlbaszorzo())
 {
  printf("Sikertelen fajlnyitas.\n");
 }else{printf("Minden OK!\n");}
}


int fajlbaszorzo()
{
 int s, o, i, j; FILE *fm;
 if(!(fm=fopen("szorzotabla","w"))){return(1);}
 printf("Sorok szama: "); scanf("%d", &s);
 printf("Oszlopok szama: "); scanf("%d", &o);
 for(i=1;i<=s;i++)
 {
  for(j=1;j<=o;j++)
  {
   fprintf(fm,"%d*%d=%d\t", i, j, i*j);
  }
  fprintf(fm,"\n\n");
 }
 fflush(fm); fclose(fm);
 return (0);
}
```

A fájl helyét illetően annyit kell most szem előtt tartanunk, hogy amennyiben nem a futtatható állomány „mellé” kívánjuk az írandó fájlt menteni, akkor akár teljes (abszolút) elérési utat is megadhatunk, valahogy így:

```
fm=fopen("/home/levi/Asztal/szorzotabla","w");
```

Azért jó, ha tudjuk, hogy a fenti helyzetben, ha van rá lehetőség, jobb eleve a progi mappájánál vagy a relatív elérési útnál maradni, mivel az abszolúthoz alaposabban ki kell ismernünk magunkat az adott mappaszerkezetben, ami minden rendszernél más és más. Fontos észben tartanunk továbbá – már, ha valaki Windows-zal kívánja megkeseríteni a saját életét – , hogy ott a mappákat – legalábbis régebben így volt – a \ jellel kell elválasztanunk egymástól! Ennek az a következménye, hogy az elérési út megadásakor / jelek helyett, \ jeleket kell bevinnünk a karakterláncba, s itt a bökkenő, hiszen a karakterláncban a \ szimbólum valami mással együtt szokott szerepelni (gondoljunk csak például a `printf("\n");` -re!). Ha azt szeretnénk, hogy egy karakterlánc tartalmazza a \ jel ASCII kódját – ami egyébként 92 – , akkor az idézőjelek közé, oda, ahová a \ jelet tenni szeretnénk, két darab \ jelet kell tennünk, így:\\
Az elérési út megadása tehát valahogy így fog festeni Windows alatt:

```
fm=fopen("C:\\Users\\Levi\\szorzotabla.txt","w");
```

... vagy valami hasonló (nincs Win a gépemen ...)

De felejtsük is el a Windows roppant ”átgondolt” rendszerét, s térjünk vissza a mi kis jól bevált, ingyenes, nyílt forráskódú, vírusmentes és hatékony Linuxunkhoz, mely alatt persze elég csak simán, egy darab / alkalmazása a fenti helyzet esetén.

Írni már tudunk fájlba. Most lássuk, mit kell tennünk, ha be szeretnénk olvasni egyet! Nézzük az alábbi rövid kódot, mely egy fájlt teljes egészében beolvas és ki is ír a képernyőre! Tegyük fel, hogy az előző program által kiírt `szorzotabla` fájl már/még létezik! Írjunk programot, mely a fájl tartalmát kiírja a képernyőre!

```
#include<stdio.h>

int olvaso();

main()
{
 if(olvaso())
 {
  printf("Sikertelen fajlnyitas.\n");
 }else{printf("Minden OK!\n");}
}

int olvaso()
{
 FILE *fm; char k;
 if(!(fm=fopen("szorzotabla","r"))){return(1);}
 while(!feof(fm)){fscanf(fm,"%c", &k); printf("%c", k);}
```

```
  fclose(fm);
  return(0);
}
```

Ami egyből feltűnik odafent, az a `"w"` helyére beékelt `"r"`, ami a *read*-ben gyökeredzik, hiszen ezúttal olvasásra nyitottunk meg egy fájlt. Az egyetlen újdonságot tartalmazó sor a `while` ciklust magába foglaló

```
while(!feof(fm)){fscanf(fm,"%c", &k); printf("%c", k);}
```

sor, melyben legelőször a `feof()` függvényen akadhat meg a szemünk. E függvény, a számára paraméterül megadott csatornát vizsgálja, úgy mégpedig, hogy a visszatérési értéke (ami `int` típusú) akkor **nem** 0, ha elérkeztünk a csatorna végéhez. Innen a neve is, melyben az „`eof`” rész, az „End Of File”-ból származtatható. Nyilván a fájl (csatorna) „végigpásztázása” végett szükséges ciklusba foglalnunk az olvasást. (A fájlban „léptetnünk” viszont, ahogy az látható, nem (nekünk) kell.) A ciklusmagban található a már ismert `printf` függvény, melynek első paraméterét ezúttal karakter kiírására „élesítettük” a `%c` beírása által. (Ugye emlékszünk még, hogy ez mit jelent? A `printf`, a `k` változó értékére – ami egy egész szám – a `%c` miatt ASCII kódként tekint, s a kódnak megfelelő karaktert írja ki a képernyőre.) Maga a `k` változó az `fscanf` függvény által kapja meg az értékét, úgy mégpedig, hogy az `fscanf`, a beleírt `%c` miatt az állományból beolvasott karakter ASCII kódjának megfelelő egész számot írja bele a `k` változóba érték gyanánt. Ezt a változót kapja meg aztán a fent említett `printf`. Természetesen megoldhattuk volna a beolvasást és kiírást akár az alábbi (és más egyéb) módokon is:

- `while(!feof(fm)){fscanf(fm,"%c", &k); putchar(k);}`
- `while(!feof(fm)){k=fgetc(fm); putchar(k);}`
- `while((k=fgetc(fm))!=EOF){putchar(k);}`
- `while(!feof(fm)){putchar(fgetc(fm));}`

, ahol az `fgetc` mindig a következő karaktert olvassa be a fájlból, mely karakter ASCII kódja a függvény visszatérési értéke lesz. Az `EOF` pedig a fájl végét lezáró „láthatatlan” karakter. Olyasmi, mint a `\n`, vagy a `\0` (a karakterláncot lezáró karakter, amit betett sokszor helyettünk a fordító a sztring végére) vagy akár a `\t`, stb. A `putchar` függvény, pedig a paraméterként kapott értéket ASCII kódként kezeli, s a neki megfelelő karaktert írja ki a szabványos kimenetre (a képernyőre). Létezik persze több függvény is, melyek alkalmazásával elérhető ugyanaz a cél,

például a fentinél megmaradván, akár soronként is beolvashatnánk a fájlt, valahogy így:

```
int olvaso()
{
 FILE *fm; char n=50, k[n];
 /*Az n elembol egy darab, a lezarobajt lesz!*/
 if(!(fm=fopen("szorzotabla","r"))){return(1);}
 while(!feof(fm)){fgets(k, n, fm); printf("%s", k);}
 fclose(fm);
 return(0);
}
```

FONTOS:

Az `fgets` és még néhány, itt nem említett függvény működésének utánanézhet bárki, de én megelégszem a kevesebb, ám értő kézzel, megfelelően használt függvénnyel is.

Mire gondolok?

Arra, hogy jóllehet a `printf` és `scanf` függvényeken kívül is létezett egyéb, ki- és beviteli „képernyős” eszköz (`putchar`, `getchar`, stb, stb), mi mégis, kizárólag a `printf` és `scanf` alkalmazásával is célt értünk mindig (esetenként persze lehetett volna egyszerűbb is az élet...).

Hasonlóképpen, az `fprintf` és az `fscanf` függvényeknek is megvannak a hasonló kollégáik, ám az esetek tekintélyes részében nélkülözhetőek. Világos, hogy így nem egyszer igencsak át kell gondolni az általunk alkalmazott módszert, ám az is, hogy ha most minden függvényt a nyakatokba öntök, akkor könnyen lehet, hogy csak a fókuszt szélesítem ki jobban a kelleténél, s a végén teljesen *elvesztek a részletekben, mint vasorrú bába a mágneses viharban ...*

Maradjunk annyiban, hogy aki még bizonytalan, az tanulja meg a fenti kettőt rendesen használni, aki pedig már nagyon magabiztos és kifinomultabb, specializáltabb módszerekre vágyik, az úgyis simán utána tud nézni a többinek, mert már rendelkezik az ehhez szükséges szilárd alapokkal.

Amit most itt, első körben még mindenképpen meg kell említenünk, az a „hozzáfűzés mód” -ban való megnyitás.

Azért van erre szükség, mert ha írásra nyitunk meg egy már létező fájlt, akkor mindent elpusztítunk (felülírunk), amit addig az tartalmazott. Ezt elkerülendő, ilyenkor a hozzáfűzés -mód használata javallott, mely annyiban merül ki egyelő-

re, hogy a `fopen` függvény 2. paraméterébe `"w"` vagy `"r"` helyett, `"a"` -t írunk (*append*). Ilyenkor a fájl megnyílik, s a bele való írás a végétől indulhat, ha pedig addig nem létezett, akkor még az írása előtt „létrejön”.

Majd’ elfelejtettem: az összes fájlkezelő függvény az `stdio.h` lakója, míg a memóriafoglalásnál tárgyaltak a `stdlib.h` -hoz tartoznak.

Természetesen – ha már fájlokkal dolgozunk – nem elégedhetünk meg csupán annyival, hogy a tartalmukat kiíratjuk a képernyőre, elvégre, ennyi erővel akár rá is klikkelhetnénk egy adott ikonra, s ugyanott tartanánk. Nyilván akkor van értelme a fájlból való olvasásnak, ha az ott lévő adatokat fel is tudja dolgozni a programunk.

Nincs más teendőnk hát, mint, hogy a beolvasottakat értékként átadjuk a programunk változóinak, melyekkel már a folyamat is tud mit kezdeni.

Nézzünk egy példát, melyben „legyártunk” egy `n` darab véletlenszámot tartalmazó fájlt, majd ebből a fájlból beolvasva a számokat, növekvő sorrendbe rendezzük azokat, s az így kapott sorozatot visszaírjuk a fájlba, a már meglévő rendezetlen sorozat alá! *Lehetséges megoldás:*

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>  /*A veletlenszam-generalas,
orajel szerinti inicializalasahoz*/

int kiirveletlen(int *m);
int rendez(int m);
int kiirrendezett(int n, int *sz);

int main(){
 int meret=0;
 if(kiirveletlen(&meret))
 {printf("Sikertelen a fajl irasa.\n"); return(0);}
 if(rendez(meret))
 {
  printf("Sikertelen fajlolvasas vagy memoriakezeles.\n");
  return(0);
 }
 return(0);
}


int kiirveletlen(int *m)
```

```
{
 FILE *f; int i;
 srand(time(NULL)); /*orajelhez "igazitott"
 inicializalas*/
 printf("Mennyi lesz? "); scanf("%d", m);
 if(!(f=fopen("veletlen", "w"))){return(1);}
 for(i=0;i<*m;i++)
 {
  fprintf(f, "%d\t", rand());
 } /*veletlenszamok generalasa*/
 fflush(f); fclose(f); return(0);
}



int rendez(int m)
{
 FILE *f; int *szamok, i, j, min, minindex, s;
 if(!(szamok=(int *)malloc(m*sizeof(int)))){return(1);}
 if(!(f=fopen("veletlen", "r"))){ return(1); }
 for(i=0;i<m;i++){fscanf(f, "%d", &szamok[i]);}
 fclose(f);

 for(i=0;i<m-1;i++)
 {
  for(j=i;j<m;j++)
  {
   if(j==i || szamok[j]<min){ min=szamok[j]; minindex=j; }
  }
  if(min<szamok[i])
  {
   s=szamok[i];
   szamok[i]=szamok[minindex]; szamok[minindex]=s;
  }
 }

 return(kiirrendezett(m, szamok));
}



int kiirrendezett(int m, int *sz)
{
```

```
 FILE *f; int i;
 if(!(f=fopen("veletlen", "a"))){return(1);}

 fprintf(f, "\n\nA fenti szamok rendezetten:\n\n");

 for(i=0;i<m;i++){ fprintf(f, "%d\t", sz[i]); }
 fprintf(f, "\n"); fflush(f); fclose(f); free(sz);
 return(0);
}
```

Ami itt most fontos, hogy az `fscanf` -es beolvasásnál nem kellett foglalkoznunk a „láthatatlan” karaktereket képviselő `\t` által megadott tabulátorral, s ez igaz a többi hasonlóra is. Összességében elmondható, hogy tabulátorból, újsorból (stb-ből) – tehát, ún *whitespace* karakterből – akármennyi követheti egymást két beolvasás között, az `fscanf` jól fog működni.

Azonban(!!!) ..... lehetőség van itt elkövetni egy nehezen lefülelhető hibát, jelesül, az ember esetleg könnyen abba a kísértésbe eshet, hogy a beolvasást az alábbi ciklussal hajtassa végre:

```
while(!feof(f)){ fscanf(f, "%d", &szamok[i++]);}
```

, s ne a kódban szereplő

```
for(i=0;i<m;i++){fscanf(f, "%d", &szamok[i]);}
```

ciklust futtasa le. Mi lehet itt a gond? Kipróbálva őket, láthatólag mindkét változat esetében megfelelően működik a programunk. Pedig ez csak a látszat. Gondoljunk csak bele, miként írtuk ki a fájlba a számokat, s jusson eszünkbe, hogy az alkalmazott kód az alábbi módon festett:

```
for(i=0;i<*m;i++)
 {
  fprintf(f, "%d\t", rand());
 }
```

Ez azt jelenti, hogy az utolsó szám után, de még a fájl végét jelző – EOF – bájt *előtt* tartalmazni fog a fájl egy `\t` – "whitespace" – karaktert. Mi ezzel a gond? Nos az, hogy a

```
while(!feof(f)){ fscanf(f, "%d", &szamok[i++]);}
```

ciklus szépen elpásztáz egészen az EOF-ig, s csak akkor nem végez beolvasást, ha már odaért. *Előtte* viszont, amikor épp a `\t`-n "tapos", tekintve, hogy a ciklus fejében erre az esetre még igaz értéket szül a `!feof(f)` kifejezés, lefut még

egyszer utoljára a ciklusmag. Igen ám, de a tömb, ahová olvas már "tele van", hiszen beolvastunk minden számot már, épp annyit, amennyinek helyet foglaltunk a tömbben.

Mindez azt jelenti, hogy a legutolsó beolvasással – amikor már a `\t` karakternél tartunk – az `i` index értékével már kifutunk a tömbből!! Ráadásul még csak értelmes beolvasnivaló sincs már azon a helyen, ahonnan ilyenkor olvasnánk....!

Az egy dolog, hogy a C nyelv van annyira kedves, hogy rendszerint tolerálja az effajta vétséget (egy-egy elem erejéig), viszont attól még a hiba – egy hosszabb program esetén – alattomos, sunyi, észrevétlen módon "forrásnak" indulhat, s csak a jóisten a megmondhatója, hogy miféle gigszert okozhat végül....

Konklúziónk tehát az, hogy *ilyen* esetben, amikor pontosan (és jól) tudjuk, mekkora a tömbünk, melyet épp töltögetünk egy fájlból, ne a fájl méretére, hanem *a tömb méretére alapozzuk* a beolvasó ciklus működési feltételét! Épp, ahogy a kódban is tettük volt:

```
for(i=0;i<m;i++){fscanf(f, "%d", &szamok[i]);}
```

Nnnna... :)
Lassan itt a **második zárthelyi** ...
Ideje hát megnéznünk egy ahhoz hasonló feladatmegoldó készséget igénylő feladatsort! Fontos, hogy innentől már egyszer sem fogunk eltekinteni a hibajelzések küldésétől és feldolgozásától sem!

1. feladat
Írjunk programot, mely megkérdi tőlünk, hogy melyik állománnyal szeretnénk foglalkozni, s miután beírtuk a fájlnevet – esetünkben a `roxfort`-ot – a konzolra, nyissa meg azt, majd töltse fel az ott lévő adatokkal a mi saját dinamikus listánkat (struktúratömbünket)! Ezután jelenítse meg a tömb tartalmát a képernyőn, szépen kiírva például, hogy

```
nev: Lekvarzsibbaszto                evfolyam: 2
nev: Kandisz_Nora                    evfolyam: 1
stb,
.
.
```

Megjegyzés 1: Nyilvánvaló, hogy a dinamikus struktúratömb számára lefoglalandó hely mérete attól függ, hány személyt kell elszállásolni benne. A személyek száma itt most a megnyitott állományban lévő sorok számával egyezik, magyarán, ha meg akarjuk állapítani a lakók számát, nincs más dolgunk, mint végigpásztázni a fájlt, megvizsgálni minden egyes karakter ASCII kódját, s ahányszor

újsor karaktert találunk, annyiszor kell léptetni a sorok (személyek) számát tároló változó értékét. Itt kezdődik a kavarodás! Ugyanis, amikor "kézzel" legyártjuk a beolvasandó szöveges állományt, a fájl utolsó, szöveget tartalmazó sorának végére – az EOF elé – a Linux *gedit* szerkesztője akkor is odabiggyeszt egy újsor karaktert, ha mi nem nyomunk ott *enter*-t. Azt *hiszem*, Windows alatt, ha például jegyzettömbbel szerkesztjük az állományt, nem ez a helyzet, ott nekünk kell még beküldenünk egy *enter*-t, ha szeretnénk betenni az utolsó sor végére is egy újsor karaktert, "lezárandó" az adott sort. De ugyanezt tapasztalnánk például a Linux *kate* szerkesztőjében is. Magyarán, ha kézzel írjuk meg a fájlt, akkor rendszer / szerkesztő függő az utolsó sor végének a sorsa, így magunknak kell letesztelnünk, mi a szitu az általunk használt szövegszerkesztő esetében! Amennyiben viszont `fprintf`-fel írjuk meg a fájlt – Linux alatt legalábbis – csak akkor lesz az utolsó sor végén újsor karakter, ha azt az `fprintf`-fel beletesszük, egy `\n` -nel. Feltehetőleg, ez a legtisztább helyzet a mi szempontunkból.

Megjegyzés 2: Tekintve, hogy itt a tömb számára történő memóriafoglalás előtt - kiderítendő a nyilvántartásban szereplő személyek, s ezzel a leendő tömb elemeinek a számát - még a fájlt is át kell vizsgálni, az áttekinthetőség kedvéért érdemes szétszedni két külön függvénybe a memóriafoglalást és a struktóratömb feltöltését. Magam is így fogok eljárni alább.

Megjegyzés 3: Ha valaki használná, a fejezet elején említett `%ms`-es beolvasás az `fscanf`-nél is működik.

A `roxfort` állomány minden sora tartalmaz egy nevet és egy évfolyamot jellemző számot, mely két adatot egy *tab* vagy egy *space* karakter választ el.

2. feladat
Hívjunk a `main`-ben egy függvényt, amely szimulál egy versenyt, úgy mégpedig, hogy a dinamikus lista objektumainak eddig fel nem töltött – pontszámokat tároló – mezőit véletlenszámokkal hinti be, majd az eredményül kapott – „kibővített" – listát kiírja a képernyőre és a `verseny` nevű szöveges állományba!

3. feladat
Fűzzük hozzá egy függvény segítségével a `verseny` nevű állományhoz a legtöbb pontot szerző versenyző(k) adatait és írassuk ki azokat a képernyőre is!

4. feladat
Írjunk egy kis függvényt, mely – miután az előző függvény belsejében meghívtuk – megjeleníti a képernyőn a hívó függvény által imént kiegészített szöveges állomány teljes tartalmát!

5. feladat

Hívjunk a `main`-ben egy függvényt, amely bekéri a minket érdeklő nevet, s az adott illető(k) adatait kiírja a képernyőre és a `verseny` nevű állomány tartalmához is hozzáfűzi azt!

6. feladat Írjunk egy függvényt, mely kiírja egy `zh` nevű állományba, a magát a függvényt is tartalmazó program teljes forráskódját!
Kitétel: a `string.h` függvényei nem használhatóak!

Lehetséges megoldás:

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

typedef struct kupa{char n[50]; int e; int p;}kupa;

int foglal(kupa **v, char *fn, int *m);   /*1. feladat*/
int beolvas(kupa *v, char *fn, int m);    /*1. feladat*/
void konzolra(kupa *v, int m);            /*1. feladat*/
int versenyeztet(kupa *v, int m);         /*2. feladat*/
int maximum(kupa *v, int m);              /*3. feladat*/
int megjelenit(char *fajlnev);            /*4. feladat*/
int kereso(kupa *v, int m);               /*5. feladat*/
int sztringhasonlit(char *a, char *b);    /*5. feladat*/
int zhkiir(kupa *v, int m);               /*6. feladat*/

int main()
{
 char fajlnev[50]; int meret=0; kupa *verseny=NULL;

 if(foglal(&verseny, fajlnev, &meret))     /*1. feladat*/
 {
  printf("Sikertelen␣fajlnyitas␣vagy␣tarfoglalas.\n");
  return(0);
 }

 if(beolvas(verseny, fajlnev, meret))      /*1. feladat*/
 {
  printf("Sikertelen␣fajlnyitas.\n");
  return(0);
 }

 konzolra(verseny, meret);    /*1. feladat*/
```

```
 if(versenyeztet(verseny, meret))  /*2. feladat*/
 {
  printf("Sikertelen␣fajlnyitas.\n");return(0);
 }

 if(maximum(verseny, meret))   /*3. feladat*/
 {
  printf("\nSikertelen␣fajlnyitas.\n");return(0);
 }

 if(kereso(verseny, meret))  /*5. feladat*/
 {
  printf("\nSikertelen␣fajlnyitas.\n");return(0);
 }

 if(zhkiir(verseny, meret)) /*6. feladat*/
 {
  printf("\nSikertelen␣fajlnyitas.\n");return(0);
 }
 free(verseny); return(0);
}


int foglal(kupa **v, char *fn, int *m)/*1. feladat*/
{
 FILE *f=NULL; char c;

 printf("Melyik␣fajlt␣nyissam␣meg?␣");
 scanf("%49s", fn);

 if(!(f=fopen(fn, "r"))){ return(1); }

 while((c=fgetc(f))!=EOF){if(c=='\n'){(*m)++;}}

 if(!(*v=(kupa *)malloc(*m*sizeof(kupa))))
 {
  return(1);
 }

 fclose(f);
 return(0);
}
```

```
int beolvas(kupa *v, char *fn, int m)/* 1. feladat */
{
 FILE *f=NULL; int i;

 if(!(f=fopen(fn, "r"))){return(1);}

 for(i=0;i<m;i++)
 {
  fscanf(f, "%49s%d", v[i].n, &v[i].e);
 }

 fclose(f);
 return(0);
}


void konzolra(kupa *v, int m)    /* 1. feladat */
{
 int i;
 for(i=0;i<m;i++)
 {
  printf("\nnev: %s\tevfolyam: %d\n", v[i].n, v[i].e);
 }
}


int versenyeztet(kupa *v, int m) /* 2. feladat */
{
 FILE *f=NULL; int i; srand(time(NULL));
 if(!(f=fopen("verseny", "w"))){return(1);}
 printf("\nA verseny eredmenye:\n");
 fprintf(f, "\nA verseny eredmenye:\n");
 for(i=0;i<m;i++){
  v[i].p=rand();
  printf("\nnev: %s\t", v[i].n);
  printf("evf.: %d\t\t", v[i].e);
  printf("pontszam: %d\n", v[i].p);
  fprintf(f, "\nnev: %s\t", v[i].n);
  fprintf(f, "evf.: %d\t\t", v[i].e);
  fprintf(f, "pontszam: %d\n", v[i].p);
 }
 fflush(f); fclose(f); return(0);
}
```

```
int maximum(kupa *v, int m)  /*3. feladat*/
{
 FILE *f=NULL; int i, max=v[0].p;
 for(i=1;i<m;i++)
 {
  if(max<v[i].p){ max=v[i].p; }
 }
 if(!(f=fopen("verseny", "a"))){return(1);}

 fprintf(f, "\nA legtobb pontot -");
 fprintf(f, " (%d-t) - kapta(k):\n", max);
 printf("\nA legtobb pontot -");
 printf(" (%d-t) - kapta(k):\n", max);

 for(i=0;i<m;i++)
 {
  if(max==v[i].p)
  {
   fprintf(f, "%d.", v[i].e);
   fprintf(f, " evfolyambol, %s\n\n", v[i].n);
   printf("%d.", v[i].e);
   printf(" evfolyambol, %s\n\n", v[i].n);
  }
 }
 fflush(f); fclose(f);
 return (megjelenit("verseny"));/*4. feladathoz*/
}


int megjelenit(char *fajlnev)/*4. feladat*/
{
 FILE *f=NULL; char c;
 printf("\n\nA fajl tartalma: ");
 if(!(f=fopen(fajlnev, "r")))
 {
  printf("nem jelenitheto meg.\n\n"); return(1);
 }
 printf("\n\n********************\n");
 while((c=fgetc(f))!=EOF){putchar(c);}
 printf("********************\n");
 fclose(f); return(0);
}
```

```
int kereso(kupa *v, int m)  /*5. feladat*/
{
 FILE *f=NULL; int i=0; char kit[50], van_e=0;
 if(!(f=fopen("verseny", "a"))){return(1);}
 printf("\nKit␣keresunk?␣"); scanf("%49s", kit);
 for(i=0;i<m;i++)
 {
  if(sztringhasonlit(kit, v[i].n))
  {
   van_e++;
   fprintf(f, "\nA␣keresett␣nev:␣%s,\t", v[i].n);
   fprintf(f, "evf.:␣%d,\t", v[i].e);
   fprintf(f, "pont:␣%d\n", v[i].p );
   printf("\nA␣keresett␣nev:␣%s,\t", v[i].n);
   printf("evf.:␣%d,\t", v[i].e);
   printf("pont:␣%d\n", v[i].p );
  }
 }
 if(!van_e)
 {
  printf("Nincs␣ilyen␣nev␣a␣listan.\n");
 }
 fflush(f); fclose(f);
 return(0);
}


int sztringhasonlit(char *a, char *b)/*5. feladat*/
{
 int ahossz=0, bhossz=0, i=0;
 while(a[ahossz]!=0){ahossz++;}
 while(b[bhossz]!=0){bhossz++;}
 if(ahossz==bhossz)
 {
  while(i<ahossz){if(a[i]!=b[i]){break;}i++;}
  if(i==ahossz){return(1);}
 }
 return(0);
}
```

```
int zhkiir(kupa *v, int m) /*6. feladat*/
{
 FILE *ezt=NULL, *ide=NULL; char c;

 /*Ha "program.c", a forras neve:*/
 if(!(ezt=fopen("program.c", "r"))){return(1);}

 if(!(ide=fopen("zh", "w"))){return(1);}

 while((c=fgetc(ezt))!=EOF)
 {
  fputc(c, ide);
 }

 /*vagy:*/
 /*while(!feof(ezt))
  {
   fscanf(ezt, "%c", &c);
   fprintf(ide, "%c", c);
  }*/

 fflush(ide);
 fclose(ide);
 fclose(ezt);

 return(0);
}
```

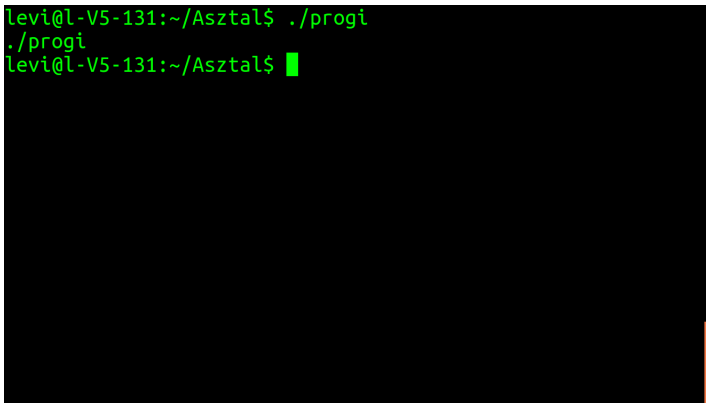
12. fejezet

A `main` függvény paraméterei

Esett már szó sok mindenről e szemeszterben, ám a `main` paramétereinek bemutatását mindeddig szemérmes hallgatás övezte. Megtörvén a csendet, igyekszem alább röviden vázolni a lényeget. Mielőtt bármit is mondanék, futtassuk le az alábbi programocskát!

```
#include<stdio.h>

int main(int argc, char *argv[])
{
 printf("%s\n", argv[0]);
 return(0);
}
```

A program futása az alábbihoz hasonló parancssort eredményez:



A program kiírta a nevét, ami nálam épp `progi` volt. Valójában azonban azt a

parancsot írta ki, ami elindítja, amikor parancssorból indítjuk. Ezúttal a kulcsszó a **parancssor**. Hamarosan kiderül az is, hogy miért. A `main` függvény egy `int` típusú *számot* (ezt jelöltük `int argc` -vel a paraméterezéskor) és egy *mutatót* kap az operációs rendszertől (ez volt a `char *argv[]`). A második paraméter (az `argv`) egy olyan mutató, mely egy mutatókból álló tömb (egy mutatótömb) első (0 indexű) elemét jelöli. (Tehát egy mutatót jelölő mutató.) A tömbben „lakó" mutatók pedig egy-egy karakterlánc első elemére mutatnak. E karakterláncok mindegyike egy azok közül, melyeket a program indításakor beírunk a parancssorba (beleértve magát a program nevét is). Ezért is hívják a `main`, paramétereket tartalmazó részét *parancssori argumentumoknak*. Na de ha az `argv` egy mutatótömb, akkor mi az `argc`? Nos, az `argc` nem más, mint a mutatótömb elemeinek – másként mondva, a `main` függvény számára, a parancssori indításkor átadott karakterláncoknak – a száma. Nézzük csak vissza az előbbi kis program eredményét! Amikor kiírattuk vele a mutatótömb első elemét, tehát az első paramétert, a `printf("%s\n", argv[0]);` sor hatására kiírta a saját nevét vagyis a parancsot, mellyel a parancssorból indítottuk. Valóban, már a program neve is egy parancssori paraméter, méghozzá az említett mutatótömb első (és esetünkben egyetlen) eleme. Az alábbi programocskával ki is próbálhatjuk, hogy valóban csak egy elemet tartalmaz-e az `argv`.

```
#include<stdio.h>

int main(int argc, char *argv[])
{
 printf("Az 'argv' elemeinek a szama: %d", argc);
 printf(", erteke: %s\n", argv[0]);
 return(0);
}
```

Az eredmény:

Játszunk el egy kicsit azzal, hogy a program „átfutja” a saját paramétereit tartalmazó tömböt, s ki is írja, amit ott talál. A kód a következő:

```
#include<stdio.h>

int main(int argc, char *argv[])
{
 int i;
 printf("A␣parancssorban␣kapott␣parameterek");
 printf("␣szama:␣%d\n", argc);
 printf("Alabb␣sorolom␣oket:\n\n");
 for(i=0;i<argc;i++)
 {
  printf("%d.␣parameter:␣%s\n", i+1, argv[i]);
 }
 return(0);
}
```

Az eredmény:

```
levi@l-V5-131:~/Asztal$ ./progi -Hogy_vagyunk,_hogy_vagyunk
? -Jól. -Bővebben? -Nem_jól.
A parancssorban kapott parameterek szama: 5
Alabb sorolom oket:

1. parameter: ./progi
2. parameter: -Hogy_vagyunk,_hogy_vagyunk?
3. parameter: -Jól.
4. parameter: -Bővebben?
5. parameter: -Nem_jól.
levi@l-V5-131:~/Asztal$
```

Szép és jó dolog ugyan az ilyen játék, de miután kinőtte az ember, felvetődik benne, hogy esetleg egyéb haszna is van a `main`, parancssorban való „paraméterezhetőségének”.

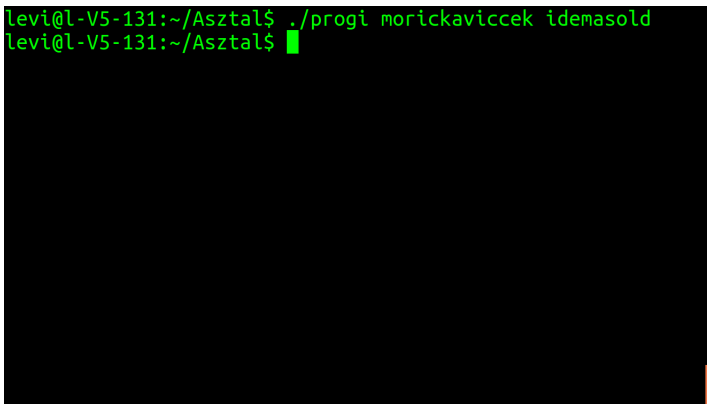
Mi lenne például, ha mindenféle gügyögés helyett inkább fájlneveket írnánk be?

Készítsünk egy szöveges állományt, mentsük el mondjuk `morickaviccek` néven, majd a program indításakor adjuk meg a készített állomány nevét, valamint azét a fájlét, amibe át szeretnénk másolni azt, akkor is, ha az még nem is létezik!

Nézzük az ilyen használatra írt kódot!

```
#include<stdio.h>

int main(int argc, char *argv[])
{
 char k, i=1; FILE *f1=NULL, *f2=NULL;
 if(!(f1=fopen(argv[i++], "r")))
 {
  printf("Sikertelen fajlnyitas."); return(1);
 }

 if(!(f2=fopen(argv[i], "w")))
 {
  printf("Sikertelen fajlnyitas."); return(1);
 }

 while((k=fgetc(f1))!=EOF)
 {
  fprintf(f2, "%c", k);
 }

 fclose(f1); fflush(f2); fclose(f2);

 return(0);
}
```

Ez esetben így néz ki a parancssor az indításkor, illetve utána:



Ha kinézünk a parancssorból, azt tapasztaljuk, hogy megjelent egy `idemasold` nevű állomány abban a mappában, ahová a `progi`-t is elmentettük, s tartalmaz mindent, amit a `morickaviccek` is tartalmazott (hiszen karakterről karakter-

re átmásoltuk azt). Természetesen, lehet számokat, stb-t is átmásolni vagy velük műveleteket végezni, azonban ilyen esetben inkább az `fscanf` és az `fprintf` alkalmazása ajánlott, hiszen nem csupán egy-egy karaktert akarunk „rakosgatni”[1].

A fent említett helyzetben valami ilyesmi lenne a ciklusban:

```
while((!feof(f1))
{
 fscanf(f1, "%d", &szam);
 fprintf(f2, "%d\t", szam);
}
```

... vagy bármi más – a beolvasott adatokkal elvégzett – művelet ...

Végül – zárandó e fejezetet – megnézzük, miként festene az előző fejezet ”zh-szerű” feladatsorának megoldása, ha a felhasználandó / feldolgozandó állományok neveit – kulturált módon – a program indításakor adnánk meg a parancsorban, beírván például, hogy:
`./progi roxfort verseny progi.c zh`
A feladat most is ugyanaz, mint az előző fejezetben, így nem írom le újra. (Egyetlen különbség, hogy parancsorból ”jönnek” a fájlnevek. A program indításakor nyilván csak az első kettő paraméter (és a `progi.c`) által megnevezett állománynak kell léteznie.) Magát a kódot viszont teljes egészében megjelenítem, hogy ne kelljen visszalapozgatni folyton, s egyben látható legyen az egész. A `scanf` / `fscanf` beolvasásnál pedig azt a változatot használom, ami Win alatt is működik. Nosza:

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

typedef struct kupa{char n[50]; int e; int p;}kupa;

int foglal(kupa **v, char *fajlnev, int *m);     /*1. f.*/
int beolvas(kupa *v, char *fajlnev, int m);      /*1. f.*/
void konzolra(kupa *v, int m);                   /*1. f.*/
int versenyeztet(kupa *v, int m, char *fajlnev); /*2. f.*/
```

[1]Lehet persze másképp is boldogulni, hiszen, ha pl. maradunk az `k=fgetc(f1)` - es módszernél, akkor az adott karakter ASCII kódját csípjük ki, s ha az egy számot ábrázol, akkor – ahhoz, hogy azzal a számmal műveleteket tudjunk végezni – előbb számmá kell alakítanunk azt, vagyis meg kell keresnünk az kinyert ASCII érték által kódolt számjegyet, s azt, mint értéket megadni egy változónak. Persze, bonyolódik a helyzet, ha több jegyből áll a szám, mert akkor előbb minden számjegy esetében meg kell állapítani, hogy hol áll ”helyiértékileg”, s az adott értékkel meg kell szorozni, majd a kapott szorzatokat össze kell adni, hogy megkapjuk magát a szám értékét. .. és a törtekről még nem is beszéltünk(!).

```
int maximum(kupa *v, int m, char *fajlnev);        /*3. f.*/
int megjelenit(char *fajlnev);                     /*4. f.*/
int kereso(kupa *v, int m, char *fajlnev);         /*5. f.*/
int sztringhasonlit(char *a, char *b);             /*5. f.*/
int zhkiir(kupa *v, int m, char *ebbol, char *ebbe);/*6.*/

int main(int argc, char *argv[]){
 int meret=0, i=1; kupa *verseny=NULL;

 if(foglal(&verseny, argv[i], &meret))     /*1. feladat*/
 {
  printf("Sikertelen␣fajlnyitas␣vagy␣tarfoglalas.\n");
  return(0);
 }

 if(beolvas(verseny, argv[i++], meret))     /*1. feladat*/
 {
  printf("Sikertelen␣fajlnyitas.\n"); return(0);
 }

 konzolra(verseny, meret);    /*1. feladat*/

 if(versenyeztet(verseny, meret, argv[i]))  /*2. feladat*/
 {
  printf("Sikertelen␣fajlnyitas.\n");return(0);
 }

 if(maximum(verseny, meret, argv[i]))    /*3. feladat*/
 {
  printf("\nSikertelen␣fajlnyitas.\n");return(0);
 }

 if(kereso(verseny, meret, argv[i++]))  /*5. feladat*/
 {
  printf("\nSikertelen␣fajlnyitas.\n");return(0);
 }

 if(zhkiir(verseny, meret, argv[i], argv[i+1])) /*6. feladat*/
 {
  printf("\nSikertelen␣fajlnyitas.\n");return(0);
 }
 free(verseny); return(0);
}
```

```
int foglal(kupa **v, char *fajlnev, int *m)/* 1. feladat */
{
 FILE *f=NULL; char c;

 if(!(f=fopen(fajlnev, "r"))){ return(1); }

 while((c=fgetc(f))!=EOF){if(c=='\n'){(*m)++;}}

 if(!(*v=(kupa *)malloc(*m*sizeof(kupa))))
 {
  return(1);
 }

 fclose(f); return(0);
}


int beolvas(kupa *v, char *fajlnev, int m)/* 1. feladat */
{
 FILE *f=NULL; int i;

 if(!(f=fopen(fajlnev, "r"))){return(1);}

 for(i=0;i<m;i++)
 {
  fscanf(f, "%49s%d", v[i].n, &v[i].e);
 }

 fclose(f);
 return(0);
}


void konzolra(kupa *v, int m)    /* 1. feladat */
{
 int i;
 for(i=0;i<m;i++)
 {
  printf("\nnev: %s\tevfolyam: %d\n", v[i].n, v[i].e);
 }
}
```

```
int versenyeztet(kupa *v, int m, char *fajlnev) /*2. f.*/
{
 FILE *f=NULL; int i; srand(time(NULL));
 if(!(f=fopen(fajlnev, "w"))){return(1);}
 printf("\nA verseny eredmenye:\n");
 fprintf(f, "\nA verseny eredmenye:\n");
 for(i=0;i<m;i++){
  v[i].p=rand();
  printf("\nnev: %s\t", v[i].n);
  printf("evf.: %d\t\t", v[i].e);
  printf("pontszam: %d\n", v[i].p);
  fprintf(f, "\nnev: %s\t", v[i].n);
  fprintf(f, "evf.: %d\t\t", v[i].e);
  fprintf(f, "pontszam: %d\n", v[i].p);
 }
 fflush(f); fclose(f); return(0);
}

int maximum(kupa *v, int m, char *fajlnev)  /*3. feladat*/
{
 FILE *f=NULL; int i, max=v[0].p;
 for(i=1;i<m;i++){ if(max<v[i].p){ max=v[i].p; } }
 if(!(f=fopen(fajlnev, "a"))){return(1);}
 fprintf(f, "\nA legtobb pontot -");
 fprintf(f, " (%d-t) - kapta(k):\n", max);
 printf("\nA legtobb pontot -");
 printf(" (%d-t) - kapta(k):\n", max);

 for(i=0;i<m;i++)
 {
  if(max==v[i].p)
  {
   fprintf(f, "%d.", v[i].e);
   fprintf(f, " evfolyambol, %s\n\n", v[i].n);
   printf("%d.", v[i].e);
   printf(" evfolyambol, %s\n\n", v[i].n);
  }
 }
 fflush(f); fclose(f);
 return (megjelenit(fajlnev));/*4. feladathoz*/
}
```

```
int megjelenit(char *fajlnev)/*4. feladat*/
{
 FILE *f=NULL; char c;
 printf("\n\nA fajl tartalma: ");
 if(!(f=fopen(fajlnev, "r")))
 {
  printf("nem jelenitheto meg.\n\n"); return(1);
 }
 printf("\n\n********************\n");
 while((c=fgetc(f))!=EOF){putchar(c);}
 printf("********************\n");
 fclose(f); return(0);
}


int kereso(kupa *v, int m, char *fajlnev)  /*5. feladat*/
{
 FILE *f=NULL; int i=0; char kit[50], van_e=0;
 if(!(f=fopen(fajlnev, "a"))){return(1);}
 printf("\nKit keresunk? "); scanf("%49s", kit);
 for(i=0;i<m;i++)
 {
  if(sztringhasonlit(kit, v[i].n))
  {
   van_e++;
   fprintf(f, "\nA keresett nev: %s,\t", v[i].n);
   fprintf(f, "evf.: %d,\t", v[i].e);
   fprintf(f, "pont: %d\n", v[i].p );
   printf("\nA keresett nev: %s,\t", v[i].n);
   printf("evf.: %d,\t", v[i].e);
   printf("pont: %d\n", v[i].p );
  }
 }
 if(!van_e)
 {
  printf("Nincs ilyen nev a listan.\n");
 }
 fflush(f); fclose(f);
 return(0);
}
```

```
int sztringhasonlit(char *a, char *b)/*5. feladat*/
{
 int ahossz=0, bhossz=0, i=0;
 while(a[ahossz]!=0){ahossz++;}
 while(b[bhossz]!=0){bhossz++;}
 if(ahossz==bhossz)
 {
  while(i<ahossz){if(a[i]!=b[i]){break;}i++;}
  if(i==ahossz){return(1);}
 }
 return(0);
}


int zhkiir(kupa *v, int m, char *ebbol, char *ebbe) /*6.*/
{
 FILE *ezt=NULL, *ide=NULL; char c;

 if(!(ezt=fopen(ebbol, "r"))){return(1);}

 if(!(ide=fopen(ebbe, "w"))){return(1);}

 while((c=fgetc(ezt))!=EOF)
 {
  fputc(c, ide);
 }

 /*vagy:*/
 /*while(!feof(ezt))
  {
   fscanf(ezt, "%c", &c);
   fprintf(ide, "%c", c);
  }*/

 fflush(ide);
 fclose(ide);
 fclose(ezt);

 return(0);
}
```

Természetesen, ezenkívül még sok egyéb távlatot is megnyit előttünk a `main`

paraméterezésének okos felhasználása. Kinek - kinek csak a fantáziája szabhat határt az elképzelhető lehetőségek ügyes kiaknázásában.

Persze, felvetődhet valakiben a kérdés: minek ez a parancssori paraméteresdi, miért nem jó a "`scanf` / `getchar`-os" beolvasás? Nos, Zidarics tanár úr – a velünk párhuzamosan futó "villamos progi1" kurzuson – szintén szembesült e kérdéssel (többször is), így válaszát ki is tette a kurzus honlapjára[2]. Alább, tőle idézek:

> "A jó program alaposan tesztelt[3]. A teszteknek bármikor megismételhetőknek kell lenniük. Ha a program csak a `scanf` - `getchar` párost használja, nagyon nehéz mindenre kiterjedő tesztet készíteni hozzá. Az argumentumokkal történő paraméterátadás sokrétű tesztek gyártását teszi lehetővé. Ha csak a `scanf` - `getchar` párost ismeri[4], akkor az argumentummal történő átadás esetén zavarba jöhet, ha nem ismeri."

E kurzusban, mindenképp illik szót ejtenünk még az *önmagukra hivatkozó adatszerkezetekről* (pl.: láncolt listákról) is. Erről szól a következő fejezet.

[2]Irgalmatlanul hosszú a link, így inkább azt írom ide, amit a *google*-be kell írnunk, ha az első találatok közt szeretnénk látni a hivatkozott honlapot: *Zidarics Zoltán Programozás I.*

[3]lábjegyzet tőlem: No, igen.. a tesztvezérelt fejlesztés megérne egy külön misét ... sajnos ez is áldozatul esik a "tanidő" rövidségének, de mint ahogy elmondtam a kurzus elején: tűnjék bármilyen nehéznek az anyag, ez mind csak alapozás!

[4]lábjegyzet tőlem: a hallgató

13. fejezet

Önmagukra hivatkozó adatszerkezetek

Ilyen típusú adatszerkezeteket úgy hozhatunk létre például, hogy egy általunk alkotott struktúra egyik mezőjének egy olyan mutatót adunk meg, mely az adott mutatót tartalmazó struktúra típusával egyező típusú struktúrát képes kijelölni a memóriában.

Ezekkel a mutatókkal lánccá (esetünkben, a lentiekben bemutatott, egyszeresen láncolt listává) fűzhetjük össze az egyes – azonos struktúratípusúnak deklarált – objektumokat.

Nézzük, hogy nézne ki a Roxfort boszi-képzőjének versenyzőit nyilvántartó verem, ha így tárolnánk a vonatkozó adatokat!

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

typedef struct kupa{
                    char nev[32]; int evf; int p;
                    struct kupa *tovabbi;}kupa;

kupa *elso=NULL;

int betesz(char *n, int e, int p);
int kivesz();
```

```
int main()
{
 char n[32], c; int e, p;

 printf("\nVerem␣feltoltese:\n");

 do
 {
  printf("\nNev:␣"); scanf("%31s", n);
  printf("Evfolyam:␣"); scanf("%d", &e);
  printf("Pontszam:␣"); scanf("%d", &p);

  if(betesz(n, e, p))
  {
   printf("Sikertelen␣memoriafoglalas."); return(0);
  }

  printf("\nVan␣meg␣hallgato?␣(i/n)␣");
  while((c=getchar())=='\n'); system("clear");

 }while(c=='i');

 printf("\n\nVerem␣tartalmanak␣kiirasa␣es␣uritese:\n\n");

 while(!kivesz());

 return(0);
}


int betesz(char *n, int e, int p)
{
 kupa *ujelem; int i=0;
 if(!(ujelem=(kupa *)malloc(sizeof(kupa)))){return(1);}
 strcpy(ujelem->nev, n);
 ujelem->evf=e;
 ujelem->p=p;
 ujelem->tovabbi=elso;

 elso=ujelem;
 return(0);
}
```

```
int kivesz()
{
 kupa *torolni;
 if(elso==NULL){return(1);}

 torolni=elso;
 printf("\nNev: %s\t", elso->nev);
 printf("evf.: %d\t", elso->evf);
 printf("pontszam: %d\n", elso->p);

 elso=elso->tovabbi;
 free(torolni);
 return(0);
}
```

A program működésének megértését megkönnyítheti a „Problémaosztályok, algoritmusok” tantárgy keretében, a veremről, mint adatszerkezetről tanultak felelevenítése!

Ami fontos, hogy a struktúrát jelölő mutató készítése folyamán, a

```
typedef struct kupa{
                    char nev[32]; int evf; int p;
                    struct kupa *tovabbi;}kupa;
```

blokkban még nem használhatjuk a `kupa` nevet a mutató által jelölt típus megadásakor, csak `struct kupa`-t!

Futtatáskor láthatjuk, hogy „visszafelé”, azaz épp az általunk beírt sorrend szerint az utolsótól az első felé haladva jelennek meg a verem által tárolt elemek.

Azért van ez így, mert mind az írás, mind a kiolvasás a lánc(olt lista) egyazon végén történik (vagy ha – fejünket elfordítva – „függőleges” láncként tekintünk a szerkezetre akkor: a verem tetején). Így, amit utoljára tettünk be, azt vesszük ki először. Innen származik az ilyen jellegű folyamat neve, jelesül: **LIFO** (Last In First Out)

Nézzük, hogyan is működik a fenti kis programunk!

Láthatjuk, hogy a program indításakor, amikor még üres a verem (tulajdonképpen, ekkor még nem is létezik a verem), addig az első (s egyben felső) elemet jelölő mutató – az `elso` – értéke NULL. Mint azt látni fogjuk, ez a mutató min-

dig a leg(f)első elemét jelöli majd a veremnek. Ez a dolga. Nyilván, egy ilyen szerepben, láthatónak kell lennie a vermet kezelő összes függvény számára, *ezért globális* változóként kell deklarálnunk.

A `betesz()` függvény, a hívásakor megkapja a verem új elemének tartalmát (leszámítva a lista következő elemét jelölő mutatót), mely paraméterek értékei az általunk megkoreografált struktúra mezőibe kerülnek majd. Igen ám, de ilyenkor még nincs hova elhelyezni a kapott paraméterek által hordozott értékeket, ezért először a `malloc()` függvény hívása által helyet kell foglalni a memóriában a verem (vagy egyszeresen láncolt lista) új elemének.

A lefoglalt hely címét pedig, mint az látható, az `ujelem` nevű mutató kapja meg. (Vagy, ha tetszik, az `ujelem` mutatót ráfordítjuk a verem új elemének a helyére.) Most, hogy van már hova beírni a kapott paramétereket, a `strcpy(ujelem->nev, n);` az `ujelem->evf=e;` és az `ujelem->p=p;` utasításokkal értéket adunk a verem új elemének (illetve az új elem mezőinek), ami egy `kupa` típusú struktúra (legalábbis a lefoglalt hely mérete megegyezik a `kupa` típus helyigényével).

Vegyük észre, hogy nem pontokkal, hanem nyilakkal hivatkoztunk a struktúra mezőire, aminek az az oka, hogy itt egy pointerrel dolgozunk, ami egy struktúrát jelöl, s mint azt az eddig tanultakból már tudhatjuk, ha egy függvénynek egy struktúrát, egy azt jelölő pointerrel adunk át, akkor a pontokat nyilak kell, hogy felváltsák!

A fenti esetben ugyan nem kapott a `betesz()` függvény semmiféle struktúrajelölő pointert a paraméterei közé, de ettől még – bizonyos szempontból – a helyzet ugyanaz, mint, ha kapott volna, ugyanis, ahogy abban az esetben sem egy struktúrával, hanem egy azt jelölő pointerrel dolgozna, úgy most is csak azt teszi. Ezért kell hát e helyütt is „nyilaznunk”.

A verem épp létrehozott, s feltöltött elemével kapcsolatban már csak az van hátra, hogy „beláncoljuk” a listába. Ezt persze könnyű mondani, nekünk viszont még ki kell találnunk, hogy miként is kellene elérnünk ezt!

A megoldás pofonegyszerű, mindössze azt kell csak szem előtt tartanunk, hogy az `elso` nevű mutató hivatott jelölni a verem leg(f)első elemét, s meg is kell maradnia ennél a szerepnél. Ezt úgy oldjuk meg, hogy az `elso=ujelem;` utasítással „ráfordítjuk” azt a verem legújabb elemére. Mielőtt azonban ezt megtennénk, az `elso` régi értékét (ami NULL volt) beleírjuk a legújabb elem, következő elemet jelölő mutatójába az `ujelem->tovabbi=elso;` utasítás kiadása által. Ha most belegondolunk az algoritmus működésébe, s végigzongorázzuk, mit jelente-

nek a fenti néhány sorban leírtak, akkor könnyen rájöhetünk, hogy ez az egész „mutatósdi" kettős célt szolgál:

- egyfelől, világos, hogy amikor már van legalább egy eleme a veremnek (tehát már legalább a másodikat „töltjük") , akkor az történik, hogy az `ujelem->tovabbi=elso;` utasítással az új elem, következő elemet jelölő mutatóját ráfordítjuk arra az elemre, ami addig legfelül volt a veremben (amit addig az `elso` jelölt), majd az `elso=ujelem;` utasítással az `elso`-t ráfordítjuk a legfrissebb elemre, így a verem új elemét, mintegy beszendvicseljük az `elso` nevű mutató és az addig legfelül/legelöl helyet foglaló elem „közé", magyarán a lefrissebb elem kerül mindig a láncolt lista elejére, a verem tetejére.

- másfelől, a lista legelsőként feltöltött („legrégebbi") elemének a következő elemet jelölő pointere (a `tovabbi` nevű) NULL értékű marad, hiszen akkor kapott értéket az `ujelem->tovabbi=elso;` utasítás által, amikor az `elso` értéke még NULL volt, s e pointer értékét a verem további töltése során már nem fogjuk megbolygatni, ahogy persze a többi elem tovabbi mutatójáét sem (kivéve persze a veremhez való „hozzáfűzésüket), de itt most nem ez a lényeg, hanem az, hogy a verem legrégebbi (legalsó) elemének `tovabbi` nevű mutatója mindig NULL értékű. Tulajdonképpen nem más ez, mint egy „csomó" a lánc végén, ami jelzi, hogy meddig „nyújtózkodhatunk" a lista bejárásakor, továbbá, mint azt majd látni fogjuk, ennek ségítségével tudhatjuk meg, hogy üres-e már/még a verem, hiszen, ha az `elso` nevű pointer értéke NULL, akkor nincs elem a veremben (de még csak verem sincs :)).

A `betesz()` függvény tehát teszi a dolgát, ameddig tennie kell (ameddig újrafuttatjuk a hívását tartalmazó hátultesztelő ciklust).

Miután befejeztük a verem „mélyítését", a program kiírja annak elemeit, s egyben ki is üríti a vermet. T ermészetesen, nem muszáj ilyenkor „kipucolni" a vermet, csupán azért tesszük meg, hogy lássuk, mi a módja.

Tényleg! Mi is a módja? Nézzük csak meg közelebbről a folyamatot! Tehát, ott tartottunk, hogy – nem kívánván több versenyzőt nyilvántartásba venni – kiléptünk a `betesz()` függvényt hívogató `do{}while();` ciklusból, csak, hogy legott belevessük magunkat egy elöltesztelő ciklusba, mely a következőképpen fest: `while(!kivesz());` . A ciklusnak nincsen magja, mivel mindössze annyi a feladata, hogy addig hívja újra és újra a `kivesz()` függvényt, amíg az0-t ad vissza (ezúton is jelezvén, hogy nincs hiba a működésében). Amennyiben a

`kivesz()` függvény visszatérési értéke 1, a vezérlés kilép a ciklusból, ami azt jelenti, hogy többet nem hívja meg a függvényt. Lássuk, mitől is függ, melyik értéket adja vissza a függvény, s azt is, hogy mit jelentenek ezek az értékek!

Először is: a `kivesz()` függvénynek az a dolga, hogy „kivegyen” egy elemet a veremből. Mielőtt azonban belekezdenénk a `kivesz()` működésének elemzésébe, érdemes felfigyelnünk rá, hogy mennyire hasznos az egyszeresen láncolt listára veremként tekinteni, ugyanis: amikor kiolvassuk, s egyben (mint azt majd tesszük ezúttal) ki is „ürítjük” a listát, ez a kép (mármint az, hogy ez egy verem) nagyban egyszerűsíti a folyamat megértését.

Hiszen, maradván ennél a képnél, mit is kell tennie a `kivesz()` függvénynek? Ki kell vennie a láncolt listánk egyik elemét, mely feladat, amennyiben veremként tekintünk a listára, nyilvánvalóan ekvivalens a veremből való kiemeléssel, amivel kapcsolatban világos, hogy csakis a verem tetején (a lista első eleménél) történhet. Mindig. Akárhányadszorra is végezzzük el a „kiemelést”, mindig csak a verem tetejéről „csipegethetünk”, hisz csak ahhoz férünk hozzá.

Ebből kifolyólag, nincs más dolgunk hát, mint a `kivesz()` függvényt újra és újra ráuszítani a verem tetején lévő elemre. Ehhez persze meg kell neki mutatni, hol találja meg azt! Erre szolgál a `torolni=elso;` parancs, hiszen, mint azt tudjuk, mindig az `elso` jelöli a verem tetejét. Ezután kiíratjuk a képernyőre a listából kiiktatandó elem mezőinek tartalmát a `printf()` függvénnyel, majd a verem tetejét jelölő `elso` pointer „nyilát” átirányítjuk a törölni kívánt elem alatti elemre az `elso=elso->tovabbi;` utasítás kiadása által. Ezzel elérjük, hogy mostantól az eddig első elemnek tekintett elem *alatti* elem legyen a verem teteje (hiszen ráállítottuk a verem tetejét jelölni hivatott `elso` nevű mutatót). Vegyük észre azonban, hogy ugyan eme művelettel kiiktattunk a listából egy elemet, a rendelkezésünkre álló memória méretét mégsem növeltük meg. Ugyanis a veremből száműzött elem – noha nem tagja már a listának – a számára a `malloc()` által hajdan biztosított memóriaterületet továbbra is lefoglalva tartja. Következésképp, ha nem akarunk félmunkát végezni, ki kell adnunk a `free(torolni);` parancsot is!

Miután mindezeken a `kivesz()` függvény túlesett, otthagyja a 0-t a `while()` feltételében, amit a `!` jel „igazzá tesz”, így újra meghívásra kerül a függvény, s újra csak kezelésbe veszi a verem tetején talált elemet, egészen addig, amíg azzal nem szembesül, hogy a verem tetejét jelölő mutató értéke NULL. (Ez ugye azután történik, miután az utolsó/legalsó elemhez érvén, az azt jelölő `elso` nevű mutató által kijelölt elemhez tartozó `tovabbi` értékével teszi egyenlővé az `elso`-t az `elso=elso->tovabbi;` utasítás által, mivel ekkor az `elso->tovabbi` értéke már NULL.)

Tehát, amennyiben a `kivesz()` függvény azt „látja”, hogy az `elso` mutato értéke NULL, hibajelzést (egy 1-es értéket) küld a `while()`-ba, ezáltal jelezvén, hogy a verem már üres, nem kell hát tovább üríteni. Ezt az 1-est a `!` jel teszi hamissá, miáltal a vezérlés kilép a ciklusból, majd kisvártatva a program is befejezi a működését.

Természetesen felmerülhet a kérdés, hogy mire jó az életünk – láncolt listák általi – bonyolítása? Miért ne maradhatnánk meg a jó kis struktúratömbjeinknél, melyek a memóriába sorfolytonosan[1] kerültek elmentésre? Nos, esetenként épp ezért. Mármint, a sorfolytonos memóriafoglalás miatt. Csak, hogy értsük: képzeljük el, miként szúrnánk be egy új elemet egy „normál” struktúratömbbe! Nehezítsük a dolgot, s találjuk ki, miként tehetnénk ezt meg úgy, hogy a tömb elejéhez fűzzük hozzá az új elemet! Fokozzuk az élvezeteket, s gondoljunk ki eljárást arra, hogy teszem azt, a második és a harmadik elem közé kerüljön az új elem!

Persze, hogy megoldható a dolog, na de épp csak a szemünket nem üti ki az, hogy mennyivel egyszerűbben kivitelezhetők az efféle manőverek olyan listák esetében, melyek elemei nem kell, hogy egymás mellett legyenek a memóriában, mivel azok az egyik mezőjük által mutatják a következő elemüket. Ilyenkor egyszerűen csak új értéket kell adnunk egy-egy pointernek, s kész is a beszúrás. A fenti esetben a második elem pointerét ráállítjuk a beszúrni szándékozott elemre, s ennek az új elemnek a mutatóját pedig arra, amelyik addig a harmadik volt a listában.

Vannak persze kétszeresen láncolt listák is, ahol az egyes elemek két pointert is tartalmaznak, melyek közül egyik az előző, míg a másik a következő elemre mutat. Ismeretesek továbbá cirkulárisan láncolt listák is, de ezek tárgyalásától eltekintünk, mert az már túlmutat e szemeszteren. Megemlítjük azonban, hogy az önhivatkozó adatszerkezetek létezését nem csupán a beszúrás könnyebbé tétele indokolja, hanem segítségükkel olyan szerfelett érdekes és hasznos szerkezetek valósíthatók meg, mint például a fák is.

[1] A tömbelemek fizikailag is egymás melletti helyen vannak a memóriában, folytonosan kitöltve a tömb számára lefoglalt blokkot.

14. fejezet

Egy feladat, s néhány érdekesség

Kép a **Kapcsolat** című filmből

E fejezet tartalmát, kizárólag azoknak a figyelmébe ajánlanám, akik nem riadnak vissza egy kis szellemi kihívástól, mi több, igénylik is azt. Természetesen a többiek is elolvashatják, ártani nem fog.

Íme az alapvetés: Egy titkos kormányügynökség megkeres minket egy munkával. Miután aláírtuk a munkaszerződés titoktartási nyilatkozatot tartalmazó részét (erre amúgy a hétköznapokban is sor kerülne egy „mezei” kutató-fejlesztő cégnél is), beavatnak minket a részletekbe:

Az egyik új részecskegyorsítóban, miután elérték az 100 TeV-es energiaküszöböt (a CERN LHC-je a jelenlegi csúcstartó, a maga 14 TeV-es szintjével) mikroszkopikus féregjáratok megnyílására utaló jeleket detektáltak. A megfigyelt járatok egyikét aztán egy új technológia segítségével – mely, a Casimir–effektusra ala-

pozva, megváltoztatván az elektromágneses tér zérusponti energiájának homogenitását, „belenyúl” a tér szerkezetébe – sikerült stabilizálni és makroszkopikus méretűre „felfújni”.

Ezt követően a projektben dolgozó kutatók arra lettek figyelmesek, hogy a féreglyukból elektromágneses jelcsomagok érkeznek, melyek mindegyike azonos frekvenciájú, ám különböző mennyiségű fotont tartalmaz.

Felmerült hát a gyanú, hogy esetleg intelligens forrás állhat a jelsorozat mögött.

A mi feladatunk az, hogy megalkossunk egy programot, mely képes eldönteni azt, hogy intelligens forrásról lehet-e szó vagy csupán véletlenszerű fehérzaj okozza a tapasztalt jelenséget...

Szerencsére, fizikából és az információelmélet tantárgy keretei között is tanultunk az entrópiáról, melyet most segítségül hívhatunk a kérdés eldöntésére.

Maga az entrópia, a fizikában csupán egy – monoton növő – állapotfüggvény melynek értéke annál nagyobb, minél rendezetlenebb a rendszer, melyet jellemez. Tehát – konyhanyelven – mondhatnánk azt is, hogy az entrópia a rendezetlenség mértéke. De mi köze ennek az információelmélethez?

Röviden, annyi, hogy minél rendezetlenebb egy rendszer (pl. minél nagyobb a „kupi” a szobánkban), annál kevesebb információt tudunk „kinyerni” belőle.

Vagyis – végképp nagyon pongyolán fogalmazva – az entrópia növekedésével csökken a kinyerhető információ mértéke (nem magának a teljes információnak a mennyisége, hanem csak a kinyerhető információé).

Ebből az következik, hogy ha egy jelsorozatnak ki tudjuk számolni a nemsokára említésre kerülő Shannon- entrópiáját, s emellett képesek vagyunk megmondani azt is, hogy mennyi egy adott jelszámhoz (esetünkben a kapott fotoncsomagok számához) tartozó, maximálisan kapható Shannon-entrópia értéke, akkor kezünkben a megoldás! Miért is? Azért, mert egy intelligens forrás esetében, ha az kommunikálni igyekszik, akkor az általa használt bármilyen jelrendszer Shannon-entrópiájának valahova a 0 és a maximális érték által határolt tartomány derekára kell esnie.

Megjegyzés: Egy csomagban amúgy rém nehéz lenne megállapítani a fotonszámot, mert ha a fotonok Poisson- eloszlásban jönnek (pl. lézerek esetében), akkor egy foton detektálásakor nem zárhatjuk ki egy másik foton jelenlétét sem, na és persze a detektorok sem tökéletesek.

Tehát: van egyszer a fizikában megismert entrópia és van a kinyerhető információ, mely két mennyiség „kétfelé tart”, vagyis egymással fordított arányban

állnak. Az információelméletben azonban bevezetésre kerül a Shannon-entrópia, mely mennyiség viszont, az eddig tárgyalt entrópiával szemben „együtt" halad a kinyerhető információ mennyiségével azaz, arányos azzal. A **Shannon-entrópia** ugyanis nem más, mint egy adott jelsorozat egyetlen jelére eső átlagos információ, bit-ben megadva.

Miről is van itt szó egyáltalán?

Arról, hogy meg ha meg tudjuk mondani egy adott jelsorozat egyetlen jelére eső átlagos információt (a Shannon- entrópiát), akkor – anélkül, hogy akár egy kukkot is értenénk az adott „nyelv"-ből – el tudjuk dönteni, hogy az valóban egy kommunikációs jelrendszer-e vagy sem.

Ugyanis ha például egy jelsorozat csakis egyfajta jelet tartalmaz, akkor ott egy jel információértéke nyilván 0 bit. Ha viszont minden jel különbözik és csak egyetlen egyszer fordul elő a sorozatban, akkor megint csak bajban vagyunk, mivel itt aztán látszólag tényleg magas az információérték, ám nem valószínű, hogy értelmes jelrendszerről van szó, hiszen nincsenek ismétlődő jelek benne.

Olvassuk csak vissza az eddigi szöveget, s láthatjuk, hogy az egyes betűk, szavak, stb, jó néhányszor szerepelnek benne, s ami fontos, van amelyik gyakrabban, van amelyik kevésbé gyakrabban (már persze, ha elfogadjuk, hogy ez az egész egy értelmes rendszert alkot.. :))

Maga a Shannon-entrópia (jele, rendszerint: H) az alábbi módon számítható:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i,$$

ahol a p_i-k, az egyes elemek előfordulási valószínűségei (relatív gyakoriságai), mely értékeket úgy kaphatjuk meg, hogy megszámoljuk, hogy az adott fajtájú/értékű elem hányszor szerepel a jelsorozatban, s a kapott értéket elosztjuk a jelsorozat elemszámával, n-nel. Az $(1, 2, 3, 4, 2, 3, 5, 7, 9, 9)$ számsorozatban például a 3-as szám relatív gyakorisága: $1/5$.

Nézzük, hogy például a *„Szeretem e szemesztert!"* mondatra alkalmazva a fenti formulát, mit kapunk H- ra!

Az jelsorozat 23 elemből áll. Az egyes elemek relatív gyakorisága a következő: $p(s) = 3/23$, $p(z) = 3/23$, $p(e) = 7/23$, $p(r) = 2/23$, $p(t) = 3/23$, $p(m) = 2/23$, $p(space) = 2/23$, $p(!) = 1/23$.

Ezekből azt kapjuk, hogy – ha jól számoltam – erre a jelsorozatra $H = 2.788077$. Ha csupa egyforma betűből állna a jelsorozat, akkor ez az érték 0 lenne. Ha pedig mindegyik betű különbözne (pl.: *„aysxdcfvgbhnjmklépqower”*), akkor egy ilyen hosszúságú betűsorozatra $H(max) = 4.523562$ lenne, ami szép és jó (meg persze nagyobb is), épp csak nem egy intelligens jelsorozat, hanem inkább egy „véletlenszerű” eloszlás jellemzője.

Látható, hogy – persze csak kellően nagy mintavétel esetén – bármilyen jelsorozatról eldönthető, hogy „beszél”-e hozzánk valaki/valami vagy sem, anélkül, hogy dekódolnánk/értenénk az üzenetet, már, ha valóban az[1].

Immáron eleget tudunk ahhoz, hogy elvégezzük a ránk bízott munkát.

Előbb írunk egy demo-változatot a programból, ami az általunk elképzelt algoritmus teszteléséhez jöhet jól, s a következőt teszi.

- Generál egy tetszőlegesen hosszú véletlenszám–sorozatot, amit kiír egy szöveges fájlba.
- Ezt a fájlt beolvassa, s az egyes elemek számértékét fotonszámként kezelve, minden különböző elemnek (különböző értékű elemnek) kiszámítja a relatív gyakoriságát.
- Ebből kiszámolja a jelsorozat Shannon-entrópiáját, amit kiír a fájlba és a konzolra is.

Nyilván a generált véletlenszámok egy fix sorozatból jönnek, tehát álvéletlenek, de arra azért ügyeltek a C nyelv megalkotói, hogy egy-egy ilyen sorozat valóban véletlenszerűnek tűnjön, így ha jól működik a programunk, akkor az általa számolt Shannon-entrópiának az adott hosszúságú sorozatot jellemző maximumérték közelében kell lennie!

A program működését – nem a leghatékonyabban ugyan, de – úgy oldhatjuk meg, hogy miután beolvastuk a számokat egy tömbbe, előbb növekvő sorrendbe rendezzük azokat.

Ezután – azt követően, hogy megszámoltuk őket – a rendezett tömbből egy másikba kigyűjtjük a különböző elemek darabszámát.

(Azért jó a rendezett tömb, mert egyfelől könnyebb így megszámolni a különböző elemek darabszámát, másfelől az értékén keresztül ellenőrizhető, hogy még

[1] Egy valódi eset persze jóval összetettebb, hisz nem csak független betűket, de szavakat, mondatokat, kifejezéseket, stb-t is tartalmazhat.

mindig ugyanannál az elemnél tartunk-e a tömbben vagy pedig már egy új szám jött, s bővítenünk kell azt a tömböt, ahová épp kigyűjtjük a különböző elemek darabszámait.)

Mindezek után, már az új tömböt bejárván – az abban lévő darabszámokból, valamint a teljes jelszámból – könnyedén kiszámolhatjuk a jelsorozat Shannon-entrópiáját.

Íme a program, melynek önálló megértése, illetve megalkotása képezné az említett szellemi kihívás/játék lényegét. Természetesen, mint mindig, megoldható a kérdés más módon is!

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

int jelekkiir(int *m);
int jelekrendez(int m, int **jel);
int jelekesdarabszamok(int m, int *jel);
int szamolkiirentropia(int osszjelszam, int jelfajtaszam, int *jsz);


int main()
{
 int meret, *jelek=NULL;

 if(jelekkiir(&meret))
 {
  printf("Sikertelen␣fajliras.\n"); return 0;
 }

 if(jelekrendez(meret, &jelek))
 {
  printf("Sikertelen␣fajlolvasas␣vagy␣memoriakezeles.\n");
  return 0;
 }

 if(jelekesdarabszamok(meret, jelek))
 {
  printf("Sikertelen␣fajlolvasas␣vagy␣memoriakezeles.\n");
  return 0;
 }

 return 0;
}
```

```
int jelekkiir(int *m)
{
 FILE *f=NULL; int i; srand(time(NULL));
 printf("Mennyi␣lesz?␣"); scanf("%d", m);
 if(!(f=fopen("jelek", "w"))){return 1;}
 for(i=0;i<*m;i++){ fprintf(f, "%d\t", rand()); }
 fflush(f); fclose(f);
 return 0;
}

int jelekrendez(int m, int **jel)
{
 FILE *f=NULL; int i=0, j, min, minindex, s;
 if(!(*jel=(int *)malloc(m*sizeof(int)))){return 1;}
 if(!(f=fopen("jelek","r"))){ return 1; }
 while(!feof(f)){fscanf(f, "%d", &(*jel)[i++]);}
 fclose(f);
 for(i=0;i<m;i++)
 {
  for(j=i;j<m;j++)
  {
   if(j==i || (*jel)[j]<min)
   {
    min=(*jel)[j]; minindex=j;
   }
  }
  if(min<(*jel)[i])
  {
   s=(*jel)[i]; (*jel)[i]=(*jel)[minindex];
   (*jel)[minindex]=s;
  }
 }
 return 0;
}

int jelekesdarabszamok(int m, int *jel)
{
 int i, j; int *jelszamok=NULL;

 for(i=0,j=0;i<m;i++)
 {
  if(i==0)
  {
   if(!(jelszamok=(int *)realloc(jelszamok, ++j*sizeof(int))))
   {
    return 1;
   }
   jelszamok[j-1]=1;
  }
```

```
  else
  {
   if(jel[i-1]==jel[i]){ jelszamok[j-1]++; }
   else
   {
    if(!(jelszamok=(int *)realloc(jelszamok, ++j*sizeof(int))))
    {
     return 1;
    }
    jelszamok[j-1]=1;
    }
   }
  }
  return (szamolkiirentropia(m, j, jelszamok));
}


int szamolkiirentropia(int osszjelszam, int jelfajtaszam, int *jsz)
{
 FILE *f=NULL; int i;
 float ossz=osszjelszam, relativgyakorisag, entropia=0;
 float maxentropia= -log2f(1/ossz);

 for(i=0;i<jelfajtaszam;i++)
 {
  relativgyakorisag=jsz[i]/ossz;
  entropia-=relativgyakorisag*log2f(relativgyakorisag);
 }

 if(!(f=fopen("jelek", "a"))){ return 1; }

 fprintf(f, "\n\nAz␣itt␣lehetseges␣legnagyobb␣");
 fprintf(f, "entropiaertek:␣%f\n", maxentropia);
 fprintf(f, "Ennek␣kozeleben␣csupan␣veletlenszeru␣");
 fprintf(f, "\"feherzajrol\"␣beszelhetunk.\n\n");
 fprintf(f, "0␣entropia␣eseten␣1␣db␣jelet␣vettunk");
 fprintf(f, "␣vagy␣mindig␣ugyanazt.\n\n");
 fprintf(f, "A␣(0,␣%f)␣tartomany", maxentropia);
 fprintf(f, "␣\"derekan\"␣intelligens␣a␣forras.\n\n");
 fprintf(f, "Ezzel␣osszevetve,␣a␣jelsorozat␣forrasanak");
 fprintf(f, "␣entropiaja:␣%f\n", entropia);

 fflush(f); fclose(f);

 printf("\n\nAz␣itt␣lehetseges␣legnagyobb␣");
 printf("entropiaertek:␣%f\n", maxentropia);
 printf("Ennek␣kozeleben␣csupan␣veletlenszeru␣");
 printf("\"feherzajrol\"␣beszelhetunk.\n\n");
 printf("0␣entropia␣eseten␣1␣db␣jelet␣vettunk");
```

```
 printf("␣vagy␣mindig␣ugyanazt.\n\n");
 printf("A␣(0,␣%f)␣tartomany", maxentropia);
 printf("␣\"derekan\"␣intelligens␣a␣forras.\n\n");
 printf("Ezzel␣osszevetve,␣a␣jelsorozat␣forrasanak");
 printf("␣entropiaja:␣%f\n", entropia);
 return 0;
}
```

Megfigyelhetjük, hogy mindig nagyjából $H(max)$-ot kapunk.

Az, hogy egyre nagyobb elemszám esetén egy picit csak, de egyre inkább elmarad a számított érték $H(max)$-tól azért van, mert nagy elemszámok esetén már becsúszhat egy-egy ismétlődés is.

Nos, ez mind szép és jó, de ne érjük be ennyivel!

Próbáljuk meg általánosabbá tenni a kódot! Nyilvánuljon ez meg legfőképp abban, hogy – némileg „földibb" témánál maradván – legyen a program bevethető tetszőleges forrásból (például újságcikkből) kimásolt tartalommal bíró szöveges típusú állomány esetében is! Továbbá, jó lenne, ha már a program indításakor, a parancssorban lehetőségünk nyílna megadni a vizsgálandó állomány azonosítóját! Alább vázolom ennek egy lehetséges módját:

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

int beolvasrendez(int *m, int **jel, char *fajlnev);
int meretez(char *fajlnev, int *m);
int jelekesdarabszamok(int m, int *jel);
void szamolkiirentropia(int osszjelszam, int jelfajtaszam, int *jsz);

int main(int argc, char *argv[])
{
 int meret=0, *jelek=NULL;
 if(beolvasrendez(&meret, &jelek, argv[1]))
 {
  printf("Sikertelen␣fajlolvasas␣vagy␣memoriakezeles.\n");
  return 0;
 }

 if(jelekesdarabszamok(meret, jelek))
 {
  printf("Sikertelen␣fajlolvasas␣vagy␣memoriakezeles.\n");
  return 0;
 }
 return 0;
}
```

```
int beolvasrendez(int *m, int **jel, char *fajlnev)
{
 FILE *f=NULL; int i=0, c, j, min, minindex, s;

 if(meretez(fajlnev, m)){return 1;}
 if(!(f=fopen(fajlnev,"r"))){return 1;}
 if(!(*jel=(int *)malloc(*m*sizeof(int)))){return 1;}
 while((c=fgetc(f))!=EOF){ (*jel)[i++]=c; }
 fclose(f);

 for(i=0;i<*m;i++)
 {
  for(j=i;j<*m;j++)
  {
   if(j==i || (*jel)[j]<min)
   {
    min=(*jel)[j]; minindex=j;
   }
  }
  if(min<(*jel)[i])
  {
   s=(*jel)[i]; (*jel)[i]=(*jel)[minindex];
   (*jel)[minindex]=s;
  }
 }

 return 0;
}



int meretez(char *fajlnev, int *m)
{
 FILE *f;
 if(!(f=fopen(fajlnev, "r"))){return 1;}
 while(fgetc(f)!=EOF){(*m)++;}
 fclose(f);
 return 0;
}



int jelekesdarabszamok(int m, int *jel)
{
 int i, j; int *jelszamok=NULL;
```

```
 for(i=0,j=0;i<m;i++)
 {
  if(i==0)
  {
   if(!(jelszamok=(int *)realloc(jelszamok, ++j*sizeof(int))))
   {
    return 1;
   }
   jelszamok[j-1]=1;
  }
  else
  {
   if(jel[i-1]==jel[i]){jelszamok[j-1]++;}
   else
   {
   if(!(jelszamok=(int *)realloc(jelszamok, ++j*sizeof(int))))
    {
    return 1;
    }
   jelszamok[j-1]=1;
   }
  }
 }
 szamolkiirentropia(m, j, jelszamok);
 return 0;
}


void szamolkiirentropia(int osszjelszam, int jelfajtaszam, int *jsz)
{
 int i; float ossz=osszjelszam, relativgyakorisag, entropia=0;
 float maxentropia= -log2f(1/ossz);

 for(i=0;i<jelfajtaszam;i++)
 {
  relativgyakorisag=jsz[i]/ossz;
  entropia-=relativgyakorisag*log2f(relativgyakorisag);
 }
 printf("\n\nAz itt lehetseges legnagyobb entropiaertek:");
 printf(" %f\n", maxentropia);
 printf("Ennek kozeleben csupan veletlenszeru");
 printf(" \"feherzajrol\" beszelhetunk.\n\n");
 printf("0 entropia eseten 1 db jelet vettunk");
 printf(" vagy mindig ugyanazt.\n\n");
 printf("A (0, %f) tartomany ", maxentropia);
 printf("\"derekan\" intelligens a forras.\n\n");
 printf("Ezzel osszevetve, a jelsorozat forrasanak");
 printf(" entropiaja: %f\n\n", entropia);
}
```

Nyilván, ez a kis kód nem egy általánosan „bevethető” analitikai program, hiszen könnyen becsapható, de szemléltető eszköznek megfelel. Mindenesetre, ez a kis program már képes rá, hogy egy alapos önvizsgálatot tartson. :-) Mire gondolok? Például arra, hogy ha elég vicces és kíváncsi kedvünkben vagyunk, akkor a program indításakor akár azt is kipróbálhatjuk, hogy mit szól hozzá, ha a futtatható állomány nevének beírása után, a saját forrását adjuk meg vizsgálandó állományként. Valahogy így `./futtathatonev forrasnev.c` Az futtatás eredményéből kitűnik, hogy legújabb kis rutinunknak nincsenek önértékelési problémái. :-)

Természetesen bármelyik kódon lehet még finomítani, de ebben az esetben ez kiváltképp érvényes. Ez az ára annak, hogy nem mentem túl a kurzus anyagán.

15. fejezet

Útravaló

Elértünk hát e kis – programozásba való – bevezető kurzusunk végéhez. Zárás gyanánt, ha megengeditek, leírok még néhány, általam fontosnak vélt dolgot a programozásról. Ha valaki mindazt, amit a kurzus során eddig kifejtettem érti és készségszinten tudja alkalmazni is, az megfelelő képzettségi szintet ért el ahhoz, hogy elkezdjen belemélyedni a programozásba. Félreértés ne essék ugyanis, e kurzus tartalma – tűnjön esetleg bármily nehéznek is a tanulás folyamán – mind, mind puszta alapozás. Nem szerepelnek a tantárgy tematikájában (és feltehetőleg az időbe sem férnének bele, csak úgy, ha hiper–sebességgel rohannánk át a tananyagon, ami viszont annak megértését tenné végképp felszínessé) sem a függvénypointerek, sem az unionok, sem a listákkal, bináris fákkal való játszadozás, sem a bitműveletek, sem a program „külvilággal” való kapcsolata, ahogy a nagyobb programok tervezése, az annak folyamán történő adatszerkezet-megválasztás, az alkalmazói programok készítése a specifikálástól a dokumentálásig, programjaink alapos tesztelésének csínja-bínja és még sok egyéb dolog sem.

De még az általam, tananyagba foglalt ismeretanyag is minimalista volt, abban az értelemben, hogy a tematika megválasztása során az a cél vezérelt, hogy minél több eszköz helyett, inkább minél olajozottabb gondolkodási készségre tegyen szert az, aki lelkiismeretesen végigcsinálja a benne szereplő feladatokat.

Igyekeztem a tárgy oktatása során úgy egyensúlyozni, hogy a képzési követelményeknek jól megfelelő hallgató, aki azelőtt sosem programozott, egyfelől szert tegyen egy – relatíve(!) – magas szintű problémakezelő készségre (hiszen a szellemi képességek - akár a testiek – fejleszthetők), másfelől, rendelkezzen annyi, C nyelv terén szerzett ismerettel és C-re való rálátással is, melyekre építve, ha úgy hozza a szükség, már bőven képes lehet önállóan tanulni, illetve eligazodni a bőségesen rendelkezésre álló, C nyelvvel kapcsolatos források világában.

Jó hír gyanánt, érdemes szem előtt tartani azt, hogy ha a C nyelvben eligazodik valaki, akkor több más, C-szerű szintaktikával (nem szemlélettel(!)) bíró nyelv (pl.: C++, Java, PHP, stb) sem okozhat gondot neki.

Most pedig, hogyan tovább?

A következő lépcsőfok az objektumorientált szemlélet keretei között történő programozási gyakorlat elsajátítása lehet, ami egy szép, tartalmas és igen hasznos szellemi „edzés”, melynek viszont már nem a C nyelv, hanem az objektumorientált programozási paradigmát kifejezetten támogató egyéb nyelvek (pl.: Java, C++, stb) keretei között érdemes lezajlania. Az igazság kedvéért persze nem árt megjegyezni, hogy lehet C-ben is objektumorientált módon programozni, s ha valaki efelé szeretne orientálódni, csak bíztatni tudom. Ha viszont nem bánjátok, ebbe a témába már nem csapnék bele.

Annak, aki **tud** OOP-ben programozni (tehát annak, aki valóban tisztában van az OOP-vel), s érdekli, hogy miként lehet azt C-ben megvalósítani, javaslom, hogy „lapozza fel” a `http://stackoverflow.com` vonatkozó „fejezetét”!

Végezetül: ha csak egyvalaki is hasznosnak találta az általam leírtakat, már nem dolgoztam hiába.