Pohl László

C labor jegyzet

BME 2005-2006

Tartalom

Bevezetés

Ez a jegyzet azokat a gyakorlati ismereteket kívánja bemutatni, melyek a C programozás tanulása során felmerülnek, és a tárgy tematikának részei. A jegyzet felépítése eltér a hagyományostól, ezt azért fontos megjegyezni, mert a kezdő programozó talán elborzad, ha belepillant az első fejezetben, ahol a fejlesztőkörnyezet indítását bemutató rész után rögtön egy kétszáz soros programba botlik. Ettől nem kell megijedni, eleinte nem cél ekkora programok készítése, viszont cél, hogy megértsük a működésüket.

A hagyományos programozás oktatásban az adott témakörhöz tartozó példák általában nagyon rövidek. Ezek előnye, hogy könnyű megérteni a működésüket, és könnyű hasonlót létrehozni. Hátrányuk, hogy a gyakorlati életben általában ennél sokkal hosszabb programokat készítünk, a rövid program esetében nem használunk olyan eszközöket, melyek egy hosszabbnál elengedhetetlenek. Ilyen például a megfelelő formátum, magjegyzések, változónevek, függvénynevek használata, a hibatűrő viselkedés (lásd „*Most nem kell hibaellenőrzés, de majd a nagyháziban igen.*”).

Az első fejezetben bemutatott program célja, hogy kipróbáljuk rajta a különféle hibakeresési funkciókat. Az 1.4 részben szerepel a program leírása, mely egyaránt tartalmaz a kezdő és a haladó programozók számára szóló ismereteket. Az első gyakorlat során a cél annyi, hogy nagyjából követni tudjuk a program működését, hogy megértsük, hogy amikor az 1.5 során végiglépkedünk a sorokon, mi miért történik, ezért ekkor csak fussuk át a leírást. Javaslom azonban, hogy a félév második felében, mikor már nagyobb programozói rutinnal rendelkezünk, lapozzunk vissza az 1.4 fejezethez, és olvassuk el ismét.

A továbbiakban vegyesen fognak szerepelni kisebb és nagyobb példaprogramok, a nagyokat mindig csak megérteni kell, nem kell tudni hasonlót írni, bár a félév végére már igen. A hallgató feladata egy úgynevezett „Nagy házi feladat” elkészítése, egy ilyen mintafeladat is szerepel a jegyzetben.

A C nyelvet, hasonlóan a beszélt nyelvekhez, nem lehet elsajátítani önálló tanulás nélkül, ezért mindenképpen oldjunk meg gyakorló feladatokat óráról órára, egyedül, otthon!

A példaprogramokat nem kell begépelni. A jegyzet elektronikus formában a http://www.eet.bme.hu/~pohl/ oldalon megtalálható, a pdf-ből ki lehet másolni a szöveget, ha az Adobe Reader felső eszköztárán a Select Text-et választjuk, de ha minden igaz, akkor a példaprogramok zip-elve is letölthetők ugyanonnan, így még a formázás is megmarad. A jegyzettel kapcsolatos kritikákat, hibalistákat a pohl@eet.bme.hu címre várom.

1. Integrált fejlesztőkörnyezet

A kényelmes és gyors munka érdekében programfejlesztésre lehetőség szerint olyan szoftvereket használunk, mely egybeépítve tartalmazza a programkód szerkesztésére alkalmas szövegszerkesztőt, a programmodulokból gépi kódot létrehozó fordítót (compiler), a gépi kód modulokból futtatható programot létrehozó linkert, és a programhibák felderítését megkönnyítő debugger rendszert.

Ebben a fejezetben két ilyen környezettel fogunk megismerkedni. Az első még a ’90-es évek elején készült Borland C++, melyet egyszerű kezelhetősége miatt használnak manapság is az oktatásban. Hátránya, hogy DOS operációs rendszerhez készült, emiatt a futtatható program által használt memóriaméret igen alacsony, mindössze néhány száz kilobájt, de például egy tömb, vagy bármely adatstruktúra mérete nem haladhatja meg a 64 kB-t. További részletekért, lásd [1].

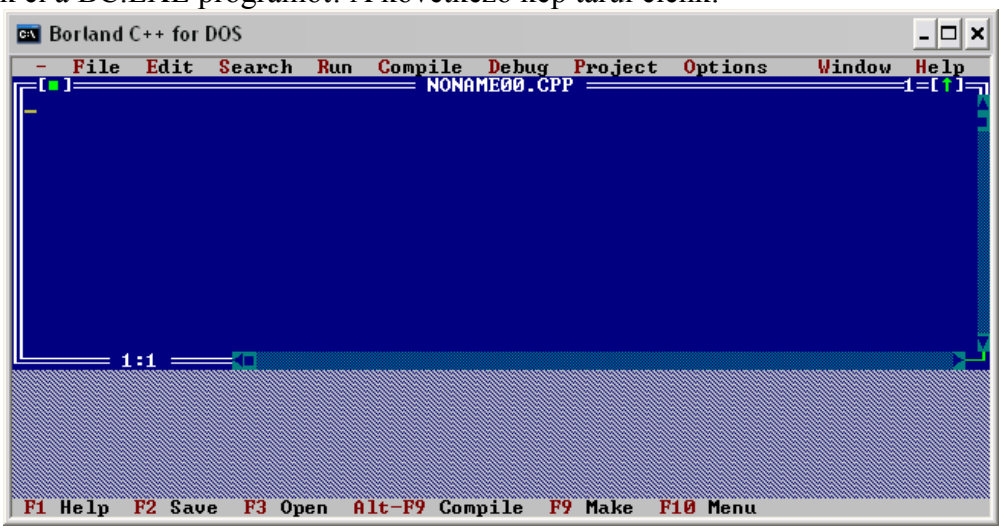
A másik a Microsoft Visual C++ 6.0, mely a HSzK összes termében a hallgatók rendelkezésére áll. Ennél léteznek újabb Visual C++ fordítók is, a Visual C++ .Net, a .Net 2003, illetve a .Net 2005. (Ezek a fordítók 32 bites kódot készítenek. A 32 bites programok maximum 2 GB memóriát kezelhetnek, ez általában már nem jelent számunkra problémát, nem úgy, mint a 16 bites DOS 64 kB-os, illetve néhány száz kB-os korlátai.)

Ahogy a programok nevéből látszik, ezek C++ fordítók. A C++ nyelv felülről kompatibilis a C-vel, tehát minden C program egyben C++ program is. Napjainkban már csak kevés esetben találkozunk olyan fordító programmal, mely csak a C-t ismeri, és a C++-t nem.

Aki ingyenes Windowsos fordítóra vágyik, annak érdemes kipróbálnia a [3] rendszert, mely a DevC++-ra épülő grafikus felület, Delphi- (ill. C++ Builder)-szerűen hozhatunk létre benne grafikus Windowsos programokat, de dolgozhatunk konzol módban is. Hátránya, hogy lassú a fordító (ez legyen a legnagyobb bajunk…). (Erre a rendszerre Nagy Gergely hívta fel a figyelmem, ezúton is köszönet neki.)

1.1 Program létrehozása Borland C++-ban

Indítsuk el a BC.EXE programot! A következő kép tárul elénk:



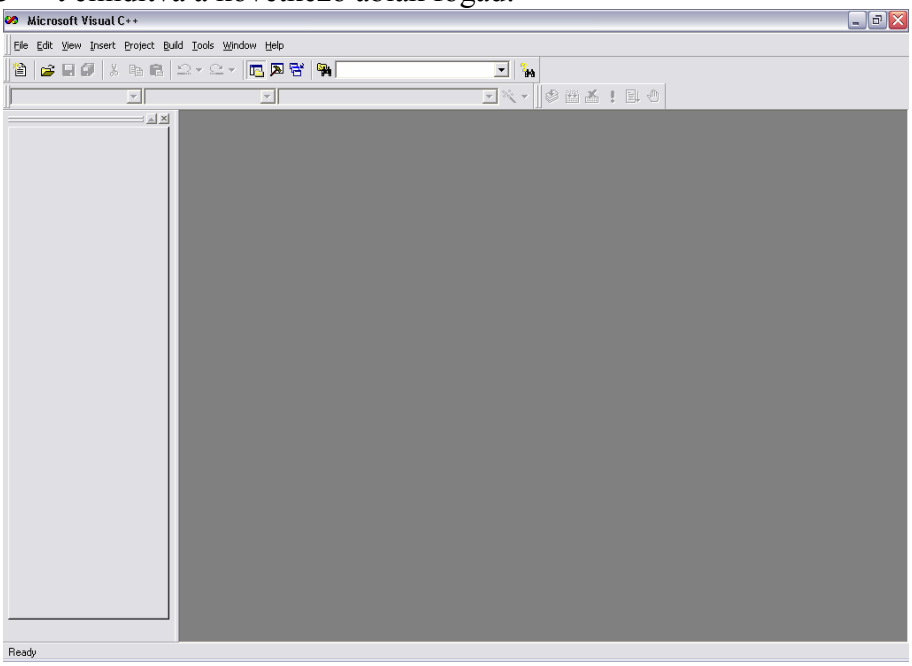
1. ábra

A [■] gombra kattintva tudjuk bezárni a szerkesztőmezőt, a [↑] gombbal pedig kinagyíthatjuk, hogy befedje az ablak egész belső részét. Ha véletlenül (vagy szándékosan) becsuktuk a szerkesztőmezőt, a File menü New parancsával hozhatunk létre újat.

Most nincs más dolgunk, mint begépelni a kódot. A készülő programot érdemes néhány percenként elmenteni.

1.2 Program létrehozása Visual C++ 6.0-ban

A Visual C++-t elindítva a következő ablak fogad:



2. ábra

Ezúttal a helyzet bonyolultabb, ugyanis nem kezdhetünk el egyből gépelni. Előbb létre kell hozni a program projektjét. Ehhez kattintsunk a File menü New… parancsára!

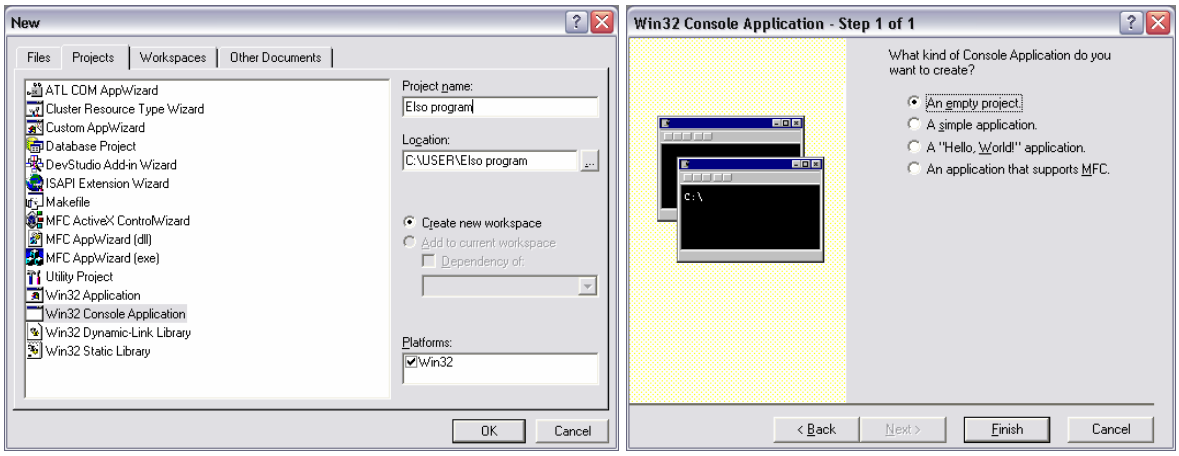
> Ne használjuk a New ikont! Ez egy txt fájlt hoz létre, amivel nem tudunk mit kezdeni, mert nem része a projektnek. Ha véletlenül mégis egy ilyen ablakba kezdtük írni a programit, ne essünk kétségbe, hanem mentsük el a fájlt (cpp kiterjesztéssel! tehát pl. valami.cpp), csukjuk be, majd a következőkben leírtak szerint hozzuk létre a projektet. Ha ez megvan, válasszuk a Project menü Add To Project almenüjéből a Files… parancsot! Keressük meg az elmentett fájlt, és adjuk a projekthez!

A New ablakban alapértelmezés szerint a Projects fül nyílik meg. Ha mégsem, akkor válasszuk ki azt. A listából válasszuk ki a Win32 Console Application-t.

> Figyelem! Ne a sima Win32 Application-t válasszuk, mert ellenkező esetben kezdhetjük előről az egész projekt létrehozást!

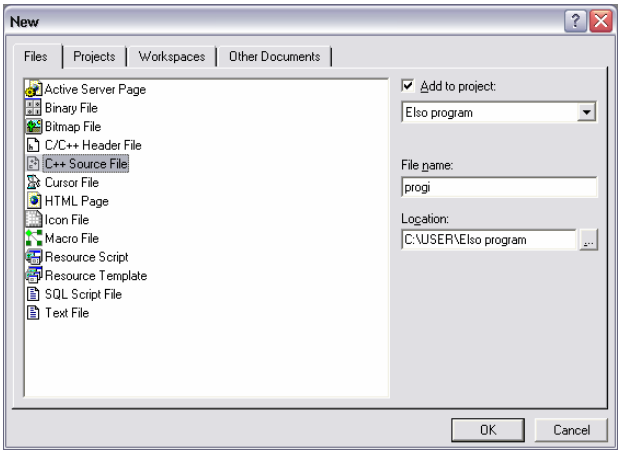
Válasszuk ki a mappát, ahova a projektet tenni szeretnénk, és adjuk meg a projekt nevét. A kiválasztott mappában létre fog jönni egy új mappa a megadott névvel, ebbe kerülnek a program fájljai.

Kattintsunk az OK-ra! Ekkor a jobb oldali ablakot kapjuk (3. ábra). Itt maradjunk meg az „An empty project” beállításnál!



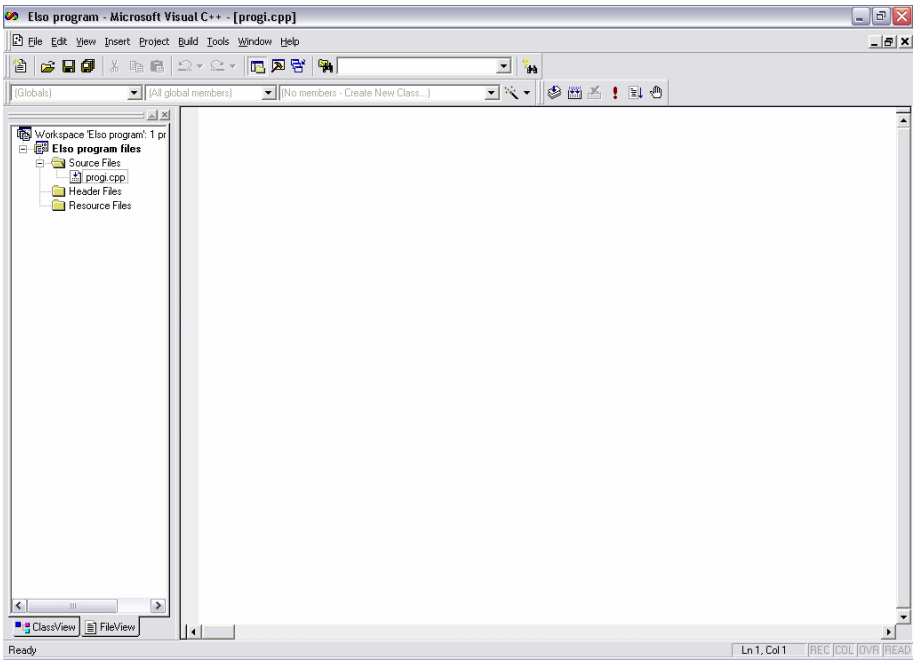
3. ábra

Ha a projekt létrejött, akkor ismét válasszuk ki a File menü New… parancsát!



4. ábra

Ezúttal a Files fülön kattintsunk, ha nem az lenne kiválasztva. Válasszuk ki a C++ Source File-t a listából. Adjunk neki nevet a File name szerkesztőmezőben, és nyomjuk meg az OK-t. Az eredményt az 5. ábrán látjuk.



5. ábra

1.3 Az első program

Másoljuk be az alábbi programot az általunk használt fejlesztőkörnyezetbe. A program letölthető a http://www.eet.bme.hu/~pohl/ weboldalról.

> Aki Borland fordítót használ, nem tudja használni a [Ctrl+C] [Ctrl+V] (másolás+beillesztés) funkciókat. A Borland fordító menüjében is találunk olyan funkciót, hogy Paste ([Shift+Ins]), de ez saját vágólapot használ, nem a Windowsét, ezért a Windows vágólapra helyezett szöveg nem másolható be a Borland fordítóba.

```
//*******************************************************
#include <stdio.h>       // Header fájlok beszerkesztése
#include <stdlib.h>
#include <math.h>
//*******************************************************


//*******************************************************
// prototípusok, azaz függvénydeklarációk
//*******************************************************
void osszead();
void kivon();
void szoroz();
void oszt();
void gyok();
void szinusz();
void osztok();
int egy_int_beolvasasa();
double egy_double_beolvasasa();
void ket_double_beolvasasa(double*,double*);
//*******************************************************


//*******************************************************
// függvénydefiníciók
//*******************************************************
//*******************************************************
int main(){
//*******************************************************
        int valasztott_ertek=0;  // egész típusú változó 0 kezdőértékkel

        printf("Udvozoljuk! On a szamologepet hasznalja. Jo munkat!\n"); // kiírás
        while(valasztott_ertek!=10){ // ciklus, míg a v.é. nem egyenlő 10-zel

                // A menü kiírása

                printf("\nKerem, valassza ki a muveletet!\n");
                printf("1  Osszeadas\n");
                printf("2  Kivonas\n");
                printf("3  Szorzas\n");
                printf("4  Osztas\n");
                printf("5  Negyzetgyok\n");
                printf("6  Szinusz\n");
                printf("7  Egesz szam osztoi\n");
/*
                printf("8  Természetes logaritmus\n");
                printf("9  Exponenciális\n");
*/
                printf("10 Kilepes\n");

                // A választott érték beolvasása

                fflush(stdin); // standard input puffer ürítése
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                // A választott művelet végrehajtása

                switch(valasztott_ertek){ // a v.é.-nek megfeleő case után folytatja
                        case  1: osszead();      break; // az osszead függvény hívása
                        case  2: kivon();        break;
                        case  3: szoroz();       break;
                        case  4: oszt(); break;
                        case  5: gyok(); break;
                        case  6: szinusz();      break;
                        case  7: osztok();       break;
/*
                        case  8: logarit();      break; // természetes logaritmus
```

```
                        case  9: exponen();       break;
*/
                        case 10:                  break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
        printf("\nTovabbi jo munkat!\n");
        return 0;
}


//*********************************************************
void osszead(){
//*********************************************************
        double a,b;       // két valós értékű változó
        printf("\nOsszeadas\n");
        ket_double_beolvasasa(&a,&b); // függvényhívás
        printf("\nAz osszeg: %g\n",a+b); // összeg kiírása
}


//*********************************************************
void kivon(){
//*********************************************************
        double a,b;
        printf("\nKivonas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA kulonbseg: %g\n",a-b);
}


//*********************************************************
void szoroz(){
//*********************************************************
        double a,b;
        printf("\nSzorzas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA szorzat: %g\n",a*b);
}


//*********************************************************
void oszt(){
//*********************************************************
        double a,b;
        printf("\nOsztas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA hanyados: %g\n",a/b);
}


//*********************************************************
void gyok(){
//*********************************************************
        double a;
        printf("\nGyokconas\n");
        a=egy_double_beolvasasa();
        printf("\nA negyzetgyok: %g\n",sqrt(a));
}


//*********************************************************
void szinusz(){
//*********************************************************
        double a;
        printf("\nSzinusz\n");
        a=egy_double_beolvasasa();
        printf("\nA szinusz: %g\n",sin(a));
}


//*********************************************************
void osztok(){
//*********************************************************
        int a,i;
        printf("\nOsztok szamitasa\n");
        a=egy_int_beolvasasa();
        printf("\n%d osztoi:\n",a);
        for(i=1;i<=a/2;i++)
                if(a%i==0)printf("%d\t",i);
        printf("%d\n",a);
}


//*********************************************************
int egy_int_beolvasasa(){
//*********************************************************
        int OK=0,szam=0;
```

```
        while(OK==0){
                printf("\nKerem a szamot: ");
                fflush(stdin);
                if(scanf("%d",&szam)!=1){printf("\nHibas szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }
        return szam;
}


//*****************************************************
double egy_double_beolvasasa(){
//*****************************************************
        int OK=0;
        double szam;
        while(OK==0){
                printf("\nKerem a szamot: ");
                fflush(stdin);
                if(scanf("%lg",&szam)!=1){printf("\nHibas szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }
        return szam;
}


//*****************************************************
void ket_double_beolvasasa(double * a, double * b){
//*****************************************************

        // Első szám bekérése

        int OK=0;
        while(OK==0){
                printf("\nKerem az elso szamot: ");
                fflush(stdin);
                if(scanf("%lg",a)!=1){printf("\nHibas elso szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }

        // Második szám bekérése

        OK=0;
        while(OK==0){
                printf("\nKerem a masodik szamot: ");
                fflush(stdin);
                if(scanf("%lg",b)!=1){printf("\nHibas masodik szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }
}
```

Próbáljuk ki a programot, futtassuk le! Ezt Borland fordítónál a Run menü Run parancsával tehetjük meg, Visual C++ esetén pedig a Debug menü Start, vagy Start Without Debugging parancsával.

1.4 A program működése

Ebben a pontban áttekintjük a fenti program működését, megismerkedünk az egyes részek szerepével, funkciójával. Most még nem feladat, hogy hasonlót tudjunk írni, de a következő pontban bemutatott hibakereséshez értenünk kell a működést.

Az ebben a jegyzetben használt program színeket a Visual C++fordító használja, de a Borland is hasonlóképpen más-más színnel jelöli a kód egyes részeit. Mindez a jobb áttekinthetőséget szolgálja. Aki Visual C++-t használ, és a kódja fekete-fehér, az nem cpp fájlként nyitotta meg a kódot. Ebben az esetben tekintse át ismét az 1.2 fejezetben leírtakat!

1.4.1 Megjegyzések, formázás a C programban

Az ANSI/ISO C nyelv kétféle megjegyzéstípust támogat, de a régebbi fordítók (pl. TurboC) csak a /* */ típust ismerik. Ha ilyen fordítóval van dolgunk, akkor kézzel kell kicserélni a // típusú megjegyzéseket a /* */ típusúakra.

A megjegyzések a programba helyezett olyan szövegek, melyek a programozók számára áttekinthetőbbé teszik a forráskódot, továbbá lehetővé teszik, hogy egyes programrészeket ideiglenesen eltávolítsanak a kódból, például hibakeresési célból.

Az egyik megjegyzésfajta a /* operátorral kezdődik. A két karakter közé nem tehetünk más karaktert, pl. szóközt. A /* utáni szöveg, bármit is tartalmaz, megjegyzésnek számít, tehát a fordító nem veszi figyelembe. A megjegyzés az első */ operátorig tart, tehát többsoros is lehet (lásd a fenti programot).

A /* */ típusú megjegyzések nem ágyazhatók egymásba, ami azt jelenti, hogy egy ilyen kód:

```
/*
              /*      case  8: logarit();   break; // természetes logaritmus     */
                      case  9: exponen();   break;
*/
```

helytelen, ugyanis a második /*-ot a fordító nem veszi figyelembe, mert egy megjegyzésen belül volt, és a fordító semmit sem vesz figyelembe, ami egy megjegyzésen belül van. A fordítóprogram a fordítás során ezt két üzenettel is jelzi:

```
f:\vc7\szgl\main.cpp(69) : warning C4138: '*/' found outside of comment
f:\vc7\szgl\main.cpp(69) : error C2059: syntax error : '/'
```

A másik megjegyzésfajtát, a //-t a C nyelv a C++-ból vette át. Ez a megjegyzés a sor végéig tart.

A /* */ megjegyzésben nyugodtan lehet // típusú megjegyzés, mert a /*-gal kezdett megjegyzés mindig a */-ig tart, és a köztük lévő //-t éppúgy nem veszi figyelembe a fordító, mint a /*-ot az előbbi példában (de a lezáró */-t ne a // után tegyük!). // után csak akkor kezdhetünk /* megjegyzést, ha azt még a sor vége előtt be is fejezzük, mert ekkor a /*-t sem veszi figyelembe a fordító, pl.:

```
case  4: oszt(); break; // ez /* így jó */ két megjegyzés egyben
```

Sem a megjegyzést jelző operátorok elé, sem mögé nem kötelező szóközt írni, itt mindössze azért szerepel, mert jobban átlátható kódot eredményez.

A bemutatott programban a // típusú megjegyzéseket használtuk arra, hogy a programkód áttekinthetőbb legyen. Ezt a célt szolgálja az egyes programrészeket jelző csillagsor (természetesen más karakterek is használhatók, pl. //----------, //#########, //@@@@@, kinek mi tetszik), a megjegyzést követő kódrész funkcióját leíró `// Második szám bekérése`, vagy a sor végére biggyesztett, a sor funkcióját leíró megjegyzés is:

```
case  8: logarit();     break; // természetes logaritmus
```

A /**/ típusú megjegyzésekkel olyan kódrészeket távolítottunk el, melyeket most nem akarunk használni, de később esetleg igen, vagy valamilyen más okból nem kívánjuk törölni.

Az áttekinthetőséget javító megjegyzések általában egysorosak, de például, ha egy teljes függvény funkcióit, paramétereit akarjuk leírni, akkor is legfeljebb párosorosak. Az eltávolított kódrészek hossza viszont akár több száz sor is lehet. Emiatt alakult ki az, hogy melyik megjegyzéstípust mire használjuk. Ugyanis az eltávolított kódrészben valószínűleg jó néhány, az áttekinthetőséget javító megjegyzés is található, és ha erre a célra is a /**/ megjegyzést használnánk, akkor minden */-rel zárult, áttekinthetőséget javító megjegyzés megszakítaná a kód eltávolítását.

Most pedig a formátumról. A C programokban minden olyan helyre, ahova egy szóközt tehetünk, oda többet is tehetünk, sőt akár soremelést, vagy tabulátort is. Az utasítás végét a ;, vagy a } jelzi. például:

```
if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;
```

Átírható lenne:

```
if
(          scanf
```

```
        (
"%d"                            ,       & valasztott_ertek
   )            !=      1
)       valasztott_ertek       =
0
;
```

módon is, csak ettől jóval átláthatatlanabb lenne a kód.

Annak érdekében, hogy jól látsszon, meddig tart egy utasítás vagy egy utasításblokk, az adott blokkhoz tartozó utasításokat beljebb kezdjük, ahogy ez a példaprogramban is látszik (nem szükséges ennyivel beljebb kezdeni, elég 2-3-4 szóköznyi távolság).

A { elhelyezésére két jellemző szokás alakult ki. Az első:

```
void szoroz(){
        double a,b;
        printf("\nSzorzas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA szorzat: %g\n",a*b);
}
```

A második:

```
void szoroz()
{
        double a,b;
        printf("\nSzorzas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA szorzat: %g\n",a*b);
}
```

Az első kisebb helyet foglal, a másodiknál jobban látszik, hogy melyik kapocshoz melyik kapocs tartozik. Abban az esetben, ha valaki nem tartja be ezeket a szabályokat, és mindent a bal szélen kezd, nagyon nehéz dolga lesz, ha valahol véletlenül lefelejt egy nyitó vagy záró kapcsot, mert így a párnélkülit elég nehéz megtalálni.

Ha egy utasítást osztunk ketté, mert egy sorban nehezen fér el, akkor a második felét beljebb kell kezdeni. Például:

```
        fflush(stdin);
        if(scanf("%d",&valasztott_ertek)!=1)
                valasztott_ertek=0;
        printf("\nSzorzas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA szorzat: %g\n",a*b);
```

Itt jól látszik, hogy a valasztott_ertek=0; az if utasításhoz tartozik, ha nem tennénk beljebb:

```
        fflush(stdin);
        if(scanf("%d",&valasztott_ertek)!=1)
        valasztott_ertek=0;
        printf("\nSzorzas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nA szorzat: %g\n",a*b);
```

Így azt hihetnénk, hogy a valasztott_ertek=0; egy külön utasítás, mert a ; hiánya az előző sor végén nem feltűnő.

1.4.2 Header fájlok beszerkesztése

```
//*****************************************************
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//*****************************************************
```

A C programok fordítása két lépésben történik. Először az ún. előfordítónak, vagy preprocesszornak szóló utasításokat figyelembe véve készül egy köztes C program, majd a fordító ebből a köztes kódból készíti el a gépi kódú, lefordított fájlt. Az összes, előfordítónak

szóló utasítás # (hash mark, vagy kettős kereszt) szimbólummal kezdődik, és az adott sorban csak szóköz vagy tabulátor karakterek vannak előtte.

A #include az utána megadott nevű fájlt hozzászerkeszti a forráskódhoz. A fájl neve kétféle formában szerepelhet (a fenti kódban csak az egyiket használtuk):

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

Az első esetben, tehát <> között megadott névnél, a fordító a fejlesztőkörnyezetben beállított mappá(k)ban keresi a fájlt. (Ez Borland C++ esetén az Options menü Directories pontjában állítható, Visual C++ esetén a Tools/Options/Directories-ban.). Ha "" között szerepel, akkor pedig abban a mappában, ahol a program forráskódja is van. Tehát "" esetén a programunkhoz tartozik a header fájl, <> esetén pedig a rendszerhez.

A header fájlok konstansokat és függvények prototípusait tartalmazzák. Például ha megnyitjuk az stdio.h-t (ez az általam használt számítógépben "C:\Program Files\Microsoft Visual Studio .NET\Vc7\include\stdio.h"), akkor pl. ezt láthatjuk benne:

```
_CRTIMP int __cdecl printf(const char *, ...);
_CRTIMP int __cdecl putc(int, FILE *);
_CRTIMP int __cdecl putchar(int);
_CRTIMP int __cdecl puts(const char *);
_CRTIMP int __cdecl _putw(int, FILE *);
_CRTIMP int __cdecl remove(const char *);
_CRTIMP int __cdecl rename(const char *, const char *);
_CRTIMP void __cdecl rewind(FILE *);
_CRTIMP int __cdecl _rmtmp(void);
_CRTIMP int __cdecl scanf(const char *, ...);
_CRTIMP void __cdecl setbuf(FILE *, char *);
```

A printf() és a scanf() ezek közül szerepel is a programban. Ugyancsak az stdio.h-ban szerepel a fflush() függvény is. A sin() és az sqrt() a math.h része. A program nem használja az stdlib.h-t, tehát ezt az include sort törölni is lehet.

A standard függvények (printf, scanf stb.) forráskódja nem áll rendelkezésünkre, azt a gyártó nem mellékeli a fordítóprogramhoz. Ezeknek a függvényeknek a lefordított kódja .lib fájlokban található, melyet a linker szerkeszt a programhoz (csak azokat a függvényeket, melyeket valóban használunk is).

Hogy mely headerekben milyen függvények találhatók, az pl. a megfelelő C nyelv könyvekben megtalálható (pl. [1]).

stdio=Standard Input Output, stdlib=Standard Library

1.4.3 Függvénydeklaráció, függvénydefiníció

Az első programnyelvekben az egész program csak utasítások sorozatából állt. Ez azonban már a közepes méretű programokat is átláthatatlanná tette. Ezért születtek a strukturált programozást támogató programnyelvek, mint a C (bizonyos méret fölött ez is áttekinthetetlenné válik, ezért jöttek létre az objektumorientált nyelvek, mint amilyen a C++).

A strukturáltság azt jelenti, hogy a programot adott funkciót ellátó részekre, függvényekre osztjuk. A függvények előnye, hogy ha azonos műveletet kell többször megismételni, akkor elég csak egyszer leírni az ehhez szükséges kódot, és ezt a programban akárhányszor „meghívhatjuk", vagyis végrehajthatjuk.

A fenti program második részét a függvénydeklarációk alkotják:

```
//*****************************************************
// prototípusok, azaz függvénydeklarációk
//*****************************************************
void osszead();
void kivon();
void szoroz();
```

```
void oszt();
void gyok();
void szinusz();
void osztok();
int egy_int_beolvasasa();
double egy_double_beolvasasa();
void ket_double_beolvasasa(double*,double*);
//*****************************************************
```

Gondot szokott okozni a „deklaráció” és a „definíció” elnevezések megkülönböztetése. Ebben segíthet, hogy a francia „la declaration de la Republic Hongroise” azt jelenti magyarul, hogy a Magyar Köztársaság kikiáltása. A deklaráció tehát kikiáltást jelent: megmondjuk, hogy van ilyen. Ha nem csak azt akarjuk tudni, hogy van ilyen, hanem azt is tudni akarjuk, milyen, akkor definiálni kell a függvényt.

A kékkel szereplő szavak a kulcsszavak, melyeket a C nyelv definiál. Feketével a többi programrész szerepel, a zöld pedig a megjegyzés (1.4.1).

Itt minden egyes sorban egy-egy függvény neve szerepel, mely három részből áll:

1. Visszatérési érték. Ha van egy matematikai egyenletünk, pl. x=sin(y), akkor ebben az esetben a sin függvény visszatérési értéke y szinusza, és ez kerül be x-be. Programozás közben gyakran írunk olyan függvényeket, melyeknek szintén van visszatérési értéke, pl. maga a szinusz függvény a fenti programban is szerepel.
   A fenti függvények között kettő olyan van, melynek szintén van visszatérési értéke: az egy_int_beolvasasa() int típusú, tehát egész számot ad vissza, az egy_double_beolvasása() double típusú, tehát valós számot ad vissza. Azok a függ vények, melyek neve előtt a void szerepel, nem adnak visszatérési értéket. Például, ha egy függvénynek az a feladata, hogy kiírjon valamit a képernyőre, akkor nincs szükség visszatérési értékre.
2. A függvény neve. Ez angol kis és nagybetűkből, számokból, valamint _ jelekből állhat, szóköz vagy egyéb elválasztó karakter nem lehet benne, és nem kezdődhet számmal (aláhúzással igen).
3. A () zárójelek között vannak a függvény paraméterei. Például a sin(y) esetén a szinusz függvény meg kell, hogy kapja y értékét, mivel annak szinuszát kell kiszámítania. A fenti deklarációk közül csak a ket_double_beolvasasa(double*,double*); függvénynek vannak paraméterei, méghozzá két double típusú pointer (a * jelzi, hogy pointerről, vagyis memóriacímről van szó).

A deklarációnak mindig előbb kell szerepelnie, mint ahol a függvényt használjuk, a definíció lehet később is, vagy akár másik programmodulban, vagy .lib fájlban. Fontos megjegyezni, hogy **a definíció egyben deklaráció is**, tehát ha a függvények definícióját a main() függvény elé tettük volna (ez igen gyakran megesik), ahol azokat használjuk, akkor felesleges odaírni külön a deklarációt. Ebben a programban például nem írtunk deklarációt a main() függvényhez (ez nem is szokás).

Lássuk most a függvénydefiníciót! Például:

```
//*****************************************************
void ket_double_beolvasasa(double * a, double * b){
//*****************************************************

	// Első szám bekérése

	int OK=0;
	while(OK==0){
		printf("\nKerem az elso szamot: ");
		fflush(stdin);
		if(scanf("%lg",a)!=1)
			{printf("\nHibas elso szam, adja meg helyesen!\n"); continue;}
		OK=1;
	}
```

```
        // Második szám bekérése

        OK=0;
        while(OK==0){
                printf("\nKerem a masodik szamot: ");
                fflush(stdin);
                if(scanf("%lg",b)!=1)
                        {printf("\nHibas masodik szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }
}
```

Összehasonlítva a deklarációval, a következők a különbségek:

1. A paraméterlistában nem csak a pereméter típusa szerepel, hanem egy változónév is. A függvény kódjában ezen a néven hivatkozunk a paraméterre.

   Deklaráció esetén is oda szabad írni a változó nevét, de ott nem kötelező, definíciónál viszont igen. (A deklarációnál megadott változónév nem kell, hogy megegyezzen a definícióban megadottal.)
2. A) után nem ; áll, hanem egy kapcsos zárójelek közé zárt utasításblokk.

Ha a függvénynek van visszatérési értéke (tehát nem void), akkor a függvényben kell, hogy szerepeljen return utasítás, a return után kell írni a visszaadott értéket. Pl.:

```
//*****************************************************
int egy_int_beolvasasa(){
//*****************************************************
        int OK=0,szam=0;
        while(OK==0){
                printf("\nKerem a szamot: ");
                fflush(stdin);
                if(scanf("%d",&szam)!=1)
                        {printf("\nHibas szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }
        return szam;
}
```

Visszatérési értéket nem adó (void típusú) függvény esetén is lehet return utasítás, ekkor a return-t pontosvessző követi.

Egy függvényben több return utasítás is lehet. Ez például akkor hasznos, ha valamilyen feltétel esetén csak a függvény egy részét kell végrehajtani.

1.4.4 A main() függvény

Minden C nyelvű program futtatása a main() függvény futtatását jelenti. Amikor rákattintunk az exe-re, akkor a program main() függvénye indul el. A main természetesen más függvényeket is hívhat, és azok is hívhatnak más függvényeket, stb.

Ha egy függvénynek nem írunk visszatérési típust, akkor C fordító esetében alapértelmezés szerint int, C++ fordító esetében void lesz ez a típus. Mivel mi C++ fordítót használunk, de C programot fordítunk, a fordító általában rugalmasan kezeli ezt a kérdést, és mindkettőt elfogadja, a return paraméterezése alapján dönti el, hogy mit is akarunk: ha egy egész értéket adunk vissza (mint a fenti programban), akkor int main()-nek tekinti, ha simán return;-t írunk, vagy nem használjuk a return-t, akkor void main()-nek.

Lássuk, hogyan működik!

```
//*****************************************************
```

```
int main(){
//*******************************************************
	int valasztott_ertek=0;
```

Először létrehozunk (definiálunk) egy valsztott_ertek nevű, egész értékkészletű változót, és 0 kezdőértéket adunk neki. Ha nem adunk kezdőértéket, akkor bármi lehet benne, ez nincs meghatározva.

Borland C++ esetén -32768 és +32767 közötti egész számokat tárolhatunk benne, mert itt az int 16 bites, Visual C++ esetén pedig -2147483648 és +2147483647 közötti, mert az int 32 bites. Ezek az értékek úgy adódnak, hogy 2^{15} ill. 2^{31} értékek a határok, a 16. ill. a 32. bit jelenti az előjelet, továbbá pozitív irányban azért eggyel kevesebb értékünk van, mert a nullának is kell hely.

Amennyiben 64 bites rendszerben fordítjuk a programot, az int általában továbbra is 32 bites marad, mert a ±2 milliárd egész értékű műveleteknél általában elég szokott lenni, ritkán van szükség ennél nagyobbra. Ha mégis szükség volna erre, akkor a long típust használhatjuk (32 ill. 16 bites rendszerben a long 32 bites, de a legújabb C++ szabványban létezik long long típus, mely 64 bites, de ez egyik, általunk használt fordítóban sem használható. Ehelyett Visual C++-ban az _int64 típust használhatjuk, de ez nem szabványos.)

További, a C nyelvben használt típusokkal kapcsolatban lásd a szakirodalmat, pl. [1].

```
	printf("Udvozoljuk! On a szamologepet hasznalja. Jo munkat!\n");
```

A printf függvény a standard kimenetre kiírja a paraméterként megadott, idézőjelek között szereplő szöveget. A standard kimenet általában a képernyő. Ha azonban a programot úgy indítjuk el, hogy program.exe > kimemet.txt, akkor a képernyőn nem jelenik meg semmi, ehelyett a szöveg a kimenet.txt nevű fájlba kerül.

Az idézőjelek közötti szövegben használhattunk volna ékezeteket, csak sajnos ezt a Windows hibásan jeleníti meg, az ékezetek nélküli szöveg olvashatóbb. (**Próbáljuk ki!**)

A kiírt szövegekben nem minden íródik ki a képernyőre, a \ (fordított per, vagy back slash) és a % jel után speciális, a kiíratást vezérlő karakter(ek) szerepel(nek). Erről még később lesz szó. Ebben a stringben (a string karakterlánc, vagyis az, ami a két idézőjel között van) egy ilyen szerepel, a \n. A \n után következő szöveg új sorba kerül. (Jelen esetben ez a szöveg egy későbbi printf()-ben kerül kiírásra, de akár itt is írhattunk volna utána valamit.)

```
	while(valasztott_ertek!=10){
```

A while utáni {, és a hozzá tartozó } közötti utasítások addig ismétlődnek, míg a paraméterként megadott kifejezés igaz, jelen esetben, amíg a valasztott_ertek nem egyenlő 10-zel. Tehát a != azt jelenti, hogy nem egyenlő.

A while utáni {-hez tartozó }-t a while w betűjével egyvonalban függőlegesen lefelé haladva találjuk meg. Pont azért írtuk ide, hogy jól látsszon, mi van a while-on belül, lásd az 1.4.1 pontot, a formázással kapcsolatban.

Ne felejtsük, hogy a valasztott_ertek létrehozásakor 0 kezdőértéket adtunk, ami ugye nem egyenlő tízzel, tehát a feltétel igaz, ezért végrehajtódnak az utasítások a kapcsos zárójelek között. Ha ehelyett `int valasztott_ertek=10;` szerepelne, akkor a feltétel hamis lenne, tehát a {} közötti utasítások (vagyis a ciklusmag) egyszer sem hajtódna végre. (**Próbáljuk ki!**)

Fontos! Ha nem adtunk volna kezdőértéket, tehát `int valasztott_ertek;` szerepelt volna, akkor valasztott_ertek kezdőértéke meghatározatlan lenne, tehát akár 10, akár bármi más is lehetne, tehát kiszámíthatatlanná válna a program működése, és ez súlyos hiba, amiért a ZH-ban pontlevonás jár!

```
		// A menü kiírása
```

```
                printf("\nKerem, valassza ki a muveletet!\n");
                printf("1  Osszeadas\n");
                printf("2  Kivonas\n");
                printf("3  Szorzas\n");
                printf("4  Osztas\n");
                printf("5  Negyzetgyok\n");
                printf("6  Szinusz\n");
                printf("7  Egesz szam osztoi\n");
/*
                printf("8  Természetes logaritmus\n");
                printf("9  Exponenciális\n");
*/
                printf("10 Kilepes\n");
```

Az első printf() \n-nel kezdődik, és mivel az előző printf() \n-nel végződött, ez azt jelenti, hogy egy üres sor marad a két szövegsor között. A két \n-t egy stringbe is tehettük volna, így: \n\n, ez ízlés kérdése (ne felejtsük azt sem, hogy ez egy ciklus belsejében van, tehát valószínűleg többször ismétlődni fog a kiírás).

Az egyes kiírt sorok csak azért kerültek külön-külön prontf-be, hogy átláthatóbb legyen a program. (Ez az „átláthatóbb legyen a program” igen gyakori hivatkozási alap, ugyanakkor rendkívül fontos, hogy gyorsan megtaláljunk bármit a programunkban, mert egy rosszul felépített program nagyságrendekkel megnövelheti a hibakereséssel vagy bővítéssel töltött időt. A hibakeresés ideje (persze a bővítés is) egy jól megírt program esetén is összemérhető, sőt, akár nagyobb is lehet, mint maga a program megírása. Gondoljuk csak meg, milyen sok ez egy rosszul megírt programnál!)

Tehát megírhattuk volna pl. így is, az eredmény ugyanaz, csak a látvány nem:

```
printf("1  Osszeadas\n2  Kivonas\n3  Szorzas\n4  Osztas\n");
printf("5  Negyzetgyok\n6  Szinusz\n7  Egesz szam osztoi\n");
```

A 8-as és 9-es kiírását ideiglenesen eltávolítottuk a kódból, mert az ezekhez szükséges függvényeket nem valósítottuk meg. Ez később az olvasó feladata lesz.

```
// A választott érték beolvasása

fflush(stdin);
if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;
```

Ezt a két sort fordítva tárgyaljuk, mert a scanf ismerete nélkül nem érthető az fflush.

Az if utasítás, hasonlóan a while-hoz, egy feltételt vár paraméterként. Jelen esetben azt nézi, hogy vajon a scanf() visszatérési értéke 1-e, vagy sem. Ha az if feltétele igaz, akkor a) után szereplő utasítás végrehajtódik, ha nem igaz, akkor kihagyja. (Az if és a while közötti különbség tehát az, hogy az if utáni utasítás csak egyszer fut le, míg a while utáni mindaddig, amíg a feltétel igaz. Ha több utasítást szeretnénk az if után írni, akkor azokat tegyük {} közé, mint a while esetében!)

Tehát ha a scanf visszatérési értéke nem 1, akkor a valasztott_ertek 0-val lesz egyenlő.

A scanf() függvény tulajdonképpen a printf() ellentéte, tehát a billentyűzetről kérhetünk be adatokat, jelen esetben azt, hogy hányas számot ír be az, aki a programot lefuttatja. (A scanf igazándiból a standard bemenetről olvas, mely alapesetben a billentyűzet, de itt is átirányíthatunk egy fájlt: program.exe < bemene.txt. Természetesen ebben az esetben a bemenet.txt-nek léteznie kell, és megfelelő adatokat kell tartalmaznia.)

A scanf első paramétere egy string – csakúgy, mint a printf-é. Ez a string azonban csak vezérlő karaktereket és elválasztó jeleket tartalmazhat, mert a **scanf soha semmit sem ír ki**, és ha mást is írunk bele, akkor hibásan fog működni!

Jelen esetben egy darab %d szerepel itt, vagyis egy darab egész számot akarunk beolvasni. %d helyett írhattunk volna %i-t is, mindkettő ugyanazt jelenti (a d a decimal-ra utal, vagyis 10-es számrendszerbeli számra, az i pedig int-re).

A string után, vesszővel elválasztva szerepel azoknak a változóknak a neve, ahová a beolvasott értéket tenni akarjuk. Annyi változót kell megadni, ahány beolvasását a stringben jeleztük! (A scanf függvény megszámolja, hogy a stringben hány beolvasandó értéket adtunk meg, és ezután annyi darab változót keres a string után, amennyi a string alapján várható. Sajnos a fordító nem tudja megállapítani, hogy annyit adtunk, kevesebbet, vagy többet, és abban az esetben, ha nem pontosan annyit adtunk, akkor ez a program futtatása közben jelentkező hibát okozhat.)

Figyeljük meg a változó neve előtt szereplő & jelet! Ez az operátor ugyanis az utána írt változó memóriacímét adja, vagyis egy pointert (mutatót), mely a változóra mutat a memóriában. Ha ezt nem írnánk elé, akkor a scanf() függvény azt az értéket kapná, ami a valasztott_ertek-ben van, vagyis első futáskor 0-t, hiszen ez volt a kezdőérték. Ezzel nem sok mindent tudna kezdeni, mert neki az lenne a dolga, hogy a változóba tegyen be egy értéket, nem az, hogy a változóból kivett értékkel bármit csináljon. Emiatt a memóriacímet kell neki átadni, így ugyanis tudni fogja, hogy hol van a valasztott_ertek, amibe a beolvasott értéket pakolnia kell.

A scanf() függvény visszatérési értéke azt mondja meg, hogy hány darab változót olvasott be sikeresen. Jelen esetben egy darabot szeretnénk beolvasni. Mikor történik az, ha nem 1 a visszatérési érték? Például, ha az udvarias „Kerem valassza ki a muveletet!” kérésünkre a felhasználó ezt válaszolja „Anyád!”. Ezt ugyanis a scanf nem képes egész számként értelmezni, ezért a valasztott_ertek-be nem tesz semmit, viszont 0-t ad vissza.

Az fflush(stdin) funkciója az, hogy kiürítse a standard input puffert. Ehhez meg kell magyarázni, hogy mi a standard input puffer, és hogyan működik a scanf.

A standard input puffer egy olyan hely a memóriában, ahonnan a scanf beolvassa a beírt szöveget, és átalakítja például egész, vagy valós számmá, vagy meghagyja szövegnek, attól függően, hogy a scanf-ben pl. %d, %lg, vagy %s szerepelt-e. Ha a scanf ezt a puffert üresen találja, akkor szól az operációs rendszerek (a DOS-nak, Windowsnak, Unixnak/Linuxnak stb.), hogy olvasson be a standard inputról. Az operációs rendszer pedig mindaddig gyűjti a lenyomott billentyűket, míg az <Enter>-t le nem nyomjuk, aztán az egészet bemásolja a standard input pufferbe.

Az operációs rendszer egyik legfontosabb feladata, hogy ilyen jellegű szolgáltatásokat biztosítson. Ezek igazából függvényhívások. Ugyanis nagyon sok program szeretne pl. billentyűzetről olvasni, és elég macerás lenne, ha minden egyes billentyűműveletet a billentyűzet áramköreinek programozásával, az adott programozónak kellene megoldania. Tehát minden billentyűzet, lemez, képernyő stb. műveletet egy driver segítségével az operációs rendszer dolgoz fel, és a programok az operációs rendszer megfelelő függvényeit hívják. DOS esetén még csak néhányszáz ilyen volt, de manapság már sokezer. Ez teszi lehetővé például, hogy minden Windowsos program ugyanolyan ablakot nyit, ha az Open… parancsot választjuk a File menüben. Ez a fájlnyitás ablak például a commdlg.dll-ben, vagyis a common dialog dinamic linked library-ben található.

Ha nem írtuk volna be az fflush(stdin); utasításokat mindenegyes scanf elé, akkor előfordulhatna, hogy a felhasználó mondjuk azt írja be, hogy „1 2 3 4”. Ebben az esetben ez a scanf gond nélkül beolvassa az 1-et a valasztott_ertek-be (az 1 után szóköz következik, ami nem számjegy, ezért itt abbahagyja a beolvasást, a puffer további tartalmát nem bántja). Ezután (ahogy látni fogjuk) a program megvizsgálja a valasztott_ertek-et, és mivel 1-et talál benne, az osszead() függvényre ugrik. Ez a függvény azzal kezdi, hogy beolvas két valós számot, az összeadandókat. fflush nélkül ezek természetesen a 2 és a 3 lesznek. Kiszámolja az összeget, kiírja, majd újra felteszi a kérdést, hogy melyik műveletet választjuk. Mivel a 4-es még a pufferben maradt, azt beolvassa, és ezután várni fogja az osztandót és az osztót, hiszen a 4-es az osztást jelenti.

Ha az fflush-t használjuk, akkor az osszead() által meghívott ket_double_beolvasasa() függvényben lévő fflush() kiüríti a puffert, tehát az utána következő scanf várni fogja, hogy most írjuk be a két összeadandó értéket.

```
// A választott művelet végrehajtása
```

```
		switch(valasztott_ertek){
			case  1: osszead();   break;
			case  2: kivon();     break;
			case  3: szoroz();    break;
			case  4: oszt();      break;
			case  5: gyok();      break;
			case  6: szinusz();   break;
			case  7: osztok();    break;
/*
			case  8: logarit();   break; // természetes logaritmus
			case  9: exponen();   break;
*/
			case 10:              break;
			default: printf("Hibas muveletszam (%d). Probalja ujra!",
                                                                valasztott_ertek);
		}
```

A switch utasítás paramétere egy egész típusú mennyiség, jelen esetben a valasztott_ertek. A program végrehajtása azzal az utasítással folytatódik, ahol a case után szereplő érték megegyezik a valasztott_ertek-kel. Ha egyik case-szel sem egyezik meg, akkor a default utáni utasítás jön.

Minden sor végén látjuk a break; utasítást. Ha kitörölnénk, akkor miután az adott case sorra kerül a vezérlés, és lefut az ott szereplő utasítás (például az osszead() függvény) akkor a következő sorra menne tovább a program, tehát a kivon(); függvényre. Ha van break, akkor a switch-et lezáró } után folytatódik a végrehajtás.

```
	printf("\nTovabbi jo munkat!\n");
	return 0;
}
```

Mivel a main() visszatérési típusa int, egy egész értéket kell visszaadnunk. A 0 azt jelzi, hogy nem történt hiba. A hibát általában -1-gyel jelezhetjük. Ezt a visszatérési értéket az operációs rendszer kapja meg. Windowsban szinte sohasem foglalkozunk vele, Unixban kicsit gyakrabban.

1.4.5 A többi függvény

A többi függvényt két csoportra oszthatjuk. Az egyik csoportba azok a függvények tartoznak, amelyek a menüből kiválasztott matematikai műveletet végrehajtják, a másikba azok, amelyek bekérik a felhasználótól a számításokhoz szükséges számokat.

Először nézzük ez utóbbi csoportot. Egy vagy két szám bekérése tulajdonképpen egy viszonylag általános feladat, nem is igazán kötődik ehhez a programhoz. A matematikai műveleteknek megfelelően három ilyen függvényre lesz szükség, mert vannak műveletek, melyekhez két valós szám szükséges, vannak, melyekhez egy, és van egy olyan is, melyhez egy egész szám. A három függvény működése nagyon hasonló egymáshoz, tulajdonképpen ha az egyik megvan, akkor a másik kettő Copy+Paste-tel és némi módosítással elkészíthető.

Vegyük az egy valós számot bekérő függvényt!

```
//******************************************************
double egy_double_beolvasasa(){
//******************************************************
	int OK=0;
	double szam;
	while(OK==0){
		printf("\nKerem a szamot: ");
		fflush(stdin);
		if(scanf("%lg",&szam)!=1)
			{printf("\nHibas szam, adja meg helyesen!\n"); continue;}
		OK=1;
```

```
        }
        return szam;
}
```

A while ciklus addig fut, míg az OK egész értékű változó értéke 0. Figyeljük meg, hogy annak vizsgálatára, hogy az OK vajon egyenlő-e nullával, két darab egyenlőségjelet írtunk. **Jegyezzük meg, hogy az egyenlőség vizsgálatára mindig 2 db = jelet használunk! Egy darab egyenlőségjel az értékadást jelenti!**

Mi történne, ha azt írnánk: while(OK=0)? Nos, ebben az esetben az OK változóba 0 kerülne, és a kifejezés azt jelentené, mintha ezt írtuk volna: while(0). Van-e ennek értelme? Igen, a C nyelvben van. **A C nyelvben ugyanis az egész számoknak igaz/hamis jelentése is van: a 0 hamisat jelent, minden más egész szám igazat!** És mivel ebben az esetben 0, azaz hamis volna a zárójelben, a while ciklus egyszer sem futna le. Ennél is kellemetlenebb következménye lehet, ha nem nulla van a while feltételében, hanem mondjuk 1: while(1). Ez ugyanis azt jelenti, hogy a ciklus örökké fut, vagyis **végtelen ciklust** kaptunk. Ez pedig a program lefagyását eredményezheti. Előfordul, hogy szándékosan csinálunk végtelen ciklust, ekkor azonban gondoskodunk arról, hogy ki tudjunk lépni belőle. Ciklusból kilépni a break vagy a return utasítással tudunk (ezutóbbival természetesen az egész függvényből kilépünk).

A printf, fflush és scanf működését korábban már áttekintettük: kiírja a szöveget, kiüríti a standard input puffert, és bekér egy double típusú számot. A scanf-ben lg-vel jelöltük, hogy double típusú valós számot akarunk beolvasni. Ehelyett használhatjuk az le és az lf változatot is, ezek beolvasásnál ugyanazt jelentik (printf esetén különböző a jelentésük).

Az if után ezúttal két utasítást is tettünk, ezért azokat {} közé kellett zárni. Ezek közül a continue az új. Azt jelenti, hogy a ciklus elejére ugrunk, a feltételvizsgálathoz, tehát az utána következő OK=1; utasítás már nem hajtódik végre, ha nem sikerült beolvasni az egy darab valós számot (mert a felhasználó nem számot írt).

Ha sikerült beolvasni a szam-ot, akkor az if feltétele hamis, tehát a {}-be tett rész nem hajtódik végre, hanem a következő sorra lépünk, OK=1 lesz, a while feltétele tehát hamis lesz: nem fut le többször a ciklus, hanem a return szam; következik.

Az egész számot bekérő függvény gyakorlatilag ugyanez.

Nézzük a két valós számot bekérő függvényt!

```
//******************************************************
void ket_double_beolvasasa(double * a, double * b){
//******************************************************

        // Első szám bekérése

        int OK=0;
        while(OK==0){
                printf("\nKerem az elso szamot: ");
                fflush(stdin);
                if(scanf("%lg",a)!=1)
                        {printf("\nHibas elso szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }

        // Második szám bekérése

        OK=0;
        while(OK==0){
                printf("\nKerem a masodik szamot: ");
                fflush(stdin);
                if(scanf("%lg",b)!=1)
                        {printf("\nHibas masodik szam, adja meg helyesen!\n"); continue;}
                OK=1;
        }
}
```

Itt kétszer ugyanaz szerepel, mint az előző függvényben, azaz csak majdnem. A különbség az, hogy ezúttal két darab beolvasott értéket kell visszaadnunk. A return viszont csak egyet tud visszaadni. Mit tehetünk ebben az esetben? Két pointert (mutatót) adunk paraméterként a függvénynek, melyek egy-egy valós számok tárolására szolgáló változóra mutatnak.

Nézzük meg a scanf függvények használatát! **Itt az a ill. b változó előtt nem szerepel az & jel,** mert a és b nem valós szám, hanem valós számra mutató pointer, pont az, amire a scanf-nek szüksége van. Látni fogjuk, hogy az & jel a ket_double_beolvasasa() függvény meghívásakor fog szerepelni. (Egyébként lehetséges lett volna úgy is megoldani, hogy csak az egyik számot adjuk vissza így, pointerrel, a másikat pedig a return-nel.)

Következnek a számoló függvények.

```
//*******************************************************
void osszead(){
//*******************************************************
        double a,b;
        printf("\nOsszeadas\n");
        ket_double_beolvasasa(&a,&b);
        printf("\nAz osszeg: %g\n",a+b);
}
```

Létrehozunk két változót az összeg két összeadandója számára, bekérjük a két számot (látjuk az & operátorokat: a és b címét vesszük), majd kiírjuk az összeget.

Az összeg kiírására használt printf kicsit bővebb, mint eddig láttuk: a scanf-re hasonlít, itt is látunk benne egy %-os vezérlő szekvenciát, és a szöveg után vesszővel elválasztva, az összeadást.

A float vagy double típusú számok kiírására a %e, %f, %g szolgál, ezek kicsit más formátumban írnak ki. A %le, %lf, %lg pedig a long double számok kiírására szolgál. Ez egy logikátlanság, hiszen a scanf-nél láttuk, hogy ott ezeket a sima double számok beolvasására használtuk. Részletesebb ismertetés, lásd pl. [1].

A printf esetén a felsorolás is különbözik a scanf-től, hiszen itt nem azt kell megmondanunk, hogy hol van a memóriában, amit kiírni szeretnénk, hanem az, hogy mit akarunk kiírni, ezért nem pointert, hanem értéket adunk meg, tehát nem kell az & a változó neve elé, sőt, ahogy itt is látható, kifejezés is szerepelhet.

Nézzünk egy egy paramétert beolvasó függvényt:

```
//*******************************************************
void gyok(){
//*******************************************************
        double a;
        printf("\nGyokconas\n");
        a=egy_double_beolvasasa();
        printf("\nA negyzetgyok: %g\n",sqrt(a));
}
```

Ezúttal az érték az a=egy_double_beolvasasa(); módon kerül az a-ba. A négyzetgyököt pedig az sqrt() függvénnyel számítjuk ki, melynek prototípusa (deklarációja) a math.h-ban található (maga a függvény pedig valamelyik .lib fájlban, amit a linker szerkeszt a kész programhoz).

```
//*******************************************************
void osztok(){
//*******************************************************
        int a,i;
        printf("\nOsztok szamitasa\n");
        a=egy_int_beolvasasa();
        printf("\n%d osztoi:\n",a);
```

```
        for(i=1;i<=a/2;i++)
                if(a%i==0)printf("%d\t",i);
        printf("%d\n",a);
}
```

Az osztók számítását végző függvény az egyetlen, ahol valamiféle algoritmust használunk, melyet a for-ral kezdődő sorban találunk.

A for egy újabb ciklus típus (a while volt a másik). A () közötti rész három részre osztódik, ezeket ; választja el egymástól. Az i=1 a kezdeti értékadás, az i<=a/2 a feltétel (a ciklus addig fut, míg i értéke kisebb vagy egyenlő a felénél), a harmadik rész pedig i értékét 1-gyel növeli (i=i+1 is szerepelhetett volna, ugyanazt jelenti, csak így rövidebb). Egy for ciklus mindig átírható while ciklusra. Ez a ciklus while ciklussal így nézne ki:

```
        i=1;
        while(i<=a/2){
                if(a%i==0)printf("%d\t",i);
                i++;
        }
```

Például, ha a=12, akkor a ciklus 6-szor fut végig, miközben i értéke rendre 1, 2, 3, 4, 5, 6. Mikor i értéke eléri a 7-et, a feltétel hamissá válik, így a ciklus nem fut le többet.

A for ciklus magjában egyetlen utasítás található, az if, ezért nem kellett {} közé tenni. Az if feltételében ez szerepel: a%i==0, vagyis a és i osztásának maradéka egyenlő-e nullával, hiszen ekkor osztója i a-nak. Ha ez így van, akkor kiírjuk i-t.

1.5 Fordítás, futtatás, hibakeresés

Miután begépeltük a C nyelvű programot, abból futtatható programot kell készíteni (Microsoft operációs rendszerek alatt ezek .exe, Unix alatt .o kiterjesztésűek). A C nyelvben a programok gyakran több forrásfájlból állnak, az is előfordulhat, hogy ezeket a modulokat más-más ember készíti. A C nyelv így támogatja a csoportmunkát. Emiatt a felépítés miatt a futtatható állomány létrehozása két lépésből áll:

1. Fordítás: ekkor minden egyes .c (esetünkben .cpp) fájlból egy lefordított object fájl jön létre (általában .obj kiterjesztéssel).
2. Szerkesztés (linkelés): az object fájlokat, és a rendszerfüggvényeket tartalmazó .lib fájlokból a programban használt függvényeket összeszerkeszti, és ezzel létrehozza a futtatható programot.

Lehetőség van arra is, hogy több object fájlból mi magunk hozzunk létre függvénykönyvtárat, vagyis .lib fájlt, ezzel a kérdéssel azonban ebben a jegyzetben nem foglalkozunk.

Borland C++ esetén a fordítással összefüggő parancsok az IDE Compile menüjében találhatók. Az elemek funkciója a következő:

- Compile: lefordítja a szerkesztőben lévő, éppen aktív .C vagy .CPP fájlt, és létrehozza az .OBJ fájlt.
- Make: A Borland C++ támogatja a projectek kezelését. A project a programhoz tartozó forrásfájlok kapcsolatát írja le. Egy project fájl segítségével nem kell külön-külön lefordítanunk az egyes forrásállományokat, majd azokat „kézzel” összeszerkeszteni, mert ezt a Make parancs megteszi helyettünk. Ha nem kreáltunk külön projectet, csak egy egyszerű .C(PP) fájlunk van, akkor a Make előbb lefordítja, majd linkeli. Project esetén a Make csak azokat a fájlokat fordítja le, melyek megváltoztak a legutóbbi fordítás óta.
- Link: a lefordított obj. fájlokból létrehozza a futtatható programot.

- Build all: Működése hasonló a Make-hez, mindössze annyi a különbség, hogy akkor is újrafordítja a forrásfájlokat, ha azok nem változtak meg a legutóbbi fordítás óta.

Microsoft VC++ 6.0 esetén a fordítással összefüggő parancsok az IDE Build menüjében, illetve az eszköztáron is megtalálhatók (ha a Build eszköztár be van kapcsolva).

- Compile *valami*.cpp: lefordítja a *valami*.cpp fájlt, obj. fájlt hoz belőle létre.
- Build *Elso program*.exe: lefordítja azokat a forrásfájlokat, melyek módosultak a legutóbbi fordítás óta, majd a linker létrehozza az .exe-t. Gyakorlatilag ugyanazt csinálja, mint a Borland Make parancsa.
- Rebuild All: a forrásfájlokat akkor is újrafordítja, ha nem módosultak a legutóbbi fordítás óta. Ez például akkor lehet hasznos, ha időközben megváltoztattuk a fordítót. Például az Intel C++ Compiler feltelepítve beépül a Visual C++-ba, és ezután magunk választhatjuk ki, hogy az eredeti Microsoft fordítóval készüljön az adott menetben a program, vagy az Intelével (ezutóbbi segítségével jelentősen felgyorsíthatjuk a programunkat, mivel kihasználja az MMX, SSE, SSE2, SSE3 utasításkészleteket, és további jónéhány optimalizációs eljárást is tartalmaz, melyeket a Microsoft fordítója nem). Az Intel fordítójának időlimites, amúgy teljes értékű próbaváltozata regisztráció után letölthető a vállalat honlapjáról.

Mind Borland, mind Visual C++ esetében számos lehetőség áll rendelkezésre, melyben a lefordított program tulajdonságait állíthatjuk be. Ezek a lehetőségek a BC++ esetében az Options menüben vannak, például az Options/Compiler/Advanced Code Generation-t választva kiválaszthatjuk, hogy a fordító használja a 386-os processzor utasításkészletét, és a 387-es matematikai koprocesszor utasításkészletét (a matematikai koprocesszor a Pentium óta minden processzor része, tehát nincs értelme ennél alább adni.

Visual C++ esetén a Build menü Set Active Configuration pontjában két alapértelmezett konfiguráció közül választhatunk. Létrehozhatunk többet is, de a két alapértelmezettnél többre csak a legritkább esetben van szükség. A két konfiguráció a következő:

- Debug: minden optimalizáció ki van kapcsolva, az .exe fájlba belekerülnek a debuggolást segítő kódok is. Emiatt ez így kapott exe nagy és lassú lesz.
- Release: a véglegesnek szánt változat, optimalizációval és a debuggolást segítő kódok nélkül: release állásban nem tudunk debuggolni, csak ha megváltoztatjuk a konfigurációt, de akkor már inkább a debug módot használjuk.

A konfigurációkhoz tartozó beállításokat a Project menü Settings pontjában állíthatjuk be. Itt teljesen átszabhatjuk a beállításokat, akár olyannyira, hogy Release üzemmódban legyen olyan, mint alapértelmezés szerint Debug módban, és fordítva. Ha a C/C++ fülön kattintunk, és a Category legördülő listából kiválasztjuk a Code Generation-t, akkor a Processor menüben a 386-ostól a Pentium Pro-ig áll rendelkezésünkre a választási lehetőség. Aki ennél többet szeretne, annak ott az Intel Compiler, jó pénzért.

Amennyiben a programunk szintaktikai hibát tartalmazott, tehát valamit rosszul írtunk, esetleg lefelejtettünk egy zárójelet vagy pontosvesszőt, az erre vonatkozó hibaüzenetek a képernyő alján jelennek meg. A hibaüzenetre kattintva a kurzor arra a sorra ugrik, ahol a hibát elkövettük a fordító szerint, de előfordulhat, hogy a hiba az előző sorban volt, pl. ott hagytuk le a pontosvesszőt (ez a jelenség inkább a Borland fordítóját érinti). **Ha több hibát talál a fordító, akkor mindig fölülről lefelé haladjunk ezek kijavításában**, mert gyakran előfordul, hogy a lejjebb leírt hibák nem is hibák, hanem csak egy előbb lévő hiba következtében mondja azt a fordító. Ilyen például akkor fordul elő, ha lehagyunk egy zárójelet.

Ha sikerült lefordítani a programot, az még nem jelenti azt, hogy helyesen is működik. Vannak olyan esetek, amikor a program szintaktikailag helyes, tehát le lehet fordítani, de a fordító talál olyan gyanús részeket, ahol hiba lehet. Ilyen esetekben figyelmeztetést (warning) ír ki. A warningokat is nézzük át, és csak akkor hagyjuk figyelmen kívül, ha biztosak vagyunk benne, hogy jó, amit írtunk.

A kezdő programozónak gyakran a szintaktikai hibák kijavítása is nehéz feladat, de ők is rá fognak jönni, hogy a szemantikai hibák (bugok) felderítése sokkal nehezebb, hiszen itt nem áll rendelkezésre a fordító segítsége, nem mondja meg, hogy itt és itt van a hiba, hanem a rendellenes működésből erre nekünk kell rájönnünk. A fejlesztőkörnyezet azonban nem hagy ilyen esetekben sem eszközök nélkül.

A legegyszerűbb eszköz a hibakezelésre, ha soronként futtatjuk végig a programot. Ehhez Borland C++ esetén a Run/Step over ill. Run/Trace into,

Visual C++ esetén a Build menü Start Debug almenüjében pl. a Step Into-t választva megjelenik a Debug menü (és a debug eszköztár), itt megtaláljuk a Step Over és a Step Into parancsot is.

A Step/Trace Into és Over között az a különbség, hogy amennyiben egy adott programsorban függvényhívás szerepel, az Into paranccsal belépünk a függvénybe, és annak sorain is végigmehetünk. Ezzel szemben az Over parancs az egész függvényt egy utasításnak tekinti, és egyben végrehajtja.

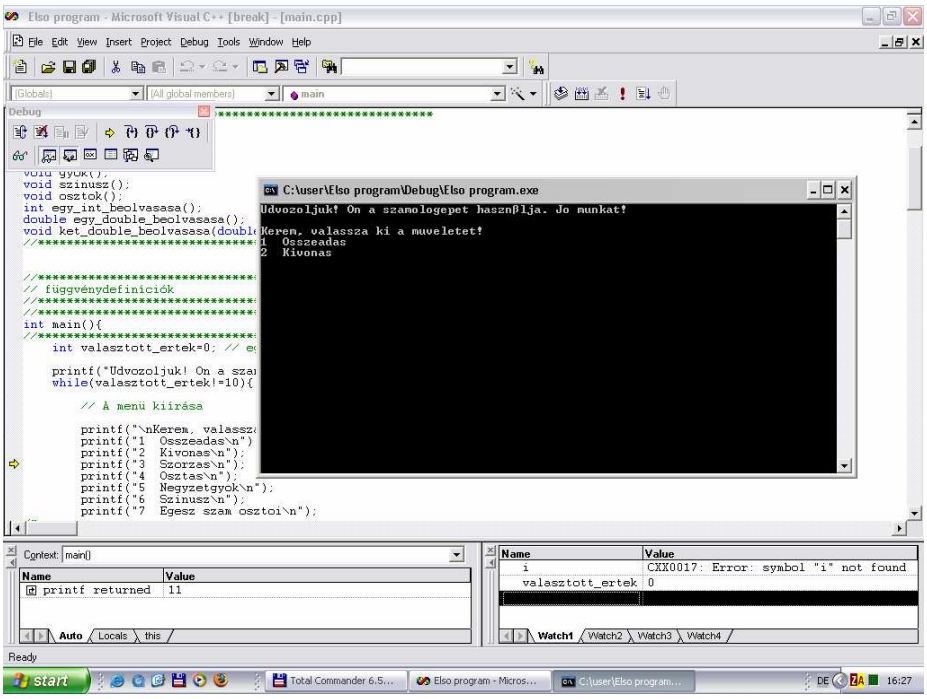
BC++-nál egy-egy sor lefutása után megnézhetjük a kimeneti képernyőt, ha lenyomjuk az Alt+F5 kombinációt. Innen az any key :-) lenyomásával térhetünk vissza a programhoz.

További fontos parancs a Go to cursor (BC) ill. Run To Cursor (VC). Ha valamelyik programsorra beállítjuk a kurzort, és kiadjuk ezt a parancsot, akkor a program normálisan fog futni egész addig a sorig, ahol megáll, és onnantól léptethetjük tovább Step Intoval, ill. Step Overrel.

Az eddigi parancsokkal csak azt vizsgálhattuk, hogy mely utasítások hajtódnak végre, és melyek nem, azonban ennél is több lehetőségünk van: megnézhetjük a változók értékét, és folyamatosan figyelemmel kísérhetjük azt.

Borlandnál válasszuk a Debug/Watches/Add watch pontot, és írjuk be mondjuk azt, hogy valasztott_ertek. Alul megjelenik egy kis ablak, benne a változó nevével, és ha a program egy olyan részén lépkedünk, ahol a változó létezik, akkor megjelenik az aktuális értéke.

Visualnál miután Debug üzemmódba kerültünk (pl. a Step Into-val, vagy a Run To Cursorral), alul megjelenik egy kettéosztott ablak. Bal oldalon a fejelszőkörnyezet automatikusan kiválasztja azokat a változókat, melyeket fontosnak tart, és eltűnnek, mikor úgy dönt, hogy már nem fontosak. Jobb oldalon (alul Watch1 címkével) pedig mi magunk írhatunk be változóneveket:



6. ábra

A fenti ábrán épp a harmadik printf előtt áll a kurzor, ezt mutatja a sárga nyilacska. Automatikus érték az előző printf visszatérési értéke (a kiírt karakterek száma), az általunk választott változók az i és a valasztott_ertek. A main függvényben nem használunk i nevű változót, ezt jelzi a hibaüzenet, a valasztott_ertek most a kezdeti 0 értéket tartalmazza.

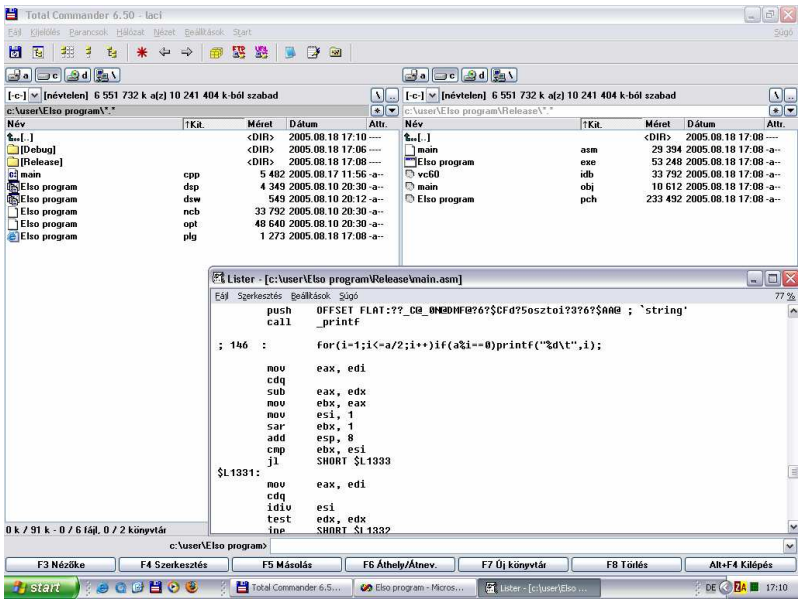
Nem csak változók, hanem kifejezések is vizsgálhatók, például t[23], *yp vagy a+b is.

A Debuggolás a Run/Program reset (BC) ill. Debug/Stop Debugging (VC) paranccsal megszakítható, vagy a program normálisan futtatható tovább a Run/Run ill. Go paranccsal.

Következő eszközünk a hibakeresésre a töréspontok behelyezése. A töréspont (breakpoint) behelyezéséhez állítsuk a kurzort a kívánt sorra, majd BC esetén Debug/Toggle Breakpoint, VC esetén jobb klikk a soron, és Add/Remove Breakpoint. Ezután BC-nél Debug/Breakpoints…/Edit, ill. VC-nél Edit/Breakpoints…/Condition választása esetén beállíthatjuk, hogy milyen feltétel teljesülése esetén álljon meg a program az adott sornál (pl. i==10), hányszor menjen végig a soron megállás nélkül, miközben a feltétel igaz (ha a feltételt üresen hagyjuk, akkor azt nézi, hogy összesen hányszor ment már végig a soron, mielőtt megállna, ha az ismétlésszámot hagyjuk üresen, akkor az első alkalommal megáll itt, amikor a feltétel teljesül, ha mindkettőt üresen hagyjuk, akkor pedig mindig megáll itt, amikor ideér a program: ez olyan, mintha a Run To Cursort választottuk volna).

Feladat: lépkedjünk végig a programon a Step Into, a Step Over, a Run To Cursor utasításokat kipróbálva. Próbáljuk ki a töréspontok behelyezését is!

Lehetőség van arra, hogy a fordítóprogramok Assembly nyelvű kódot generáljanak (BC esetén Options/Compiler/Code generation/Options/Generate assembler source, VC esetén Project/Settings/(C/C++)/Category/Listing Files/Listing file type/Assembly with Source Code. Ez azoknak jelent segítséget, akik ismerik valamennyire az Assembly nyelvet.



7. ábra

Fájlkezeléshez rendkívül hasznos eszköz a Total Commander, sokkal hatékonyabban lehet vele dolgozni, mint a „My Computer”-rel. Számtalan hasznos funkciója közül említést érdemel, hogy két fájlt könnyedén össze tudunk hasonlítani.

A Borland C++ fordító csak azokat a fájlokat generálja, melyekről eddig szó volt. Ezzel szemben a Visual C++ rengeteg olyat is létrehoz, melyek a fordító munkáját könnyítik meg. Mi kell ebből nekünk, és mi nem?

Nos: a Debug és a Release mappát minden szívfájdalom nélkül törölhetjük. Ha kell az exe, akkor azt azért előbb másoljuk át valahova. Amikor legközelebb újrafordítjuk a programunkat, ezek automatikusan ismét létrejönnek.

A projekt főkönyvtárából a .cpp és .h fájlokra van szükségünk. Ha a többit letöröljük, akkor legközelebb ismét létre kell hoznunk a konzol alkalmazás projektet az 1.2 pontban bemutatottak szerint, majd a projekthez hozzá kell adnunk a .cpp és .h fájlokat.

Ha tehát haza akarjuk küldeni napi munkánkat, akkor célszerű először letörölni a Debug és Release mappát, majd a maradékot becsomagolni (Total Commanderrel Alt+F5), és ezt csatolni a levélhez. Ha nagyon szűkös a sávszélességünk, akkor csak a .cpp és .h fájlokat tömörítsük.

Feladat: Egészítsük ki a programot a logaritmus és az exponenciális függvény használatának lehetőségével. Ehhez:

a. **Töröljük ki a /**/ párosokat a megfelelő printf és case részekről!**
b. **Hozzuk létre a logarit() és az exponen() függvények prototípusait!**
c. **Hozzuk létre a logarit() és az exponen() függvényeket a gyok() vagy a szinusz() függvények Copy+Paste-jével, és módosítsuk úgy hogy a függvények a logaritmus ill. az exponenciális számításnak megfelelően működjenek! (A logaritmus kiszámítására a log(), az exponenciális kiszámítására az exp() függvény használható.**

2. Beolvasás és kiírás, típusok, tömbök

Az előző fejezetben találkoztunk már a két legfontosabb függvénnyel, melyekkel a beolvasást és a képernyőre írást megoldják, ezek voltak a scanf() és a printf(). Ezeket használjuk leggyakrabban, most részletesebben is megismerkedünk velük. Megismerünk néhány további beviteli (input) és kiviteli (output) függvénnyel is.

A fejezetben megismerkedünk a C nyelv adattípusaival, és az első összetett adatszerkezettel, a tömbbel.

2.1 Programértés

A „programértés” az idegennyelv tanulásnál ismert „szövegértés” megfelelője. Ebben a részben tehát a program megértése, lefuttatása, kipróbálása a feladat.

```
//*******************************************************
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>      // egész típusok értékkészlete
#include <float.h>       // a valos tipusok tulajdonsagai
//*******************************************************


//*******************************************************
void egesz();
void valos();
void string();
void tomb();
//*******************************************************


//*******************************************************
int main(){
//*******************************************************
        int valasztott_ertek=0;  // egész típusú változó 0 kezdőértékkel

        printf("I/O, tipusok, tombok\n"); // kiírás
        while(valasztott_ertek!=10){ // ciklus, míg a v.é. nem egyenlő 10-zel

                // A menü kiírása

                printf("\nKerem, valassza ki a muveletet!\n");
                printf("1  Egesz tipusok\n");
                printf("2  Valos tipusok\n");
                printf("3  Stringek\n");
                printf("4  Tombok\n");
                printf("10 Kilepes\n");

                // A választott érték beolvasása

                fflush(stdin); // standard input puffer ürítése
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                // A választott művelet végrehajtása

                switch(valasztott_ertek){ // a v.é.-nek megfeleő case után folytatja
                        case  1: egesz();        break; // az osszead függvény hívása
                        case  2: valos();        break;
                        case  3: string();       break;
                        case  4: tomb();         break;
                        case 10:                 break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
        printf("\nTovabbi jo munkat!\n");
        return 0;
}


//*******************************************************
void egesz(){
// Egész szám I/O
//*******************************************************
```

```
char c1;
unsigned char c2;
signed char c3;

int i1;
unsigned i2;              // vagy unsigned int i2;

short s1;                 // vagy short int l1;
unsigned short s2;        // vagy unsigned short int l2;

long l1;                  // vagy long int l1;
unsigned long l2;         // vagy unsigned long int l2;

// char család

printf("*********************************************************************************\n");
printf("\nchar min=\t%d\nchar max=\t%d\n",CHAR_MIN,CHAR_MAX);
printf("\nunsigned char min=\t0\nnunsigned char max=\t%d\n",UCHAR_MAX); // a min mindig 0
printf("\nsigned char min=\t%d\nsigned char max=\t%d\n",SCHAR_MIN,SCHAR_MAX);

printf("\nA scanf karakterkent es nem szamkent kezeli a char tipust.\n");
printf("Adjunk meg egy karaktert, majd nyomjuk meg az ENTER-t!\n");
fflush(stdin);
if(scanf("%c",&c1)!=1)printf("\nSikertelen beolvasas\n");

printf("A(z) %c karakter szamkent=\t%d\tvagy\t%u\n",c1,c1,c1);
c2=(unsigned char)c1;
c3=(signed char)c1;
printf("A(z) %c karakter elojel nekluli karakterre konvertalva=\t%u\n",c1,c2);
printf("A(z) %c karakter elojeles karakterre konvertalva=\t%d\n",c1,c3);

// int család

printf("*********************************************************************************\n");
printf("\nint min=\t%d\nint max=\t%d\n",INT_MIN,INT_MAX);
printf("\nunsigned int min=\t0\nnunsigned int max=\t%u\n",UINT_MAX); // a min mindig 0
printf("\nA signed int ugyanaz, mint az int\n");

printf("\nAdjunk meg egy egesz szamot az int ertekkeszletben!\n");
fflush(stdin);
if(scanf("%d",&i1)!=1)printf("\nSikertelen beolvasas\n");
i2=(unsigned)i1;
printf("%i elojel nelkulive konvertalva=\t%u\n",i1,i2);

printf("\nAdjunk meg egy nemnegativ egesz szamot az unsigned ertekkeszletben!\n");
fflush(stdin);
if(scanf("%u",&i2)!=1)printf("\nSikertelen beolvasas\n");
i1=(int)i2;
printf("%u elojelesse konvertalva=\t%d\n",i2,i1);

// short család

printf("*********************************************************************************\n");
printf("\nshort min=\t%hd\nshort max=\t%hd\n",SHRT_MIN,SHRT_MAX);
printf("\nunsigned short min=\t0\nnunsigned short max=\t%hu\n",USHRT_MAX); // a min mindig 0
printf("\nA signed short ugyanaz, mint az short\n");

printf("\nAdjunk meg egy egesz szamot a short ertekkeszletben!\n");
fflush(stdin);
if(scanf("%hd",&s1)!=1)printf("\nSikertelen beolvasas\n");
s2=(unsigned)s1;
printf("%hi elojel nelkulive konvertalva=\t%hu\n",s1,s2);

printf("\nAdjunk meg egy nemnegativ egesz szamot az unsigned short ertekkeszletben!\n");
fflush(stdin);
if(scanf("%hu",&s2)!=1)printf("\nSikertelen beolvasas\n");
s1=(int)s2;
printf("%hu elojelesse konvertalva=\t%hd\n",s2,s1);

// long család

printf("*********************************************************************************\n");
printf("\nlong min=\t%ld\nlong max=\t%ld\n",LONG_MIN,LONG_MAX);
printf("\nunsigned long min=\t0\nnunsigned long max=\t%lu\n",ULONG_MAX); // a min mindig 0
printf("\nA signed long ugyanaz, mint az long\n");

printf("\nAdjunk meg egy egesz szamot a long ertekkeszletben!\n");
fflush(stdin);
if(scanf("%ld",&l1)!=1)printf("\nSikertelen beolvasas\n");
l2=(unsigned)l1;
printf("%li elojel nelkulive konvertalva=\t%lu\n",l1,l2);

printf("\nAdjunk meg egy nemnegativ egesz szamot az unsigned long ertekkeszletben!\n");
fflush(stdin);
if(scanf("%lu",&l2)!=1)printf("\nSikertelen beolvasas\n");
l1=(int)l2;
printf("%lu elojelesse konvertalva=\t%ld\n",l2,l1);
printf("*********************************************************************************\n");
```

```
}


//*******************************************************
void valos(){
// Valós (lebegőpontos) szám I/O
//*******************************************************
        float f,f2;
        double d,d2;
        long double l,l2;

        // float

        printf("*********************************************************************************\n");
        printf("\nA legkisebb float pozitic ertek\t\t%g\n",FLT_MIN);
        printf("A legnagyobb float pozitic ertek\t%g\n",FLT_MAX);

        printf("\nAdjunk meg valos szamot!\n");
        fflush(stdin);
        if(scanf("%g",&f)!=1)printf("\nSikertelen beolvasas\n"); // %e, %f, %g egyaránt használható
        printf("A szam harom alakban\t%e\t%f\t%g\n",f,f,f);

        // double

        printf("*********************************************************************************\n");
        printf("\nA legkisebb double pozitic ertek\t%g\n",DBL_MIN);
        printf("A legnagyobb double pozitic ertek\t%g\n",DBL_MAX);

        printf("\nAdjunk meg valos szamot!\n");
        fflush(stdin);
        if(scanf("%lg",&d)!=1)printf("\nSikertelen beolvasas\n"); // %le, %lf, %lg egyaránt használható
        printf("A szam harom alakban\t%e\t%f\t%g\n",d,d,d);

        // long double

        printf("*********************************************************************************\n");
        printf("\nA legkisebb long double pozitic ertek\t%g\n",LDBL_MIN);
        printf("A legnagyobb long double pozitic ertek\t%g\n",LDBL_MAX);

        printf("\nAdjunk meg valos szamot!\n");
        fflush(stdin);
        if(scanf("%Lg",&l)!=1)printf("\nSikertelen beolvasas\n"); // %le, %lf, %lg egyaránt használható
        printf("A szam harom alakban\t%Le\t%Lf\t%Lg\n",l,l,l);

        // A "valós" számok diszkrét értékkészlete

        printf("*********************************************************************************\n");
        for(f=1.0,f2=0.0;f!=f2;f*=1.2F,f2=f+1.0F);
        printf("Ket float ertek kozott >=1 a kulonbseg elelett az ertek felet: %g\n",f);
        for(d=1.0,d2=0.0;d!=d2;d*=1.2,d2=d+1.0);
        printf("Ket double ertek kozott >=1 a kulonbseg elelett az ertek felet: %g\n",d);
        for(l=1.0,l2=0.0;l!=l2;l*=1.2L,l2=l+1.0L);
        printf("Ket long double ertek kozott >=1 a kulonbseg elelett az ertek felet: %Lg\n",l);
        printf("*********************************************************************************\n");
}


//*******************************************************
void string(){
// String I/O
//*******************************************************
        char t[100],c;

        printf("*********************************************************************************\n");
        printf("Irjon be egy szoveget!\n");
        fflush(stdin);
        if(scanf("%s",t)!=1)printf("\nSikertelen beolvasas\n");// Nincs & a t neve előtt
        printf("\nEzt irta: %s\n",t);
        printf("Ha volt benne szokoz, csak az elso szot olvasta be.\n");

        printf("\nIrjon be meg egy szoveget!\n");
        fflush(stdin);
        if(gets(t)==NULL)printf("\nSikertelen beolvasas\n");
        puts("\nEzt irta: ");
        puts(t);
        puts("\nMost az egesz sort beolvasta\n");

        printf("\nIrjon be egy karaktert(vagy tobbet), aztan ENTER!\n");
        fflush(stdin);
        if((c=getchar())==EOF)printf("\nSikertelen beolvasas\n");
        printf("Ezt irta: ");
        putchar(c);
        putchar('\n');
        printf("*********************************************************************************\n");
}


//*******************************************************
```

```
void tomb(){
// Tömb I/O
//*****************************************************
        double d[20];
        int i,j;

        printf("*********************************************************************************\n");
        printf("Adjon meg max. 20 szamot! Ha befejezte a szamok magadasat, irjon 0-t!\n");
        for(i=0;i<20;i++){
                d[i]=0.0;
                fflush(stdin);
                printf("%02d. szam= ",i+1);
                if(scanf("%lg",&d[i])!=1){
                        printf("\nNem szamot adatt, probalja meg ujbol!\n");
                        i--; // Ismét ugyanabba a tömbelembe olvasson, ahová most nem sikerült
                        continue; // a ciklus további részét kihagyja
                }
                if(d[i]==0.0)break; // ha 0-t írtunk, megáll
        }

        printf("Ezeket a szamokat irta:\n");
        for(j=0;j<i;j++)printf("%02d. szam= %g\n",j+1,d[j]);
        system("PAUSE"); // nem szabványos, UNIX-ban nem megy
        printf("A szamok forditott sorrendben:\n");
        for(j=i-1;j>=0;j--)printf("%02d. szam= %g\n",j+1,d[j]);
        printf("*********************************************************************************\n");
        system("PAUSE"); // nem szabványos, UNIX-ban nem megy
}
```

2.1.1 Egész típusok

A C nyelv a következő egész szám típusokat ismeri:

- char: mérete az adott számítógép legkisebb adatblokkja, tipikusan egy byte. Fontos tudni róla, hogy mivel döntően szövegfeldolgozásra használjuk, nincs definiálva, hogy előjeles vagy előjel nélküli. A fordítón múlik, hogy melyik. Ha ez fontos, használjunk a következő típusokat:
  - signed char: előjeles char (értékkészlet általában -128 – +127-ig)
  - unsigned char: előjel nélküli char (értékkészlet általában 0-255-ig)
- int: a számítógép számára optimális méretű előjeles egész szám. Tipikusan 16 vagy 32 bites.
  - signed: más néven signed int ugyanaz, mint az int
  - unsigned: más néven unsigned int, előjel nélküli int
- short: más néven short int, előjeles egész, mérete ≤ int mérete. Tipikusan 16 bites.
  - signed short: ugyanaz, mint a short
  - unsigned short: előjel nélküli short
- long: más néven long int, előjeles egész, mérete ≥ int mérete. Tipikusan 32 bites.
  - signed long: ugyanaz, mint a long
  - unsigned long: előjel nélküli long

Az adott típusban tárolható legkisebb és legnagyobb értéket a limits.h tartalmazza, pl.:

```
printf("\nchar min=\t%d\nchar max=\t%d\n",CHAR_MIN,CHAR_MAX);
printf("\nunsigned char min=\t0\nunsigned char max=\t%d\n",UCHAR_MAX); // a min mindig 0
printf("\nsigned char min=\t%d\nsigned char max=\t%d\n",SCHAR_MIN,SCHAR_MAX);
```

Itt a char típusra jellemző konstansokat láthatjuk (CHAR_MIN, CHAR_MAX, stb.).

Char típusok esetén ugyanúgy végezhetünk matematikai műveleteket, mint a többi egész esetén, de figyeljünk arra, hogy a char típus értékkészlete igen kicsi, nem erre találták ki, hanem, ahogy a neve is mutatja, karakterek tárolására. Figyeljük meg a char kettős természetét: van, amikor számként, és van amikor karakterként kezeljük. Futtassuk a fenti programot, válasszuk az 1-es opciót, és amikor azt kéri, hogy adjunk meg egy karaktert, adjuk meg a 0-t. Azt fogja

válaszolni, hogy **a 0 karaktert a 48-as szám tárolja**. Mi ennek az oka? Minden számítógép azokat a karaktereket, melyeket például ebben a szövegben is olvashatunk, egy számmal tárolja. Hogy melyik karakterhez hányas tartozik azt az ASCII szabvány rögzíti. Eszerint a '0'-t a 48, az '1'-et a 49, a ' ' szóközt a 32, az 'A'-t a 65, az 'a'-t a 97 jelenti.

Egy karaktert beolvasni vagy kiírni a %c suffixszel lehet a scanf ill. printf függvényben. Karakterbeolvasásra ill. kiírásra szolgál a getchar ill. putchar függvény, lásd 2.1.3

Típuskonverzió:

```
c2=(unsigned char)c1;
c3=(signed char)c1;
```

Ezt írhattuk volna így is:

```
c2=c1;
c3=c1;
```

A másodikat **implicit típuskonverziónak** nevezzük, mert nincs odaírva, hogy milyen típusra szeretnénk, az elsőt pedig **explicit típuskonverziónak**, mert oda van írva a céltípus. Így nem csak karakterről karakterre, hanem bármely más egész, vagy lebegőpontos típusra, ill. -ról tudunk konvertálni. Ha nem írjuk oda zárójelben a céltípust, bizonyos esetekben a fordító warninggal fogja jelezni, hogy az adott irányú konverzió veszteséggel járhat, hiszen ha pl. az int változó értéke 1000, akkor ez az érték nem fog beleférni a charba. Ekkor is bele fog tenni valamit, de nyilván nem 1000-et. Ha odaírjuk a zárójeles részt, akkor nem kapunk warningot, de az értékadás ugyanúgy hibás lesz, ha a céltípusba nem fér bele a forrás. Ha valós számról konvertálunk egészre, akkor kerekítés történik.

A függvény következő része az int típussal foglalkozik. Ez a rész (és a short ill. longgal foglalkozó rész) hasonló a charhoz, ezért külön nem térünk ki rá, de a kódot mindenki nézze meg!

Viszont itt kell kitérnünk arra, hogy hogyan is lesz a karakterből szám. Amikor pl. az int típusnál a %d (vagy %i, teljesen mindegy) szuffixszel beolvasunk egy egész számot, akkor a begépelt karakterekből elő kell állítani az egész számot. Például beírjuk, hogy 123, és megnyomjuk az ENTER-t. a scanf megszámolja, hogy hány karakterből áll a szám, aztán így konvertálja: (első_karakter-48)*100+(második_karakter-48)*10+(harmadik_karakter-48). Azért 48, mert, ahogy az előbb láttuk, a '0' karakter kódja 48, az '1'-é 49, és így tovább. Printf esetében pont fordítva jár el, ott az egész számból kell karaktereket készíteni.

2.1.2 Lebegőpontos számok

A C nyelv három lebegőpontos szám típust használ:

- float: egyszeres pontosságú lebegőpontos szám, tipikusan 32 bites. Csak akkor használjuk, ha nagyon muszáj, mert pontatlan.
- double: dupla pontosságú lebegőpontos szám, tipikusan 64 bites, a legtöbb esetben megfelelő pontosságú.
- long double: még nagyobb pontosságú lebegőpontos szám, az újabb fordítókban ez is 64 bites, és csak külön kérésre 80 bites, mert a 80 bites méret memória-hozzáférés szempontjából nem ideális.

A **lebegőpontos** azt jelenti, hogy a tizedespont nem marad a helyén, hanem előre ugrik az első értékes számjegy után, és az egészet megszorozzuk egy kitevővel. Például a 6847.12 ebben a formában fixpontos, 6.84712×10^3 formában pedig lebegőpontos. Ezutóbbi számot C-ben így írhatjuk: 6.84712e3, esetleg: 6.84712e+003. Nagy E-t is használhatunk. A lebegőpontos számokat a gép természetesen kettes számrendszerben tárolja, így ez a szám $1.10101011111100011110101\times2^{12}$, ha floatban tároljuk. Float esetén ugyanis 23 bit áll az

értékes számjegyek tárolására (mantissza) + egy előjelbit, a pont előtt álló 1-est, és a pontot nem kell tárolni, a kitevő pedig -128 és +127 közötti (ez 8 bit). Double esetén a mantissza 51+1 bites, a kitevő (karakterisztika) pedig 12 bit.

A példaprogramban láthatjuk azt a furcsaságot, hogy double típus esetén a scanf és a printf eltérő suffixet használ: printf-nél a float és a double egyaránt %e, %f és %g-vel írandó ki, míg beolvasni a double-t %le, %lf és %lg-vel kell. A fordítóprogramok el szokták fogadni kiírásnál is a %le stb. változatot.

A három kiírásmód jelentése:

- %e: kitevős alak, pl. 6.84712e+003
- %f: fixpontos alak, pl.: 6847.12. Nullához közeli számoknál 0.00000-t ír ki, de nagy számoknál ez is kitevős alakot használ: 2.45868e+066. Ez egész számoknál is kiteszi a pontot, pl. 150.0000.
- %g: a kiírás alkalmazkodik a számhoz. Nullához közel és távol kitevős alak: 8.754e-019. A kettő között fixpontos alak, pl. 6847.12. Az egész számokat egészként írja, pl. 150.

A következő programrészlet egy igen fontos dologra hívja fel a figyelmünket:

```
// A "valós" számok diszkrét értékkészlete

for(f=1.0,f2=0.0;f!=f2;f*=1.2F,f2=f+1.0F);
printf("Ket float ertek kozott >=1 a kulonbseg elelett az ertek felet: %g\n",f);
for(d=1.0,d2=0.0;d!=d2;d*=1.2,d2=d+1.0);
printf("Ket double ertek kozott >=1 a kulonbseg elelett az ertek felet: %g\n",d);
for(l=1.0,l2=0.0;l!=l2;l*=1.2L,l2=l+1.0L);
printf("Ket long double ertek kozott >=1 a kulonbseg elelett az ertek felet: %Lg\n",l);
```

Bár a könnyebb érthetőség érdekében gyakran mondjuk a számítógép által használt lebegőpontos számra, hogy „valós”, ez nem igaz, ugyanis csak a racionális számok egy töredékét tárolhatják, hiszen véges számú bitet használunk. Ez az esetek nagy részében nem jelent problémát, azonban mindig figyelemmel kell lennünk erre. A fenti programrészlet megmutatja, hogy körülbelül hol van az a legkisebb érték, amihez 1-et hozzáadva nem változik meg a szám, mert két tárolható érték között túl naggyá válik a távolság. Egy egyszerű 10-es számrendszerbeli példa a következő: ha pl. 1.2345×10^6 ilyen pontosságban tárolható. Adjunk hozzá egyet: $1.2345\times10^6=123450$ => +1 => 123451 => 1.2345×10^6. Tehát eltűnt a hozzáadott 1, ugyanazt kaptuk vissza.

Ez például olyan esetben jelenthet gondot, ha egy olyan gyökkereső algoritmust futtatunk, amelyik két oldalról közelíti meg a gyököt, és mondjuk az a leállás feltétele, hogy a két közelítés különbsége kisebb legyen, mint mondjuk 1e-5. Ha lefuttatjuk a fenti programot, láthatjuk, hogy float esetén már 2e+007 előtt 1 lesz két tárolható szám között a különbség, double esetén ez 9.8e+015 (long double esetén VC++ 7.0-val ugyanennyi).

Módosítsuk a fenti programot, hogy azt írja ki, hogy mennyinél lesz 1e-5-nél nagyobb a különbség!

A módosítás után nekem az jött ki, hogy float esetén 164.845 esetén már ennél nagyobb volt a különbség! Tehát ennél nagyobb gyökök esetén kiakadhat a gyökkereső program! Ezért ne használjunk floatot! A double esetén 1.46e+011 a határ, ami már elegendően nagy érték, hogy ne legyen gondunk.

Hogy működik ez a for ciklus? Ez egy kicsit bonyolultabb, mint az eddigiek, ezért nézzük meg külön! A pontosvesszőket figyeljük, abból továbbra is kettő van a zárójelen belül (minden for ciklusban kettő, és csak kettő van).

Az első pontosvessző előtt van a kezdeti inicializálás, ahol ezúttal két változónak is értéket adunk. Ez a rész összesen egyszer fut le, a tényleges ciklus előtt.

A két pontosvessző között van a feltétel. A ciklus addig fut, míg ez igaz, vagyis a szám és a számnál eggyel nagyobb szám nem egyforma.

A második pontosvessző utáni rész mindig a ciklusmag után fut le (a ciklusmag ezúttal üres, mert a) után csak egy ; áll). Itt előbb az egyik változó értékét 1,2-vel szorozzuk, a másik változó pedig eggyel nagyobb értéket kap, mint az első. Az f*=1.2F ugyanazt jelenti, mintha azt írtuk volna: f=f*1.2F. A konstans után álló F betű azt jelzi, hogy float típusról van szó, ha nem írunk ilyet, akkor double, ha L-et írunk, long double (f ill. l is használható). Egész számok esetén az L a long intet, U az unsigned intet jelenti. Nyilván UL vagy LU az unsigned long. A shortnak nincs suffixe. A szabványról bővebben pl. [2].

2.1.3 Stringek

Már a kezdet kezdetétől használunk stringeket, azaz inkább **string konstans**okat. A string konstansok "" közé zárt szövegek, például azok, amiket a printf-ben használunk. A stringek (nem csak a konstansok) a következőképp helyezkednek el a memóriában: (pl. a "Jo munkat!\n"):

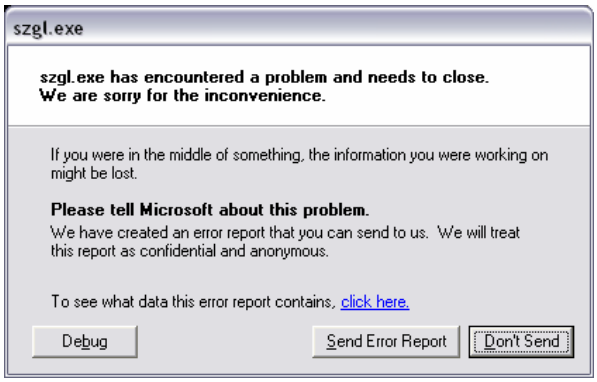
'J'	'o'	' '	'm'	'u'	'n'	'k'	'a'	't'	'!'	'\n'	'\0'

Itt minden egyes cella egy bájtot, vagyis egy char típusú változót jelent a memóriában. Figyeljük meg, hogy az egyes karaktereket gondolatjelek közé zártuk. C-ben így jelöljük a **karakter konstans**okat.

A string végén szerepel egy '\0', ami a 0 számot jelöli (emlékezzünk, hogy a '0' karakter, melyhez a 48-as szám tartozik, a 0-s számhoz viszont a '\0' karakter). Ezzel jelezzük, hogy itt van a string vége. A későbbiekben mi is fogunk olyan függvényeket készíteni, amelyek stringeket dolgoznak fel, és addig mennek, míg egy '\0'-ba nem ütköznek. Emiatt **a '\0' miatt a stringeknek mindig 1 karakterrel több hely kell, mint a string tényleges hossza!**

Mit tegyünk akkor, ha nem konstans stringre van szükség, hanem például a felhasználótól szeretnénk megkérdezni, mondjuk a nevét? Ehhez először is szükség van egy változóra, ahol tárolhatjuk. Ennek a változónak ugyanolyan formátumúnak kell lennie, mint ahogy a „Jo munkat!\n" esetében látjuk, vagyis sok egymás mellett álló karakterre van szükségünk, azaz egy **karakter tömb**re. Így adhatunk meg egy 100 elemű karakter tömböt: `char t[100];`

Ebben a tömbben tehát maximum 100 karakter fér el, azaz egy 99 karakterből álló string+ a lezáró '\0'. Kisebb string is lehet benne, legfeljebb a végét nem használjuk ki. Figyeljünk azonban arra, hogy akkora tömbméretet adjunk meg, amelybe biztosan belefér a string, mert ha túl kicsire csináljuk a tömböt, és többet írunk bele, akkor a program ezt nem fogja észrevenni, és jó esetben ezt a kedves üzenetet fogjuk megkapni:



8. ábra

Itt célszerű a Don't Sendet választani, legalábbis ha nem az a célunk, hogy a Microsoftot bosszantsunk, hiszen ők nem sokat tudnak kezdeni egy olyan program hibájával, amit mi magunk írtunk ☺.

Tehát jó esetben ezt az üzenetet kapjuk, rossz esetben a túlírás megmarad a programunk saját memóriaterületén, így ezt az operációs rendszer nem veszi észre, viszont más változóink adatai tönkremennek, és az ilyen rejtélyes hibák felderítése nehéz feladat.

A string beolvasása scanf-fel:

```
scanf("%s",t)
```

Figyeljük meg, hogy a t elé nem írtuk az & operátort! **A tömb neve (nem csak karaktertömbé, bármilyené) a [] nélkül a tömb 0. elemére mutató pointer.** A scanf pedig pont egy karaktertömb 0. elemére mutató pointert vár. (A C-ben a tömbök elemeinek sorszámozása 0-tól n-1-ig megy, ahol az n az elemszám, jelen esetben 100.) Természetesen megadhatjuk közvetlenül is a 0. elemre mutató pointert, ez ugyanazt jelenti:

```
scanf("%s",&t[0])
```

A scanf mellett használhatjuk a gets függvényt is szövegbevitelre. A scanf hátránya, hogy csak az első szóközig olvas, tehát ha valakitől a nevét szeretnénk megtudni, akkor a scanf-fel bajban lennénk, még akkor is, ha kétszer egymás után hívjuk, hiszen van akinek egy keresztneve van, van akinek kettő, de előfordulhat több is. Szerencsére használhatjuk a gets függvényt, mellyel egy egész sort olvashatunk be.

A gets párja a puts. A puts(t) ugyanazt csinálja, mint a printf("%s",t).

Karakter beolvasására használhatjuk a scanf("%c",&c) helyett a c=getchar()-t is. Sikertelen beolvasás esetén a getchar EOF (End Of File) értéket ad vissza.

A getchar ellentéte a putchar.

2.1.4 Tömbök

Abban az esetben, ha ugyanabból a változótípusból több ezerre, netán több millióra van szükség, nem csak macerás, hanem lehetetlen is minden egyes elemnek külön változónevet használni, ha sikerülne is, a kezelésük rendkívül megbonyolítaná a program működését.

A tömb olyan összetett változó típus, mely adott számú, azonos típusú elemből áll. Létrehozásakor a név után szögletes zárójelben kell megadni az elemek számát:

```
double d[20];
```

Ez egy 20 elemű, double elemeket tároló tömb. **A [] közé tett nemnegatív egész szám a tömbelem indexe**. Fontos, hogy a tömb indexelése 0-tól kezdődik, ezért a legnagyobb indexű elem indexe eggyel kisebb, mint a tömb mérete, jelen esetben 19.

```
double d[20];
d[ 0]=12.3; // A tömb első eleme
d[19]=-8.1; // A tömb utolsó eleme
d[20]=3.14; // HIBÁS!!! Következmény lásd a 8. ábrán!
```

Ezt jól jegyezzük meg!

A tömb méretének megadásához kizárólag konstans, tehát a fordítás idején ismert értéket használhatunk! (Nem kérdezhetjük meg a felhasználót, hogy az általa megadott mérettel definiáljuk a tömb méretét!)

A tömb további előnye az egyedi változókhoz képest az is, hogy a tömbelem indexe nem csak konstans, hanem egész típusú változó is lehet. Így egyszerű egy ciklussal a tömb elemeit feldolgozni. A példaprogramban először feltöltjük egy ciklussal.

```
for(i=0;i<20;i++){
        d[i]=0.0;
        fflush(stdin);
        printf("%02d. szam= ",i+1);
        if(scanf("%lg",&d[i])!=1){
                printf("\nNem szamot adatt, probalja meg ujbol!\n");
                i--; // Ismét ugyanabba a tömbelembe olvasson, ahová most nem sikerült
                continue; // a ciklus további részét kihagyja
        }
        if(d[i]==0.0)break; // ha 0-t írtunk, megáll
}
```

Először nullázzuk az aktuális tömbelemet (kezdetben, hasonlóan a közönséges változókhoz, a tömb elemei „memóriaszemetet” tartalmaznak). Ebben az algoritmusban ez tulajdonképpen nem lényeges, akár ezt a sort el is hagyhatjuk.

A printf-ben %02d-vel írjuk ki az elem sorszámát, ez azt jelenti, hogy legalább 2 helyet foglaljon a szám (természetesen, ha 555-öt akarnánk kiírni, az hármat foglalna, de az 5 kettőt). A 0 pedig azt jelenti, hogy az üres helyeket 0-val tölti ki, tehát 01-et, 02-t stb. ír ki. Több kiírási formázásért nézzük meg [1]-et!

Sikertelen beolvasás esetén i értékét eggyel csökkentjük, hogy amikor a ciklus végén az i++-szal növeljük, akkor ugyanannyi maradjon az értéke, hogy legközelebb ismét ugyanoda töltsünk.

Ha 0-t írt be a felhasználó, akkor kilépünk a ciklusból. Ez kényelmesebb, mintha a for feltételét bővítettük volna ki ez ellenőrzéssel.

A ciklus után i értéke 20, ha a felhasználó egyszer sem írt volna 0-t, vagy annak a tömbelemnek az indexe, amelyikbe a 0-t olvastuk be.

A következő két ciklusban kiírjuk a tömbelemeket előbb a beolvasás sorrendjében, aztán fordított sorrendben.

```
for(j=0;j<i;j++)printf("%02d. szam= %g\n",j+1,d[j]);
for(j=i-1;j>=0;j--)printf("%02d. szam= %g\n",j+1,d[j]);
```

DOS vagy Windows alatt az alábbi utasítással elérhetjük, hogy a programunk kiírja, hogy Press any key to continue…

```
system("PAUSE"); // nem szabványos, UNIX-ban nem megy
```

2.2 Programírás

1. Írjunk programot, mely bekéri egy másodfokú egyenlet paramétereit, és kiszámítja a gyökeit
   a. Ha a diszkrimináns negatív, közli, hogy nincs valós megoldás, és kilép.
   b. Ha a diszkrimináns negatív, komplex eredményt ad.
2. Írjunk programot, amely bekér max. 15 számot, majd kiírja
   a. a legnagyobbat
   b. a legkisebbet
   c. az átlagot
   d. azokat a számokat. melyek kisebbek az átlagnál

3. Írjunk programot, amely bekér egy pozitív egész számot, és kiszámítja a faktoriálisát. A faktoriálist double típusú változóval számoljuk, hogy nagy megadott számoknál is működjön!
4. Írjunk C programot, amely bekér három pozitív számot, és eldönti, hogy lehetnek-e egy háromszög oldalai.
5. Írjon C programot, amely a felhasználó által Celsiusban megadott hőmérsékletet Farenheitben adja vissza.
6. Írjon C programot, amely eldönti egy bekért pozitív egész számról, hogy prím-e!
7. Írjon C programot, amely kiírja a felhasználótól bekért pozitív egész szám prímtényezős felbontását.

3. Feltételes elágazások, ciklusok, operátorok

if, case, rendezés, bitszámlálás

3.1 Programértés

Ezúttal több különálló programot nézünk.

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

//*******************************************************
void error(char s[]){
//*******************************************************
        printf("\n\nHiba: %s\n",s);
        exit(-1); // kilépünk a programból
}

//*******************************************************
main(){
//*******************************************************
        double a,b,c,max;
        printf("Adjon meg harom szamot!\n+);
        printf(" a : "); if(scanf("%lf",&a)!=1)error("Hibas adat!");
        printf(" b : "); if(scanf("%lf",&b)!=1)error("Hibas adat!");
        printf(" c : "); if(scanf("%lf",&c)!=1)error("Hibas adat!");
        if(a>b){ // ha a nagyobb mint b, akkor
                if(a>c)max=a;
                else max=c;
        }
        else{ // egyébként (vagyis ha a nem nagyobb, mint b)
                if(b>c)max=b;
                else max=c;
        }
        printf("Maximum: %f\n",max);
}
```

Három szám közül kiírja a legnagyobbat. Még egy variáció ugyanerre:

```
//*******************************************************
main(){
//*******************************************************
        double a,b,c,max;
        printf("Adjon meg harom szamot!\n+);
        printf(" a : "); if(scanf("%lf",&a)!=1)error("Hibas adat!");
        printf(" b : "); if(scanf("%lf",&b)!=1)error("Hibas adat!");
        printf(" c : "); if(scanf("%lf",&c)!=1)error("Hibas adat!");
        max=c;
        if(a>b&&a>c)max=a;
        else if(b>c)max=b;
        printf("Maximum: %f\n",max);
}
```

Az error() függvényt és az include-okat csak az első programnál írtuk ide, de természetesen a másodikhoz is hozzá tartoznak. Az error() függvény bemenő paramétere egy karaktertömb. Figyeljük meg, hogy nem adtuk meg a tömb méretét!

A következő program – ismét két változatban – megmondja, hogy a begépelt karakter kisbetű, nagybetű vagy szám.

```
#include <stdio.h>
main(){
        int ch;
        printf("karakter: ");
        fflush(stdin);
        ch=getchar();
        if(ch>='0'&&ch<='9')printf("szám     : %d %c\n",ch,ch);
```

```
        if(ch>='A'&&ch<='Z')printf("nagy betű: %d %c\n",ch,ch);
        if(ch>='a'&&ch<='z')printf("kis betű : %d %c\n",ch,ch);
}


#include <stdio.h>
#include <ctype.h>
main(){
        int ch;
        printf("karakter: ");
        fflush(stdin);
        ch=getchar();
        if(isdigit(ch))printf("szám     : %d %c\n",ch,ch);
        else if(islower(ch))printf("kis betű : %d %c\n",ch,ch);
        else if(isupper(ch))printf("nagy betű: %d %c\n",ch,ch);
        else printf("egyik sem!\n");
}
```

A második változatban a ctype.h-ban definiált függvények segítségével döntjük el, hogy milyen karakterről van szó.

Az && logikai ÉS kapcsolatot jelent. Tehát ha ch>='0' és ch<='9', akkor igaz az állítás. Ennek párja a VAGY kapcsolat, amit két függőleges vonallal jelzünk: || (ez a magyar billentyűzeten Alt_Gr+W). A VAGY kapcsolat azt jelenti, hogy ha a két állításból bármelyik igaz (akár mindkettő is), akkor igaz a teljes állítás.

Feladat: bővítsük ki az első programot úgy, hogy abban az esetben, ha se nem szám, se nem betű a beírt karakter, akkor írja ki, hogy egyik sem (hasonlóan a második programhoz)!

Feladat: bővítsük ki a második programot úgy, hogy azt is írja ki, ha felhasználó whitespace karaktert (szóköz, tabulátor, soremelés) adott meg! Ehhez használja az isspace() függvényt!

Végül ismét egy összetett program:

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


//*********************************************************
void biztonsagos();
void abszolut();
void operatorok();
//*********************************************************


//*********************************************************
int main(){
//*********************************************************
        int valasztott_ertek=0;  // egész típusú változó 0 kezdőértékkel

        printf("I/O, tipusok, tombok\n"); // kiírás
        while(valasztott_ertek!=10){ // ciklus, míg a v.é. nem egyenlő 10-zel

                // A menü kiírása

                printf("\nKerem, valassza ki a muveletet!\n");
                printf("1  Switch-case pelada\n");
                printf("2  Abszolut ertek\n");
                printf("3  Operatorok\n");
                printf("10 Kilepes\n");

                // A választott érték beolvasása

                fflush(stdin); // standard input puffer ürítése
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                // A választott művelet végrehajtása

                switch(valasztott_ertek){ // a v.é.-nek megfeleő case után folytatja
                        case  1: biztonsagos();  break; // az osszead függvény hívása
                        case  2: abszolut();     break;
                        case  3: operatorok();   break;
                        case 10:                 break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
        printf("\nTovabbi jo munkat!\n");
        return 0;
}
```

```
//*******************************************************
void biztonsagos(){
//*******************************************************
        int c; // Ezúttal int-et használunk karakter tárolására, de lehetne char is

        printf("*********************************************************************************\n");
        printf("\nA kovetkezo kerdesre igen eseten I,i,Y,y valasz adhato, nem eseten N es n.\n");
        printf("Miztonsagos?\n");

        fflush(stdin);
        c=getchar();

        switch(c){
                case 'i':
                case 'I':
                case 'y':
                case 'Y':       printf("Biztonsagos.\n");
                                break;
                case 'n':
                case 'N':       printf("Cseppet sem biztonsagos.\n");
                                break;
                default: printf("Nem tudom.\n");
        }
        printf("*********************************************************************************\n");
}


//*******************************************************
void abszolut(){
//*******************************************************
        double d; // Ezúttal int-et használunk karakter tárolására, de lehetne char is

        printf("*********************************************************************************\n");
        printf("\nKerek egy szamot!\n");

        for(fflush(stdin);scanf("%le",&d)!=1;fflush(stdin));

        d=(d<0.0)?-d:d;

        printf("Az abszolut ertek: %g\n",d);
        printf("*********************************************************************************\n");
}

#define MAX_KAR 100

//*******************************************************
void operatorok(){
//*******************************************************
        int a=3,b=6,n;
        char c[MAX_KAR];

        // bitenkénti operatorok

        // and, or, xor, not => 2, 7, 5, -4
        printf("%d\t%d\t%d\t%d\n",a&b,a|b,a^b,~a);

        // logikai operatorok

        // and, or, not => 1, 1, 0
        printf("%d\t%d\t%d\n",a&&b,a||b,!a);

        printf("\nKerek egy egesz szamot: ");
        fflush(stdin);
        if(scanf("%d",&n)!=1){printf("\nHibas adat!\n");return;}
        n%2==0||printf("nem ");
        printf("paros szam\n");

        // bitek eltolása (shift)

        // 6 12 24
        printf("%d\t%d\t%d\n",a<<1,a<<2,a<<3);
        // 3 1 0
        printf("%d\t%d\t%d\n",b>>1,b>>2,b>>3);

        a<<=1;//a*=2
        b>>=1;//a/=2

        // Hany bites az int?

        for(n=0,b=1;b;b<<=1,n++);
        printf("%d bites az int\n",n);

        // egész szám binárissá konvertálása

        printf("Kerek egy egesz szamot!\n");
        if(scanf("%d",&a)!=1){printf("\nSikertelen beolvasas\n");return;}
```

```
        n=n<MAX_KAR?n:MAX_KAR;

        printf("Binaris formaban:");
        b=0;
        do{
                c[b++]=(a&1)?'1':'0';
                a=a>>1;
        }while(a!=0&&b<n);
        for(;b--;)putchar(c[b]);
        printf("\n");
}
```

A switch-case szerkezetet korábban is láttuk, most még egyszer fussunk át rajta:

```
//*******************************************************
void biztonsagos(){
//*******************************************************
        int c; // Ezúttal int-et használunk karakter tárolására, de lehetne char is

        printf("*********************************************************************************\n");
        printf("\nA kovetkezo kerdesre igen eseten I,i,Y,y valasz adhato, nem eseten N es n.\n");
        printf("Miztonsagos?\n");

        fflush(stdin);
        c=getchar();

        switch(c){
                case 'i':
                case 'I':
                case 'y':
                case 'Y':       printf("Biztonsagos.\n");
                                break;
                case 'n':
                case 'N':       printf("Cseppet sem biztonsagos.\n");
                                break;
                default:        printf("Nem tudom.\n");
        }
        printf("*********************************************************************************\n");
}
```

Ha több értékhez ugyanazt az utasítást szeretnénk használni, így tehetjük meg. Ha beírnánk, mondjuk a case 'I': után, hogy printf("I-t irt\n");, akkor 'i' vagy 'I' beírása esetén kiíródna az „I-t irt" szöveg, és a „Biztonsagos." is!

Itt a karakter tárolására int típusú változót használunk.

A következő függvényben a ?: operátort kívántuk kihangsúlyozni.

```
        d=(d<0.0)?-d:d;
```

Jelentés: ha d kisebb mint nulla (tehát negatív), akkor –d kerül d-be, egyébként d. Ha a ? előtti kifejezés igaz, akkor a ? és a : közé írt érték (kifejezés vagy függvényhívás is lehet), ha hamis, akkor a : után írt érték lesz az egész kifejezés értéke.

Operátorok:

```
        int a=3,b=6,n;
        char c[MAX_KAR];

        // bitenkénti operatorok

        // and, or, xor, not => 2, 7, 5, -4
        printf("%d\t%d\t%d\t%d\n",a&b,a|b,a^b,~a);
```

Az a változó értéke 3, binárisan 0011, a b változó pedig 6, binárisan 0110.

- A bitenkénti ÉS (and) azt jelenti, hogy ÉS logikai műveletet végzünk a két egész szám megfelelő bitjei között, tehát az eredményben ott lesz 1, ahol mindkettőben 1 volt, a többi helyen 0 lesz. Jelen esetben 0010 az eredmény, decimálisan 2.
- A bitenkénti VAGY (or) akkor 1, ha legalább az egyik bit 1 volt, itt: 0111, azaz 7.
- A bitenkénti KIZÁRÓ VAGY (xor) akkor egy ha csak az egyik volt 1: 0101, tehát 5. (A^B=A|B-A&B).
- A bitenkénti negálás vagy invertálás akkor 1, ha a bit 0, és akkor 0, ha a bit 1: 1100=-4. (Az első 1 jelenti a negatív előjelet. Vigyázat: -3=1101, -2=1110, -1=1111, 0=0000)

```
// logikai operatorok

// and, or, not => 1, 1, 0
printf("%d\t%d\t%d\n",a&&b,a||b,!a);
```

A logikai operátorok két oldalán egy-egy logikai művelet, vagy egész értékű kifejezés szerepel. Ha egész értékeket helyezünk a logikai operátor két oldalára, akkor nem számít az egyes bitek értéke, csak az, hogy a szám 0 volt-e, vagy bármi más. **A 0 ugyanis hamisat jelent, a bármi más igazat.** (Ezt gyakran kihasználjuk, az if utasításban, vagy a ciklusokban. Ne lepődjünk meg, ha ilyet látunk: if(a)…, vagy for(j=10;j;j--), esetleg while(1), ezek ezt jelentik: if(a!=0)…, for(j=10;j!=0;j--) és while(1==1) (ezutóbbi egy szándékos végtelen ciklus, amiből breakkel vagy returnnel szokás kilépni)). **Logikai művelet visszatérési értéke is tekinthető egész számnak: IGAZ=1, HAMIS=0.**

Itt a=3=>IGAZ, b=6=> igaz, tehát

- a ÉS b: a&&b=IGAZ=1
- a VAGY b: a||b=IGAZ=1
- NEM a: !a=HAMIS=0

Az alábbi, felettébb szokatlan kódrésszel a logikai ÉS, és logikai VAGY egy fontos vonására szeretném felhívni a figyelmet:

```
n%2==0||printf("nem ");
printf("paros szam\n");
```

A || és a && jobb oldalán álló kifejezés csak akkor hajtódik végre. ha a bal oldali kifejezés alapján nem lehet eldönteni, hogy mi a teljes kifejezés értéke. A || műveletnél, ha a bal oldalán IGAZ érték van, akkor a teljes kifejezés is biztosan igaz, tehát a jobb oldal nem fut le. A && műveletnél, ha a bal oldalán HAMIS érték van, akkor a teljes kifejezés is biztosan hamis, tehát a jobb oldal nem fut le.

A fenti példában, ha a bal oldalon n 2-vel vett osztási maradéka 0, vagyis a bal oldalon igaz áll, a jobb oldalon álló printf nem hajtódik végre. Ha a maradék nem 0 (azaz 1), akkor viszont nem tudhatjuk, hogy a || értéke igaz, vagy hamis lesz, csak akkor, ha a jobb oldal is lefut, azaz kiíródik a „nem ” is. Ezt a sort átírhatnánk így is: n%2&&printf(… A || vagy && bármely oldalára csak olyan függvényt tehetünk, melynek egész visszatérési értéke van. (A printf visszatérési értéke a kiírt karakterek száma.)

Figyelem! Ez a tulajdonság egy további következménnyel is jár: az && és a || kiértékelési pontot is jelent. (A harmadik operátor, mely kiértékelési pontot jelent, a vessző (,)). Ez azt jelenti, hogy az előtte álló kifejezés kiértékelődik. Például: if((a=i++)>0&&(b=i+2)) kifejezésben, ha i mondjuk 4 volt, akkor a=4, és b=7 lesz, mert a ++ posztinkremens, tehát a kifejezés kiértékelődése után növeli az értéket, de az && kiértékelési pont, tehát itt megtörténik a növelés, tehát b=5+2 lesz. Néhány éve egy ilyesféle kifejezés szerepelt a nagy ZH „Mit ír ki?” feladatában.

```
// bitek eltolása (shift)

// 6 12 24
printf("%d\t%d\t%d\n",a<<1,a<<2,a<<3);
// 3 1 0
printf("%d\t%d\t%d\n",b>>1,b>>2,b>>3);

a<<=1;//a*=2
b>>=1;//a/=2
```

Az << és >> operátor adott irányban, a jobb oldalán szereplő darab bittel eltolja a számot, a kiesett 1-esek elvesznek. Pl. a=3=000011 => a<<1=000110, a<<2=1100, a<<3=011000; b=6=0110 => b>>1=0011, b>>2=0001, b>>3=0000. Ezek a műveletek tulajdonképpen a 2 hatványaival történő szorzásnak ill. osztásnak felelnek meg, ha erre van szükség, célszerűbb ezt használni (csak egész számoknál!). Ha mégsem ezt írjuk, a fordítók általában elég okosak ahhoz, hogy erre cseréljék.

Figyelem! Ha negatív egész számot >> művelettel tolunk, nem 0, hanem 1 lép be balról! Pl.: 1000 => 1100 => 1110 => 1111. De ha ugyanaz a bináris szám unsigned típusú, akkor 0 fog belépni: 1000 => 0100 => 0010 => 0001 => 0000.

```
// Hany bites az int?

for(n=0,b=1;b;b<<=1,n++);
printf("%d bites az int\n",n);
```

A fenti program megszámolja, hogy hány bites egy egész szám. Kezdetben 1-et rakunk b-be, aztán minden lépésben eggyel balra toljuk: 0001=>0010=>0100=>1000=>0000. Amikor végül kiesik az egyes a bal oldalon, akkor b 0 lesz, tehát a for feltétele hamis.

Figyelem! Ha középre ezt írnánk: b<<n, ez nem működne, mert ha b 32 bites, és n=32, akkor ez a művelet úgy működik, mintha b<<0-t írtunk volna (gyakorlatilag valójában b<<(n%32) a művelet.

```
// egész szám binárissá konvertálása

printf("Kerek egy egesz szamot!\n");
if(scanf("%d",&a)!=1){printf("\nSikertelen beolvasas\n");return;}

n=n<MAX_KAR?n:MAX_KAR;

printf("Binaris formaban:");
b=0;
do{
        c[b++]=(a&1)?'1':'0';
        a=a>>1;
}while(a!=0&&b<n);
for(;b--;)putchar(c[b]);
printf("\n");
```

A fenti kód egy egész számot ír ki bináris formában. A c tömb:

```
char c[MAX_KAR];
```

Ahol

```
#define MAX_KAR 100
```

A c tömb tehát 100 elemű, amit a MAX_KAR preprocesszor konstanssal definiáltunk.

```
n=n<MAX_KAR?n:MAX_KAR;
```

Itt n kezdőértéke az az érték, hogy hány bites az egész szám, melyet az imént számoltunk ki. Ha a MAX_KAR-ral definiált tömb mérete kisebb ennél, akkor gondoskodnunk kell arról, hogy semmiképp se lépjük túl a tömb méretét. n-be tehát a két érték közül a kisebb kerül.

```
b=0;
do{
        c[b++]=(a&1)?'1':'0';
        a=a>>1;
}while(a!=0&&b<n);
```

Ezúttal a hátultesztelő do-while ciklust használjuk. Ez abban különbözik a sima while-tól, hogy a ciklusmag egyszer mindenképpen végigfut, és csak utána történik a feltétel ellenőrzése.

> Pascalosok figyelmébe: a do-while ciklus is addig ismétlődik, míg a while feltétele igaz, szemben a Pascal repeat-until-jával.

Bármely ciklust igénylő algoritmus megvalósítható a for, while, vagy do-while ciklus bármelyikével. Mindig azt használjuk, amelyiket kényelmesebbnek érezzük.

A fenti kód úgy működik, hogy megvizsgáljuk a legkisebb helyiértékű bitet (LSB=Least Significant Bit), az a&1 pont ezt adja. Ha 1, akkor ’1’ karakter kerül a tömbbe, egyébként ’0’. Ezután eltoljuk egy bittel jobbra, így a jobbra álló bit kerül legalulra. Mindezt addig ismételjük, míg a ki nem ürült, vagy el nem értük n-et. Ezutóbbi feltétel elsősorban negatív számok esetén érdekes, hiszen akkor 1-esek jöttek be balról, tehát a szám sosem lesz 0. (Másik lehetőség, ha kisebb a tömb, mint ahány bites az egész, de ekkor meg eleve nem lesz jó a kiírt szám, hisz az eleje hiányzik.)

```
for(;b--;)putchar(c[b]);
```

Ez a ciklus kiírja a tömbben tárolt karaktereket hátulról előre, hiszen fordított sorrendben kaptuk azokat.

3.2 Programírás

1. Írjunk programot, amely kiírja a Fibonacci sorozat első n elemét, ahol n-et a felhasználó adja meg.
2. Írjunk programot, amely bekér két számot, és kiírja az összes közös osztójukat.
3. Írjunk programot, amely bekér egy pozitív egész számot, és kiírja római számként. (Nehéz feladat)
4. Írjunk C programot, amely bekér egy osztályzatot, és kiírja, hogy az elégtelen, elégséges, közepes, jó, vagy jeles-e.
5. Írjon programot, amely kiírja az 1901 és 2099 közötti összes olyan nap dátumát, amelyik péntek 13-ra esik! (A programíráshoz használja a fejét, ne használjon erre a célra mások által kidogozott algoritmust!)(Nehéz feladat)
6. Írjunk C függvényt, amely két, paraméterként adott pozitív egész számról eldönti, hogy barátságos számok-e.
   a. Egészítsük ki teljes programmá, mely 1 és 1.000.000.000 között kiírja az összes ilyen párost (nem baj, ha lassú).
7. Írjon C programot, amely kiírja két bekért pozitív egész szám legnagyobb közös osztóját és legkisebb közös többszörösét!

4. Többdimenziós tömbök, pointerek

4.1 Programértés

Ezúttal is több kisebb programot fogunk megnézni.

4.1.1 Mátrix összeadás

A program véletlen számokkal feltölt két tömböt, majd összeadja őket, végül kiírja a három mátrixot.

```
//*********************************************************
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
//*********************************************************

//*********************************************************
// konstansok
//*********************************************************
const int MSOR=4,MOSZL=3;

//*********************************************************
// globális változók
//*********************************************************
int a[MSOR][MOSZL],c[MSOR][MOSZL];

//*********************************************************
int main(){
//*********************************************************
        int i,j,b[MSOR][MOSZL]; // lokális változók

        // véletlenszám generátor inicializálása

        srand((unsigned)time(NULL));

        // mátrixok feltöltése

        for(i=0;i<MSOR;i++)for(j=0;j<MOSZL;j++){a[i][j]=rand();b[i][j]=rand();}

        // összeadás

        for(i=0;i<MSOR;i++)for(j=0;j<MOSZL;j++){c[i][j]=a[i][j]+b[i][j];}

        // kiírás

        printf("A=\n");
        for(i=0;i<MSOR;i++){
                for(j=0;j<MOSZL;j++)printf("%12d",a[i][j]);
                printf("\n");
        }

        printf("\nB=\n");
        for(i=0;i<MSOR;i++){
                for(j=0;j<MOSZL;j++)printf("%12d",b[i][j]);
                printf("\n");
        }

        printf("\nA+B=\n");
        for(i=0;i<MSOR;i++){
                for(j=0;j<MOSZL;j++)printf("%12d",c[i][j]);
                printf("\n");
        }

}
```

Ebben a programban használunk először **globális változók**at (nem mintha szükség lenne rájuk, csak a példa kedvéért). Ezek tehát olyan változók, melyek a függvényeken kívül találhatók. Jellegzetességük, hogy ezeket a függvények közösen használhatják, tehát például a main()-ben adunk neki értéket, és egy másik függvényben kiíratjuk az értékét anélkül, hogy paraméterként át

kellene adni a függvénynek. Ebből látjuk a hátrányukat is: gyakorlatilag bármelyik függvény láthatja, és meg is változtathatja az értéküket, ami borzasztó nehezen megkerülhető hibákhoz vezethet, és a programot is jóval átláthatatlanná teszi, megakadályozza az egyes programmodulok könnyű cseréjét, vagy újra felhasználását más programokban. Tulajdonképpen a globális változók előnyét megtartja (több függvényből látható, nem kell paraméterként átadni), és a hátrányokat kiküszöböli (csak azok a függvények változtathatják meg a változók értékét, melyeknek megengedjük) a C++-ban bevezetett osztályrendszer (az objektumorientáltság megvalósítása). **A globális változók használatát lehetőség szerint kerüljük!**

A globális változók a függvényekhez hasonlítanak atekintetben, hogy a definíciójuk helyétől kezdve használhatók, és nekik is van deklarációjuk, melyet az extern kulcsszóval adhatunk meg. Pl.: extern int a;

Ezt (a függvénydeklarációhoz hasonlóan) más programmodulba (másik C vagy cpp fájlba) is beépíthetjük, és ekkor, a függvényhez hasonlóan, a linker teremti meg a kapcsolatot a két modul között.

Egy függvényen belül lehet definiálni a globálissal megegyező nevű változót, akár más típusút is, ekkor a globális változó nem elérhető a függvényből.

Konstansok: korábban már találkoztunk a preprocesszor konstansokkal, melyeket a #define kulcsszó vezetett be, pl.:

#define MAX 10

Ezt a konstanst a preprocesszor helyettesíti be, gyakorlatilag egy szövegbehelyettesítő művelettel, anélkül, hogy a tartalomra a legkisebb figyelemmel is lett volna. Van azonban lehetőség típusos konstansok bevezetésére is, ha a változó neve elé a const kulcsszót írjuk. Ekkor természetesen kötelező a kezdőérték, hiszen később már nem változtathatjuk meg az értékét. A fenti példában két ilyen konstanst használunk, a csupa nagybetű természetesen nem kötelező.

Véletlenszám generáláshoz az stdlib.h-ban deklarált rand() függvényt használjuk. A véletlenszám generáláshoz használt kezdőértéket az srand() függvény állítja be, ha ezt nem használjuk, a rand() a program minden futtatásakor ugyanazokat a számokat fogja előállítani. (**Próbáljuk ki!**)

A kezdőérték megadásához a time() függvényt használjuk (prototípus a time.h-ban), mely az 1970 óta eltelt másodpercek számát adja vissza.

A tömbök indexeléséhez minden esetben két, egymásba ágyazott for ciklust használunk. A mátrixokat a következőképp definiáljuk:

```
int a[MSOR][MOSZL],c[MSOR][MOSZL];
```

Tehát első zárójelben a sorok, másodikban az oszlopok száma. Az a memóriában így helyezkedik el:

[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]	[2][0]	[2][1]	[2][2]	[3][0]	[3][1]	[3][2]

Ezzel a módszerrel akárhány dimenziós tömböt készíthetünk, pl.: t[6][8][7][3] egy négydimenziós tömb lesz.

Dinamikus módszerrel is tudunk többdimenziós tömböt létrehozni, de annak kezelése elég macerás.

A printf()-ben használt %12d azt jelenti, hogy minden kiírt szám 12 helyet foglal el, így könnyű kialakítani az egyforma oszlopszélességet a mátrixszerű kiíráshoz.

4.1.2 Tömbök bejárása pointerrel, stringek

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


//*******************************************************
void fuggveny(char s[]){
//*******************************************************
        printf("A tomb merete: %d\tHoppa! Ez nem annyi!\n", sizeof(s)/sizeof(char));

        // Számoljuk meg kézzel, első módszer:

        {
                int i;
                for(i=0; s[i]!='\0'; i++); // '\0' helyett egyszerűen 0-t is írhatunk
                printf("1. A karakterek szama: %d\tEz eggyel kevesebb lett. Miert?\n",i);
        }


        // Számoljuk meg kézzel, második módszer:

        {
                char *p=s;
                while(*(p++)!='\0');
                printf("2. A karakterek szama: %d\n",p-s);

                // írjuk még rövidebben:

                p=s;
                while(*(p++));
                printf("3. A karakterek szama: %d\n",p-s);
        }

}


//*******************************************************
void stringmasolo_1(char * cel, char * forras){
//*******************************************************
        int i;
        for(i=0;forras[i]!=0;i++)cel[i]=forras[i];
        cel[i]=0;//A stringet lezáró 0-t is be kell tenni
}


//*******************************************************
void stringmasolo_2(char * cel, char * forras){
//*******************************************************
        while(*(cel++)=*(forras++));
}


//*******************************************************
void nullaz_1(double * d,int meret){
//*******************************************************
        int i;
        for(i=0;i<meret;i++)d[i]=0.0;
}


//*******************************************************
void nullaz_2(double * d,int meret){
// ugyanaz, csak rondább
//*******************************************************
        int i;
        for(i=0;i<meret;i++)*(d++)=0.0;
}


//*******************************************************
int main(){
//*******************************************************
        char s[]="Ez van a tombben";// kezdőérték a tömbnek
        char s2[100],s3[100];
        double d[]={365,-82,3.14};
        double d2[4];

        printf("A tomb merete: %d\n", sizeof(s)/sizeof(char));
        fuggveny(s);
        stringmasolo_1(s2,s);
        printf("A lemasolt szoveg 1: \"%s\"\n",s2);
        stringmasolo_2(s3,s);
        printf("A lemasolt szoveg 2: \"%s\"\n",s3);
        printf("A d tomb merete: %d\n", sizeof(d)/sizeof(double));
```

```
        printf("A d2 tomb merete: %d\n", sizeof(d2)/sizeof(double));
        nullaz_1(d,sizeof(d)/sizeof(double));
        nullaz_2(d2,4);
        return 0;
}
```

Kezdjük a main() függvénnyel! Itt két olyan tömb is szerepel, melynek kezdőértéket adunk, az s és a d. Az s-nek is adhattunk volna úgy kezdőértéket, ahogy a d-nek: (s[]={'E','z',' ','v','a','n', stb.), de így egyszerűbb. A tömb méretét nem adtuk meg közvetlenül, mert a fordító is meg tudja számolni.. Ennek ellenére megtehettük volna, hogy beírjuk a [] közé a méretet. Ha ez kisebb, mint az elemek száma, akkor fordítási hibát kapunk, ha nagyobb, akkor a tömb végén lévő elemek nem kapnak kezdőértéket.

A sizeof() operátor (nem függvény!) a megadott változó méretét adja, bájtban. Ha tömb méretéről van szó, akkor ahhoz, hogy a tömb elemszámát megkapjuk, el kell osztani a tömbben lévő elemek méretével. **Figyelem! A sizeof csak a valódi tömbök esetén adja a tömb méretét, pointerrel mutatott tömb esetén** (mégha az nem dinamikus is) **a pointer méretét adja vissza**.

Tömbök (és stringek) másolása: **A tömböket csak elemenként tudjuk másolni**, tehát a[i]=b[i] módon. Az a=b nem működik, mert a tömb neve a [] nélkül a tömb első elemére mutató pointer, tehát egy olyan változó, mely a tömb első (0 indexű) elemének memóriacímét tartalmazza.

Egy pointerből az általa mutatott értéket a pointer neve elé tett * karakterrel kapjuk vissza. Tehát ha a fenti példában ezt írnánk: *s='A';, akkor a string tartalma „Az van a tombben"-re változna, mert az első elemet felülírjuk. Természetesen ez nem csak char tömbökre működik, *d=25; is jó. C-ben a pointereknek van típusa (kivéve a void* pointereket), ezért a fordító tudja, hogy mit jelent a mutatott érték, tehát a megfelelő műveletet tudja végrehajtani (és a *d=25; esetén double-t másol, nem pl. int-et).

A d tömb 1 indexű elemére a d+1 pointer mutat, tehát d+1 ugyanaz, mint &d[1]. Vagyis *(d+1) ugyanaz, mint d[1]. A kettő közül bármelyiket használhatjuk, de célszerűbb az utóbbit használni, mert átláthatóbb kódot eredményez.

A stringek kezelésére általában a string.h-ban szereplő rutinokat használjuk, pl.: strcpy: stringek másolására, strcat: két string összefűzése, strcmp: két string összehasonlítása, strlen: string hosszának számítása, stb. Bővebben pl. [1].

```
//*******************************************************
void fuggveny(char s[]){
//*******************************************************
        printf("A tomb merete: %d\tHoppa! Ez nem annyi!\n", sizeof(s)/sizeof(char));
```

Ez a tömb méretére 2-t vagy 4-et fog adni (ha valaki 64 bites rendszerben fordítja le ezt a programot, akkor bizonyára 8-at, hiszen ha a 64 bites rendszerekben valami tényleg 64 bites, akkor a pointer biztos az). Mi ennek az oka? Az, hogy itt látszólag ugyan egy tömböt adunk át, de valójában egy karakterre (egy pointerből nem lehet eldönteni, hogy az egy darab értékre mutat, vagy egy tömbre, mert az egy darab érték olyan, mint egy egyelemű tömb) mutató pointert. A tömb méretét tehát nem adjuk át automatikusan. Ha ez szükséges, akkor egy int változóban adjuk át a méretet is (lásd a nullaz függvényeket). A fenti módon, char s[]-ként megadott paraméter tehát konstans pointer, ami azt jelenti, hogy s pointer értékét nem lehet megváltoztatni (az s által mutatott értékeket, vagyis a tömb elemeit természetesen igen).

```
        // Számoljuk meg kézzel, első módszer:

        {       int i;
```

```
        for(i=0; s[i]!='\0'; i++); // '\0' helyett egyszerűen 0-t is írhatunk
        printf("1. A karakterek szama: %d\tEz eggyel kevesebb lett. Miert?\n",i);
}
```

Ez a rész csak azért került egy blokkba, hogy lássuk: nem csak a függvény elején lehet változókat definiálni, hanem bármely blokk elején, tehát a { után. Az így definiált változók csak az adott blokkon belül léteznek, ezért a } után már nem használhatjuk, annak ellenére, hogy még a függvényen belül vagyunk!

A for ciklus addig megy, ameddig 0-t nem talál, hisz ez jelzi a string végét. A kérdésre a válasz pedig: A '\0' is benne van a tömbben, tehát kell neki a hely, azonban a ciklus leáll, mikor megtalálja a 0-t, tehát i a 0 indexét tartalmazza, az utolsó elem indexe pedig eggyel kisebb, mint a méret, hiszen 0-val kezdődik az indexelés.

```
{       char *p=s;
        while(*(p++)!='\0');
        printf("2. A karakterek szama: %d\n",p-s);

        // írjuk még rövidebben:

        p=s;
        while(*(p++));
        printf("3. A karakterek szama: %d\n",p-s);
}
```

Most nézzünk néhány „C-szerűbb" megvalósítást. Ezek használatát mindig alaposan fontoljuk meg, mert nagymértékben ronthatják a kód olvashatóságát, futási sebességük pedig azonos.

Ezúttal egy pointerrel járjuk be a tömb elemeit. A definíciónál írt char *p=s esetén az értékadás a p-nek történik, nem a *p-nek! A *(p++)!='\0' jelentése: megnézzük a p által mutatott tömbelemet, hogy az vajon '\0'-e, ha nem, akkor folytatódik a ciklus. Miután megnéztük, p pointer értéke eggyel nő a ++ miatt, ami azt jelenti, hogy a p a tömb következő elemére fog mutatni. Mivel p típusos pointer, a program tudja, hogy hány bájttal kell előrébb mutatni a következő tömbelemre. **Tehát p+1 mindig a következő tömbelemre mutat**, és a típusától függően 1, 2, 4, 8 vagy akár még több (struktúratömb esetén) bájttal nagyobb pointer értéket jelent.

A művelet fordítva is működik: p-s megmondja, hogy hány tömbelem távolságra van egymástól a két pointer. A két pointer típusa meg kell, hogy egyezzen! (Jelen esetben mindkettő char * típusú)

A második esetben *(p++) van a while feltételében. Amikor p egy olyan karakterre mutat, melynek értéke '\0', vagyis 0, akkor ez HAMIS-at jelent (emlékszünk még: ha egy egész szám 0, az HAMIS, ha bármi más, akkor igaz).

Fontos! A következő definíciók ugyanazt jelentik:

```
char *a,b; char* a,b; char * a,b;
```

Ezek közül a második kettő megtévesztő lehet, mert azt az érzetet keltheti, mintha itt két pointert definiálnánk, de ez nem igaz! Itt a b egy közönséges, karakter típusú változó! Ha két pointert akarunk, ezt írjuk:

```
char *a,*b;
```

A két következő függvénnyel stringeket másolhatunk (hiszen = jellel nem lehet!).

```
//*****************************************************
void stringmasolo_1(char * cel, char * forras){
//*****************************************************
        int i;
        for(i=0;forras[i]!=0;i++)cel[i]=forras[i];
        cel[i]=0;//A stringet lezáró 0-t is be kell tenni
}
```

A függvény két karaktertömböt kap bemenetként – pontosabban karakterre mutató pointert, de a kettő gyakorlatilag ugyanaz. Feltételezzük, hogy a cel stringbe belefér a forras string. Erről a hívó függvénynek kell gondoskodnia. Ha nem ilyet ad, akkor az futási hibát fog okozni.

Ezúttal a for ciklus feltételében !=0-t vizsgáljuk. Ehelyett nyugodtan lehetne !='\0' is, hiszen ugyanazt jelenti. Sőt, lehetne egyszerűen a forras[i] is, mert forras[i] egész típusú, és pont akkor ad 0-t, amikor a ciklust be kell fejezni (emlékszünk: 0=HAMIS, nem 0=IGAZ).

Ha a ciklus eléri a 0-t, akkor azt már nem másolja át, pedig azt is kell, hiszen az zárja a stringet. Ha ezt elhagynánk, akkor a string tartalma „Ez van a tombben╗ς*eĚÝ.☺q%″ś@§ér☹”, vagy valami hasonló krix-krax lenne, mert az ott lévő, korábbról ottmaradt memóriatartalom (memóriaszemét) is a stringhez tartozónak minősülne, a krix-krax a következő, véletlenül ott maradt '\0'-ig tartana.

Apropó: a C-ben a 'a' és az "a" között az a különbség, hogy az első egy karakter konstans, míg a második egy string konstans, mely két karakterből áll: 'a'+'\0', és valójában a rá mutató pointert jelenti.

```
//*****************************************************
void stringmasolo_2(char * cel, char * forras){
//*****************************************************
        while(*(cel++)=*(forras++));
}
```

Itt egy sorban megoldjuk az egészet. A while feltételében léptetjük a két, tömbre mutató pointert (ezek most nem konstans pointerek, mint s[] esetén a korábbi fuggveny()-ben, ezért megváltoztatható az értékük. A while akkor áll le, amikor '\0'-t másolunk a forrásból a célba (emlékezzünk, hogy C-ben létezik az a=b=c; típusú értékadás, a fenti esetben is ez történik, csak ott 'a' helyett a while kapja meg másolatot, és az alapján dönt). További előny, hogy a stringet lezáró 0 is átmásolódik, mert a while kiértékelése csak a másolás után következik be, ami így le is áll.

A main függvényben a stringmasolo_2 függvényt s3 és s változókkal hívtuk, és itt megváltoztattuk cel és forras értékét, melyek a függvény végén már a lezáró '\0' utáni bájtra mutatnak a memóriában. Ez hatással van s3-ra és s-re is? Ők ezután a két string után fognak mutatni? Nem:

Amikor C-ben meghívunk egy függvényt, akkor **a paraméterként megadott változókról vagy konstansokról mindig másolat készül**, tehát a másolatot megváltoztatva az eredeti változó értéke nem változik, csak a lemásolté:

```
void nem(int a){      // másolat készül a paraméterről (a=b)
        a=8;          // a lemásolt 'a' fog változni, 'b' nem!
}

void igen(int * a){   // a másolat a paraméterként megadott 'b'-re mutató pointerről készül!
        *a=8;         // a lemásolt pointer által mutatott 'b' értéke változik!
}

main(){
        int b=3;
        valami(b);  // ezután 'b' értéke továbbra is 3
        valami(&b); // de itt már 8-ra változik b!!! (b címét adjuk át)
}
```

Ha változást akarunk, a változó címét kell átadnunk, mert akkor a címről készül másolat. (Ekkor a függvény bemenete pointerre változtatandó).

```
//*****************************************************
void nullaz_1(double * d,int meret){
//*****************************************************
        int i;
```

```
        for(i=0;i<meret;i++)d[i]=0.0;
}
```

Mivel a tömb méretét nem adja át automatikusan a rendszer, nekünk kell megtenni egy külön változóban, ez a meret. Ez a függvény tehát 0 értékekkel tölti tele a megadott tömböt (**a tömbről nem készül másolat, csak a tömb címéről**, erre figyeljünk, nehogy akaratlanul megváltoztassuk a tömb tartalmát!)

```
//*****************************************************
void nullaz_2(double * d,int meret){
//*****************************************************
        int i;
        for(i=0;i<meret;i++)*(d++)=0.0;
}
```

Itt pointerrel csináljuk ugyanazt. Ezt a formát nem érdemes használni, mert nem gyorsabb, és nem is kisebb az előzőnél, csak kevésbé átlátható.

4.2 Programírás

1. Írjon C programot, amely
   a. Megkérdezi a felhasználót, hogy hány koordinátát (x,y) akar megadni. Max. 500-at engedjen.
   b. Olvasson be ennyi koordinátát (double típust használjon, az értékeket tömbben tárolja)!
   c. Számítsa ki az origótól mért távolságuk átlagát!
   d. Írja ki azokat a koordinátákat, melyek az átlagnál közelebb találhatók az origóhoz.

2. Készítsen C függvényt, amely két stringet kap paraméterként, és az első végére fűzi a másodikat (ugyanazt tegye, mint az **strcat()** a string.h-ban).

   Feltételezzük, hogy az első paraméterben megadott string olyan tömbben van, amely elég nagy, hogy beleférjen a hozzáfűzött rész is.

3. Készítsen C programot, amely összeszoroz két, véletlenszerű értékekkel feltöltött mátrixot!
4. Készítsen C függvényt, amely paraméterként egy stringre mutató pointert és egy karaktert kap, megkeresi a stringben a karakter első előfordulási helyét, és visszatérési értékül egy erre mutató pointert ad vissza. Ha nem találja, NULL pointert ad vissza (return NULL;) A függvény tehát ugyanazt teszi, mint az strchr függvény.
5. Készítsen C függvényt amely két stringet kap paraméterként, visszatérési értéke int típusú, értéke pedig 0, ha a két string megegyezik (nem a címe, hanem a tartalma!). Pozitív szám (mindegy, mennyi), ha az első string abc rendben nagyobb, mint a második, és negatív, ha a második string a nagyobb. (Ha az egyik string ugyanaz, mint a másik eleje, akkor a másik a nagyobb.)
6. Készítsen C függvényt, amely a paraméterként kapott stringben minden nagybetűt kisbetűre cserél!
7. Készítsen C függvényt, amely a paraméterként kapott stringben minden kisbetűt nagybetűre cserél!
8. Készítsen C függvényt, amely a paraméterként kapott stringben minden szót nagy kezdőbetűssé alakít, míg a szó többi betűjét kisbetűvé. (Szó az, mely előtt nem betű van, ill. a string szóval kezdődik, ha s[0] betű.)

9. Készítsen C függvényt, amely négy stringet kap paraméterként, az elsőbe helyezi a második módosított változatát: a másodikban kapott stringben kicseréli a harmadik paraméterben megadott szavakat a negyedik paraméterben megadott szavakra. Ellenőrizze azt is, hogy az első két stringre mutató pointer megegyezik-e, ha igen, akkor írjon ki hibaüzenetet, és állítsa le a programot az exit() függvénnyel! (Nehéz feladat.)

Olvassuk el ismét az 1. fejezetet!

5. Rendezés, keresés

5.1 Programértés

qsort, Hashinggel történő adatfelvétel és keresés implementálása.

5.1.1 Közvetlen kiválasztásos és buborék rendezés

Az alábbi program bemutatja a közvetlen kiválasztásos és a buborék rendezést. Az // adm jelzésű sorok csak arra szolgálnak, hogy nyomon tudjuk követni a program futását, ezek törölhetők.

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


//*******************************************************
void kozvetlen(double * t,int meret);
void buborek(double * t,int meret);
void kiir(double * t,int meret);
//*******************************************************


//*******************************************************
int main(){
//*******************************************************
        double t1[]={863,-37,54,9520,-3.14,25,9999.99};
        double t2[]={863,-37,54,9520,-3.14,25,9999.99};

        kozvetlen(t1,sizeof(t1)/sizeof(double));
        buborek(t2,sizeof(t2)/sizeof(double));
        return 0;
}


//*******************************************************
void kozvetlen(double * t,int meret){
//*******************************************************
        int i,j,minindex;
        int masolas=0,osszehasonlitas=0; // adm

        printf("Kozvetlen kivalasztasos rendezes, kezdetben:\n"); // adm
        kiir(t,meret); // adm
        for(i=0;i<meret-1;i++){
                minindex=i;
                for(j=i+1;j<meret;j++){
                        osszehasonlitas++; // adm
                        if(t[j]<t[minindex])minindex=j;
                }
                if(minindex!=i){
                        double temp=t[minindex];
                        t[minindex]=t[i];
                        t[i]=temp;
                        masolas+=3; // adm
                }
                printf("\n%d. lepes utan:\n",i+1); // adm
                kiir(t,meret); // adm
        }
        printf("Kesz\nOsszehasonlitasok: %d\tMasolasok: %d\n",osszehasonlitas,masolas); // adm
        system("PAUSE"); // adm
}


//*******************************************************
void buborek(double * t,int meret){
//*******************************************************
        int i,j,nemvoltcsere;
        int masolas=0,osszehasonlitas=0; // adm
        double temp;

        printf("\nBuborek rendezes, kezdetben:\n"); // adm
```

```
        kiir(t,meret); // adm
        for(i=0;i<meret-1;i++){
                nemvoltcsere=1;
                for(j=0;j<meret-1-i;j++){
                        osszehasonlitas++; // adm
                        if(t[j]>t[j+1]){
                                temp=t[j];t[j]=t[j+1];t[j+1]=temp;
                                masolas+=3; // adm
                                nemvoltcsere=0;
                        }
                }
                if(nemvoltcsere)break;
                printf("\n%d. lepes utan:\n",i+1); // adm
                kiir(t,meret); // adm
        }
        printf("Kesz\nOsszehasonlitasok: %d\tMasolasok: %d\n",osszehasonlitas,masolas); // adm
        system("PAUSE"); // adm
}


//*******************************************************
void kiir(double * t,int meret){
//*******************************************************
        int i;
        for(i=0;i<meret;i++)printf("%d. elem: %.2f\n",i+1,t[i]);
        system("PAUSE");
}
```

Hogyan működik a közvetlen kiválasztásos rendezés?

1. lépés: Végigmegyünk a tömb elemein, és megnézzük, melyik a legkisebb. Ha megtaláltuk, akkor kicseréljük a tömb első elemére.
2. lépés: A tömb első eleme tehát a legkisebb, ebben a körben ezt már nem vizsgáljuk. A maradék elemek között megkeressük a legkisebbet, és ezt kicseréljük a tömb második elemével.
3. lépés: A maradékból kikeressük a legkisebbet, és ezt tesszük a maradék elejére.
4. lépés: A 3. lépést ismételjük, míg a tömb végére nem érünk.

Lássuk ezt C nyelven:

```
//*******************************************************
void kozvetlen(double * t,int meret){
//*******************************************************
        int i,j,minindex;

        for(i=0;i<meret-1;i++){
                minindex=i;
                for(j=i+1;j<meret;j++){
                        if(t[j]<t[minindex])minindex=j;
                }
                if(minindex!=i){
                        double temp=t[minindex]; t[minindex]=t[i]; t[i]=temp; // csere
                }
        }
   }
```

A külső ciklus végigmegy a tömb elemein. Az i azt mutatja, hogy melyik az az elem, amelyet ebben a lépésben ki fogunk cserélni a legkisebbre. A ciklus meret-1-ig megy. Ha nem írjuk oda a -1-et, akkor sincs probléma, csak feleslegesen fut végig még egyszer a ciklus. (Gondoljuk át, miért!)

A minindex változóba először i-t teszünk, vagyis egyelőre az i indexű elem a maradék legkisebb eleme.

Ezután következik a belső ciklus. A j i+1-től indul: önmagával nem hasonlítjuk össze az elemet, és az i-nél kisebb indexű elemek már sorban vannak, ezekkel nem foglalkozunk. Ez a ciklus meret-ig megy, mert a legkisebbet az összes maradék közül keressük. A ciklusmagban egyetlen utasítás van, melyben megnézzük, hogy az aktuális elem kisebb-e az eddigi legkisebbnél (ha csökkenő sorrendbe akarnánk tenni az elemeket, akkor mindössze ezt az egy relációs jelet kellene megfordítanunk). Ha kisebb, akkor ezentúl a minindex erre az elemre mutat.

A belső ciklusban tehát az i utáni elemek közül kiválasztottuk a legkisebbet. Ezután megnézzük, hogy a legkisebb elem az i-edik-e (tehát már eleve is a helyén volt, csak ezt nem tudtuk), ha nem, akkor fel kell cserélni őket.

Ha nem világos a működés, futtassuk le a programot. Ha még így sem, használjuk a debuggert, és úgy kövessük a működést, miközben figyeljük a változók értékének változását!

A buborékrendezés a következőképpen működik:

1. lépés: Megvizsgáljuk a tömb első és második elemét. Ha az első nagyobb, akkor felcseréljük őket. Ezután megvizsgáljuk a második és harmadik elemét, ha a második nagyobb volt, ismét cserélünk (figyeljük meg, hogy ha az első lépésben és a másodikban is volt csere, akkor az eredetileg az első helyen álló elem már a harmadikon van: ezúttal tehát a legnagyobb elem mozog a tömb vége felé, nem pedig a legkisebb az eleje felé, mint a közvetlen kiválasztásos rendezésnél!). Ezt a cserélgetést a tömb végéig ismételgetjük.
2. lépés: A tömb utolsó eleme tehát a legnagyobb. A következő lépésben szintén a tömb legelejéről kezdjük (ellentétben a közvetlen kiválasztásossal), de ezúttal csak az utolsó előttiig megyünk, hiszen az utolsó a legnagyobb.
3. lépés: A maradékban végigcserélgetjük a szomszédos elemeket, így a maradék legnagyobb eleme a maradék végére kerül.
4. lépés: A 3. lépést ismételgetjük, míg a maradék egy elem nem lesz.

Az algoritmust gyorsabbá tehetjük, ha figyeljük, hogy az adott lépésben volt-e csere. Ha nem volt, akkor a tömb rendezett, tehát nem kell tovább folytatnunk a rendezést.

Ha nem világos, érdemes megnézni a http://www.cs.bme.hu/~gsala/alg_anims/3/bsort-e.html weblapon szereplő animációt.

C nyelven az algoritmus a következőképp néz ki:

```
//******************************************************
void buborek(double * t,int meret){
//******************************************************
        int i,j,nemvoltcsere;
        double temp;

        for(i=0;i<meret-1;i++){
                nemvoltcsere=1;
                for(j=0;j<meret-1-i;j++){
                        if(t[j]>t[j+1]){
                                temp=t[j];t[j]=t[j+1];t[j+1]=temp; // csere
                                nemvoltcsere=0;
                        }
                }
                if(nemvoltcsere)break;
        }
}
```

Itt a külső ciklus szerepe, hogy beállítsa, meddig menjen a belső. A nemvoltcsere változóba 1-et állítunk be, és ha a belső ciklusban nem nullázzuk le (vagyis nem volt csere), akkor a breakkel kilépünk a külső ciklusból, mert a tömb már rendezett. A külső ciklus ezúttal is mehetne meret-ig, de akkor a belső meret-i-1-e az utolsó lépésben 0 lenne, tehát a belső le sem futna (bár ez az időveszteség jelentéktelen, úgyhogy mindegy).

A belső ciklus minden lépésben eggyel kevesebbig megy, mert az előző lépésben a maradék legnagyobb eleme került abban a lépésben utolsó helyre. Figyeljük meg, hogy ezúttal t[j] és t[j+1] elemeket cseréljük, ezért **itt szükséges a -1**, vagyis j<meret-1-i.

A működés jobb megértéséhez debuggerrel soronként is léptessük végig a futást, és közben nézzük a változók értékeit!

A teljes programban a „Press any key to continue…" kiírására, és a billentyűleütésre várakozásra a

```
        system("PAUSE"); // adm
```

függvényhívást használtuk. A system függvény (mely az stdlib.h-ban elérhető) az operációs rendszernek ad parancsot, tipikusan elindít egy programot. Ha pl. a PAUSE helyére beírjuk, hogy mspaint.exe, akkor a Paint fog elindulni új ablakban, a programunk pedig addig várakozik, míg a Paint-ből ki nem lépünk. A PAUSE a DOS/Windows rendszerek sajátja, Unix-ban ehelyett használhatjuk pl. az alábbi kódot:

```
{       char s[100];
        printf("Press ENTER to continue...\n");
        fflush(stdin);
        gets(s);
}
```

Melyik algoritmusnak mi az előnye? Ha a fenti programot futtatjuk, közvetlen kiválasztásos rendezés esetén az összehasonlítások száma 21, a másolásoké 12. Buborék rendezésnél az összehasonlítások száma 18, a másolásoké viszont 24. Ezeken az adatokon láthatóan a közvetlen kiválasztásos rendezés jobban teljesít. A másolás és az összehasonlítás ideje adatstruktúrától és számítógép felépítéstől függhet A fenti program esetén valószínűleg a másolás tart tovább, mert írni is kell, és az írás általában időigényesebb, mint az olvasás, de az összehasonlítás is jelentős időt vehet igénybe. A közvetlen kiválasztásos rendezésnél az összehasonlítások száma csak az adatok számától függ n(n-1)/2 (jelen esetben 6+5+4+3+2+1=21), míg buborék rendezés esetén előbb is véget érhet, ha a kiinduló tömb már eleve részben rendezett, és az utolsó néhány lépést nem kell végrehajtani (a fenti példában is azért 18 az összehasonlítás, mert az utolsó két lépés elmaradt). Közvetlen kiválasztásos rendezésnél a másolások száma max az összes adat háromszorosa, mert lépésenként legfeljebb egyszer cserélünk (de részben rendezett tömbnél ennyiszer sem), míg buboréknál –ahogy a fenti példa is mutatja – ez akár sokkal több is lehet.

5.1.2 Qsort

A C rendelkezik egy beépített rendező függvénnyel, a qsorttal. A q a quickre utal, általában hatékonyabb, mint a fenti kettő. Az alábbi program bemutatja a használatát:

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


//*******************************************************
void kiir(double * t,int meret){
//*******************************************************
        int i;
        for(i=0;i<meret;i++)printf("%d. elem: %.2f\n",i+1,t[i]);
}

int osszehasonlitas=0; // adm
```

```
//*******************************************************
int hasonlit(const void *a,const void *b){
//*******************************************************
        double *ia=(double *)a;
        double *ib=(double *)b;

        osszehasonlitas++; // adm

        if(*ia<*ib)return -1;  //<0
        if(*ia==*ib)return 0;  //==0
        return 1;              //>0
}


//*******************************************************
int main(){
//*******************************************************
        double t[7]={863,-37,54,9520,-3.14,25,9999.99};

        printf("Rendezes elott:\n");
        kiir(t,7);

        qsort(t,7,sizeof(double),hasonlit);

        printf("Rendezes utan:\n");
        kiir(t,7);
        printf("%d osszehasonlitas tortent.",osszehasonlitas);
        return 0;
}
```

A qsort meghívásán túl egy dolgunk van: el kell készíteni egy olyan függvényt, melynek bemenő paramétere két void* típusú pointer, visszatérési értéke int.

A void* típus nélküli pointer, akkor használjuk, ha többféle típusra is mutathat. Bármilyen pointerből könnyedén készíthetünk void*-ot, ha zárójelben elé írjuk: (void*). Ez az átalakítás a pointer által tartalmazott memóriacímet nem érinti, mert a pointer típusa csak arra szolgál, hogy tömb esetén ki tudja számolni a következő elem helyét. Mivel ennek a pointernek nincs típusa, az általa mutatott értékre nem hivatkozhatunk, csak ha a pointert visszaalakítjuk valamilyen típusúvá, ahogy ezt a fenti hasonlit függvényben látjuk.

Ezt a függvényt a qsort függvény hívja meg, és két tömbelemet hasonlít össze. Ha az első nagyobb, mint a második, akkor pozitív egész számot kell visszaadni a returnnel, ha egyformák, akkor 0-t, ha a második nagyobb, akkor negatívot.

Először a void* pointereket double*-gá alakítjuk, mert a tömbben double értékek vannak:

```
double *ia=(double *)a;
double *ib=(double *)b;
```

Ezután visszaadjuk a megfelelő értéket:

```
if(*ia<*ib)return -1;  //<0
if(*ia==*ib)return 0;  //==0
return 1;              //>0
```

-1 és 1 helyett más negatív és pozitív értéket is visszaadhattunk volna. Ha lefuttatjuk a programot, láthatjuk, hogy ugyanúgy 21 összehasonlítás történt, mint a közvetlen kiválasztásos rendezésnél.

A main-ben a következőképp hívjuk a qsortot:

```
qsort(t,7,sizeof(double),hasonlit);
```

Első paraméter a tömb neve, második az elemek száma, harmadik egy tömbelem mérete, negyedik pedig az összehasonlítást végző függvény neve (bármely függvény neve a zárójelek nélkül – hasonlóan a tömbökhöz – a függvényre mutató pointert jelenti).

A stringek qsorttal való rendezésére lássunk azt a programot, melyet a Visual C++ helpje tartalmaz:

```
/* QSORT.C: This program reads the command-line
 * parameters and uses qsort to sort them. It
 * then displays the sorted arguments.
 */

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

int compare( const void *arg1, const void *arg2 );

void main( int argc, char **argv )
{
   int i;
   /* Eliminate argv[0] from sort: */
   argv++;
   argc--;

   /* Sort remaining args using Quicksort algorithm: */
   qsort( (void *)argv, (size_t)argc, sizeof( char * ), compare );

   /* Output sorted list: */
   for( i = 0; i < argc; ++i )
      printf( "%s ", argv[i] );
   printf( "\n" );
}

int compare( const void *arg1, const void *arg2 )
{
   /* Compare all of both strings: */
   return _stricmp( * ( char** ) arg1, * ( char** ) arg2 );
}
```

Az _stricmp függvény nem szabványos, ez nem különbözteti meg a kis és nagybetűket, ha szabványossá akarjuk alakítani, ehelyett használjuk a strcmp-t.

Látható, hogy az összehasonlító függvény nagyon egyszerű, mert a strcmp (ill. stricmp) függvény pont olyan értékeket ad vissza. ami nekünk kell (ez nem véletlen egybeesés).

A fenti programban a main() egy olyan változatával találkozunk, amit eddig még nem láttunk: két bemenő paramétere van (van egy három paraméteres változat is, de azzal nem foglalkozunk ebben a jegyzetben). Aki csak a Windows használatához szokott, talán még nem találkozott azzal a lehetőséggel, hogy paramétereket adjon egy programnak, de remélhetőleg ők is megértik, miről van szó. Ha parancssorból akarunk másolni egy fájlt, ezt írjuk:

copy c:\c.txt d:\y.txt

Bizonyára a copy.exe is egy copy.c (vagy copy.cpp) volt egyszer, és a két paraméterét a main() paramétereként olvasták be.

A main első paramétere (itt argc-nek nevezzük, de tetszőleges változónevet adhatunk) egy egész szám, mely megmondja, hogy hány eleme van a második paraméterként kapott pointer tömbnek. Az argv tömb stringekre mutató pointereket tartalmaz.

Az argv[0] a program nevét tartalmazza, a fenti esetben copy (esetleg útvonallal és/vagy .exe kiterjesztéssel).

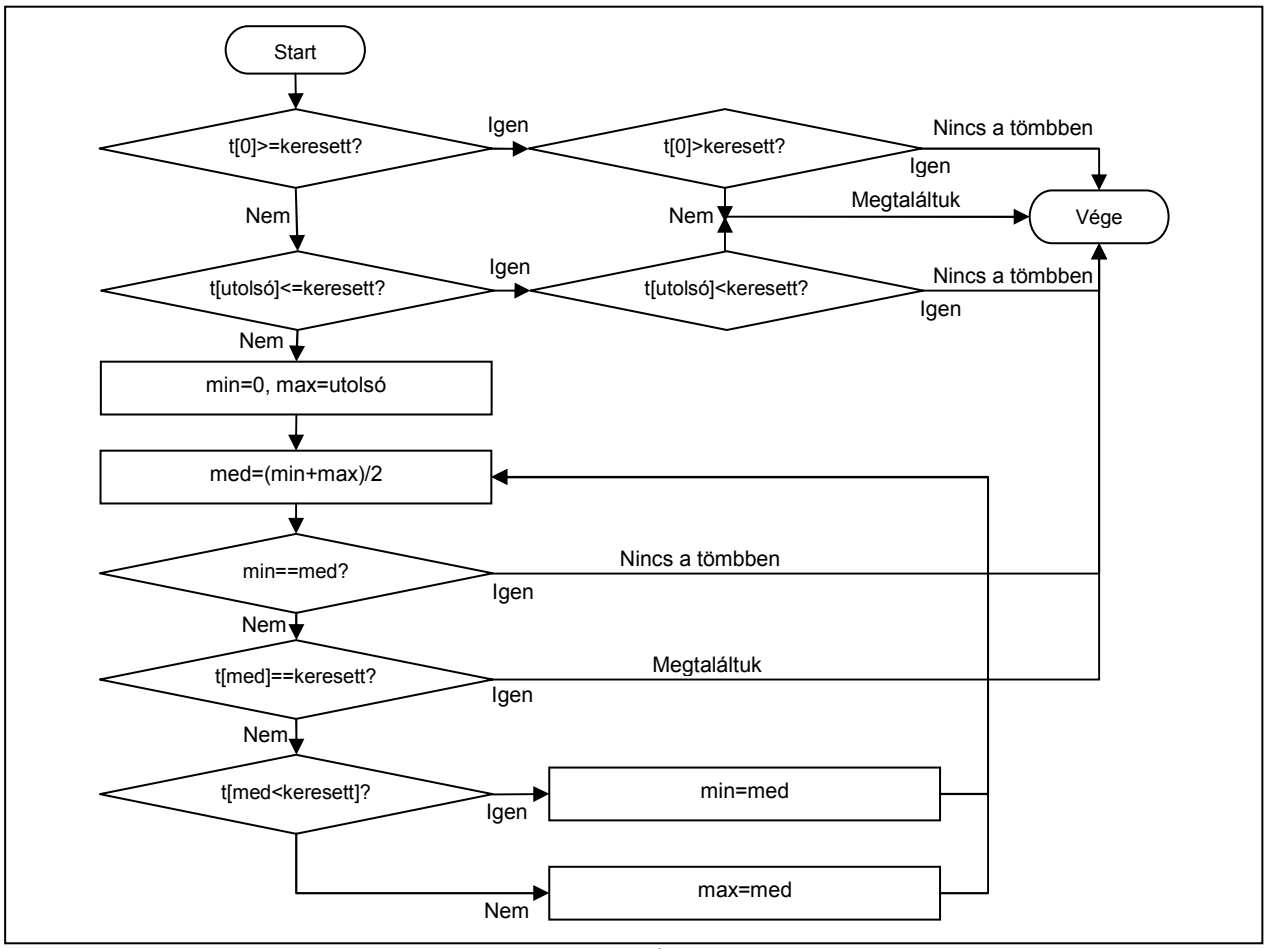
Az argv[1]-argv[argc-1] pedig a paraméterekre mutat. A fenti esetben argv[1]=” c:\\c.txt”, argv[2]=” d:\\y.txt”. (A \\ egy darab back slasht jelent.)

A program első lépésben kiveszi a tömbből az argv[0]-t, majd az ismert módon meghívja a qsortot.

5.1.3 Keresés

Abban az esetben, ha az adatstruktúránk nincs valamiképpen rendezve vagy indexelve, csak egy keresési mód jöhet számításba: a lineáris keresés. Ekkor végiglépkedünk az összes elemen, és összehasonlítjuk a keresett elemmel. Ezzel a módszerrel még példa szintjén sem foglalkozunk ebben a fejezetben, ugyanis aki az eddig tanultak és a leírás alapján nem tudja megírni programban, az jobb, ha előröl kezdi a félévet!

Mi a helyzet, ha rendezett tömbről van szó? Ebben az esetben a leghatékonyabb keresési módszer a bináris keresés, mely a következőképpen működik:



9. ábra

Azaz: a min és max indexek kezdetben a tömb két szélén állnak, majd minden lépésben feleződik köztük a távolság, körülkerítik a keresett elemet. A keresés ideje $\log_2 n$ nagyságrendű, ahol n a tömb számossága. (Lineáris keresésnél n/2 nagyságrendű).

Bináris keresést valósít meg a C nyelv standard könyvtárához tartozó bsearch függvény, lásd pl. [1].

5.1.4 Hashing

Bár a hashing megvalósítására célszerűbb lenne dinamikus adatszerkezetet használni, de demonstrációs célra a fix tömbök is megfelelnek. A hashing lényege, hogy a tárolt adathoz előállít egy egész számot a 0–(N-1) tartományban, N egy alkalmasan választott érték. Ez az

index érték választja ki, hogy az N elemű tömb melyik eleme tárolja azt az adatstruktúrát, vagy adatstruktúrára mutató pointert, amelyben az adat tárolódik. Ez az adatstruktúra lehet tömb, de lehet pl. láncolt lista is. Ezen belül az adatstruktúrán belül az adatok rendezetlenül helyezkednek el.

A hashing előnye, hogy a keresési idő a lineárishoz képest kb. 1/N-ed részére csökken a lineáris kereséshez képest, ha az adatok viszonylag egyenletesen helyezkednek el az N blokkban.

Az alábbi programban emberek nevét és címét tároljuk hashelt szerkezetben.

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


//*********************************************************
#define HASH 16
#define ELEMSZAM 30
//*********************************************************


//*********************************************************
typedef struct{
//*********************************************************
        int van;
        char nev[30];
        char cim[70];
} adat; // Itt kötelező a pontosvessző!!!


//*********************************************************
adat t[HASH][ELEMSZAM];
//*********************************************************


//*********************************************************
unsigned GetHash(char * s){
//*********************************************************
        unsigned i,sum=0;

        for(i=0;i<4;i++){
                if(s[i]==0)break;
                sum+=(unsigned)s[i];
        }
        return sum%HASH;
}


//*********************************************************
int add(adat * a){
//*********************************************************
        unsigned index,i,siker=0;

        index=GetHash(a->nev);
        for(i=0;i<ELEMSZAM;i++)if(t[index][i].van==0){
                t[index][i]=*a;
                t[index][i].van=1;
                siker=1;
                break;
        }
        return siker;
}


//*********************************************************
char * keres(char * s){
//*********************************************************
        unsigned index=GetHash(s),i;

        for(i=0;i<ELEMSZAM;i++)
                if(t[index][i].van==1&&strcmp(s,t[index][i].nev)==0)
                        return t[index][i].cim;
        return NULL;
}


//*********************************************************
void init(){
//*********************************************************
        int i,j;
```

```
        for(i=0;i<HASH;i++)for(j=0;j<ELEMSZAM;j++)t[i][j].van=0;
}


//*******************************************************
main(){
//*******************************************************
        init();

        while(1){
                char c;
                adat a;

                printf("\nKivan meg adatokat felvenni? ");
                do{
                        printf("(i/n) ");
                        fflush(stdin);
                        c=getchar();
                }while(c!='i'&&c!='I'&&c!='n'&&c!='N');
                if(c=='n'||c=='N')break;

                printf("\nNev: ");
                fflush(stdin);
                if(gets(a.nev)==NULL){
                        printf("Sikertelen adatbevitel\n");
                        exit(-1);
                }

                printf("\nCim: ");
                fflush(stdin);
                if(gets(a.cim)==NULL){
                        printf("Sikertelen adatbevitel\n");
                        exit(-1);
                }

                if(!add(&a))printf("Az adatot nem tudtam felvenni, nem fer a memoriaba!");
        }

        while(1){
                char c,nev[30],*cim;

                printf("\nKivan meg keresni? ");
                do{
                        printf("(i/n) ");
                        fflush(stdin);
                        c=getchar();
                }while(c!='i'&&c!='I'&&c!='n'&&c!='N');
                if(c=='n'||c=='N')break;

                printf("Kit keres?\n");
                fflush(stdin);
                if(gets(nev)==NULL){
                        printf("Sikertelen adatbevitel\n");
                        exit(-1);
                }

                if((cim=keres(nev))==NULL)
                        printf("A keresett szemely nem szerepel adatbazisunkban!\n");
                else printf("Cim: %s\n",cim);
        }
}
```

Gyakran előfordul, hogy összetartozó, különböző típusú adatokat akarunk tárolni. Erre szolgál a struktúra (vagy rekord) adattípus. A struktúrákkal kapcsolatos részletek: lásd pl. [1].

```
//*******************************************************
typedef struct{
//*******************************************************
        int van;
        char nev[30];
        char cim[70];
} adat; // Itt kötelező a pontosvessző!!!
```

A néven és címen kívül egy van nevű változót is tettünk a struktúrába, ezzel jelezzük, hogy az adott tömbelemben érvényes adat van-e, vagy sem.

```
adat t[HASH][ELEMSZAM];
```

Globális változóban tároljuk az adatokat. Ez egy kétdimenziós tömb, az első adja a hashing N értékét, a második, hogy egy blokkba hány adat fér.

```
//******************************************************
unsigned GetHash(char * s){
//******************************************************
        unsigned i,sum=0;

        for(i=0;i<4;i++){
                if(s[i]==0)break;
                sum+=(unsigned)s[i];
        }
        return sum%HASH;
}
```

Azt, hogy az adat melyik blokkba kerül az N közül, a GetHash függvény számítja ki olymódon, hogy kiszámítja az első négy betű összegét (ha rövidebb, akkor kevesebbet), majd veszi az N-nel (vagyis 16-tal) való osztás maradékát (itt az N-t HASH-nek nevezzük). Ez remélhetőleg kellően egyenletes eloszlást eredményez.

```
//******************************************************
int add(adat * a){
//******************************************************
        unsigned index,i,siker=0;

        index=GetHash(a->nev);
        for(i=0;i<ELEMSZAM;i++)if(t[index][i].van==0){
                t[index][i]=*a;
                t[index][i].van=1;
                siker=1;
                break;
        }
        return siker;
}
```

Az add függvénnyel beteszünk egy ilyen adatot a t tömbbe. Ez a függvény a struktúrára mutató pointert kap. Itt nem szükséges pointerrel átadni, hogy mégis ezt tesszük, annak az az oka, hogy míg egy pointer átadása mindössze 4 (2, 8) bájtot jelent, addig a teljes struktúra 104 (102) bájt, és ezt nem akarjuk bemásolni.

Ha egy struktúra adattagjára hivatkozunk, akkor a . operátort használjuk. Például a fenti kódban t[index][i].van=1;. **Ha a struktúra pointerrel van megadva, akkor (*p).elem formában kellene rá hivatkozni** (a zárójel szükséges, mert a * precedenciája kisebb, mint a .-é). **Hogy ne kelljen ilyen bonyolultan írni, bevezették a nyíl operátort, ami ugyanazt jelenti: p->elem.** A fenti kódban a->nev esetén használjuk ezt.

Figyeljük meg a fenti kódban a

```
                t[index][i]=*a;
```

sort! **Teljes struktúrát tudunk másolni az egyenlőségjellel! Akkor is, ha benne karaktertömbök vannak!** Ilyen esetben tehát felesleges külön-külön másolni az adattagokat, a karaktertömbök esetén strcpy-vel, mert így nem csak rövidebb a kód, hanem a másolás is gyorsabb!

```
//******************************************************
char * keres(char * s){
//******************************************************
        unsigned index=GetHash(s),i;

        for(i=0;i<ELEMSZAM;i++)
                if(t[index][i].van==1&&strcmp(s,t[index][i].nev)==0)
                        return t[index][i].cim;
        return NULL;
}
```

Keresésnél ismét előállítjuk a hash indexet, majd az ehhez tartozó tömböt végignézzük, hogy szerepel-e benne az adott név. Csak azokat az elemeket nézzük, ahol a van értéke 1. Ha megtaláltuk, visszaadjuk a névhez tartozó cím memóriacímét, ha nem találtuk meg, NULL pointert. (A NULL pointer több headerben is definiálva van, értéke C kód esetén (void*)0, C++ kód esetén simán 0).

Megj.: Ez a tárolási forma lehetővé teszi, hogy az elemeket egyszerűen törölhessük: ilyen esetben egyszerűen csak a van értékét kell 0-ra állítani.

```
//******************************************************
void init(){
//******************************************************
        int i,j;

        for(i=0;i<HASH;i++)for(j=0;j<ELEMSZAM;j++)t[i][j].van=0;
}
```

Ez a függvény az összes elem van értékét 0-ra állítja, azaz törli.

A main() függvényben gets()-sel olvassuk a neveket és címeket, mert ezekben valószínűleg szóköz is van, és a scanf() csak szóközig olvasna.

5.2 Programírás

1. Írjon C függvényt, amely a paraméterként kapott, rendezett, double típusú elemeket tartalmazó tömbre megvalósítja a bináris keresést! Ha megtalálta a tömbben a megadott értéket, egy erre mutató pointert adjon vissza, egyébként NULL pointert.
2. Írjon C függvényt, mely stringeket tartalmazó tömböt rendez. A tömbben tárolt stringek maximális hossza 80 karakter.
    a. közvetlen kiválasztásos rendezéssel növekvő sorrendbe
    b. közvetlen kiválasztásos rendezéssel csökkenő sorrendbe
    c. buborék rendezéssel növekvő sorrendbe
    d. buborék rendezéssel csökkenő sorrendbe
3. Írjon C függvényt, amely stringekre mutató pointerek tömbjét kapja paraméterként, és a stringeket csökkenő sorrendbe rendezi.
4. Hozzunk létre egy nagyméretű double tömböt, töltsük fel véletlenszerű értékekkel, és rendezzük a három rendező algoritmussal. Mérjük az időt. Melyik milyen gyors? (a méretet kísérletezéssel dönthetjük el, célszerű pl. 1000, 10000, 100000, stb. elemű tömböt létrehozni, hogy ne fussunk bele egy túl lassú futásba (a méret tízszerezése kb. százszorosára növeli a futási időt!) Csak 32 (vagy több) bites fordítóval érdemes kísérletezni, DOS-ban max 64 kB-os tömbjeink lehetnek).
5. Írjon C függvényt, amely stringekre mutató pointerek tömbjét, valamint egy stringet kap paraméterként, és bináris kereséssel megkeresi a stringet. A függvény egész számot adjon vissza, melynek értéke 1, ha megtalálta, és 0, ha nem találta meg.
6. Egészítsük ki a hashinget demonstráló kódot az elemtörlés lehetőségével.
7. Írjuk át a hashinget demonstráló programot oly módon, hogy menüből lehessen kiválasztani, hogy most épp felvenni, keresni, vagy törölni, valamint kilépni akarunk-e.

6. Fájlkezelés, dinamikus tömbök

6.1 Programértés

A C nyelv kétféle fájltípus kezelését támogatja:

- Általános (bináris) fájlok: A bináris fájl egy adatfolyam (stream). A beépített fájlkezelő függvények nem foglalkoznak azzal, hogy mi van a fájlban, ők csak beolvasnak vagy kiírnak annyi bájtot, amennyire mi utasítást adunk. Ezután azt kezdünk az adatokkal, amit akarunk. Ily módon kezelhetünk szövegeket, képeket, zenéket, filmeket, adatbázisokat, bármit, de a fájlformátumokat nekünk kell ismerni.
- Szöveges fájlok: A C nyelv egy fájltípushoz rendelkezik támogatással, és ez sem mondható túl jónak. Aki ismeri pl. a PHP-t, az látni fogja, hogy a C nyelv semmiféle olyan lehetőséggel nem rendelkezik változók mentésére, mint a serialize/unserialize. Szöveges fájlokat a scanf/printf/gets/puts/getchar/putchar stb. függvények fájlba dolgozó változatával kezelhetjük.

Amikor a C nyelv kialakult, még a képernyő sem volt elterjedten használt periféria, médiafájlokról pedig szó sem volt, ezért nem is készült ezekhez támogatás. A modernebb nyelvek általában rendelkeznek pl. grafikus fájlokat kezelő függvényekkel, osztályokkal (Java, PHP stb.), sajnos a C++ nem ilyen.

6.1.1 Színszámláló

Első programunk beolvas egy 256 színű BMP fájlt, és kiírja, hogy melyik színből hány van benne. Mivel a C nyelv nem ismeri a BMP formátumot, nekünk kell. A program (min.) 32 bites fordítóhoz készült, 16 bites fordítónál a BUFFSIZE méretet csökkentsük 64 kB alá, és az unsigned/int adatokat (unsigned) longra, továbbá a kiírásnál is a megfelelő formátumot állítsuk be.

```
//*********************************************************
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//*********************************************************


//*********************************************************
#define BUFFSIZE 1048576
//*********************************************************


//*********************************************************
unsigned char Fej[2048];
unsigned char Buff[BUFFSIZE];
unsigned Szamol[256];
//*********************************************************


//*********************************************************
void main(){
//*********************************************************
        char FileName[1024];
        FILE *fp;
        unsigned i=BUFFSIZE,j;

        printf("A fajl neve: ");
        fflush(stdin);
        if(gets(FileName)==NULL){printf("Hibas nevmegadas\n");exit(-1);}

        fp=fopen(FileName,"rb");
        if(!fp){puts("Fajlnyitas nem ment\n");exit(-1);}
```

```
        fread(Fej,sizeof(char),1078,fp);
        for(j=0;j<256;j++)Szamol[j]=0;
        while(i==BUFFSIZE){
                i=fread(Buff,sizeof(char),BUFFSIZE,fp);
                for(j=0;j<i;j++)Szamol[Buff[j]]++;
        }
        fclose(fp);
        j=0;
        for(i=0;i<256;i++)if(Szamol[i]){
                printf("%d.\t%12u\tB=%u\tG=%u\tR=%u\tA=%u\n",i,Szamol[i],
                        Fej[54+i*4],Fej[54+i*4+1],Fej[54+i*4+2],Fej[54+i*4+3]);
                j+=Szamol[i];
        }
        printf("Ossz:\t%12u\n",j);
        printf("Press ENTER to continue...\n");
        gets(FileName);
}
```

A fájl megnyitása a **fopen()** függvénnyel történik. Első paramétere a fájl neve (útvonalat is tartalmazhat), második paramétere a megnyitás módját jelenti: 'r'=read, olvasásra, 'w'=write, írásra, 'a'=append, hozzáfűzésre. Ha a betű után teszünk egy '+' jelet, akkor egyszerre írhatjuk és olvashatjuk a fájlt (ezt ritkán használjuk). Az r+ és w+ között az a különbség, hogy az első nem törli a fájl korábbi tartalmát, az utóbbi igen. A második betű lehet 'b'=binary vagy 't'=text. A 't' el is maradhat (tehát pl. a "w" ugyanazt jelenti, mint a "wt").

A fopen() visszatérési értéke egy FILE* típusú pointer. A FILE egy struktúra, melynek tartalma számunkra teljesen lényegtelen, ezzel csak a fájlkezelő függvények foglalkoznak. ha a fopen() NULL pointert ad vissza, nem sikerült a fájl megnyitása (pl. olvasásra próbáltunk megnyitni egy nem létező fájlt, vagy írásra egy írásvédettet).

Bináris fájlból olvasni az **fread()** függvénnyel tudunk (navajon akkor írni mivel tudunk?:-). A függvény **első paramétere egy pointer**, mely arra a memóriaterületre mutat, **ahová a beolvasott adatokat tenni akarjuk**. **Második paramétere**, tömbnek tekintve a beolvasott adatstruktúrát, **egy tömbelem mére**te, **harmadik paramétere** pedig **a tömb elemeinek száma** (bár ennek a szétválasztásnak nincs komoly jelentősége, a függvény úgyis annyi bájtot olvas, amennyi ennek a kettőnek a szorzata). **Negyedik paraméter a fájl pointer.**

A fenti programban 1078 bájtot olvasunk be, ennyi ugyanis a 256 színű BMP fájlok fejlécének mérete, mely tartalmazza a kép felbontását, színmélységét, egyéb adatait, valamint a 256 szín palettáját.

A fejléc beolvasása után BUFFSIZE lépésekben olvassuk be a képpontokat, és megszámoljuk az egyes színekhez tartozó pixeleket.

Ha befejeztük a munkát a fájllal, az fclose() függvénnyel be kell azt zárni.

Ezt követően kiírjuk az eredményeket.

6.1.2 Dinamikus tömb. struktúrák bináris fájlban

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


//************************************************************
typedef struct{
//************************************************************
        int ev, ho, nap;
}datum;


//************************************************************
typedef struct{
//************************************************************
        char nev[32];
        char cim[64];
        datum szul;
}adatok;
```

```
//************************************************************
adatok * dat=NULL;
unsigned elemszam=0,maxelem=0;
//************************************************************


//************************************************************
void error(const char *uzenet){
//************************************************************
        printf("\nError: %s\n",uzenet);
        exit(-1);
}


//************************************************************
void fajlolvas(const char *FileName){
//************************************************************
        FILE *fp;

        fp=fopen(FileName,"rb");
        if(fp==NULL)error("Cannot open file");

        elemszam=fread(dat,sizeof(adatok),maxelem,fp);
        fclose(fp);
}


//************************************************************
void fajlir(const char *FileName){
//************************************************************
        FILE *fp;

        fp=fopen(FileName,"wb");
        if(fp==NULL)error("Cannot open file");

        if(fwrite(dat,sizeof(adatok),elemszam,fp)!=elemszam)
                error("Cannot write data to file");
        fclose(fp);
}


//************************************************************
void billolvas(){
//************************************************************
        elemszam=0;

        if(dat!=NULL){ // ha létezik a tömb, felszabadítjuk
                free(dat);
                dat=NULL;
        }

        printf("Az elemek szama max: ");
        fflush(stdin);
        while(scanf("%u",&maxelem)!=1)fflush(stdin);
        if(maxelem==0)error("nem lehet 0");
        dat=(adatok*)calloc(maxelem,sizeof(adatok));
        if(dat==NULL)error("allokacios hiba");

        while(elemszam<maxelem){
                printf("Nev:\t");
                fflush(stdin);
                gets(dat[elemszam].nev);
                if(strcmp(dat[elemszam].nev,"")==0)break;
                printf("Cim:\t");
                fflush(stdin);
                gets(dat[elemszam].cim);
                printf("Szul. ev:\t");
                fflush(stdin);
                scanf("%d",&(dat[elemszam].szul.ev));
                printf("Szul. ho:\t");
                fflush(stdin);
                scanf("%d",&(dat[elemszam].szul.ho));
                printf("Szul. nap:\t");
                fflush(stdin);
                scanf("%d",&(dat[elemszam].szul.nap));
                elemszam++;
        }
}


//************************************************************
void kepir(){
//************************************************************
        unsigned i;
        for(i=0;i<elemszam;i++){
                printf("Nev:\t%s\n",dat[i].nev);
```

```
                printf("Cim:\t%s\n",dat[i].cim);
                printf("Szul:\t%d.",dat[i].szul.ev);
                printf("%d.",dat[i].szul.ho);
                printf("%d.\n\n",dat[i].szul.nap);
        }
}

//*********************************************************
void main(){
//*********************************************************
        billolvas();
        fajlir("adatok.xyz");
        fajlolvas("adatok.xyz");
        kepir();
        if(dat!=NULL)free(dat);
}
```

Gyakran előfordul, hogy a program készítésének időpontjában nem tudjuk, hogy hány adattal kell dolgoznunk. Eddig ezt úgy oldottuk meg, hogy akkora tömböt készítettünk, melybe „remélhetőleg belefér” a kívánt adatmennyiség. Ha nem fért volna bele, akkor hibaüzenetet adtunk. Ez nem túl szép és praktikus megoldás. A C nyelv lehetővé teszi, hogy ne fordításkor, hanem a futtatás során döntsük el, mennyi memóriára van szükségünk, vagyis dinamikusan foglaljunk memóriát. Ebben a fejezetben dinamikus tömböket foglalunk le.

Ha dinamikus tömbben tároljuk az adatokat, akkor a tömb lefoglalása előtt tudnunk kell, hogy hány adatra lesz szükségünk. Bár lehetőség van futási időben is megváltoztatni a dinamikus tömb méretét, ezt a megoldást lehetőleg kerüljük, mert időidényes művelet. A későbbiekben látni fogunk olyan adatstruktúrákat, ahol nem kell előre tudnunk az adatok számát. Ennek hátránya, hogy az adatok nem tömbben tárolódnak, tehát bejárásuk, rendezésük bonyolultabb folyamat.

Dinamikus adat lefoglalására szükségünk lesz egy pointerre, mely a lefoglalt helyre mutat majd a memóriában:

```
adatok * dat=NULL;
```

Kezdetben NULL értéket adunk neki. Ez azért hasznos, mert ha megvizsgáljuk a pointer értékét, tudni fogjuk, hogy lefoglalt memóriaterületre mutat-e, vagy sem. Erre látunk példát a billolvas() függvényben:

```
        if(dat!=NULL){ // ha létezik a tömb, felszabadítjuk
                free(dat);
                dat=NULL;
        }
```

A lefoglalt memóriaterület felszabadítására a free() függvény szolgál. Miért szabadítjuk fel előbb, mint ahogy lefoglalnánk? Nos, nem ez a helyzet, ugyanis előfordulhat, hogy egy programon belül kétszer is meghívják ezt a függvényt, és ebben az esetben, ha nem szerepelne a felszabadítás, a lefoglalt memóriára mutató pointert a következő blokkban átirányítanánk az újonnan lefoglalt részre, miközben a korábban lefoglaltakra semmi sem mutatna, tehát lenne egy memóriablokk, ami a programé, de nem képes használni. Ezen a módon nagyon könnyen betelik az egész memória. és utána elszáll a program.

Javaban jártas olvasók figyelmét felhívnánk arra, hogy a C-ben nincs automatikus szemétgyűjtés, tehát **a lefoglalt memóriát mindig fel kell szabadítani**!

```
        printf("Az elemek szama max: ");
        fflush(stdin);
        while(scanf("%u",&maxelem)!=1)fflush(stdin);
        if(maxelem==0)error("nem lehet 0");
        dat=(adatok*)calloc(maxelem,sizeof(adatok));
        if(dat==NULL)error("allokacios hiba");
```

A tömb lefoglalása a calloc() függvénnyel történik. Első paramétere a tömb elemeinek száma, második pedig egy elem mérete (tehát pont fordítva, mint a fread-fwrite függvényeknél). A calloc() lenullázza a lefoglalt memóriát. A calloc() void* pointert ad vissza, ezt konvertáljuk (adatok*) típusúra. Ha nem sikerült a foglalás, akkor NULL pointert kapunk. Ezután a dinamikus tömböt ugyanúgy kezelhetjük, mint a fix (statikus) tömböket.

A memóriát az operációs rendszertől foglaljuk le, és annak adjuk vissza.

A fájlírás és olvasás ugyanúgy történik, mint a színszámláló programnál.

Memóriát foglalni a calloc()-on kívül a malloc() függvénnyel is lehet. A malloc() csak egy értéket vár: azt, hogy hány bájtot akarunk lefoglalni. A fenti kódban ezt is írhattuk volna:

```
dat=(adatok*)malloc(maxelem*sizeof(adatok));
```

Ez egy dolgot kivéve ugyanazt eredményezi, mint a calloc(): a malloc() nem tölti fel 0 bájtokkal a lefoglalt memóriát, szemben a calloc()-kal. (Mondhatjuk azt is, hogy a malloc hasonló a nemdinamikus változókhoz: kezdeti értékük memóriaszemét, míg calloc()-nál csupa 0, ezért calloc()-nál elhagyható a kezdeti érték beállítása).

Lehetőség van a lefoglalt memória méretének megváltoztatására is a realloc() függvénnyel. Ennek első paramétere a korábban lefoglalt, megváltoztatandó méretű memóriaterületre mutató pointer, második paramétere az új méret, pl.:

```
dat=(adatok*)realloc(dat,maxelem*sizeof(adatok)+10);
```

A realloc() függvény úgy működik, hogy lefoglal egy, a második paraméternek megfelelő méretű memóriablokkot a malloc()-kal, megállapítja, hogy mekkora az első paraméterként kapott memóriaterület (erre nincs szabványos C függvény), ha az új memóriaterület nagyobb, mint az eredeti, akkor az egészet átmásolja, ha kisebb, akkor csak annyit, amennyi éppen belefér az újba. Ebből látszik, hogy **a realloc() használata nagyobb blokkoknál elég időigényes**.

A memóriaterületek közti másolásra írhatunk pl. for ciklust (for(i=0;i<n;i++)a[i]=b[i];), de használhatjuk a C nyelv beépített függvényeit is: a memcpy()-t, és a memmove()-ot.

```
memcpy(ide,innen,bajtszam);
memmove(ide,innen,bajtszam);
```

A kettő között az a különbség, hogy ha a forrás és cél memóriaterület részben vagy egészben átfedi egymást, a memcpy nem biztos, hogy jó eredményt ad, a memmove pedig igen. (Ez akkor fordulhat elő, ha egy tömb egyik részét másoljuk át önmaga egy másik részébe.)

6.1.3 Szöveges fájlok

Valósítsuk meg az előző programot úgy, hogy ezúttal szöveges fájlba ír, és onnan olvas. Ehhez mindössze két függvényt kell módosítani, a fajlolvas()-t, és a fajlir()-t.

```
//*********************************************************
void fajlir(const char *FileName){
//*********************************************************
        FILE *fp;
        unsigned i;
```

```
        fp=fopen(FileName,"wt");
        if(fp==NULL)error("Cannot open file");

        for(i=0;i<elemszam;i++){
                fprintf(fp,"%s\n%s\n%d\t%d\t%d\n",dat[i].nev,dat[i].cim,
                        dat[i].szul.ev,dat[i].szul.ho,dat[i].szul.nap);
        }
        fclose(fp);
}
```

Ezúttal nem tudjuk egy utasítással elintézni az írást, mint a bináris fájlnál. Egy ciklussal végiglépkedünk a tömb elemein, és az fprintf() függvény segítségével kiírjuk. Az fprintf() ugyanúgy használható, mint a printf(), azzal a különbséggel, hogy az eredmény az fp-vel megadott fájlba kerül.

Ne zavarjon senki, hogy csak egy fprintf()-et használtunk, nyugodtan használhatunk akár ötöt is.

Miért írtuk három sorba az adatokat? Mert a név és a cím valószínűleg szóközöket is tartalmaz, valamiképpen el kell választani őket egymástól. Azért ezt a módszert választottuk, mert így egy egyszerű fgets()-sel be fogjuk tudni olvasni a nevet és a címet, és több dolgunk nincs. Egy másik megoldás lehet, ha mondjuk idézőjelek közé írjuk a nevet és a címet, aztán beolvasáskor karakterenként olvasunk, és tudjuk, hogy az első idézőjeltől a másodikig tart a szöveg. Ez láthatóan bonyolultabb lenne. A számokat egy sorba írjuk, és egyszerű tabulátorral választjuk el őket egymástól. Írhatnánk ezeket is külön sorba, de erre nincs szükség.

Ha adatokat írunk szöveges fájlba, melyeket később vissza szeretnénk olvasni, mindig azt tartsuk szemünk előtt, hogy hogyan lesz a legkényelmesebb a visszaolvasás. Gondoljuk meg, hogy ha pl. egy ilyen kiírást használnánk:

```
        fprintf(fp,"%d%d%d%d%d",1,23,456,7,89);
```

Ekkor ez kerülne a fájlba: 123456789. Nincs az a program, ami ebből kideríti, hogy mik voltak az eredeti számok, hiszen nem tettünk közéjük semmilyen elválasztójelet.

```
//*********************************************************
void fajlolvas(const char *FileName){
//*********************************************************
        FILE *fp;
        unsigned i;
        char s[256];

        fp=fopen(FileName,"rt");
        if(fp==NULL)error("Cannot open file");

        for(i=0;i<elemszam;i++){
                if(fgets(dat[i].nev,31,fp)==NULL)error("Nev nem olvashato");
                if(fgets(dat[i].cim,63,fp)==NULL)error("Cim nem olvashato");
                if(fgets(s,255,fp)==NULL)error("Szuletesi ido nem olvashato");
                if(sscanf(s,"%d %d %d",&dat[i].szul.ev,&dat[i].szul.ho,&dat[i].szul.nap)!=3)
                        error("Szuletesi ido nem megfelelo formatumu.");
        }
        fclose(fp);
}
```

Ahogy látjuk, a beolvasás bonyolultabb. Mindhárom sort a fgets()-sel olvassuk, ennek az az oka, hogy ha fscanf()-et használnánk a fgets()-sel vegyesen, akkor a három szám után a sorvégjel ott maradna a következő beolvasásra várva, ezt pedig a következő fgets() kapná meg.

Az fgets() három paraméterrel rendelkezik: a karaktertömbre mutató pointer, ahová a beolvasott szöveg kerül, a maximális karakterszám (ha ennél több van a sorban, akkor csak ennyit olvas, a többit a következő fgets() vagy fscanf() stb. olvassa), és a fájlpointer.

A dátum három számjegyét egy sorba írtuk, és ezeket a gets()-sel olvassuk be az s tömbbe. Az s stringből a sscanf() függvény segítségével olvassuk ki a számokat. Ez ugyanúgy működik, mint a scanf vagy a fscanf, csak a bemenet nem a billentyűzet vagy egy fájl, hanem egy string.

Ha kipróbáljuk a fenti programot Borland C++, vagy Visual C++ fordítóval, láthatjuk, hogy kiírásnál a név után és a cím után is egy-egy üres sor következik, vagyis a fscanf() otthagyott egy soremelést a beolvasott string végén. Nem tudom, hogy ez Unix alatt is így van-e, ha nem, akkor ez annak a következménye, hogy a Microsoft operációs rendszerei alatt (korábban biztos, mert újabban az ilyen dolgokat kezdik kijavítani, például a WinXP alatt nyugodtan megadhatjuk az útvonalat / jelekkel is, nem kell \) a fájlba írt \n hatására valójában egy \n\r páros kerül kiírásra (soremelés, kocsivissza, ez utóbbi a nyomtatóra utal), és beolvasáskor az fgets() csak az egyiket nyeli el. (A billentyűzetről olvasó gets()-nél nem tapasztalunk ilyesmit.) Ha a beolvasott nevek között pl. keresni szeretnénk, akkor célszerű eltávolítani azt a felesleget (a \n kódja 13, a \r-é 10, tehát pl. for(i=strlen(s)-1;s[i]==10|| s[i]==13;i--)s[i]=0;)

Ha nem írunk a fájlba stringeket, csak számokat, vagy a stringben biztos nincs whitespace karakter, akkor a beolvasáshoz használjuk nyugodtan a fscanf()-et, az kényelmesebb. (De a legkényelmesebb a bináris fájlok kezelése, úgyhogy ha nem muszáj, ne használjunk szöveges fájlt.)

Ha lefuttattuk az előző két programot, akkor fogunk találni a winchesteren egy adatok.xyz, és egy adatok.zyx nevű fájlt. Az előbbi bináris, az utóbbi szöveges. Nézzünk beléjük a Total Commanderrel, az F3-at megnyomva! (Ennek hiányában a Notepad is megteszi.) Láthatjuk, hogy a szöveges fájl ezzel a programmal jól olvasható, a bináris pedig tele van mindenféle krix-kraxszal, ez azért van, mert a szövegszerkesztő/olvasó programok ismerik a szöveges fájlok kódolását, de az általunk definiált struktúrájú binárisét nem.

Még egy fontos következmény, talán nem mindenkinek nyilvánvaló: a text fájlok kiterjesztése nem csak .txt, hanem bármi lehet, a bináris fájloké sem csak .dat, az is bármi lehet. Ezeket a formákat csak azért szoktuk használni, hogy könnyebben eligazodjunk a fájlnevek dzsungelében.

Akár szöveges, akár bináris fájloknál használható a fájlon belüli pozícionálás, illetve pozíció lekérdezése. Az ftell() függvény egy fájl pointert kap paraméterként, és megmondja, hogy hanyadik bájt következik, visszatérési értéke long típusú:

```
long n=ftell(fp);
```

Ez például akkor hasznos, ha egy sort kétszer akarunk beolvasni. Akkor az első olvasás előtt lekérdezzük a pozíciót, ezt eltároljuk egy változóban, majd a beolvasás után az fseek()-kel visszaállítjuk:

```
fseek(fp,n,SEEK_SET);
```

A SEEK_SET azt jelzi, hogy n a fájl kezdetétől fogva értelmezett. Harmadik paraméterként két másik lehetőség is van: SEEK_END: a fájl végétől számít (ha pozitív értéket adunk meg, akkor a fájl vége után ennyivel áll be a pozíció, ha tehát pl. a fájl utolsó 100 bájtját akarjuk kiolvasni, akkor fseek(fp,-100,SEEK_END; szükséges, tehát **n lehet negatív is**!). Harmadik lehetőség a SEEK_CUR, ami a jelenlegi pozícióhoz képesti beállítást tesz lehetővé.

6.1.4 Többdimenziós dinamikus tömb

```
//*************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <time.h>
//*************************************************************

//*********************************************************************
void error(const char *s,...){
//*********************************************************************
        va_list p;

        va_start(p,s);

        printf("\n\nError: ");
        vfprintf(stderr,s,p);
        printf("\n\n");

        va_end(p);
        printf("Press ENTER to exit...");
        fflush(stdin);
        getchar();
        exit(-1);
}


//*************************************************************
main(){
//*************************************************************
        double **a,**b,**c;
        unsigned x,y,i,j;

        printf("Oszlopok szama: ");fflush(stdin);scanf("%u",&x);
        printf("Sorok szama:    ");fflush(stdin);scanf("%u",&y);

        if(x==0 || y==0)error("Size must be positive");

        // 2D dinamikus tömb lefoglalása

        a=(double**)calloc(y,sizeof(double*));//pointertömb
        b=(double**)calloc(y,sizeof(double*));//pointertömb
        c=(double**)calloc(y,sizeof(double*));//pointertömb
        if(a==NULL || b==NULL || c==NULL)error("Memory allocation fault (1)");

        for(i=0;i<y;i++){
                a[i]=(double*)calloc(x,sizeof(double));
                b[i]=(double*)calloc(x,sizeof(double));
                c[i]=(double*)calloc(x,sizeof(double));
                if(a[i]==NULL || b[i]==NULL || c[i]==NULL)
                        error("Memory allocation fault (2)");
        }

        // a,b tömb feltöltése véletlenszámokkal

        srand((unsigned)time(NULL));
        for(i=0;i<y;i++)
                for(j=0;j<x;j++){
                        a[i][j]=(rand()-1000.0)/1000.0;//a .0 jelzi, hogy lebegőpontos
                        b[i][j]=(rand()-1000.0)/1000.0;//a .0 jelzi, hogy lebegőpontos
                }

        // Mátrixok kivonása

        for(i=0;i<y;i++)for(j=0;j<x;j++)c[i][j]=a[i][j]-b[i][j];

        // Kiírás a

        printf("\nA=\n");
        for(i=0;i<y;i++){
                for(j=0;j<x;j++)printf("%12.3f",a[i][j]);
                printf("\n");
        }

        // Kiírás b

        printf("\nB=\n");
        for(i=0;i<y;i++){
                for(j=0;j<x;j++)printf("%12.3f",b[i][j]);
                printf("\n");
        }

        // Kiírás c
```

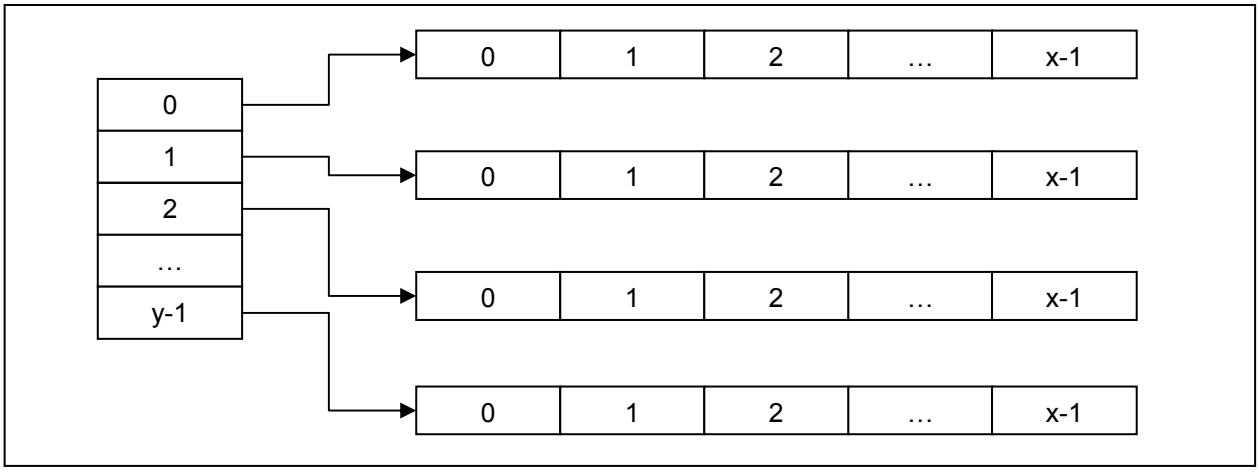
```
        printf("\nA-B=\n");
        for(i=0;i<y;i++){
                for(j=0;j<x;j++)printf("%12.3f",c[i][j]);
                printf("\n");
        }

        // Memóri afelszabadítás

        for(i=0;i<y;i++){free(a[i]);free(b[i]);free(c[i]);}
        free(a);free(b);free(c);
}
```

A fenti error() függvénnyel a 8. fejezetben foglalkozunk részletesebben.

A C nyelv csak egydimenziós dinamikus tömbök használatát támogatja közvetlenül, ezért amennyiben többdimenziós tömbre van szükségünk, ezt kézi munkával kell megvalósítani. Ez azt jelenti, hogy például a fenti esetben a kiinduló változónk egy n (jelen esetben 2) índirekciót tartalmazó pointer, ahol n a leendő tömb dimenziója. Mi 2D double tömböket szeretnénk kreálni, ezért **double pointerekre van szükség (3D tömb esetén ***típus pointerekre volna, stb.).

Első lépésben lefoglalunk egy akkora pointertömböt, ahány sora lesz a mátrixnak, vagyis amennyi a tömb első indexéhez tartozó mérete, jelen esetben y.

A második lépésben pedig ebben a tömbben minden egyes elem egy x méretű double tömbre fog mutatni, valahogy így:



10 ábra

Az is látszik, hogy ellentétben a valódi kétdimenziós tömbbel itt elképzelhető, hogy a mátrix sorai eltérő hosszúságúak legyenek.

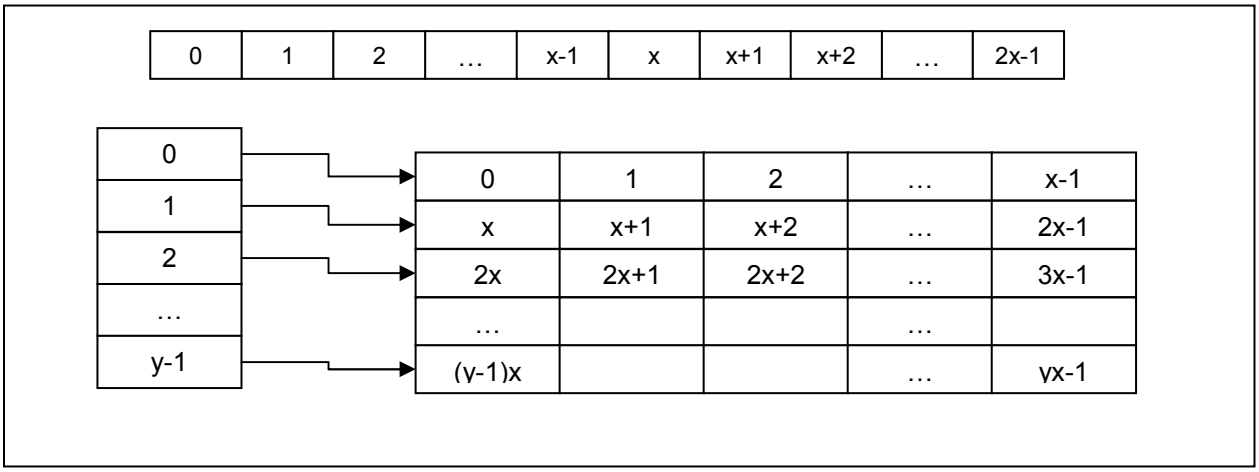
Az így létrehozott dinamikus tömb ugyanúgy használható, mint a nemdinamikus társa.

A tömb felszabadítása fordított sorrendben zajlik, mint ahogy a foglalás: először a sorokat, majd a pointertömböt töröljük.

6.1.5 Többdimenziós dinamikus tömb – másképp

```
//************************************************************
main(){
//************************************************************
        double **a,**b,**c;
        unsigned x,y,i,j;

        printf("Oszlopok szama: ");fflush(stdin);scanf("%u",&x);
        printf("Sorok szama:    ");fflush(stdin);scanf("%u",&y);

        if(x==0 || y==0)error("Size must be positive");
```

```
        // 2D dinamikus tömb lefoglalása

        a=(double**)calloc(y,sizeof(double*));//pointertömb
        b=(double**)calloc(y,sizeof(double*));//pointertömb
        c=(double**)calloc(y,sizeof(double*));//pointertömb
        if(a==NULL || b==NULL || c==NULL)error("Memory allocation fault (1)");

        a[0]=(double*)calloc(x*y,sizeof(double));
        b[0]=(double*)calloc(x*y,sizeof(double));
        c[0]=(double*)calloc(x*y,sizeof(double));
        if(a[0]==NULL || b[0]==NULL || c[0]==NULL)
                error("Memory allocation fault (2)");
        for(i=1;i<y;i++){
                a[i]=a[0]+x*i;
                b[i]=b[0]+x*i;
                c[i]=c[0]+x*i;
        }

        // a,b tömb feltöltése véletlenszámokkal

        srand((unsigned)time(NULL));
        for(i=0;i<y;i++)
                for(j=0;j<x;j++){
                        a[i][j]=(rand()-1000.0)/1000.0;//a .0 jelzi, hogy lebegőpontos
                        b[i][j]=(rand()-1000.0)/1000.0;//a .0 jelzi, hogy lebegőpontos
                }

        // Mátrixok kivonása

        for(i=0;i<y;i++)for(j=0;j<x;j++)c[i][j]=a[i][j]-b[i][j];

        // Kiírás a

        printf("\nA=\n");
        for(i=0;i<y;i++){
                for(j=0;j<x;j++)printf("%12.3f",a[i][j]);
                printf("\n");
        }

        // Kiírás b

        printf("\nB=\n");
        for(i=0;i<y;i++){
                for(j=0;j<x;j++)printf("%12.3f",b[i][j]);
                printf("\n");
        }

        // Kiírás c

        printf("\nA-B=\n");
        for(i=0;i<y;i++){
                for(j=0;j<x;j++)printf("%12.3f",c[i][j]);
                printf("\n");
        }

        // Memóri afelszabadítás

        free(a[0]);free(b[0]);free(c[0]);
        free(a);free(b);free(c);
}
```



11. ábra

A normál (nemdinamikus) két- (vagy több-) dimenziós tömbök valójában a memóriában egydimenziós tömbként találhatók, és az index(ek) elhagyásával keletkező pointert a gép nem tárolja, hanem kiszámítja, amikor szükség van rá. Például az int a[8][4]; tömb a[3] pointere valójában az a[0]+4*3 módin keletkezik, ahol 4 az oszlopok száma, 3 az index.

Létrehozhatjuk dinamikus kétdimenziós tömbünket ehhez hasonló módon is, bár a pointereket tartalmazó segédtömbre ez esetben is szükség van. A megoldást a 11. ábrán láthatjuk: létre akarunk hozni egy y×x méretű tömböt, ehhez lefoglalunk egy y méretű pointertömböt és egy y×x méretű egydimenziós tömböt, majd a pointertömböt úgy töltjük fel, hogy a szomszédos elemek között éppen egy sornyi távolság legyen az egydimenziós tömbben.

Az így kapott tömb ugyanúgy használható, mint a 6.1.4 pontban bemutatott kétdimenziós tömb, vagy a valódi 2D tömb.

6.2 Programírás

1. Írjon C programot, amely a parancssori paraméterként kapott C fájlból kiszűri a megjegyzéseket, és a második parancssori paraméterként kapott nevű fájlba írja. (Tehát ezúttal a main függvény kétparaméteres változatát(main(int argc, char ** argv) kell használni).
2. Írjon programot C nyelven! Adott a következő struktúra:

```
typedef struct{
        char Nev[50];
        unsigned Sorszam;
}Elem;
```

   a. Kérdezze meg a felhasználót, hogy hány név-sorszám párost szeretne beolvasni!
   b. Hozzon létre dinamikusan egy ekkora tömböt (a tömb neve legyen Nevsor), mely a fenti elem típusú elemekből áll!
   c. Olvassa be az adatokat a billentyűzetről a Nevsor tömbbe!
   d. Rendezze a tömb elemeit Sorszam szerinti növekvő sorrendbe! A rendezéshez készítsen függvényt, melynek bemenő paramétere a tömb!
   e. Írja ki a tömb elemeit printf("%3d.\t%s\n",Nevsor[i].Sorszam,Nevsor[i].Nev); formában!
   f. Szabadítsa fel a lefoglalt memóriát!
3. Írjon C programot!

   Adott az angmag.dic nevű szöveges fájl. Ez egy angol magyar szótár, melyben a szavak abc sorrendben találhatók. Minden sorban egy angol és egy magyar szó található = jellel elválasztva. Például:

   a=egy
   about=körülbelül
   and=és
   file=akta
   file=fájl
   zip=cipzár

   A szótárban maximum 1000 szópár fordul elő, egy-egy magyar illetve angol szó hossza maximum 30 karakter.

   Olvassa be ezt a szöveges fájlt, és készítse el belőle a magyar-angol szótárat, amit mentsen a magang.dic nevű szöveges állományba. A magang.dic fájlban is abc

sorrendben legyenek a szavak, így:

akta=file
cipzár=zip
egy=a
és=and
fájl=file
körülbelül=about

Figyelem, az ékezetes karaktereknek nem kell a magyar abc-nek megfelelő sorban lennie, de aki akarja, így is megcsinálhatja.

4. Írjon C függvényt, amely modellezi a realloc() függvény működését, de eltérően attól, paraméterként megkapja a forrás dinamikus változó méretét is:

```
void * my_realloc(void * src,unsigned src_size, unsigned new_size);
```

A megoldáshoz természetesen nem használhatja a realloc() függvényt!

5. Mentsünk egy Excel táblázatot .csv formátumban, és nézzünk bele Notepaddel! Láthatjuk, hogy egyszerű szöveges fájlt kaptunk, az egyes oszlopokat pontosvessző választja el egymástól. Az ilyen formátumú fájlokat az Excel vissza is tudja olvasni.

7. Dinamikus struktúrák: láncolt listák, bináris fák

7.1 Programértés

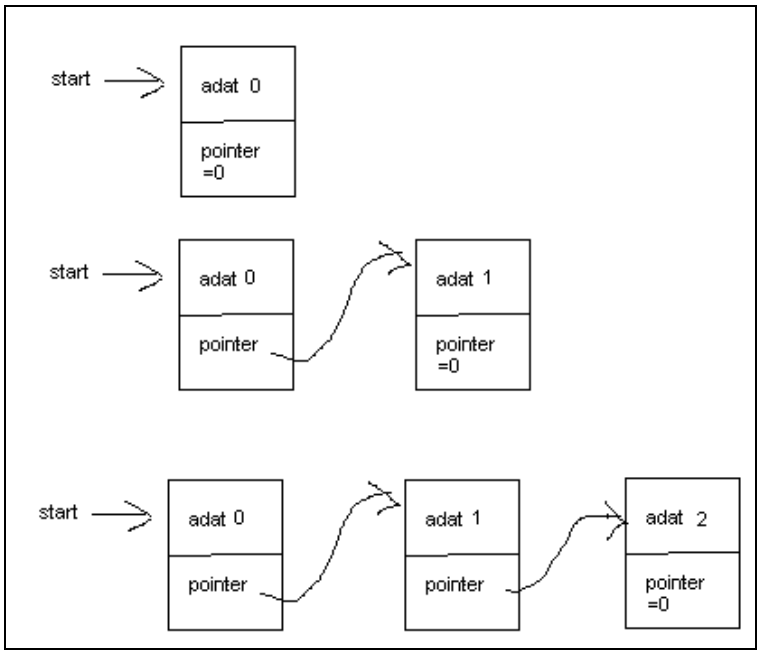
Gyakran előfordul, hogy nagyobb tömegű adatot kell kezelni, melynek nem tudjuk előre a pontos méretét. Ha létrehozunk egy tömböt, és ebbe kezdjük pakolni az adatokat, akkor előfordulhat, hogy a tömb megtelik. Ebben az esetben nem tehetünk mást, mint közöljük ezt a tényt a felhasználóval, aki ennek bizonyára nem fog örülni. Ha ezután a programból is kilépünk, annak még kevésbé.

Következő lehetőség, hogy dinamikusan foglalunk le egy tömböt. Ez előnyös, ha a tömb lefoglalása előtt tudjuk, hogy mennyi adatra számíthatunk. Ha nem így van, ismét előáll az előbb említett eset, bár itt már van lehetőség a javításra: ha megnöveljük a tömb méretét a realloc() függvénnyel. Ez nem túl elegáns megoldás, és az előfordulhat, hogy túl sokáig tart, ha nagy adatbázisról van szó. Nem kérhetjük meg a felhasználót, hogy legyen szíves, várjon öt percet, mert adminisztratív feladatot végzünk.

A C nyelv nem biztosít számunkra egyéb adattípust az eddig tanultakon kívül, mégis van mód arra, hogy tetszőleges számú adatot tároljunk anélkül, hogy átméretezésre lenne szükség. Ehhez olyan adatstruktúrára van szükség, melyben mindig rendelkezésre áll olyan pointer, mely nem mutat semmire, és ha szükség van rá, akkor ezzel mutathatunk egy újonnan lefoglalandó elemre. Az, hogy mindig legyen ilyen pointerünk, úgy valósítható meg, ha a pointer magának a dinamikusan lefoglalt elemnek a része. A C nyelvben a struktúra (rekord) típus az, amely képes pointereket más adatokkal együtt tárolni, ebben a fejezetben tehát pointereket tartalmazó adatstruktúrákkal foglalkozunk.

7.1.1 Egyszeresen láncolt lista strázsaelem nélkül

Egy láncolt listában az elemek sorban egymás után helyezkednek el. Az alábbi ábra mutatja a lista felépülését:



12. ábra

A láncolt listát alkotó struktúrák két részre oszthatók, az egyik a hasznos adat, a másik pedig a következő elemre mutató pointer. A fenti esetben mindössze egy külső pointer van, amely a listára mutat, ez a start. A lista bővítése így történik: pointer=(struktúra*)malloc(struktúra);

Készíthetünk olyan láncot is, ahol a struktúrákban két pointer is van: egyikkel a következő, a másikkal az előző elemre mutatunk, ezt nevezzük kétirányú, vagy duplán láncolt listának, míg a fenti az **egyszeresen láncolt lista**.

Az alfejezet címében szereplő **strázsaelem** (pivot) olyan elemet jelent, amelyben nem tárolunk hasznos adatot, ezt azonnal létrehozzuk a start pointer definiálása után, ami azért előnyös, mert ilyenkor pl. bejáráskor vagy bővítéskor nem kell azt vizsgálni, hogy van-e már eleme a listának. Azt ugyanakkor vizsgálni kell, hogy most épp ezen az elemen járunk-e vagy sem, összességében én nem látom előnyösnek a strázsa használatát, ezért ilyen nem is fog szerepelni egyik példaprogramban sem.

A lista használatát az alábbi példaprogram segítségével demonstráljuk:

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


//*******************************************************
typedef struct{
//*******************************************************
        char nev[64];
        char cim[128];
        char szam[32];
}adat,*padat;


//*******************************************************
typedef struct t{
//*******************************************************
        adat dat;
        struct t *kov;
}elem,*pelem;


//*******************************************************
pelem pelso=NULL,putolso=NULL,pakt=NULL;
//*******************************************************


//*******************************************************
void hozzaad(padat uj);// mindegy, hogy padat vagy
void beszur(adat * uj);// adat *, ugyanazt jelentik
void torol();
padat keres(char * nev);
padat elso();
padat kovetkezo();
void error(char * s){printf("Hiba: %s\n",s);exit(-1);}
void felvesz(padat a);
void menu_keres();
void menu_kiir();
//*******************************************************


//*******************************************************
main(){
//*******************************************************
        adat a;
        int valasztott_ertek=0;

        while(valasztott_ertek!=10){

                // A menü kiírása

                printf("\nKerem, valassza ki a muveletet!\n");
                printf("1  A lista vegehez ad egy adatot\n");
                printf("2  Beszur a listaba egy adatot\n");
                printf("3  Torli a listat\n");
                printf("4  Keres egy nevet\n");
                printf("5  Kiirja az osszeset\n");
                printf("10 Kilepes\n");

                // A választott érték beolvasása
```

```
                fflush(stdin); // standard input puffer ürítése
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                switch(valasztott_ertek){
                        case  1: felvesz(&a); hozzaad(&a);break;
                        case  2: felvesz(&a); beszur(&a); break;
                        case  3: torol();                 break;
                        case  4: menu_keres();            break;
                        case  5: menu_kiir();             break;
                        case 10:                          break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
        torol();
        return 0;
}


//*******************************************************
void felvesz(padat a){
//*******************************************************
        printf("Adja meg a nevet:         ");
        fflush(stdin);
        if(gets(a->nev)==NULL)error("Sikertelen adatbevitel");
        printf("Adja meg a cimet:         ");
        fflush(stdin);
        if(gets(a->cim)==NULL)error("Sikertelen adatbevitel");
        printf("Adja meg a telefonszamot: ");
        fflush(stdin);
        if(gets(a->szam)==NULL)error("Sikertelen adatbevitel");
}


//*******************************************************
void menu_keres(){
//*******************************************************
        char s[100];
        padat p;

        printf("Keresett nev: ");
        fflush(stdin);
        if(gets(s)==NULL)error("Sikertelen adatbevitel");
        p=keres(s);
        if(p==NULL){
                printf("%s nem talalhato az adatbazisunkban.\n",s);
                return;
        }
        printf("Nev:  %s\n",p->nev);
        printf("Cim:  %s\n",p->cim);
        printf("Szam: %s\n",p->szam);
}


//*******************************************************
void menu_kiir(){
//*******************************************************
        padat p=elso();
        while(p!=NULL){
                printf("Nev: %s\tCim: %s\tSzam: %s\n",p->nev,p->cim,p->szam);
                p=kovetkezo();
        }
}


//*******************************************************
void hozzaad(padat uj){
// A lista végéhez fűzi a pointerrel megadott struktúrát
//*******************************************************
        if(putolso==NULL){ // üres a lista
                pelso=putolso=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("hozzaad: memoriafoglalas/1 nem sikerult.");
                pakt->dat=*uj;
                pakt->kov=NULL;
        }
        else{
                putolso->kov=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("hozzaad: memoriafoglalas/2 nem sikerult.");
                putolso=pakt;
                pakt->dat=*uj;
                pakt->kov=NULL;
        }
}


//*******************************************************
void beszur(padat uj){
// név szerint az ABC-nek megfelelő helyre beszúrja a
// listába a pointerrel megadott struktúrát
```

```
//*******************************************************
        if(putolso==NULL){ // üres a lista
                pelso=putolso=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("beszur: memoriafoglalas/1 nem sikerult.");
                pakt->dat=*uj;
                pakt->kov=NULL;
        }
        else if(strcmp(pelso->dat.nev,uj->nev)>0){ // a lista elejére kell szúrni
                pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("beszur: memoriafoglalas/2 nem sikerult.");
                pakt->dat=*uj;
                pakt->kov=pelso;
                pelso=pakt;
        }
        else{
                pelem lemarad=pelso;

                pakt=pelso->kov;
                while(pakt!=NULL&&strcmp(pakt->dat.nev,uj->nev)<=0){
                        lemarad=pakt;
                        pakt=pakt->kov;
                }
                lemarad->kov=(pelem)malloc(sizeof(elem));
                if(lemarad->kov==NULL)error("beszur: memoriafoglalas/3 nem sikerult.");
                lemarad->kov->dat=*uj;
                lemarad->kov->kov=pakt;
                if(pakt==NULL)putolso=lemarad;// ha a végére szúrjuk be
        }
}


//*******************************************************
void torol(){
// Kitörli a listát
//*******************************************************
        while(pelso!=NULL){
                pakt=pelso;
                pelso=pelso->kov;
                free(pakt);
        }
        pelso=putolso=pakt=NULL;
}


//*******************************************************
padat keres(char * nev){
// Visszaadja a nev-hez tartozó struktúrát
// Ha nincs ilyen nev, NULL-t ad vissza
//*******************************************************
        for(pakt=pelso; pakt!=NULL && strcmp(pakt->dat.nev,nev)!=0; pakt=pakt->kov);
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}


//*******************************************************
padat elso(){
// Visszaadja az elsőt, aktuálist ide állítja
//*******************************************************
        pakt=pelso;
        return (pakt==NULL)?NULL:&(pakt->dat);
}


//*******************************************************
padat kovetkezo(){
// Visszaadja a következőt, aktuálist ide állítja
//*******************************************************
        if(pakt==NULL)return NULL;
        pakt=pakt->kov;
        return (pakt==NULL)?NULL:&(pakt->dat);
}
```

Ez a program egy telefonkönyv funkcióinak egy részét valósítja meg. Lássuk először a két struktúrát, amit használunk:

```
//*******************************************************
typedef struct{
//*******************************************************
        char nev[64];
        char cim[128];
        char szam[32];
}adat,*padat;
```

```
//*******************************************************
typedef struct t{
//*******************************************************
        adat dat;
        struct t *kov;
}elem,*pelem;
```

Az elsőben tároljuk az előfizető adatait. A számát is stringben tároljuk, így nyugodtan írhatunk bele kötőjeleket is. Létrehoztuk az adat és a padat felhasználói típusokat, ez utóbbi megegyezik az adat* pointerrel, tehát egyaránt használhatnánk azt, hogy adat * a; vagy azt, hogy padat a; ugyanazt jelentik.

A másodikban elhelyeztünk egy ilyen adatot, továbbá a lista következő elemére mutató pointert. Természetesen a két struktúra helyett elég lett volna egy, de így szétválasztva egyszerűbb dolgunk van, ha pl. fájlba akarnánk írni.

A második struktúrában a pointert struct t *-ként adtuk meg, mert az elem ill. pelem elnevezés később szerepel (egyébként a programban bárhol használhatnánk a „struct t”-t az elem, és a „struct t*”-ot a pelem helyett, de az elem és pelem típusokat éppen azért definiáltuk, hogy ne kelljen, mert így rövidebb).

A listához három globális pointert használunk:

```
pelem pelso=NULL,putolso=NULL,pakt=NULL;
```

A pelso a lista első elemére mutat (azaz megfelel a 10. ábrán szereplő start pointernek). A putolso a lista utolsó elemére mutat. Erre nem lenne feltétlenül szükség, de gyorsítja a lista végére történő újabb elem felvételét, mert nem kell végigmenni az összes elemen, hogy ezt megtehessük. A pakt pedig egy ideiglenes pointer, ezzel megyünk végig az elemeken, de más célra is használhatjuk. Igazából ezt célszerűbb lenne minden egyes függvényben lokális változóként megvalósítani, de lusták vagyunk, úgyhogy inkább marad globális. (A C++ többek között pont ezt oldja meg.)

A main() függvény felépítése már jól ismert számunkra a korábbi példákból:

```
//*******************************************************
main(){
//*******************************************************
        adat a;
        int valasztott_ertek=0;

        while(valasztott_ertek!=10){

                // A menü kiírása

                printf("\nKerem, valassza ki a muveletet!\n");
                printf("1  A lista vegehez ad egy adatot\n");
                printf("2  Beszur a listaba egy adatot\n");
                printf("3  Torli a listat\n");
                printf("4  Keres egy nevet\n");
                printf("5  Kiirja az osszeset\n");
                printf("10 Kilepes\n");

                // A választott érték beolvasása

                fflush(stdin); // standard input puffer ürítése
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                switch(valasztott_ertek){
                        case  1: felvesz(&a); hozzaad(&a);   break;
                        case  2: felvesz(&a); beszur(&a);    break;
                        case  3: torol();                    break;
                        case  4: menu_keres();               break;
                        case  5: menu_kiir();                break;
```

```
                        case 10:                                break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
        torol();
        return 0;
}
```

Ne felejtsük el letörölni a listát a program végén!

A felvesz() függvénnyel a-ba olvassuk a telefontulajdonos adatait:

```
//*****************************************************
void felvesz(padat a){
//*****************************************************
        printf("Adja meg a nevet:         ");
        fflush(stdin);
        if(gets(a->nev)==NULL)error("Sikertelen adatbevitel");
        printf("Adja meg a cimet:         ");
        fflush(stdin);
        if(gets(a->cim)==NULL)error("Sikertelen adatbevitel");
        printf("Adja meg a telefonszamot: ");
        fflush(stdin);
        if(gets(a->szam)==NULL)error("Sikertelen adatbevitel");
}
```

A beolvasott adatokat a hozzaad() fűzi a lista végére:

```
//*****************************************************
void hozzaad(padat uj){
// A lista végéhez fűzi a pointerrel megadott struktúrát
//*****************************************************
        if(putolso==NULL){ // üres a lista
                pelso=putolso=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("hozzaad: memoriafoglalas/1 nem sikerult.");
                pakt->dat=*uj;
                pakt->kov=NULL;
        }
        else{
                putolso->kov=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("hozzaad: memoriafoglalas/2 nem sikerult.");
                putolso=pakt;
                pakt->dat=*uj;
                pakt->kov=NULL;
        }
}
```

A függvény két részre bomlik:

1. A lista üres, ebben az esetben mindhárom pointer NULL értékű, a háromból kettő, a pelso és a putolso csak ebben az esetben NULL, a pakt máskor is lehet, ezért vagy a pelso-t, vagy a putolso-t kell vizsgálni (mindkettőt fölösleges, mert egyszerre NULL vagy nem NULL az értékük).
2. A listának van legalább egy eleme.

Első esetben mindhárom pointerrel az új elemre fogunk mutatni, melyet malloc()-kal foglalunk le. (A calloc()-ot is használhatnánk). Az uj pointerrel megadott adatokat egy egyszerű értékadással másoljuk a most lefoglalt elembe. Nem szabad megfeledkezni a kov pointer NULLázásáról, mert ebből tudjuk bejárásnál, hogy a lista végére értünk.

Második esetben a jelenlegi utolsó elem után kell fűzni az új adatot. Ez egyszerű, a putolso pont erre az elemre mutat, tehát az ő kov elemével mutatunk a frissen lefoglalt memóriára. A pakt-ot is ide állítjuk. Az utolsó elem viszont ezentúl az új elem lesz, tehát a putolso-t ide kell állítanunk. Az utolsó két művelet ugyanaz, mint az előbb.

Láncolt listát rendezni nem szokás oly módon, ahogy a tömböt, ehelyett már eleve rendezve építjük fel, ugyanis új elemet a listába szúrni kifejezetten könnyű, szemben a tömbbel (tömb

esetén a beszúrás helyét követő összes elemet eggyel hátrébb kell másolni). Ebben a programban nev szerint ABC sorrendben lehet beszúrni elemeket a beszur() függvény segítségével:

```
//*****************************************************
void beszur(padat uj){
// név szerint az ABC-nek megfelelő helyre beszúrja a
// listába a pointerrel megadott struktúrát
//*****************************************************
        if(putolso==NULL){ // üres a lista
                pelso=putolso=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("beszur: memoriafoglalas/1 nem sikerult.");
                pakt->dat=*uj;
                pakt->kov=NULL;
        }
        else if(strcmp(pelso->dat.nev,uj->nev)>0){ // a lista elejére kell szúrni
                pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL)error("beszur: memoriafoglalas/2 nem sikerult.");
                pakt->dat=*uj;
                pakt->kov=pelso;
                pelso=pakt;
        }
        else{
                pelem lemarad=pelso;

                pakt=pelso->kov;
                while(pakt!=NULL&&strcmp(pakt->dat.nev,uj->nev)<=0){
                        lemarad=pakt;
                        pakt=pakt->kov;
                }
                lemarad->kov=(pelem)malloc(sizeof(elem));
                if(lemarad->kov==NULL)error("beszur: memoriafoglalas/3 nem sikerult.");
                lemarad->kov->dat=*uj;
                lemarad->kov->kov=pakt;
                if(pakt==NULL)putolso=lemarad;// ha a végére szúrjuk be
        }
}
```

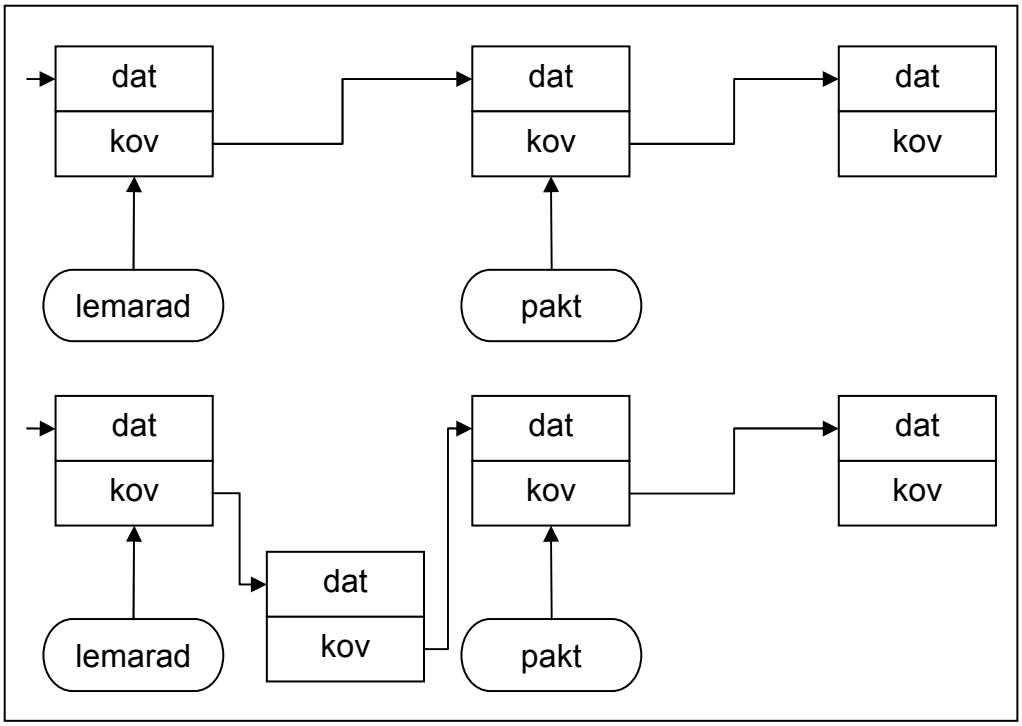
A függvény három részre oszlik. Az első ugyanaz, mint a hozzaad első blokkja, úgyhogy a fenti négy sor helyett nyugodtan írhatnánk, hogy hozzaad(uj);

A második ellenőrzi, hogy a lista legelejére kell-e beszúrni az elemet, ha igen, beteszi. Ekkor a pelso is módosul.

Harmadik eset, ha valamelyik elem után szúrunk. Egy while ciklussal végigjárjuk a listát, addig megyünk, míg a beszúrandónál nagyobbat nem találunk, vagy a lista végére nem érünk. Mivel ha találtunk egy olyan elemet, amely nagyobb a beszúrandónál, akkor ez elé kell beszúrni, ezért el kell tárolni az előző elem címét is, ez a lemarad (ugyanis visszafelé nem tudunk lépni a listában, mert nem mutat pointer az előző elemre; kétirányú listánál nem lenne szükség a lemarad-ra, mert ott vissza is tudunk lépni).

A beszúrás úgy történik, hogy a lemarad kov pointerével az új elemre mutatunk, majd az új elem kov pointerével az eddigi következő (pakt) elemre. Ha a lista végén vagyunk, akkor pakt=NULL, ebben az esetben is jó lesz a művelet (gondoljuk meg, miért), csak a putolso-t is meg kell változtatni.

A következő ábra a beszúrást illusztrálja:



13. ábra

Amikor a láncolt listát tanuljuk, ne a programkódot magoljuk be, hanem értsük meg, hogy mit kell tenni, értsük meg pl. a 11. ábrát. Ebben segíthet, ha lerajzoljuk ezeket a láncolásokat.

A listát a torol() függvénnyel törölhetjük:

```
//*******************************************************
void torol(){
// Kitörli a listát
//*******************************************************
        while(pelso!=NULL){
                pakt=pelso;
                pelso=pelso->kov;
                free(pakt);
        }
        pelso=putolso=pakt=NULL;
}
```

A fenti kód előnye, hogy nem kell külön vizsgálni, hogy a lista üres-e, mert arra is jól működik. Ha a kód alapján nem értjük a működést, rajzoljunk!

A menu_keres() függvény kikeresi a megadott névhez tartozó adatokat, ha nem találja, akkor ezt közli a felhasználóval:

```
//*******************************************************
void menu_keres(){
//*******************************************************
        char s[100];
        padat p;

        printf("Keresett nev: ");
        fflush(stdin);
        if(gets(s)==NULL)error("Sikertelen adatbevitel");
        p=keres(s);
        if(p==NULL){
                printf("%s nem talalhato az adatbazisunkban.\n",s);
                return;
        }
        printf("Nev:  %s\n",p->nev);
```

```
        printf("Cim:  %s\n",p->cim);
        printf("Szam: %s\n",p->szam);
}
```

A konkrét keresést a keres() függvény végzi:

```
//*****************************************************
padat keres(char * nev){
// Visszaadja a nev-hez tartozó struktúrát
// Ha nincs ilyen nev, NULL-t ad vissza
//*****************************************************
        for(pakt=pelso; pakt!=NULL && strcmp(pakt->dat.nev,nev)!=0; pakt=pakt->kov);
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}
```

A keresés hasonlóan zajlik, mint a beszúrás, csak itt nem <=0, hanem !=0 szerepel a feltételben, vagyis itt az egyenlőséget vizsgáljuk (a keresés is, és a beszúrás is érzékeny a betűméretre).

A menu_kiir() függvény kiírja a lista összes elemét sorban:

```
//*****************************************************
void menu_kiir(){
//*****************************************************
        padat p=elso();
        while(p!=NULL){
                printf("Nev: %s\tCim: %s\tSzam: %s\n",p->nev,p->cim,p->szam);
                p=kovetkezo();
        }
}
```

Az elso() függvény visszaadja a lista első elemét (üres listánál NULL), és beállítja pakt értékét az első elemre, hogy aztán a kovetkezo() függvény innen folytathassa:

```
//*****************************************************
padat elso(){
// Visszaadja az elsőt, aktuálist ide állítja
//*****************************************************
        pakt=pelso;
        return (pakt==NULL)?NULL:&(pakt->dat);
}


//*****************************************************
padat kovetkezo(){
// Visszaadja a következőt, aktuálist ide állítja
//*****************************************************
        if(pakt==NULL)return NULL;
        pakt=pakt->kov;
        return (pakt==NULL)?NULL:&(pakt->dat);
}
```

A láncolt listánál láttuk, hogy itt az elemek között csak lineáris keresés valósítható meg, hiszen az elemeknek nincs indexe. Ha a soros keresés ill. bejárás nem elég gyors, akkor más adatstruktúrát kell választanunk, ilyen például a bináris fa.

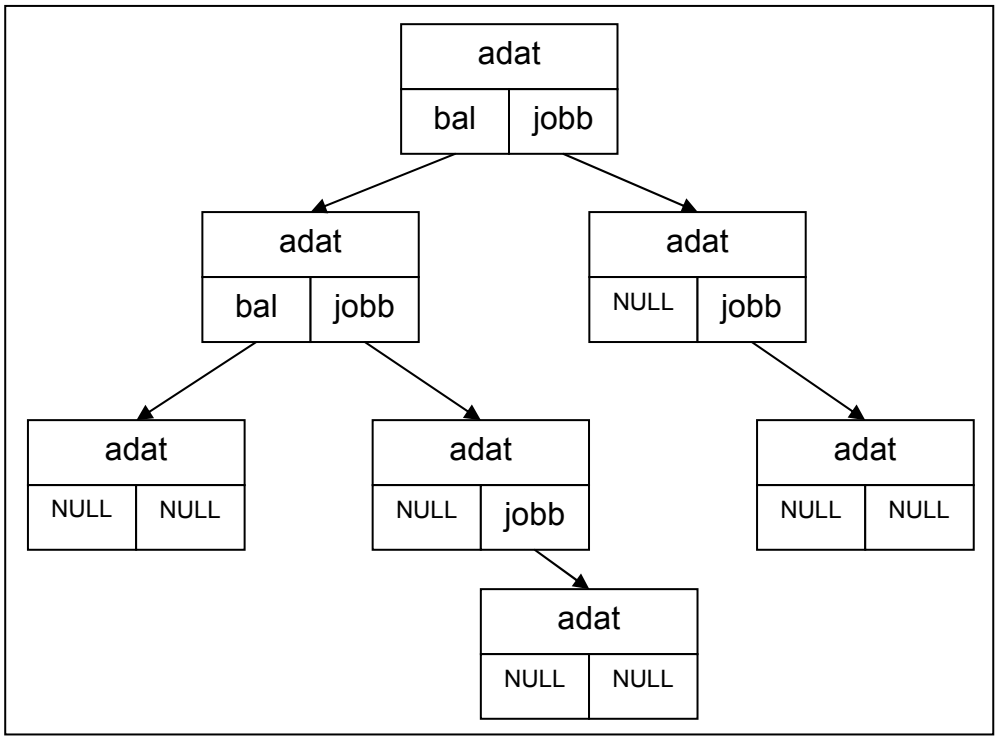
7.1.2 Bináris fa, rekurzió

A következő példát Vitéz tanár úr adta fel néhány éve:

„Egy dinamikus szerkezetben tanulo algoritmust:
Tanuljuk meg a Morse ABC nehany kodjat, es fejtsunk vissza egy szoveget:

```
S... O___ T_ E. A._ M__
... ___ . _ . . _  . ... _ .

A megoldas: epitsunk binaris fat!"
```

A bináris fákban a következőképp helyezkednek el az adatok:



14. ábra

Itt tehát minden elemhez két pointer tartozik, az egyik a jobb oldali, a másik a bal oldali elemre mutat. Felépítése tipikusan úgy történik, hogy az első elemet betesszük, aztán a következőről megvizsgáljuk, hogy nagyobb-e vagy kisebb a gyökérelemnél. Ha nagyobb, jobbra indulunk, ha kisebb balra (vagy fordítva, a lényeg, hogy konzekvensen mindig ugyanígy tegyünk).

A morzés programnál nem bal és jobb néven hívjuk a pointereket, hanem pont ill. vonalként. Az adatokat a Morse típusú struktúrában tároljuk, itt nem választottuk külön struktúrába az adatokat, hiszen csak egy karakterről van szó: arról a betűről, amelyet az a pont-vonal kombináció jelent, amilyen úton eljutottunk hozzá. A fa csúcsán álló elem (gyökérelem) nem tartalmaz karaktert sosem, ez tehát egy dummy (töltelék) elem.

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

/*********************************************************/
typedef struct dat{
/*********************************************************/
        char betu;
        struct dat *pont,*vonal;
}Morse,*MP;


MP gyoker=0;


/*********************************************************/
void delfa(MP e){
/*********************************************************/
        if(!e)return;
        delfa(e->pont);
        delfa(e->vonal);
```

```
        free(e);
}


/*******************************************************/
void feltolt(){
/*******************************************************/
        char c,aktbetu=0;
        MP futo;

        delfa(gyoker);
        if(!(gyoker=(MP)malloc(sizeof(Morse)))){
                puts("Elfogyott a memória");
                exit(-1);
        }
        gyoker->pont=gyoker->vonal=0;
        gyoker->betu=0;

        futo=gyoker;

        fflush(stdin);
        printf("Kerem a definiciot!\n");
        while((c=toupper(getchar()))!=EOF){
                switch(c){
                        case '.':{
                                if(!(futo->pont)){
                                        if(!(futo->pont=(MP)malloc(sizeof(Morse)))){
                                                puts("Elfogyott a memória");
                                                delfa(gyoker);
                                                exit(-1);
                                        }
                                        futo=futo->pont;
                                        futo->betu=0;
                                        futo->pont=futo->vonal=0;
                                }
                                else futo=futo->pont;
                                break;
                        }
                        case '_':{
                                if(!(futo->vonal)){
                                        if(!(futo->vonal=(MP)malloc(sizeof(Morse)))){
                                                puts("Elfogyott a memória");
                                                delfa(gyoker);
                                                exit(-1);
                                        }
                                        futo=futo->vonal;
                                        futo->betu=0;
                                        futo->pont=futo->vonal=0;
                                }
                                else futo=futo->vonal;
                                break;
                        }
                        default:{
                                futo->betu=aktbetu;
                                if((c>='A'&&c<='Z')||(c>='0'&&c<='9'))aktbetu=c;
                                else aktbetu=0;
                                futo=gyoker;
                        }
                }
        }
}


/*******************************************************/
void keres(){
/*******************************************************/
        char c;
        MP futo=gyoker;
        int jo=1;

        fflush(stdin);
        printf("Kerem a megfejtendo kodot!\n");
        while((c=toupper(getchar()))!=EOF){
                switch(c){
                        case '.':{
                                if(jo){
                                        if(futo->pont)futo=futo->pont;
                                        else jo=0;
                                }
                                break;
                        }
                        case '_':{
                                if(jo){
                                        if(futo->vonal)futo=futo->vonal;
                                        else jo=0;
                                }
                                break;
                        }
```

```
                        default:{
                                if(futo->betu&&jo)putchar(futo->betu);
                                else if(c==' ')putchar(' ');else putchar('?');
                                futo=gyoker;
                                jo=1;
                        }
                }
        }
}

/*********************************************************/
void main(){
/*********************************************************/
        feltolt();
        keres();
        delfa(gyoker);
}
```

Láthatjuk, hogy a program mindössze négy függvényből áll, nem is kell több. A fa gyökérelemét most is globális változóban tároljuk, melynek neve – minő meglepetés – gyoker. A main() függvény három sorában meghívja a másik három függvényt.

A fa törlését a delfa() függvény végzi:

```
/*********************************************************/
void delfa(MP e){
/*********************************************************/
        if(!e)return;
        delfa(e->pont);
        delfa(e->vonal);
        free(e);
}
```

A függvény jellegzetessége, hogy **rekurzív**, vagyis önmagát hívja. Létezik a rekurzió egy másik formája, a kölcsönös rekurzió, ahol két függvény hívja egymást kölcsönösen, természetesen ez a hívogatás nem megy örökké, a fenti példában például ha a paraméterként kapott e pointer NULL, akkor a függvény visszatér.

A függvény működése: ha nem NULL pointert kap, kitörli a jobb és baloldali részfáját, majd a gyökérelem is törlődik. Ez a módszer azért használható, mert bármelyik elemre megyünk, attól lefelé a struktúra ugyanúgy néz ki, mint a gyökérből indulva.

A rekurzió kapcsán szót kell ejtenünk a függvényhívás megvalósításáról, illetve a stackről.

Minden programhoz tartozik egy ún. stack (verem) memóriaterület, melynek méretét a fordító beállításával tudjuk szabályozni. Borlandnál a default emlékeim szerint 16 kB, Visualnál pedig 1 MB. Azt a memóriát az egész program használja. A processzor a push utasítással tud adatot beletenni, és poppal kivenni, tehát csak a tetejére lehet tenni, és csak a legfelső elemet lehet kivenni. Természetesen, ha tudjuk mi hol van benne, akkor benne is dolgozhatunk. Ez a kis ábra egy adott függvényhez tartozó stack területet mutatja. Ha ezt a függvényt egy másik hívta meg, akkor a hívó hasonló blokkjai ez alatt találhatók meg. Az adatok mérete nem feltétlenül egyforma, ezzel most egy kicsit csalunk..

A függvényhez tartozó adatok között legalulra került a paraméterek blokkja. A függvény meghívásának helyén a paramétereket belenyomjuk a stackbe, és ezeket a függvény ki tudja venni (a fenti függvénynél az e az egyetlen paraméter). C programok esetén a paraméterek közül legfelül az első szerepel, utána a második, és legalul az utolsó. Ehhez a betételnél hátulról előre kell benyomni az adatokat. Más programnyelvek gyakran nem így dolgoznak, hanem elölről hátrafelé nyomják be az adatokat, ilyen pl. a Pascal. A C-ben viszont ezt a módszert kell használni, ugyanis pl. a printf() vagy a scanf() függvény esetén bármennyi paraméter lehet, de mi

is definiálhatunk függvényeket változó paraméterlistával. Ilyen esetben csak a lista eleje biztos (az említett függvényeknél az első paraméter a formátumlista stringre mutató pointer, a függvény a formátumlista alapján dönti el, hogy milyen paraméterek következnek).

A paraméterek blokkja után jön a visszatérési cím. Ez az a memóriacím, ahová a return urasítás hatására vissza fogunk ugrani. Ha a return visszaad valamilyen értéket, akkor ő is a stackbe fogja ezt belenyomni, amit a hívás helyén ki tudunk belőle venni (ezeket a betételeket és kivételeket a fordító elintézi számunkra, nem kell ezzel törődni).

A harmadik blokkot már maga a meghívott függvény teszi a stackbe. Ide a lokális változók kerülnek. A delfa() függvényben nincsenek ilyenek, de a keres() függvényben ilyen a c, a futo és a jo.

Ha egy függvénynek nincs paramétere és lokális változója, akkor is bekerül a visszatérési cím. Ezt azért kellett itt megemlíteni, mert a rekurzió ész nélküli alkalmazásával könnyű telipakolni a stacket, és ezzel stack overflow hibát előidézni. (Ezt a hibák hackerek ki szokták használni számítógépek feltöréséhez. Ez ellen véd a legújabb mikroprocesszorokban bevezetett NX bit, persze ehhez kell az operációs rendszer támogatása is.)

Minden rekurzióval megoldott probléma megoldható rekurzió nélkül, ami gyakran gyorsabb kódot eredményez, mint a rekurzióval készült.

A feltolt() függvény a feladatban megadott formában várja a betűket és a hozzájuk tartozó morze kódokat.

```
/*********************************************************/
void feltolt(){
/*********************************************************/
        char c,aktbetu=0;
        MP futo;

        delfa(gyoker);
        if(!(gyoker=(MP)malloc(sizeof(Morse)))){
                puts("Elfogyott a memória");
                exit(-1);
        }
        gyoker->pont=gyoker->vonal=0;
        gyoker->betu=0;

        futo=gyoker;

        fflush(stdin);
        printf("Kerem a definiciot!\n");
        while((c=toupper(getchar()))!=EOF){
                switch(c){
                        case '.':{
                                if(!(futo->pont)){
                                        if(!(futo->pont=(MP)malloc(sizeof(Morse)))){
                                                puts("Elfogyott a memória");
                                                delfa(gyoker);
                                                exit(-1);
                                        }
                                        futo=futo->pont;
                                        futo->betu=0;
                                        futo->pont=futo->vonal=0;
                                }
                                else futo=futo->pont;
                                break;
                        }
                        case '_':{
                                if(!(futo->vonal)){
                                        if(!(futo->vonal=(MP)malloc(sizeof(Morse)))){
                                                puts("Elfogyott a memória");
                                                delfa(gyoker);
                                                exit(-1);
                                        }
                                        futo=futo->vonal;
                                        futo->betu=0;
                                        futo->pont=futo->vonal=0;
                                }
                                else futo=futo->vonal;
                                break;
                        }
                        default:{
                                futo->betu=aktbetu;
                                if((c>='A'&&c<='Z')||(c>='0'&&c<='9'))aktbetu=c;
                                else aktbetu=0;
                                futo=gyoker;
                        }
```

```
                }
        }
}
```

Első lépésben kitöröljük a fát, ha volt már ilyen korábban. Ezt követően létrehozzuk a gyökérelemet, majd egy ciklus következik, amely sorra olvassa a betűket. A betűk olvasásához a getchar függvényt használjuk. A beolvasás addig tart, míg EOF karaktert nem kapunk. (EOF=End Of File). ha nem valódi fájlból olvasunk, hanem a billentyűzetről, akkor a leállításhoz használjuk a Ctrl+Z kombinációt!

A futo nevű pointerrel lépkedünk az elemeken. Ha . érkezik, akkor a pont irányban lépünk, ha _ érkezik, akkor a vonal irányban. Ha az adott irányban nem létezik elem, akkor létrehozza (és betűként a '\0'-t adunk). Minden más karakter azt jelenti, hogy véget ért a morze kód. Ebben az esetben az előzőleg eltárolt betűt beírjuk az aktuális elembe, és ha a karakter betű volt, akkor ezt eltároljuk a következő beíráshoz, ezután a futo-t a visszaállítjuk a gyökérre.

```
/*********************************************************/
void keres(){
/*********************************************************/
        char c;
        MP futo=gyoker;
        int jo=1;

        fflush(stdin);
        printf("Kerem a megfejtendo kodot!\n");
        while((c=toupper(getchar()))!=EOF){
                switch(c){
                        case '.':{
                                if(jo){
                                        if(futo->pont)futo=futo->pont;
                                        else jo=0;
                                }
                                break;
                        }
                        case '_':{
                                if(jo){
                                        if(futo->vonal)futo=futo->vonal;
                                        else jo=0;
                                }
                                break;
                        }
                        default:{
                                if(futo->betu&&jo)putchar(futo->betu);
                                else if(c==' ')putchar(' ');else putchar('?');
                                futo=gyoker;
                                jo=1;
                        }
                }
        }
}
```

A kereses() függvény bekéri a megfejtendő kódot, és kiírja az ennek megfelelő szöveget írja vissza. Ha olyan karakterrel találkozik, amit nem ismer, akkor ?-et ír ki.

7.2 Programírás

1. Egészítsük ki a láncolt lista kódját azzal, hogy a megadott nevű elemet törölni lehessen a listából. (Ez gyakorlatilag a beszúrás ellentéte, nem elég egyszerűen a free()-vel felszabadítani a memóriát, előbb az elemet ki kell venni a láncból).
2. (Poppe A. példája) A standard inputon előre nem ismert számú (x,y) koordinátapárt olvasunk. Az egyes valós számértékeket whitespace karakterek határolják. Írjon olyan C programot, amely kiírja az egyes koordinátapontok távolságát az összes pont, mint pontrendszer súlypontjától mérve. Feltehető, hogy az adatok hibátlanok (helyes formátumú, páros számú valós szám). A program file-okat nem használhat. Az esetlegesen dinamkusan lefoglalt memóriát szabadítsa fel! A súlypont x_{sp} koordintája

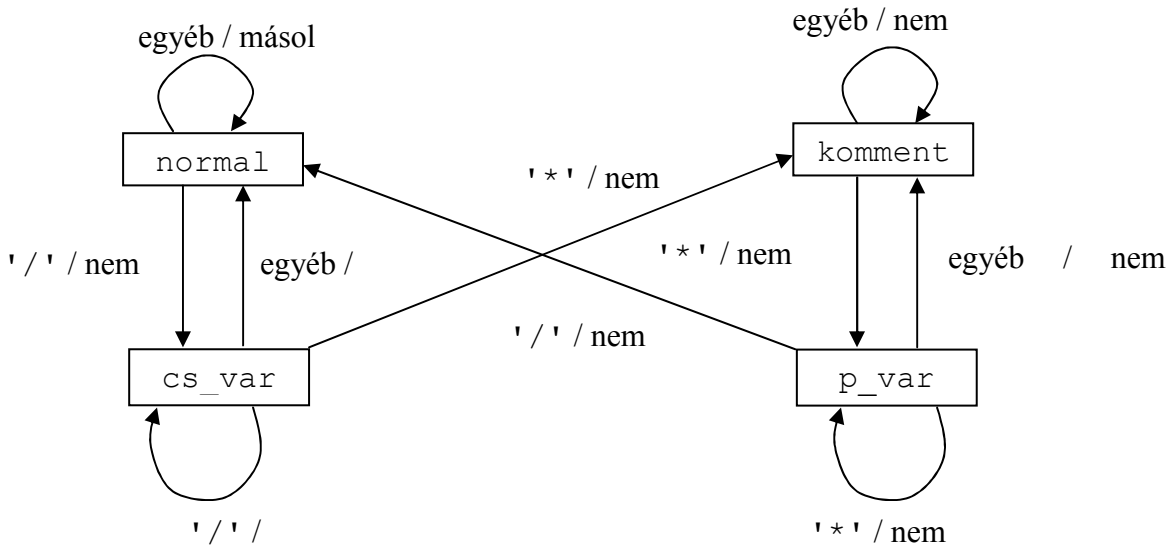
így számolható (azonos “tömegű” pontok esetén): $x_{sp} = \sum_{i=1}^{n} x_i / n$, ahol n a pontok száma.

8. Állapotgép és egyebek

8.1 Programértés

8.1.1 Poppe András állapotgépes példája: megjegyzések szűrése

A program állapotgépes modelljének állapotgráfja:



15. ábra

```
/* C forrasprogram komment mentesitese:
   A szabvanyos bemenetrol EOF-ig erkezo szoveget szuri.
   A szurt allomanyt a szabvanyos kimenetre irja.

   A programot allapotgepes modellel valositottuk meg.
*/

#include <stdio.h>
typedef enum {normal,komment,cs_var,p_var} allapotok; // allapotgepes modellhez

void main(void){
        int ch;
        allapotok allapot=normal;

        while((ch=getchar())!=EOF){
                switch (allapot){
                        case normal:    if(ch!='/')putchar(ch);else allapot=cs_var;break;
                        case cs_var:    if(ch=='*')allapot=komment;
                                        else{putchar('/');if(ch!='/'){putchar(ch);allapot=normal;}}
                                        break;
                        case komment:   if(ch=='*')allapot=p_var;break;
                        case p_var:     if(ch=='/')allapot=normal;else if(ch!='*')allapot=komment;
                                        break;
                }
        }
}
```

8.1.2 Ékezetes karaktereket konvertáló állapotgépes feladat

/*
2. Írjon programot C nyelven!
A program a szabványos bemenetről karaktereket olvas a fájlvége jelig,

kimenete a szabványos kimenet. A bemeneti stream-ben a magyar ékezetes karakterek a következő formában vannak jelen: á:a', é:e', ö:o:, ü:u:, o:o". (Az egyszerűség kedvéért csak ezzel az öt kisbetűvel foglalkozunk). A kimenetre úgy kerüljenek, hogy a két karakterből álló betűt egy magyar betűvé alakítja. Az átalakításhoz használjon állapotgépet! Abban az esetben, ha a bemenet a\', a kimenet legyen a', ha a bemenet u\:, a kimenet u: stb.
*/

/*
Rendes állapottábla:

	'a'	'e'	'o'	'u'	'\\'	':','\'','\"'	többi	EOF
alap	A	E	O	U	alap	alap	alap	ret
A	A	E	O	U	AB	alap	alap	ret
E	A	E	O	U	EB	alap	alap	ret
O	A	E	O	U	OB	alap	alap	ret
U	A	E	O	U	UB	alap	alap	ret
AB	A	E	O	U	alap	alap	alap	ret
EB	A	E	O	U	alap	alap	alap	ret
OB	A	E	O	U	alap	alap	alap	ret
UB	A	E	O	U	alap	alap	alap	ret

A fenti tábla nem tartalmazza a kiírásokt. Az állapottáblát össze tudjuk vonni a következőképpen:

	'a','e','o','u'	'\\'	':','\'','\"'	többi	EOF
alap	B	alap	alap	alap	ret
B	B	C	alap	alap	ret
C	B	alap	alap	alap	ret

Ekkor viszont alap állapotban el kell tárolni azt, hogy milyen betűt. kaptunk, ha valamelyik magánhangzó jött a fentiek közül (nem tudom, hogy ez mennyire tisztességes állapotgép, viszont egyszerűbb, áttekinthetőbb kódot kapunk. Lássuk a megoldást!
*/

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

void main()
{
        enum {alap,B,C} allapot=alap;
        int betu,elozo;/*AQ karaktereket int-ként tároljuk!*/

        while((betu=getchar())!=EOF)
        {
                switch(allapot)
                {
                        case alap:
                                switch(betu)
                                {
                                        case 'a':
                                        case 'e':
                                        case 'o':
                                        case 'u':
                                                allapot=B;
                                                elozo=betu;
                                                break;
                                        default:/* az összes többi karakternél alapba jutunk*/
```

```
                                        putchar(betu);/*allapot nem változik*/
                                }
                                break;
                        case B:
                                switch(betu)
                                {
                                        case 'a':
                                        case 'e':
                                        case 'o':
                                        case 'u':
                                                allapot=B; /*allapot nem változik, ugyhogy ezt a sort el
lehet hagyni*/
                                                putchar(elozo);/*kiírjuk az előző magánhangzót, hiszen nem
repülő ékezetes*/
                                                elozo=betu;
                                                break;
                                        case ':':
                                                switch(elozo)
                                                {
                                                        case 'o':putchar(148);break;/*'ö'*/       /*ha
papírra írjuk a programot, akkor potchar('ö');-t írjunk!*/
                                                        case 'u':putchar(129);break;/*'ü'*/
                                                        default: putchar(elozo);putchar(betu);
                                                }
                                                allapot=alap;
                                                break;
                                        case '\'':
                                                switch(elozo)
                                                {
                                                        case 'a':putchar(160);break;/*'á'*/
                                                        case 'e':putchar(130);break;/*'é'*/
                                                        default: putchar(elozo);putchar(betu);
                                                }
                                                allapot=alap;
                                                break;
                                        case '\"':
                                                switch(elozo)
                                                {
                                                        case 'o':putchar(147);break;/*'ô'*/
                                                        default: putchar(elozo);putchar(betu);
                                                }
                                                allapot=alap;
                                                break;
                                        case '\\':
                                                putchar(elozo);
                                                allapot=C;
                                                break;
                                        default:/* az összes többi karakternél alapba jutunk*/
                                                putchar(elozo);
                                                putchar(betu);
                                                allapot=alap;
                                }
                                break;
                        case C:
                                switch(betu)
                                {
                                default:
                                        putchar('\\');/* mivel break-et nem írtam, a case utáni utasítások is
végrehajtódnak!*/
                                case ':': case '\'': case '\"':/*A \-t nem írjuk ki*/
                                        allapot=alap;
                                        putchar(betu);
                                }
                                break;
                        default: printf("Programhiba: Hibas allapot!\n");exit(-1);
                }
        }
    }
```

8.1.3 Változó paraméterlista

A 6.1.4 fejezetben találkoztunk az alábbi error() függvénnyel:

```
//************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
//************************************************************

//*********************************************************************
void error(const char *s,...){
//*********************************************************************
        va_list p;
```

```
        va_start(p,s);

        printf("\n\nError: ");
        vfprintf(stderr,s,p);
        printf("\n\n");

        va_end(p);
        printf("Press ENTER to exit...");
        fflush(stdin);
        getchar();
        exit(-1);
}
```

A paramétersorban szereplő … azt jelzi, hogy ez a függvény akárhány paramétert kaphat. Eddig is sokszor használtunk ilyen függvényeket (printf(), scanf() stb.), most lássuk, hogyan is működik!

A függvények paraméterei – ahogy az előző fejezetben láttuk – a veremben kerülnek átadásra, méghozzá úgy, hogy legfelül van az első paraméter, azután a második, és így tovább. (Ez olyan programnyelveknél, amelyek nem ismerik a változó paraméterlistát, általában pont fordított a helyzet, legfelül van az utolsó. Ezt C nyelvben is elérhetjük, ha a függvény neve elé írjuk a pascal kulcsszót, de ez kizárólag a nem változó paraméterlistájú függvények esetén használható. A Windows rendszerhívások ilyen függvényhívást igényelnek.) Ennek az az oka, hogy a … kizárólag a paraméterlista végén szerepelhet, elé legalább egy rögzített paramétert meg kell adni.

A va_list típusú pointert a va_start állítja az első olyan paraméterre, amely az s után következik. A fenti függvény azonban nem végzi el a paraméterek feldolgozását, ehelyett kiír egy szöveget, és utána az egész paraméterlistát átadja a vfprintf() függvénynek, mely kiírja azt a standard error kimenetre, ami alapértelmezés szerint a képernyő.

8.1.4 Függvénypointerek

8.2 Programírás

9. Minta nagy házi feladat – telefonkönyv

9.1 Specifikáció

Telefonkönyv adatbázist kezelő program szabványos C nyelven, szöveges felhasználói felülettel, mely a következő funkciókat tudja:

- Új előfizető felvétele
- Előfizető adatainak keresése/módosítása (beleértve a törlést is), a név megadása után megkérdezi, hogy meg kívánjuk-e jeleníteni az ilyen néven szereplő előfizetőket, ha igen, kiírja őket, és mindegyiket egy számmal indexeli, és ez alapján lehet választani.
- adatok mentése
- kilépés

9.2 Tervezés

Az adatokat a következő struktúrában tároljuk:

- Név: string, 60 karakter
- Település: string, 50 karakter
- Utca: string, 50 karakter
- Házszám: string, 50 karakter (itt lehet megadni, hogy pl. 2/B 3. em. 15.)
- Irányítószám: egész
- Telefonszám, beírt formában: string, 50 karakter (kereséshez egy függvény adja vissza csak a számjegyeket)
- Megjegyzés: string, 100 karakter

Az adatokat név, cím és telefonszám szerint szeretnénk keresni, és a találatokat növekvő sorrendben kiírni, ehhez az adatokat láncolt listában tároljuk, mely mindhárom kulcs szerint rendezett (tehát mindhárom kulcs szerint láncolt). A listaelemeket a következő struktúrákba tesszük:

- adatok
- pointer a következő névre
- pointer a következő címre
- pointer a következő telefonszámra

Menü:

- Új előfizető
- Keresés/Változtatás
    - Keresés név szerint
    - Keresés cím szerint
    - Keresés telefonszám szerint
        - Név változtatása
        - Cím változtatása (részenként)
        - Telefonszám változtatása
        - Előfizető törlése
- Adatbázis mentése

- Kilépés

A programot három forrásállományra osztjuk:

- main.cpp: ez tartalmazza a main() függvényt, semmi mást
- lista.cpp: a láncolt listát kezelő függvények és változók, hozzá tartozó header a lista.h
- fuggv.cpp: a többi függvény, hozzá tartozó header a fuggv.h

Először a láncolt listát készítjük el, aztán a main()-t, aztán a maradékot.

A program fejben átlátható, ennél részletesebb tervet csak akkor szoktam készíteni, ha a program bonyolultsága ezt megkívánja. Az, hogy ki mennyire tervezi meg a programját előre, teljesen az egyén ilyen irányú beállítottságától függ. A komolyabb tervezés általában jobb struktúrájú programot eredményez, ugyanakkor ritkán fordul elő, hogy programozás közben teljes mértékben az előzetes terveknek megfelelő program készüljön, ezesetben viszont a tervek egy részét kidobhatjuk, vagy módosítanunk kell. A jobban megtervezett program nagyban segítheti a dokumentáció elkészítését.

9.3 A program

lista.h

```
//***************************************************************************
// Telefonkönyv láncolt lista függvénydeklarációi
// Létrehozva:           2005. 09. 16.
// Készítő:              Pohl László
// Licensz:              freeware
// Utolsó változtatás:   2005. 09. 27.
//***************************************************************************


//***************************************************************************
#ifndef TELKONYV_LISTA_HEADER
#define  TELKONYV_LISTA_HEADER
//***************************************************************************


//***************************************************************************
typedef struct{
//***************************************************************************
        char     nev[60];
        char     telepules[50];
        char     utca[50];
        char     hazszam[50];     // emelet és ajtó ispl.: 2/A 8.em 36.
        unsigned iranyitoszam;
        char     telefonszam[50]; // kötőjel, zárójel és szóköz is lehet benne
        char     megjegyzes[100];
}adat,*padat;


//***************************************************************************
int list_ins(adat*);
void list_clear();
int list_del();
padat list_get_akt();
padat list_get_first_nev();
padat list_get_next_nev();
padat list_get_first_cim();
padat list_get_next_cim();
padat list_get_first_num();
padat list_get_next_num();
int list_push_act();
int list_pop_act();
padat find_first_nev(const char*);
padat find_first_cim(const char*);
padat find_first_tel(const char*);
padat find_next_nev(const char * nev);
padat find_next_cim(const char * cim);
padat find_next_num(const char * num);
```

```
unsigned count_nev(const char * nev);
unsigned count_cim(const char * cim);
unsigned count_num(const char * num);
unsigned valaszt_nev(const char * nev);
unsigned valaszt_cim(const char * cim);
unsigned valaszt_num(const char * num);
padat get_n_nev(const char * nev,unsigned n);
padat get_n_cim(const char * cim,unsigned n);
padat get_n_num(const char * num,unsigned n);
//*************************************************************************


#endif
```

lista.cpp

```
//*************************************************************************
// Telefonkönyv láncolt lista függvénydefiníciói
// Létrehozva:           2005. 09. 16.
// Készítő:              Pohl László
// Licensz:              freeware
// Utolsó változtatás:   2005. 09. 30.
//*************************************************************************


//*************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lista.h"
//*************************************************************************


//*************************************************************************
// A lista elemei csak ebben a fájlban elérhetők
typedef struct el{
//*************************************************************************
        adat dat;
        struct el *kov_nev,*kov_cim,*kov_num;
}elem,*pelem;

#define MAX_VEREM 20 // a veremben lévő elemek maximális száma

//*************************************************************************
pelem start_nev=NULL,start_cim=NULL,start_num=NULL,pakt=NULL;
pelem verem[MAX_VEREM];  // list_push_act, list_pop_act verme
unsigned akt_verem=0;    // verem elemeinek aktuális száma
//*************************************************************************


//*************************************************************************
void cim2string(char * string, const padat cim){
// A címet egyesíti
//*************************************************************************
        strcpy(string,cim->telepules);
        strcat(string,cim->utca);
        strcat(string,cim->hazszam);
}


//*************************************************************************
void clear_tel_num(char * tiszta, const char * full){
// csak a számjegyeket hagyja a telefonszámban
//*************************************************************************
        for(;*full;full++)
                if(*full>='0'&&*full<='9')*(tiszta++)=*full;
        *tiszta=0; // stringet lezáró 0
}


//*************************************************************************
int cimcmp(const padat cim1,const char * cim2){
// cim1-et és cim2-t összehasonlítja, cim2 egy összemásolt cím
// vissza <0, ha cim1<cim2, 0, ha cim1==cim2, és >0, ha cim1>cim2
//*************************************************************************
        char cim1s[150];
        cim2string(cim1s,cim1);
        return strcmp(cim1s,cim2);
}


//*************************************************************************
int telcmp(const char * tel1,const char * tel2){
// az első telefonszámot tisztítja, a második már tiszta
//*************************************************************************
```

```
        char tel1c[150];
        clear_tel_num(tel1c,tel1);
        return strcmp(tel1c,tel2);
}


//*************************************************************************
int list_ins(padat uj){
// Beszúrja mindhárom listába az a-val mutatott adatot
// visszatérési érték: sikerült: 1, nem sikerült: 0
//*************************************************************************
        char cimtel[150];
        pelem beszurt;

        // üres a lista

        if(start_nev==NULL){
                start_nev=start_cim=start_num=pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL){printf("\n\nlist_ins: memoriafoglalas sikertelen (1)\n\n");return 0;}
                pakt->dat=*uj;
                pakt->kov_nev=pakt->kov_cim=pakt->kov_num=NULL;
                return 1;
        }

        // beszúrás a nev listába

        if(strcmp(start_nev->dat.nev,uj->nev)>0){ // a lista elejére kell szúrni
                pakt=(pelem)malloc(sizeof(elem));
                if(pakt==NULL){printf("\n\nlist_ins: memoriafoglalas sikertelen (2)\n\n");return 0;}
                pakt->dat=*uj;
                pakt->kov_nev=start_nev;
                start_nev=pakt;
                beszurt=pakt;
        }
        else{
                pelem lemarad=start_nev;

                pakt=start_nev->kov_nev;
                while(pakt!=NULL&&strcmp(pakt->dat.nev,uj->nev)<=0){
                        lemarad=pakt;
                        pakt=pakt->kov_nev;
                }
                lemarad->kov_nev=(pelem)malloc(sizeof(elem));
                if(lemarad->kov_nev==NULL)
                        {printf("\n\nlist_ins: memoriafoglalas sikertelen (3)\n\n");return 0;}
                lemarad->kov_nev->dat=*uj;
                lemarad->kov_nev->kov_nev=pakt;
                beszurt=lemarad->kov_nev;
        }

        // beszúrás a cim listába

        cim2string(cimtel,uj); // egyesítjük a címet a kereséshez
        if(cimcmp(&(start_cim->dat),cimtel)>0){ // a lista elejére kell szúrni
                beszurt->kov_cim=start_cim;
                start_cim=beszurt;
        }
        else{
                pelem lemarad=start_cim;

                pakt=start_cim->kov_cim;
                while(pakt!=NULL&&cimcmp(&(pakt->dat),cimtel)<=0){
                        lemarad=pakt;
                        pakt=pakt->kov_cim;
                }
                beszurt->kov_cim=pakt;
                lemarad->kov_cim=beszurt;
        }

        // beszúrás a telefonszám listába

        clear_tel_num(cimtel,uj->telefonszam);
        if(telcmp(start_num->dat.telefonszam,cimtel)>0){ // a lista elejére kell szúrni
                beszurt->kov_num=start_num;
                start_num=beszurt;
        }
        else{
                pelem lemarad=start_num;

                pakt=start_num->kov_num;
                while(pakt!=NULL&&telcmp(pakt->dat.telefonszam,cimtel)<=0){
                        lemarad=pakt;
                        pakt=pakt->kov_num;
                }
                beszurt->kov_num=pakt;
                lemarad->kov_num=beszurt;
        }
        return 1;
```

```
}


//*************************************************************************
void list_clear(){
// kitörli az egész listát
//*************************************************************************
        while(start_nev!=NULL){
                pakt=start_nev;
                start_nev=start_nev->kov_nev;
                free(pakt);
        }
        start_nev=start_cim=start_num=pakt=NULL;
}


//*************************************************************************
int list_del(){
// Kitörli az aktuális elmet.
// visszatérési érték: sikerült: 1, nem sikerült: 0
//*************************************************************************
        pelem torlendo=pakt, lemarad;

        if(pakt==NULL)return 0;

        // Kiláncolás a nev listából

        if(torlendo==start_nev){
                start_nev=pakt=start_nev->kov_nev;
        }else{
                lemarad=start_nev;
                pakt=start_nev->kov_nev;
                while(pakt!=NULL&&pakt!=torlendo){
                        lemarad=pakt;
                        pakt=pakt->kov_nev;
                }
                if(pakt==NULL)return 0; // ekkor nem találtuk meg a listában
                lemarad->kov_nev=torlendo->kov_nev;
        }

        // Kiláncolás a cím listából

        if(torlendo==start_cim)start_cim=pakt=start_cim->kov_cim;
        else{
                lemarad=start_cim;
                pakt=start_cim->kov_cim;
                while(pakt!=NULL&&pakt!=torlendo){
                        lemarad=pakt;
                        pakt=pakt->kov_cim;
                }
                if(pakt!=NULL)lemarad->kov_cim=torlendo->kov_cim;
                else return 0; // ekkor nem találtuk meg a listában
        }

        // Kiláncolás a telefonszám listából

        if(torlendo==start_num)start_num=pakt=start_num->kov_num;
        else{
                lemarad=start_num;
                pakt=start_num->kov_num;
                while(pakt!=NULL&&pakt!=torlendo){
                        lemarad=pakt;
                        pakt=pakt->kov_num;
                }
                if(pakt!=NULL)lemarad->kov_num=torlendo->kov_num;
                else return 0; // ekkor nem találtuk meg a listában
        }

        free(torlendo);
        return 1;
}


//*************************************************************************
padat list_get_akt(){
// visszaadja az aktuális elemet
//*************************************************************************
        return &(pakt->dat);
}


//*************************************************************************
padat list_get_first_nev(){
// visszaadja a név lista első elemét, pakt-ot ide állítja
//*************************************************************************
        pakt=start_nev;
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
```

```
}


//***************************************************************************
padat list_get_next_nev(){
// visszaadja a név lista következő elemét, pakt-ot ide állítja
//***************************************************************************
        if(pakt==NULL)return NULL;
        pakt=pakt->kov_nev;
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}


//***************************************************************************
padat list_get_first_cim(){
// visszaadja a cím lista első elemét, pakt-ot ide állítja
//***************************************************************************
        pakt=start_cim;
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}


//***************************************************************************
padat list_get_next_cim(){
// visszaadja a cím lista következő elemét, pakt-ot ide állítja
//***************************************************************************
        if(pakt==NULL)return NULL;
        pakt=pakt->kov_cim;
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}


//***************************************************************************
padat list_get_first_num(){
// visszaadja a telefonszám lista első elemét, pakt-ot ide állítja
//***************************************************************************
        pakt=start_num;
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}


//***************************************************************************
padat list_get_next_num(){
// visszaadja a telefonszám lista következő elemét, pakt-ot ide állítja
//***************************************************************************
        if(pakt==NULL)return NULL;
        pakt=pakt->kov_num;
        if(pakt==NULL)return NULL;
        return &(pakt->dat);
}


//***************************************************************************
int list_push_act(){
// aktuális a verembe
// visszatérési érték: sikerült: 1, tele a verem: 0
//***************************************************************************
        if(akt_verem>=MAX_VEREM)return 0;
        verem[akt_verem++]=pakt;
        return 1;
}


//***************************************************************************
int list_pop_act(){
// aktuális a veremből
// visszatérési érték: sikerült: 1, üres a verem: 0
//***************************************************************************
        if(akt_verem==0)return 0;
        pakt=verem[--akt_verem];
        return 1;
}


//***************************************************************************
int strpartcmp(const char * s1,const char * s2){
// összehasonlít a két stringben annyi karaktert, amennyi a rövidebb hossza
//***************************************************************************
        size_t n1=strlen(s1),n2=strlen(s2);
        return strncmp(s1,s2,(n1<n2)?n1:n2);
}


//***************************************************************************
```

```
padat find_first_nev(const char * nev){
// kikeresi a nevet a listából, a rá mutató pointert adja vissza. Ha
// nincs ilyen, NULL pointert. Vissza adja az olyan nevet is, amely a
// nev-ben szereplő szöveggel kezdődik vagy a talált kezdődik ezzel
//*************************************************************************
        padat p=list_get_first_nev();
        while(p!=NULL){
                if(strpartcmp(nev,p->nev)==0)return p;
                p=list_get_next_nev();
        }
        return NULL;
}


//*************************************************************************
padat find_next_nev(const char * nev){
// visszaadja a következő nevet, ha az kompatibilis a nev-vel
//*************************************************************************
        padat p;

        if(strpartcmp(nev,pakt->dat.nev)!=0)return NULL;
        if((p=list_get_next_nev())==NULL)return NULL;
        if(strpartcmp(nev,p->nev)==0)return p;
        return NULL;
}


//*************************************************************************
padat find_first_cim(const char * cim){
// kikeresi a címet a listából, a rá mutató pointert adja vissza. Ha
// nincs ilyen, NULL pointert. Vissza adja az olyan címet is, amely a
// cim-ben szereplő szöveggel kezdődik vagy a talált kezdődik ezzel
//*************************************************************************
        char cim2[150];
        padat p=list_get_first_cim();
        while(p!=NULL){
                cim2string(cim2,p);
                if(strpartcmp(cim,cim2)==0)return p;
                p=list_get_next_cim();
        }
        return NULL;
}


//*************************************************************************
padat find_next_cim(const char * cim){
// visszaadja a következő címet, ha az kompatibilis a cim-mel
//*************************************************************************
        char cim2[150];
        padat p;

        cim2string(cim2,&(pakt->dat));
        if(strpartcmp(cim,cim2)!=0)return NULL;
        if((p=list_get_next_cim())==NULL)return NULL;
        cim2string(cim2,p);
        if(strpartcmp(cim,cim2)==0)return p;
        return NULL;
}


//*************************************************************************
padat find_first_num(const char * num){
// kikeresi a számot a listából, a rá mutató pointert adja vissza. Ha
// nincs ilyen, NULL pointert. Vissza adja az olyan számot is, amely a
// num-ban szereplő szöveggel kezdődik vagy a talált kezdődik ezzel
//*************************************************************************
        char num2[150];
        padat p=list_get_first_num();
        while(p!=NULL){
                clear_tel_num(num2,p->telefonszam);
                if(strpartcmp(num,num2)==0)return p;
                p=list_get_next_num();
        }
        return NULL;
}


//*************************************************************************
padat find_next_num(const char * num){
// visszaadja a következő számot, ha az kompatibilis a num-mal
//*************************************************************************
        char num2[150];
        padat p;

        clear_tel_num(num2,pakt->dat.telefonszam);
        if(strpartcmp(num,num2)!=0)return NULL;
        if((p=list_get_next_num())==NULL)return NULL;
        clear_tel_num(num2,p->telefonszam);
```

```
        if(strpartcmp(num,num2)==0)return p;
        return NULL;
}


//*************************************************************************
unsigned count_nev(const char * nev){
// megszamolja a neveket, utana visszaallitja a pointert az elsore
//*************************************************************************
        unsigned count=0;
        padat p=find_first_nev(nev);
        list_push_act();
        for(;p!=NULL;count++)p=find_next_nev(nev);
        list_pop_act();
        return count;
}


//*************************************************************************
unsigned count_cim(const char * cim){
// megszamolja a cimeket, utana visszaallitja a pointert az elsore
//*************************************************************************
        unsigned count=0;
        padat p=find_first_cim(cim);
        list_push_act();
        for(;p!=NULL;count++)p=find_next_cim(cim);
        list_pop_act();
        return count;
}


//*************************************************************************
unsigned count_num(const char * num){
// megszamolja a telefonszamokat, utana visszaallitja a pointert az elsore
//*************************************************************************
        unsigned count=0;
        padat p=find_first_num(num);
        list_push_act();
        for(;p!=NULL;count++)p=find_next_num(num);
        list_pop_act();
        return count;
}


//*************************************************************************
unsigned valaszt_nev(const char * nev){
// Kiirja a nev-vel kompatibilis rekordokat sorszámmal, a felhasználó
// kiválasztja a kívánt sorszámot, ezt adja visszatérési értékként
//*************************************************************************
        unsigned count=1,n;
        padat p=list_get_akt();
        list_push_act();
        for(;p!=NULL;count++){
                printf("[%2d]%s, %s, %s, %s\n",count,p->nev,p->telepules,p->utca,p->telefonszam);
                if(count%20==0){
                        printf("\nNyomjon RNTER-t a folytatashoz!\n");
                        fflush(stdin);
                        getchar();
                }
                p=find_next_nev(nev);
        }
        list_pop_act();
        printf("\nHanyast valalsztja? (0=megszakit) ");
        fflush(stdin);
        while(scanf("%u",&n)!=1&&n>count){
                fflush(stdin);
                printf("Helytelen valasz, probalja ujra!\n");
        }
        return n;
}


//*************************************************************************
unsigned valaszt_cim(const char * cim){
// Kiirja a cim-mel kompatibilis rekordokat sorszámmal, a felhasználó
// kiválasztja a kívánt sorszámot, ezt adja visszatérési értékként
//*************************************************************************
        unsigned count=1,n;
        padat p=list_get_akt();
        list_push_act();
        for(;p!=NULL;count++){
                printf("[%2d]%s, %s, %s, %s\n",count,p->nev,p->telepules,p->utca,p->telefonszam);
                if(count%20==0){
                        printf("\nNyomjon RNTER-t a folytatashoz!\n");
                        fflush(stdin);
                        getchar();
                }
                p=find_next_cim(cim);
```

```
        }
        list_pop_act();
        printf("\nHanyast valalsztja? (0=megszakit) ");
        fflush(stdin);
        while(scanf("%u",&n)!=1&&n>count){
                fflush(stdin);
                printf("Helytelen valasz, probalja ujra!\n");
        }
        return n;
}


//************************************************************************
unsigned valaszt_num(const char * num){
// Kiirja a num-mal kompatibilis rekordokat sorszámmal, a felhasználó
// kiválasztja a kívánt sorszámot, ezt adja visszatérési értékként
//************************************************************************
        unsigned count=1,n;
        padat p=list_get_akt();
        list_push_act();
        for(;p!=NULL;count++){
                printf("[%2d]%s, %s, %s, %s\n",count,p->nev,p->telepules,p->utca,p->telefonszam);
                if(count%20==0){
                        printf("\nNyomjon RNTER-t a folytatashoz!\n");
                        fflush(stdin);
                        getchar();
                }
                p=find_next_num(num);
        }
        list_pop_act();
        printf("\nHanyast valalsztja? (0=megszakit) ");
        fflush(stdin);
        while(scanf("%u",&n)!=1&&n>count){
                fflush(stdin);
                printf("Helytelen valasz, probalja ujra!\n");
        }
        return n;
}


//************************************************************************
padat get_n_nev(const char * nev, unsigned n){
// Visszaadja az n. nev-vel kompatibilis rekordot
//************************************************************************
        unsigned count;
        padat p=list_get_akt();
        for(count=1;count<n;count++)p=find_next_nev(nev);
        if(p==NULL)printf("\nProgram hiba: get_n_nev -> p==NULL\n");
        return p;
}


//************************************************************************
padat get_n_cim(const char * cim, unsigned n){
// Visszaadja az n. cim-mel kompatibilis rekordot
//************************************************************************
        unsigned count;
        padat p=list_get_akt();
        for(count=1;count<n;count++)p=find_next_cim(cim);
        if(p==NULL)printf("\nProgram hiba: get_n_cim -> p==NULL\n");
        return p;
}


//************************************************************************
padat get_n_num(const char * num, unsigned n){
// Visszaadja az n. num-mal kompatibilis rekordot
//************************************************************************
        unsigned count;
        padat p=list_get_akt();
        for(count=1;count<n;count++)p=find_next_num(num);
        if(p==NULL)printf("\nProgram hiba: get_n_num -> p==NULL\n");
        return p;
}
```

fuggv.h

```
//************************************************************************
// Telefonkönyv függvénydeklarációi
// Létrehozva:          2005. 09. 16.
// Készítő:             Pohl László
// Licensz:             freeware
// Utolsó változtatás:  2005. 09. 27.
//************************************************************************
```

```
//***************************************************************************
#ifndef TELKONYV_FUGGV_HEADER
#define  TELKONYV_FUGGV_HEADER
//***************************************************************************


//***************************************************************************
#include "lista.h"
//***************************************************************************


//***************************************************************************
void betoltes();
void ujfelvesz();
void mentes();
void keres_nev_szerint();
void keres_cim_szerint();
void keres_num_szerint();
void modosit(padat p);
int IgenNem();
//***************************************************************************


#endif
```

fuggv.cpp

```
//***************************************************************************
// Telefonkönyv függvénydefiníciói
// Létrehozva:           2005. 09. 16.
// Készítő:              Pohl László
// Licensz:              freeware
// Utolsó változtatás:   2005. 09. 30.
//***************************************************************************


//***************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fuggv.h"
//***************************************************************************


//***************************************************************************
void betoltes(){
// betölti a telefon.dat fájlba korábban mentett előfizetői adatokat
//***************************************************************************
        FILE * fp;
        adat a;

        fp=fopen("telefon.dat","rb");
        if(fp==NULL){
                printf("\nNem sikerult megnyitni a telefon.dat fajlt, a beolvasas sikertelen.\n");
                return;
        }

        while(fread(&a,sizeof(adat),1,fp)==1)list_ins(&a);
        fclose(fp);
}


//***************************************************************************
void ujfelvesz(){
// megkérdezi és felveszi a listába egy új előfizető adatait
//***************************************************************************
        adat a;
        char c;

        printf("\nNev:                    ");
        fflush(stdin);
        while(gets(a.nev)==NULL)printf("Probalja ujbol\n");
        if(strcmp(a.nev,"")==0)return;

        printf("Telepules:              ");
        fflush(stdin);
        while(gets(a.telepules)==NULL)printf("Probalja ujbol\n");

        printf("Utca:                   ");
        fflush(stdin);
        while(gets(a.utca)==NULL)printf("Probalja ujbol\n");

        printf("Hazszam, emelet, ajto: ");
```

```
        fflush(stdin);
        while(gets(a.hazszam)==NULL)printf("Probalja ujbol\n");

        printf("Iranyitoszam:          ");
        fflush(stdin);
        while(scanf("%d",&a.iranyitoszam)==NULL)printf("Probalja ujbol\n");

        printf("Telefonszam:           ");
        fflush(stdin);
        while(gets(a.telefonszam)==NULL)printf("Probalja ujbol\n");

        printf("Megjegyzes:            ");
        fflush(stdin);
        while(gets(a.megjegyzes)==NULL)printf("Probalja ujbol\n");

        printf("\nNev:         %s\n",a.nev);
        printf("Cim:         %u, %s, %s %s\n",a.iranyitoszam,a.telepules,a.utca,a.hazszam);
        printf("Telefonszam: %s\n",a.telefonszam);
        printf("Megjegyzes:  %s\n\n",a.megjegyzes);

        do{
                printf("Felvehetem? (i/n):");
                fflush(stdin);
                c=getchar();
        }while(c!='I'&&c!='i'&&c!='N'&&c!='n');

        if(c=='i'||c=='I')list_ins(&a);
}


//*************************************************************************
void mentes(){
// elmenti az előfizetők adatait a telefon.dat fájlba, a legutóbbi 4
// mentés adatait is megőrzi
//*************************************************************************
        FILE * fp;
        padat p=list_get_first_nev();

        rename("telefon.ol3","telefon.ol4");
        rename("telefon.ol2","telefon.ol3");
        rename("telefon.old","telefon.ol2");
        rename("telefon.dat","telefon.old");

        fp=fopen("telefon.dat","wb");
        if(fp==NULL){
                printf("\nNem sikerult megnyitni a telefon.dat fajlt, a mentes sikertelen.\n");
                return;
        }

        while(p!=NULL){
                if(fwrite(p,sizeof(adat),1,fp)==0){
                        printf("\nA telefon.dat fajl írasa sikertelen.\n");
                        return;
                }
                p=list_get_next_nev();
        }
        fclose(fp);
}

//*************************************************************************
void modosit(padat p){
// A kiválasztott, p-ben adott előfizető adatait módosíthatjuk
//*************************************************************************
        int valasztott_ertek=0;
        adat a=*p; // csak akkor módosítjuk az eredetit, ha erre a felhasználó engedélyt ad

        while(valasztott_ertek!=8||valasztott_ertek!=9){
                printf("\nNev:  %s\n",a.nev);
                printf("Cim:  %u, %s, %s %s\n",a.iranyitoszam,a.telepules,a.utca,a.hazszam);
                printf("Tel:  %s\n",a.telefonszam);
                printf("Megj: %s\n",a.megjegyzes);

                printf("\nValasszon!\n");
                printf("1  Nev modositasa\n");
                printf("2  Telepules modositasa\n");
                printf("3  Utca modositasa\n");
                printf("4  Hazszam modositasa\n");
                printf("5  Iranyitoszam modositasa\n");
                printf("6  Telefonszam modositasa\n");
                printf("7  Megjegyzes modositasa\n\n");
                printf("8  Elofizeto torlese\n");
                printf("9  Modositasok tarolasa, es vissza a fomenube\n");
                printf("10 Modositasok figyelmen kivul hagyasa, es vissza a fomenube\n");

                fflush(stdin);
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                switch(valasztott_ertek){
```

```
                        case  1:
                                printf("\nNev:                    ");
                                fflush(stdin);
                                while(gets(a.nev)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  2:
                                printf("\nTelepules:              ");
                                fflush(stdin);
                                while(gets(a.telepules)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  3:
                                printf("\nUtca:                   ");
                                fflush(stdin);
                                while(gets(a.utca)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  4:
                                printf("\nHazszam, emelet, ajto: ");
                                fflush(stdin);
                                while(gets(a.hazszam)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  5:
                                printf("\nIranyitoszam:           ");
                                fflush(stdin);
                                while(scanf("%d",&a.iranyitoszam)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  6:
                                printf("\nTelefonszam:            ");
                                fflush(stdin);
                                while(gets(a.telefonszam)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  7:
                                printf("\nMegjegyzes:             ");
                                fflush(stdin);
                                while(gets(a.megjegyzes)==NULL)printf("Probalja ujbol\n");
                                break;
                        case  8:
                                printf("\nBiztosan torolni kivanja? (I/N)");
                                if(!IgenNem())break;
                                if(!list_del())printf("\nProgram hiba: keres_nev_szerint -> list_del\n");
                                                return;
                        case  9:
                                list_push_act();
                                if(!list_ins(&a))printf("\nProgram hiba: keres_nev_szerint -> list_ins\n");
                                list_pop_act();
                                if(!list_del())printf("\nProgram hiba: keres_nev_szerint -> list_del\n");
                        case  10:return; // a 8-as is hasznalja!
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
}


//***************************************************************************
void keres_nev_szerint(){
// Név szerint keres, ha csak az eleje stimmel, akkor is találat
//***************************************************************************
        char nev[100];
        unsigned talalat;
        padat p;

        printf("Keresett nev: ");
        fflush(stdin);
        if(gets(nev)==NULL){
                printf("\n\nSikertelen adatbevitel.\n");
                return;
        }

        talalat=count_nev(nev);
        if(talalat==0){
                printf("\nIlyen nevu ugyfel nem talalhato\n");
                return;
        }

        printf("\nA talalatok szama %u. Kivanja megjelenitreni? (I/N) ",talalat);
        if(!IgenNem())return;

        if((talalat=valaszt_nev(nev))==0)return;
        if((p=get_n_nev(nev,talalat))==NULL)return;

        modosit(p);
}


//***************************************************************************
void keres_cim_szerint(){
// Cím szerint keres, ha csak az eleje stimmel, akkor is találat
//***************************************************************************
        char cim[100];
```

```
        unsigned talalat;
        padat p;

        printf("\nKeresett cim: ");
        fflush(stdin);
        if(gets(cim)==NULL){
                printf("\n\nSikertelen adatbevitel.\n");
                return;
        }

        talalat=count_cim(cim);
        if(talalat==0){
                printf("\nIlyen cimu ugyfel nem talalhato\n");
                return;
        }

        printf("\nA talalatok szama %u. Kivanja megjelenitreni? (I/N) ",talalat);
        if(!IgenNem())return;

        if((talalat=valaszt_cim(cim))==0)return;
        if((p=get_n_cim(cim,talalat))==NULL)return;

        modosit(p);
}


//*************************************************************************
void keres_num_szerint(){
// Telefonszám szerint keres, ha csak az eleje stimmel, akkor is talál
//*************************************************************************
        char num[100];
        unsigned talalat;
        padat p;

        printf("\nKeresett num: ");
        fflush(stdin);
        if(gets(num)==NULL){
                printf("\n\nSikertelen adatbevitel.\n");
                return;
        }

        talalat=count_num(num);
        if(talalat==0){
                printf("\nIlyen telefonszamu ugyfel nem talalhato\n");
                return;
        }

        printf("\nA talalatok szama %u. Kivanja megjelenitreni? (I/N) ",talalat);
        if(!IgenNem())return;

        if((talalat=valaszt_num(num))==0)return;
        if((p=get_n_num(num,talalat))==NULL)return;

        modosit(p);
}


//*************************************************************************
int IgenNem(){
// Ha I-t nyom a felhasználó, 1-et ad vissza, ha N-t, 0-t, egyéb esetben
// újra kéri
//*************************************************************************
        fflush(stdin);
        while(1){
                switch(getchar()){
                        case 'i':
                        case 'I':
                        case 'y':
                        case 'Y': return 1;
                        case 'n':
                        case 'N': return 0;
                }
                printf("I/N? ");
        }
}
```

main.cpp

```
//*************************************************************************
// Telefonkönyv main program
// Létrehozva:          2005. 09. 16.
// Készítő:             Pohl László
// Licensz:             freeware
// Utolsó változtatás:  2005. 09. 30.
```

```
//*************************************************************************


//*************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fuggv.h"
//*************************************************************************


//*************************************************************************
void kereses(){
// A Kereses/Valtoztatas választása esetén ez hívódik meg
//*************************************************************************
        int valasztott_ertek=0;

        while(valasztott_ertek!=9){

                printf("\nKereses/Valtoztatas\n");
                printf("1  Nev szerint\n");
                printf("2  Cim szerint\n");
                printf("3  Telefonszam szerint\n");
                printf("9  Vissza a fomenube\n");

                fflush(stdin);
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                switch(valasztott_ertek){
                        case  1: keres_nev_szerint();     break;
                        case  2: keres_cim_szerint();     break;
                        case  3: keres_num_szerint();     break;
                        case  9:                                        break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
}


//*************************************************************************
main(){
//*************************************************************************
        int valasztott_ertek=0;

        betoltes();

        while(valasztott_ertek!=10){

                printf("\nValasszon!\n");
                printf("1  Uj elofizeto hozzaadasa\n");
                printf("2  Kereses/Valtoztatas\n");
                printf("3  Adatok mentese\n");
                printf("10 Kilepes\n");

                fflush(stdin);
                if(scanf("%d",&valasztott_ertek)!=1)valasztott_ertek=0;

                switch(valasztott_ertek){
                        case  1: ujfelvesz();     break;
                        case  2: kereses();               break;
                        case  3: mentes();                break;
                        case 10:                          break;
                        default: printf("Hibas muveletszam (%d). Probalja ujra!",valasztott_ertek);
                }
        }
        printf("Kivanja menteni? (I/N) ");
        if(IgenNem())mentes();
        return 0;
}
```

9.4 Használati utasítás (felhasználói dokumentáció)

A telefonkönyv programmal előfizetők adatait (nevét, címét, telefonszámát) tárolhatjuk, az adatok között kereshetünk, módosíthatjuk, vagy törölhetjük azokat.

Indítás után, amennyiben korábban elmentettük a telefonkönyvet, ez automatikusan betöltődik a telefon.dat adatállományból. Az adatállomány hiányára üzenet figyelmeztet. Ezt követően az alábbi menüből választhatunk:

1 Új előfizető hozzáadása

2 Keresés/Változtatás
3 Adatok mentése
10 Kilépés

A választáshoz írjuk be a megadott számot, majd nyomjuk meg az ENTER billentyűt.

Új előfizető hozzáadása

A menüpont választása után a következő adatokat kell megadnunk:

- Név
- Település
- Utca
- Házszám, emelet, ajtó
- Irányítószám
- Telefonszám
- Megjegyzés

Ezek többsége egyértelmű, általános megjegyzés, hogy nyugodtan használhatunk magyar ékezetes karaktereket is.

Az utca neve után írjuk oda a jellegét is (utca, út, tér, köz, dűlő, sor stb.)

A telefonszámba írhatunk kötőjelet, pluszjelet, szóközt, vagy zárójelet is. (pl.: +36(99) 123-4567). Kereséskor csak a számjegyeket veszi figyelembe a program.

Keresés/Változtatás

A következő lehetőségek közül választhatunk:
1 Név szerint
2 Cím szerint
3 Telefonszám szerint
9 Vissza a főmenübe

Az első három lehetőség valamelyikének választása esetén a program bekéri a keresett nevet/címet/telefonszámot. Fontos, hogy a program érzékeny a kisbetű-nagybetű közötti különbségekre. Bármely lehetőség választása esetén nem csak a pontosan egyező nevű/című/telefonszámú ügyfelek adatait találja meg a kereső, hanem azokét is, akinek a neve/címe/telefonszáma eleje azonos a keresettel, vagy a keresett eleje azonos ezzel. Például, ha nevet keresünk, és beírjuk, hogy Kovács, akkor az összes Kovács nevű előfizető adatait megkapjuk. Ha nem adunk meg semmit, csak leütjük az ENTER-t, akkor az összes előfizető adatait megkapjuk.

Ha cím szerint keresünk, és viszonylag részletesen szeretnénk keresni, a keresett helyet egybeírva adjuk meg! Pl.: BudapestKis utca! Telefonszám esetén a kereső csak a számjegyeket veszi figyelembe, tehát megadhatjuk, hogy 3699 1234567, ekkor meg fogja találni az előző pontban említett előfizetőt.

Ezután a program kiírja a találatok számát, és megkérdezi, hogy megjelenítse-e őket. Ha igennel válaszolunk, az előfizetők adatai a sorbarendezés tárgyának megfelelő adat szerint lesznek rendezve, minden megjelenített előfizető kap egy sorszámot. A sorszámot beírva megjelenik az előfizető összes adata, továbbá egy újabb menü:

1 Név módosítása
2 Település módosítása
3 Utca módosítása
4 Házszám módosítása
5 Irányítószám módosítása
6 Telefonszám módosítása
7 Megjegyzés módosítása
8 Előfizető törlése
9 Módosítások tárolása, és vissza a főmenübe
10 Módosítások figyelmen kívül hagyása, és vissza a főmenübe

1-7 beírása esetén módosíthatjuk az előfizető adatát, ezután a módosított adattal jelenik meg ismét az előfizető adatainak kiírása, és ismét visszakapjuk az előbbi menüt.

8-10 választása esetén visszatérünk az előző menübe, más-más hatással. 8 esetén törlődik az előfizető, 9 esetén az előfizető adatai kicserélődnek a most módosított adatokra, 10 esetén pedig egyik mostani változtatás sem lép életbe.

Adatok mentése

Ezt a menüpontot választva az adatok a telefon.dat fájlba kerülnek. Az előző mentés telefon.old néven továbbra is elérhető, sőt három korábbi is, rendre telefon.ol2, telefon.ol3 és telefon.ol4 néven. A program kilépés előtt is rákérdez, hogy mentse-e az adatokat.

9.5 Programozói dokumentáció

lista.h, lista.cpp

Az előfizetők adatait az alábbi típusú adatstruktúrákban tároljuk, melynek definíciója a lista.h fejlécállományban található:

```
//*************************************************************************
typedef struct{
//*************************************************************************
        char    nev[60];
        char    telepules[50];
        char    utca[50];
        char    hazszam[50];     // emelet és ajtó ispl.: 2/A 8.em 36.
        unsigned iranyitoszam;
        char    telefonszam[50]; // kötőjel, zárójel és szóköz is lehet benne
        char    megjegyzes[100];
}adat,*padat;
```

Az adatstruktúrákat három láncolt listára fűzzük fel: név, cím illetve telefonszám szerint sorba rendezve. A láncolt lista elemeit a következő struktúra tartalmazza:

```
//*************************************************************************
// A lista elemei csak ebben a fájlban elérhetők
typedef struct el{
//*************************************************************************
        adat dat;
        struct el *kov_nev,*kov_cim,*kov_num;
}elem,*pelem;
```

Ez a struktúra típus csak a lista.cpp függvényei számára elérhető, kívülről csak közvetve, függvényeken keresztül kaphatjuk meg az adat struktúrákra mutató pointereket, illetve adhatunk új adat elemeket a listákhoz.

A lista.h-ban deklarált, globálisan elérhető függvények definíciója a lista.cpp-ben található, melyek a következők:

```
int list_ins(adat*);
```

A paraméterként kapott, pointerrel megadott adatról másolatot készít (ehhez memóriát foglal), és ezt felfűzi mindhárom listára. Siker esetén 1, probléma esetén 0 a visszatérési érték.

```
void list_clear();
```

Törli a három listát, az összes elemmel egyetemben. A törlés során a nev listán megy végig.

```
int list_del();
```

Törli az aktuális elemet, ehhez először kiláncolja mindhárom listából, aztán felszabadítja a lefoglalt memóriát. Siker esetén 1, sikertelenség esetén 0 a visszatérési érték (ha sikertelen, az erősen programhibára utal).

```
padat list_get_akt();
padat list_get_first_nev();
padat list_get_next_nev();
padat list_get_first_cim();
padat list_get_next_cim();
padat list_get_first_num();
padat list_get_next_num();
```

Mind a hét függvény a pakt által mutatott listaelem dat adattagjára mutató pointert ad vissza, a first ill. next függvények ezen kívül előbb megváltoztatják pakt értékét is: a first függvények az adott lánc első elemére állítják, a next elemek a következőre. Ha a pakt NULL (mert a lista végén áll), akkor a visszatérési érték is NULL.

```
int list_push_act();
int list_pop_act();
```

Ahhoz, hogy a count és a valaszt függvények vissza tudják állítani pakt értékét az első, a paraméterükben megadott értéknek megfelelő listaelemre, szükség van egy kisméretű veremre, mert ezeknél a függvényeknél már nem engedjük meg, hogy közvetlenül hozzáférjenek a lista elemeihez, így pakt-hoz sem. (Akár másik modulban is lehetnének, de inkább a listához kötődnek, ezért kerültek ide.) A verem kezelését a fenti két függvény végzi. Egyéb felhasználás esetén ügyeljünk arra, hogy a verem maximum 20 elemű.

```
padat find_first_nev(const char*);
padat find_first_cim(const char*);
padat find_first_tel(const char*);
padat find_next_nev(const char * nev);
padat find_next_cim(const char * cim);
padat find_next_num(const char * num);
```

Visszaadja az első, illetve a következő, a paraméterrel kompatibilis adatot. Ha nincs ilyen, NULL-lal tér vissza. A „kompatibilis” azt jelenti, hogy a keresett és keresendő szövegből csak annyi karaktert vizsgálunk, amennyi mindkettőben megtalálható. Az összehasonlítást az strpartcmp függvény végzi. Cím esetén a címből stringet a cim2string függvény készít,

telefonszám esetén a telefonszámból a számjegyek kiszűrését a clear_tel_num függvénnyel oldjuk meg.

```
unsigned count_nev(const char * nev);
unsigned count_cim(const char * cim);
unsigned count_num(const char * num);
```

Ezek a függvények megszámolják, hogy hány, a paraméterrel kompatibilis elem található a listában, a pakt pointert az első ilyenre állítják.

```
unsigned valaszt_nev(const char * nev);
unsigned valaszt_cim(const char * cim);
unsigned valaszt_num(const char * num);
```

A függvények kiírják az összes, a paraméterrel kompatibilis elem főbb adatait, minden elem elé egy sorszámot, ezután várják, hogy a felhasználó megadja a sorszámot. Hibásan megadott sorszám esetén újra kéri. Visszatérési értéke a választott sorszám. Feltételezi, hogy a pakt az első kompatibilis néven áll. A függvény lefutása után pakt ugyanott áll, ahol a függvény meghívásakor.

```
padat get_n_nev(const char * nev,unsigned n);
padat get_n_cim(const char * cim,unsigned n);
padat get_n_num(const char * num,unsigned n);
```

Visszaadja az első paraméterrel kompatibilis n. adatot, ha nincs ilyen, NULL-t. Feltételezi, hogy a pakt az első kompatibilis néven áll. A függvény elhagyásakor pakt a kiválasztott elemen áll (ha a felhasználó törölni akarja, akkor így nem lesz szükség egyéb műveletre).

További függvények a lista.cpp-ben:

```
void cim2string(char * string, const padat cim);
```

A stringbe másolja egymás után a települést+utcát+házszámot.

```
void clear_tel_num(char * tiszta, const char * full);
```

A tisztába kerül a full szöveg összes számjegye, de semmi más.

```
int cimcmp(const padat cim1,const char * cim2){
int telcmp(const char * tel1,const char * tel2){
```

Összehasonlítja a két címet, illetve a két telefonszámot, hogy megegyeznek-e. Ezeket a függvényeket a beszúrásnál használjuk. Teljes összehasonlítást végeznek, nem kompatibilitást!

```
int strpartcmp(const char * s1,const char * s2){
```

Összehasonlítja a két string első annyi karakterét, amennyi darab mindkettőben szerepel. Ha bármelyik üres string, akkor egyeznek, hisz az üres string mindennel kompatibilis.

fuggv.h, fuggv.cpp

Ebben a csomagban különböző funkciót betöltő függvények találhatók.

```
void betoltes();
```

A telefon.dat fájlból betölti a korábban betöltött adatokat (a listákba a list_ins() függvénnyel veszi fel őket). Ha nem találja a fájlt, hibaüzenetet ír ki. A fájl bináris, adat típusú blokkokból áll.

```
void ujfelvesz();
```

Bekéri a felhasználótól az új előfizető adatait, a beírás után visszaírja a megadott információt a képernyőre, majd megkérdezi, hogy felvegye-e a listába. Igen válasz esetén a list_ins() függvénnyel beilleszti.

```
void mentes();
```

Először átnevezi a fájlokat a következő sorrendben: telefon.ol4<-telefon.ol3<-telefon.ol2<-telefon.old<-telefon.dat, majd telefon.dat néven menti az előfizetők adatait.

```
void keres_nev_szerint();
void keres_cim_szerint();
void keres_num_szerint();
```

Megkeresi a felhasználó által megadott névvel/címmel/telefonszámmal kompatibilis adatot a listában, sorszámozva kiírja, a felhasználó kiválasztja a szerkeszteni kívánt sorszámút, ezután a modosit() függvény segítségével lehetőség van az adatok módosítására.

```
void modosit(padat p);
```

A felhasználó tízféle lehetőség közül választhat:

```
printf("1  Nev modositasa\n");
printf("2  Telepules modositasa\n");
printf("3  Utca modositasa\n");
printf("4  Hazszam modositasa\n");
printf("5  Iranyitoszam modositasa\n");
printf("6  Telefonszam modositasa\n");
printf("7  Megjegyzes modositasa\n\n");
printf("8  Elofizeto torlese\n");
printf("9  Modositasok tarolasa, es vissza a fomenube\n");
printf("10 Modositasok figyelmen kivul hagyasa, es vissza a fomenube\n");
```

Az utolsó három hatására visszatér a hívó függvényhez, az első hét esetén végrehajtja a kívánt módosítást, és az eredményt visszaírja, majd ismét választhatunk.

A 9 választása esetén nem egyszerűen visszamásolja az adatokat a megfelelő rekordba, hanem list_ins()-zel beszúrja, és az eredetit törli. Erre azért van szükség, mert a lista rendezettsége így őrizhető meg, a módosítás ugyanis befolyásolhatja a rekord helyét a listákban.

```
int IgenNem();
```

Ha a felhasználó I-t nyom, 1 a visszatérési érték, ha N-t, akkor 0. Más esetben újra kéri.

main.cpp

Ebben csak két függvény kapott helyet:

```
void kereses();
```

Menüt ír ki, választhatunk név, cím, telefonszám szerinti keresés, továbbá a főmenübe való visszatérés közül.

```
main();
```

A főmenüt jeleníti meg:

```
printf("1  Uj elofizeto hozzaadasa\n");
printf("2  Kereses/Valtoztatas\n");
printf("3  Adatok mentese\n");
printf("10 Kilepes\n");
```

Vége

F.1 Segítség a feladatok megoldásához

2.3 Pl. 7 faktoriálisa 1*2*3*4*5*6*7=5040

2.5 0C=32F, 100C=212F, lineáris

3.1 A Fibonacci sorozat i-edik eleme: xi=xi-1+xi-2, első két eleme 0 és 1 (aztán 1, 2, 3, 5, 8, 13, 21…)

3.2 i akkor közös osztója a-nak és b-nek, ha (a%i==0&&b%i==0)

3.6 A és B akkor barátságos, ha A osztóinak összege=B, és B osztóinak összege=A. Barátságos számok például a 220 és 284, ugyanis

220 osztói: 1+2+4+5+10+11+20+22+44+55+110=284

284 osztói: 1+2+4+71+142=220

Továbbá pl. 1184 és 1210, 17296 és 18416, 9363584 és 9437056.

4.2 Ne feledkezzen meg arról, hogy az első stringet lezáró '\0'-t is felül kell írni, nem pedig utána tenni a másolatot, mert akkor nem ér semmit az egész, hiszen a string az első '\0'-ig tart, és lényegtelen, mi van utána. Természetesen a hozzáfűzött részt'\0'-val kell lezárni.

4.3 A két mátrix mérete legyen 3×4-es, és 4×5-ös, az eredmény nyilván 3×5-ös. A mátrixszorzás úgy történik, hogy az eredmény ij-edik eleme az első mátrix i-edik sorának, és a második mátrix j-edik oszlopának skaláris szorzata, tehát $c_{ij} = \sum_k a_{ik} b_{kj}$. Itt tehát három, egymásba ágyazott ciklusra lesz szükség, a két külső i-re és j-re a szokásos, ezeken belül ez szerepel: {c[i][j]=0; for(k=0;k<4;k++)c[i][j]+=a[i][k]*b[k][j];}.

4.4 if(s[i]==c)return &s[i]; // vagy return s+i;

4.5 pl.: for(i=0;s1[i]==s2[i]&&s1[i]!='\0';i++); return s1[i]-s2[i];

De gondolja végig, miért jó ez a megoldás. Miért elég csak s1[i]!=0-t ellenőrizni? Mi történik, ha eltérés van a két stringben? Ha a két string azonos? Ha az egyik véget ér, de addig egyformák voltak?

4.9 void csere(char * cel, char * forras, char * ezt, char * erre);

5.1 void binrend(double keresett_elem, double * tomb, double meret);

5.2 void rendez(char t[][81],int meret);

A stringeket strcpy-vel célszerű másolni, összehasonlítani pedig strcmp-vel (while(strcmp(t[i],t[j])>0).

5.3 void rendez(char ** s,int elemszam);

Itt csak a pointereket kell cserélgetni, a stringek maradnak a helyükön. Az összehasonlítás továbbra is strcmp-vel történhet.

F.2 Lehetséges nagy házi feladatok

Poppe András instrukciói (amennyiben ütközés van ez és a későbbiek között, ezek az irányadóak, de célszerű rákérdezni).

NHF: 500-1000-2000 sor terjedelmu program, ami a nyelvi
eszkozok tobbseget igenybe veszi. Tobb forrasmodulra szegmentalt
program legyen, file-kezelessel es dinamikus tarkezelessel is
valamint egy-ket nem teljesen trivialis algoritmus sajat
leprogramozasaval. Tombok es strukturak is hasznalandok.

Teljes dokumentaciot kell keszitenie.
Specifikacio, program dokumentacioja (adatszerkezetek,
fobb algoritmusok, vezerles, forrasmodulok leirasa, stb.),
felhasznaloi dokumentacio, tesztelesi leirasa, tesztelesi eredmenyek.

Javaslom, hogy a fentieket mar lassan hirdessetek ki. Mindenki
hozhat sajat problemat is. Az 5. es 7. het kozott keszuljenek el
a specifikaciok. Minden hallgato vagy 2-3 fos hallgatoi team
keszitse el a feladatkituzes utan a sajat specifikaciojat, amit ti
ellenorizzetek! Ha tul konnyu a dolog, turbozzatok fel, ha tul
nagy faba akarjak vagni a fejszet, vegyetek vissza belole! Csak
a laborvezeto altal jovahgyott elozetes specifikacio alapjan
kezdhetnek el a sracok programozni.

Beadando: a teljes dokumentacio keszlet (hardcopy-ban, bekotve),
a futo program. Bemutatando szemelyesen: a forraskod es a futo
program. Beadas: utolso elotti heten.

Pelda NHF-ek:

1) LaTeX -> HTML atkodolo program (pontosan meghatarozando az
a LaTeX parancskeszlet, amit ismernie kell).

2) Szakirodalmi adatok (publikaciok) nyilvantartasara valo program,
ami BibTeX formatumu listakat be tud olvasni, illetve sajat adatait
szinten BibTeX formaban tarolja.

3) Programozhato zsebkalkulatort emulalo program

4) Akarmilyen strukturalt adathalmaz kezelesere szolgalo
program. Tarolasi forma: XML file. Muveletek: kereses,
beszuras, torles, listazas, ket file egyesitese, resz-listak
levalogatasa
es mentese.

Ezekhez nem kell csicsas GUI. Ha valaki GUI-t is akar, rendben,
de forraskod szinten 1 file-ra korlatozodjon a GUI, a lenyegi
reszek legyenek kulon es azok ne akarjanak onalloan kommunikalni.

GUI-t igenylo, bonyolultabb feladatok:

a) Kepletszerkeszto
b) WYSIWIG editor (pl. LaTeX-hez)
c) Kapcsolasi rajz szerkeszto program
d) Folyamatabra szerkeszto program

Általános előírások

Ezek az előírások nálam érvényesek, más gyakorlatvezetőnek mások lehetnek a preferenciái.

A feladatot lehetőség szerint szabványos C nyelven oldjuk meg, de megengedett a C++ használata is. A nagy házi célja, hogy a hallgató minél jobban begyakorolja a C nyelvet, algoritmusokat valósítson meg (rendezés, adatbázis stb.), a hangsúly mindig ezen legyen.

A feladatok között előfordul több olyan is, mely grafikát kíván, ezeket nyilván csak nem szabványos módon oldhatjuk meg, hiszen a szabvány semmiféle grafikát nem támogat. A grafikára épülő feladatoknál is a lényeg a C nyelv használata.

Grafika használatához talán legegyszerűbb a Borland C++/Turbo C fordítók által biztosított függvények. Ezek leírása magyarul is olvasható [1]-ben, hátránya, hogy csak ezek a fordítók ismerik, és csak DOS alatt használható. Aki ablakozós grafikát szeretne, használja a [3] weblapról letölthető, DevC++-ra épülő, Delphi szerűen egyszerűen használható, ingyenes fejlesztőkörnyezet. Használhatja persze a Visual C++-ét is. Mindkettőre jellemző, hogy C++ tudást igényel. További lehetőség az Allegro [4], ez egy C-ben írt, játékprogramok fejlesztéséhez kidolgozott platformfüggetlen, free függvénykönyvtár.

Grafikát tartalmazó nagy házit csak azok válasszanak, akik nem most kezdték a programozás tanulását, és a hatodik héten már pontosan tudják, hogy hogyan is valósítanák meg a programot!

A program tartalmazzon **dinamikus adatszerkezetet**, **fájlkezelést**, továbbá **valamilyen bonyolultabb algoritmust** is. Lehetőleg a program több forrásfájlból álljon.

NAGYON FONTOS! Mindenki sajátkészítésű házi feladatot adjon be, akkor is, ha nem sikerül túl jól. Nálam mindenki jelest kap, aki működőképesen elkészíti azt a programot, amit bevállalt, nem jár pluszpont a fantasztikus dizájnért, és nem jár levonás az egyszerű kezelőfelületért. A bemutatott példaprogramokban szereplő menühöz hasonlóval tökéletesen elégedett lennék. Nagyon könnyen kiderül, ha valaki nem maga készítette a feladatát, mert személyesen kell elmagyarázni a program működését, és aki nem maga készítette, az nem is tudja hitelesen bemutatni. A nem saját házinak félévismétlés is lehet a következménye ebből a tárgyból.

Dokumentáció:
A dokumentáció két részből kell álljon: felhasználói és programozói dokumentáció. A két rész együttes mérete legalább 5 oldalnyi, szimpla sortávolságú, 12 pontos Times New Roman típusú betűkkel szedett A/4-es oldalnál nem lehet rövidebb. Ebbe az 5 oldalba azonban nem számítanak bele az ábrák és forráskód részletek.

Felhasználói dokumentáció: azoknak szól, akik a programot használni fogják. Tehát itt kell leírni, hogy mit és hogyan lehet csinálni a programban.

Programozói dokumentáció: annak a programozónak szól, aki tovább akarja fejleszteni a programot. A programozói dokumentáció része a tesztelési dokumentáció is.

A dokumentáció szerves részét képezik a programban elhelyezett megjegyzések. Ezek használata jelentősen gyorsíthatja a program jövőbeni továbbfejlesztését, valamint az esetleges hibák megtalálását is.

A dokumentációt nyomtatva és elektronikusan, a programot pedig elektronikus formában kérem SZEMÉLYESEN leadni!

A leadási határidő az utolsó előtti heti óra. Aki eddig nem készül el, az is köteles bemutatni az eddig elkészülteket. Ha a program készültségi foka nem éri el a 75%-ot, vagy a bemutatást e hét vége előtt elmulasztja, akkor a végső leadáskor közepes (3) osztályzatnál jobbat már nem kaphat a program készítője.

Aki elkésik, különeljárási díjat kell, hogy fizessen a Neptunon keresztül. De a késésnek is van határideje: utolsó hét csütörtök, dél (12:00 óra). Aki ezt lekési, félévet ismétel! A dokumentáció különeljárási díj nélkül is leadható az utolsó héten. Ha a dokumentáció leadása elmarad, de a programot megkaptam és elfogadtam, a nagyházi elégséges.

Az itt felsorolt példák nem kötelezően választandók, épp ellenkezőleg! Akinek saját elképzelése van, még inkább, ha valóban használni vagy eladni akarja, akkor célszerű azt választani.

Az alábbi példákat sem kell szentírásnak tekinteni, a tisztelt házi feladat készítő nyugodtan kiegészítheti saját ötleteivel, elhagyhatja a szükségtelennek ítélt részleteket. A kiválasztott nagy házi feladatát papíron vagy e-mail-ben meghatározott időpontig specifikálnia kell (magyarán kell készítenie egy rövid, max. fél oldalas összefoglalót, hogy mit fog tudni a program). Ettől a specifikációtól eltérni nem lehet, csak ha erre áldásom adtam. Nem fogadok el olyan házi feladatot, ha a delikvens azt mondja „én az amőbát választottam, de inkább CD adatbázist csináltam”. Ha valaki úgy érzi, hogy nem tudja az eredetileg vállalt feladatot elkészíteni, helyette mást szeretne, erre módja van a leadási határidőt megelőző hét szerda éjfélig, e szándékát írásban közölje az új program specifikációjával egyetemben.

1. Kétnyelvű szótár program

Készítsen programot C nyelven, amely kétnyelvű elektronikus szótárat valósít meg. A szótárat szöveges fájlban tárolja!
Javaslat: használja a *http://www.mek.iif.hu/porta/szint/egyeb/szotar/ssa-dic/eng-hun/* honlapon található angol-magyar szótár fájlt.

Funkciók: keresés mindkét nyelvben, új szó felvétele, szó jelentésének módosítása, szó törlése.

2. Fordítóprogram

Készítsen programot C nyelven, amely a paraméterként megadott szöveges fájlban található szöveget lefordítja egy másik nyelvre. Javaslat: használja a *http://www.mek.iif.hu/porta/szint/egyeb/szotar/ssa-dic/eng-hun/* honlapon található angol-magyar szótár fájlt.

Funkciók: A szótárfájlt tekintse adottnak, nem szükséges, hogy a programban ez bővíthető legyen.

A program először teljes mondatokat próbáljon behelyettesíteni (ha vannak a szótárban teljes mondatok), ezután a maradék szövegben a kifejezéseket helyettesítse be (csak teljes egyezés esetén kell cserélni). Ezután pedig a megmaradt szavakat fordítsa le.

Azokat a szavakat, melyek nem szerepelnek a szótárban, hagyja változatlanul a szövegben, de írja ki azokat egy fájlba felsorolásszerűen (minden ilyen szó csak egyszer szerepeljen ebben a fájlban.) Ha egy szónak vagy kifejezésnek több jelentése is van, ezek közül mindig az elsőt helyettesítse be.

A lefordított szöveg szintén szöveges fájlba kerüljön.

3. Pac-man

Készítsen Pac-Man játékprogramot C nyelven.

Funkciók: A játék legalább 10 pályából álljon. Legyen menthető és betölthető a játékállás. A program vezessen toplistát az elért pontszámok alapján.

4. Amőba

Készítsen amőba játékprogramot C nyelven.

Funkciók: ember-ember ellen, ember-gép ellen. A rács méretét lehessen állítani legalább 2 fokozatban (pl. 12x12 és 18x18).

5. Szöveges kalandjáték

Készítsen szöveges kalandjáték programot C nyelven. Javasolt irodalom: *F. DaCosta: A kalandprogram írásának rejtelmei*. A program lehet ennél egyszerűbb felépítésű is, például nem szükséges, hogy beírt szövegeket értelmezzen. Elég, ha a játékos egy listából választhatja ki, hogy mit tesz. Célszerű a játékkörnyezetet szöveges fájlban tárolni, így lehetőséget adva a játékosoknak saját „küldetések” készítésére.

Funkciók: a játékállás mentése és betöltése. Készítsen a játékhoz teljes „küldetést” is, legalább 10 helyiséggel.

6. Grafikus kalandjáték

Készítsen szöveges kalandjáték programot C nyelven. Javasolt irodalom: *F. DaCosta: A kalandprogram írásának rejtelmei*. Dolgozhat grafikus üzemmódban is.

Funkciók: a játékállás mentése és betöltése. Készítsen a játékhoz küldetést is, legalább 10 helyiséggel.

7. Telefonkönyv

Készítsen telefonkönyv programot C nyelven. A telefonkönyv tárolja az előfizetők nevét, címét, telefonszámát.

Funkciók: Keresés név és telefonszám alapján. Új előfizető felvétele az adatbázisba, előfizető törlése, előfizető valamely adatának megváltoztatása.

8. Tetris

Készítsen tetris programot C nyelven.

Funkciók: Legalább 5 sebességi fokozat. A toplistát fájlban tárolja.

9. Órarendkészítő program

Készítsen órarendkészítő programot C nyelven.

A program egyhetes és kéthetes (páros és páratlan hét) beosztást is kezeljen.

- Bemenő adatok: tanárok adatai, csoportok adatai, rendelkezésre álló osztálytermek (speciális labor vagy tornaterem is lehet, ekkor bizonyos órákat kötelező itt megtartani, ill. másokat nem lehet itt tartani) valamint a köztük lévő kapcsolat (melyik csoportnál, milyen rendszerességgel (heti hány óra) kell órát tartani. Vegye figyelembe, hogy nem lehet éjjel-nappal órákat tartani, ezért a csoportok megadásakor kell azt is megadni, hogy az adott csoportok mikor érnek rá, illetve a tanároknál is megadható legyen, hogy az illető mikor nem ér rá. Figyeljen arra, hogy egy terem hány férőhelyes, ill. egy csoport hány fős létszámú.
- Kimenő adatok: órarendek a tanároknak, az egyes csoportoknak, valamint a termeknek (adott teremben mikor milyen óra van). Ha valamilyen feloldhatatlan ütközés van, akkor azt a program jelzi. A keletkezett órarendeket táblázatos formában kell kiadni (text fájlba, de ha a házi feladat készítője gondolja, lehetővé teheti a képernyős megjelenítést is emellett.)

A bemenő adatokat úgy kell tárolni, hogy utólag könnyű legyen módosítani, és ne kelljen mindent újból megadni egy esetleges változáskor. Ha a feladat készítője gondolja, akkor megpróbálkozhat azzal, hogy az órarendet optimalizálja úgy, hogy minél kevesebb lyukasóra legyen, elsősorban a csoportoknál, másodsorban a tanároknál.

10. Raktári árunyilvántartó

Készítsen árunyilvántartó programot C nyelven. Az árukat vonalkód azonosítja. Az áru paraméterei közé tartoznak: beszállító, vételár, eladási ár, az áru helye a raktárban, a beszerzés és az eladás időpontja stb.

Funkciók: új árucikk felvétele, árucikk törlése, adott áru paramétereinek változtatása (pl. hány darab van raktáron, hol van, mennyibe kerül stb.), keresés legalább vonalkód alapján. Az adatbázist fájlban tárolja.

11. Média nyilvántartó.

Készítsen mp3/film/CD/DVD/stb. nyilvántartó adatbázist C nyelven. Tárolandó adatok: szerző, cím, hossz, nyelv, stb., ami az adott média jellemzője. (Csak egyféle médiát kell kezelnie a programnak, csak lusta voltam külön mp3 nyilvántartó/film nyilvántartó/… leírást készíteni).

Funkciók: új elem felvitele, elem törlése, elem módosítása, keresés legalább szerző és cím szerint.

12. Határidőnapló

13. Kártyajáték

14. Gázmodell

15. Rulett

16. Kígyó játék (két játékos irányít egy-egy kígyót, a kígyók esznek, és eközben nőnek, aki a másiknak ütközik, veszít)

17. Dallamszerkesztő

18. LaTeX -> HTML átkódoló program (pontosan meghatározandó az a LaTeX parancskészlet, amit ismernie kell).

19. Szakirodalmi adatok (publikációk) nyilvántartására való program, ami BibTeX formátumú listákat be tud olvasni, illetve saját adatait szinten BibTeX formában tarolja.

20. Programozható zsebkalkulátort emuláló program

21. Akármilyen strukturált adathalmaz kezelésére szolgáló program. Tarolási forma: XML file. Műveletek: keresés, beszúrás, törlés, listázás, két file egyesítése, rész-listák leválogatása és mentese.

22. Képletszerkesztő
23. WYSIWIG editor (pl. LaTeX-hez)
24. Kapcsolási rajz szerkesztő program
25. Folyamatábra szerkesztő program

F.3 Hivatkozások

[1] Programozzunk C nyelven! ComputerBooks
[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n843.htm
[3] http://wxdsgn.sourceforge.net/
[4] http://www.talula.demon.co.uk/allegro/