Pohl László

C gyakorlat és labor jegyzet

BME 2007

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 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. Ismerkedés a C-vel

1.1 Gyakorlat

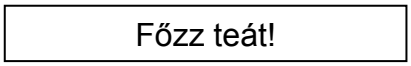
Egy hosszabb példaprogram segítségével megismerkedünk a C programok általános felépítésével.

1.1.1 Algoritmusokról

Minden program meghatározott feladatot végez. Ahhoz, hogy a számítógép azt tegye, amit mi várunk tőle, pontosan meg kell mondanunk, mi a feladat, és azt hogyan hajtsa végre. A számítógép egyszerű gép, nem gondolkodik helyettünk. Ha okosnak tűnik, az a programozó érdeme.

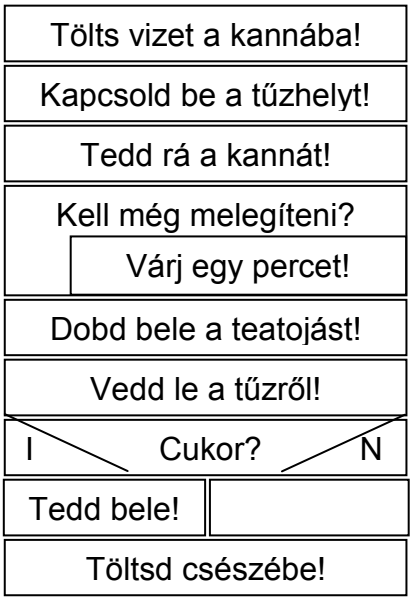
Vegyünk egy nem számítógépes példát! Van egy robotunk, akit rá akarunk venni arra, hogy főzzön nekünk teát. Hogyan programozzuk be?

1. változat:



Ha ezt mondjuk a robotnak, akkor megteszi, amit kérünk tőle? Erre a kérdésre általánosan nem lehet igennel vagy nemmel felelni, ugyanis a robottól függ! Ha van neki teafőzés utasítása (gombja), akkor elég ennyi, ha nincs, akkor nem. Ha van, akkor nem kell mást tennünk, ha nincs, akkor készítsünk egy újabb változatot!

2. változat:



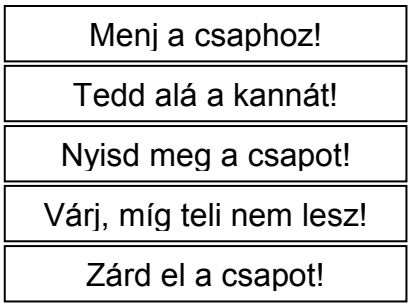
Ez itt a teafőzés algoritmusa stuktogarammal megadva. A „Kell még melegíteni?” egy ciklus feltétele, addig ismétlődik a ciklus magja („Várj egy percet!”), amíg a kérdésre adott válasz IGAZ. A „Cukor?” egy feltételes elágazás.

Ha végignézted az algoritmust, talán felvetődik benned az a megjegyzés: „Micsoda hülyeség, hogy előbb teszi a tojást a vízbe, és csak utána veszi le!”, vagy „Miért az egészbe teszi a cukrot, miért nem csak a csészébe?”. Nos, ebből is látszik, hogy a „Főzz egy teát!” utasítás nagyon sokféleképpen megvalósítható.

És miért pont ezek a műveletek? Ha lenne egy „Forrald fel a vizet!” utasítás, akkor sokkal rövidebben le lehetne írni. A „Tölts vizet a kannába!” pedig felbontható lenne: Menj a csaphoz, tedd alá a kannát, nyisd meg a csapot, várj, míg teli nem lesz, zárd el a csapot! utasításokra. Akkor miért nem így van a fenti stuktogramon? Csak. Mert én így bontottam fel.

És ezt már végre tudja hajtani a robot? Csak akkor, ha létezik „Tölts vizet a kannába!”, „Kapcsold be a tűzhelyt!” stb. utasítás, ha nem, akkor ezeket tovább kell bontani egész addig, mígnem olyan utasításokig jutunk, amelyeket a robot megért.

Nem lesz így túl bonyolult és áttekinthetetlen? Lehet, hogy az lesz. Ilyen esetben célszerű az egyes részfeladatokat különválasztva kifejteni. Ez azt jelenti, hogy ahogy az imént láttuk: ha a „Tölts vizet a kannába!” részt fel kell bontani apróbb darabokra, akkor a feldarabolást külön írjuk le:



Nevezzük ezt `tolts_vizet_a_kannaba()` függvénynek! Ha a későbbiekben azt akarjuk, hogy a robot vizet töltsön a kannába, akkor már tudni fogja, hogyan kell, egyszerűen végre kell hajtania ezt a függvényt. Miért függvény? A C nyelv így nevezi a különválasztott utasításcsoportot.

3. változat:

Lehetne még apróbb részekre bontani a feladatot? Persze, akár atomok szintjére is.

Akkor mikor hagyjuk abba a szétbontást? Addig kell bontani, míg olyan utasításokig nem jutunk, amit a robot ismer.

Mi van akkor, ha bármeddig bontjuk, nem jutunk a végére? Akkor a robot nem tudja megoldani a feladatot.

Mi köze mindennek a C nyelvhez? C-ben is függvényekre bontjuk a programot, hogy áttekinthető maradjon a kód, és a program maga, a fentiekhez hasonlóan, utasítások sorozata, beleértve a ciklust és a feltételes elágazást is.

1.1.2 A program

```
//******************************************************
#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 használja. 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");
        else 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");
        else 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");
        else 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");
        else OK=1;
    }
}
```

1.1.3 Felépítés

1.1.3.1 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 átláthatatlan 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.

1.1.3.2 Header fájlok beszerkesztése

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

Az algoritmusokról szóló részben láttuk, hogy leegyszerűsíti a program elkészítését, ha bizonyos bonyolultabb funkciók (például a `tolts_vizet_a_kannaba()`) rendelkezésre állnak, mert ezeket nem kell folyton összeállítanunk, amikor szükség van rájuk. A C nyelvben rengeteg ilyen előre összeállított függvény áll rendelkezésünkre. Ezek nélkül még a legtöbb profi programozó sem boldogulna, hiszen valójában a legegyszerűbb feladat, egy rövid szöveg kiírása a képernyőre is nagyon bonyolult. Tudni kell hozzá, hogy mely képpontokat kell bekapcsolni a képernyőn, és melyeket kikapcsolni, és ezt a be/kikapcsolást hogyan kell végrehajtani. Szerencsére a C nyelv biztosít olyan függvényeket, melyek az ilyen és hasonló, gyakran használt feladatokat elvégzik helyettünk.

A C fordító a leggyakrabban használt szabványos függvényeket sem ismeri magától, ezért a függvények deklarációit (prototípusait) meg kell adni számukra.

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. 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 mindig áll rendelkezésünkre, azt a gyártó időnként 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

1.1.3.3 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 (1.1.3.1).

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 tört számot ad vissza. Azok a függvények, 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. Ilyen egyébként az algoritmusokról szóló rész összes blokkja.
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");
        else 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");
        else 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");
        else 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.

1.1.3.4 A main() függvény

Bármilyen C nyelvű programot futtatatunk, a végrehajtás a main() nevű függvénnyel kezdődik, bárhol is van a main() függvény a programon belül. Ez azt jelenti, hogy minden programnak kell legyen egy (és csak egy) main() függvénye. Az összes többi függvényt a main() hívja meg közvetlenül vagy közvetve.

Mint minden függvénynek, így a main()-nek is van visszatérési típusa, a main() esetében azonban kétféle is van. Az eredeti C nyelvben int a visszatérési típus, C++-ban viszont void. A mai fordítók mindkét változatot elfogadják. Arra figyeljünk, hogy amennyiben az int típust választjuk, meg kell adnunk return-t is, pl.: return 0;, míg voidnál return;, de el is lehet hagyni.

A példaprogramban int a visszatérési típus. Lássuk a kódot!

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

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

Először létrehozunk (definiálunk) egy valasztott_ertek nevű, egész értékkészletű változót, és 0 kezdőértéket adunk neki. Ha nem adunk kezdőértéket, akkor a változó értéke nem meghatározott, bármi lehet! Ez azért lényeges, mert két sorral lejjebb a

while(valasztott_ertek!=10) kód szerepel. A „while” C kulcsszó, az egyik ciklustípus utasítása. A while a sor végén kezdődő {-tól kezdve a hozzá tartozó }-ig (ami a while-lal egyvonalban található lejjebb, az áttekinthetőség érdekében) lévő programrészt addig futtatja újra és újra, amíg a while utáni () közötti kifejezés értéke igaz. Jelen esetben, amíg a valasztott_ertek nem egyenlő tízzel.

Ha az „int valasztott_ertek=0;” sor végéről lemaradt volna, hogy „=0”, akkor a program működése kiszámíthatatlanná válna, ugyanis elképzelhető, hogy véletlenül épp 10 a valasztott_ertek tartalma kezdetben (hiszen bármi lehet), és ekkor a while feltétele eleve hamis, tehát a ciklus egyszer sem fut le. Ha viszont nem 10, akkor legalább egyszer lefut. Az ilyen bizonytalan működés megengedhetetlen, ZH-n, vizsgán halálfejes hibának számít!

Bármekkora egész számot tehetek egy int típusú változóba? Nem, csak akkorát, amekkora belefér! Mekkora fér bele? Ez nincs pontosan definiálva a C-ben. A későbbiekben látni fogjuk, hogyan deríthető ki, hogy az aktuális C fordító esetén mi a helyzet. Elöljáróban annyit, hogy 16 bites fordító esetén (pl. BorlandC) -32768 és +32767 közötti egész számokat tárolhatunk benne általában, 32 és 64 bites fordító esetén (VisualC++. DevC) pedig -2147483648 és +2147483647 közöttit. Ha ennél nagyobb számokkal van dolgunk, más típust kell választanunk.

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.)

```
                // 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 egy számot. (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 minden egyes 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.

1.1.3.5 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, 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");
        else 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).

Ha sikerült beolvasni a szam-ot, akkor az if feltétele hamis, tehát a printf nem hajtódik végre, hanem a következő sorra lépünk, az else ágra, ami akkor hajtódik végre, ha az if feltétele hamis. 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");
        else 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");
        else 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.

1.2 Labor

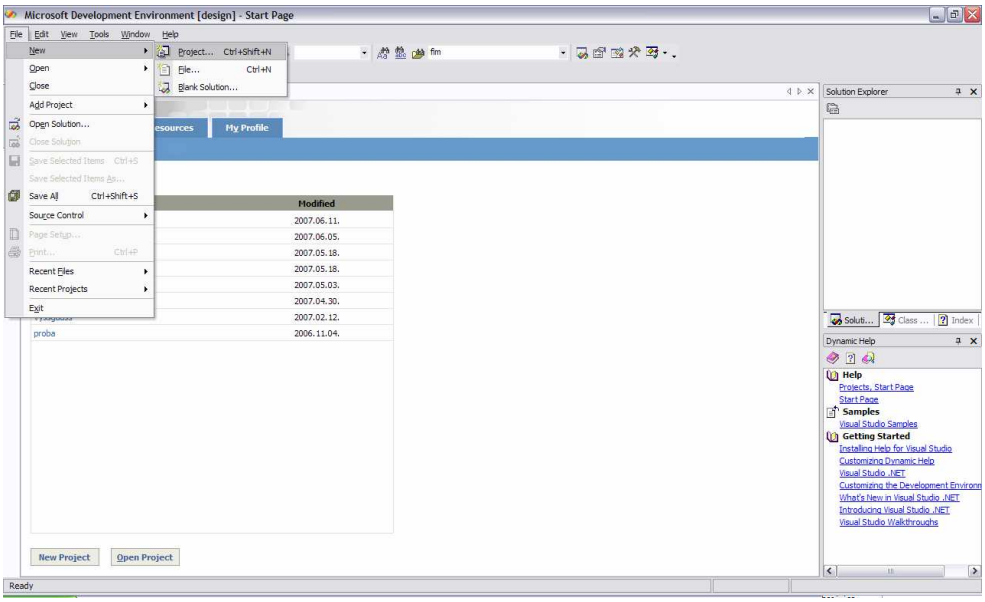
1.2.1. Integrált fejlesztőkörnyezet

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.

Három fejlesztőkörnyezettel ismerkedünk meg röviden: a Microsoft Visual C++ .NET 2003-as verziójával, mely a HSzK-ban is megtalálható, a MS VC++ .NET 2005 Express Editionnel, mely ingyenesen letölthető a MS honlapjáról, és a vele készült programokat akár el is adhatjuk, viszont sok funkciót kihagytak belőle, ami a kereskedelmi változatokban megvan, valamint a DevC++-szal.

1.2.2 Program létrehozása

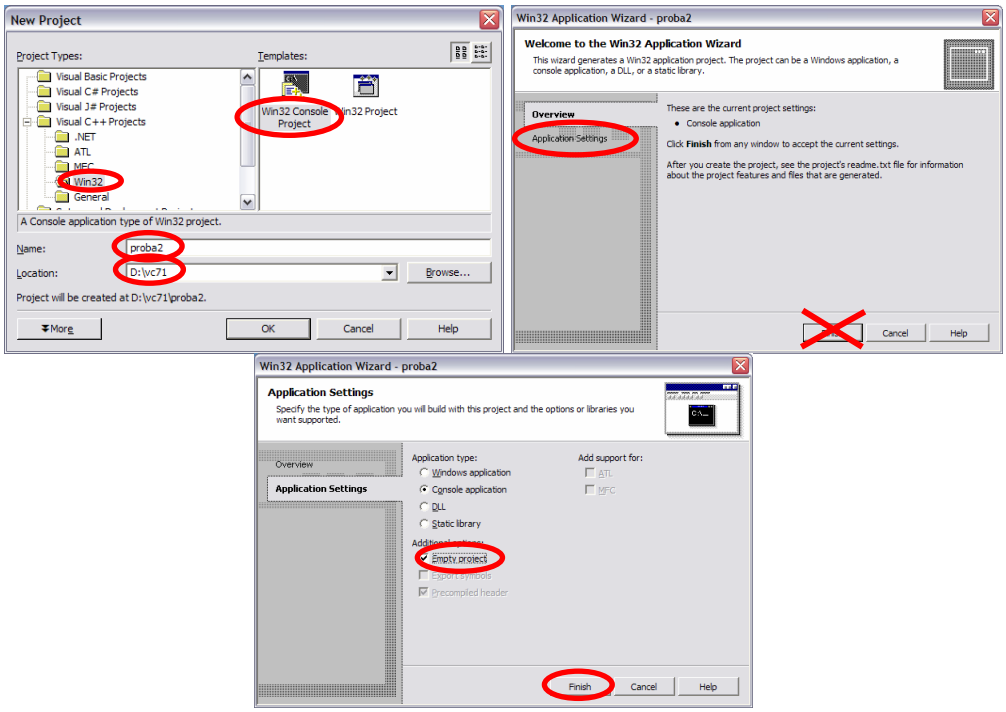
VC++ 2003-at elindítva a következő ablak fogad



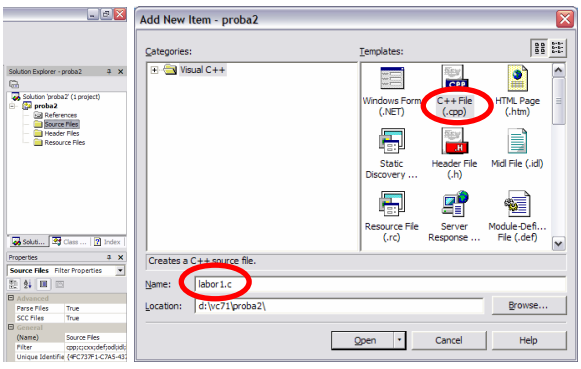
Létre kell hozni a program projektjét. Ehhez kattintsunk a File menü New/Project… parancsára!

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 (c vagy cpp kiterjesztéssel! tehát pl. valami.c), csukjuk be, majd a következőkben leírtak szerint hozzuk létre a projektet. Ha ez megvan, a jobb oldali Solution Explorerben a Source Files mappára kattintsunk jobb egérgombbal! Ezután Add/Add Existing Item => válasszuk ki az elmentett c vagy cpp fájlt!

A következő dialógusablakokat kell helyesen beállítani. Figyelem, könnyű elrontani!

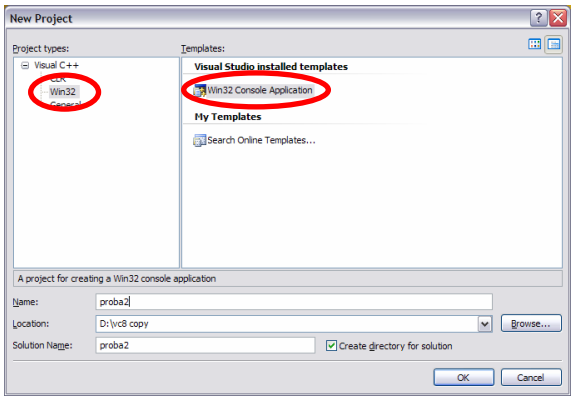


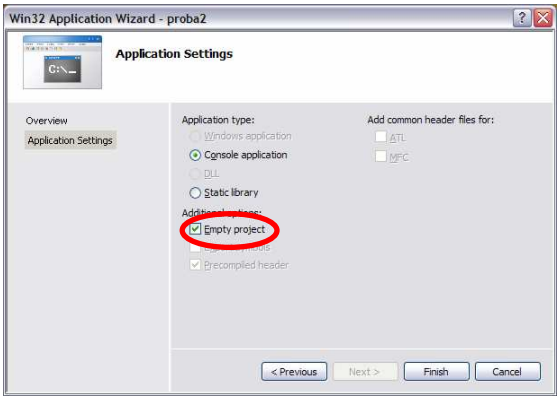
Figyeljünk, hogy Win32 Console Projectet indítsunk, és a második dialógusablakban ne a Finishre kattintsunk, hanem az Application Settings fülre! Ha jól csináltuk, akkor az ábrához hasonlóan a jobb oldalt található Solution Explorer mindegyik mappája üres lesz. Most jobb klikk a Solution Explorer Source Files mappáján => Add New Item => Válasszuk a C++ File-t, és adjunk neki nevet. Ha c kiterjesztést írunk (valami.c), akkor a fordító nem C++ fájlnak fogja kezelni.



A megjelenő szövegszerkesztő ablakban megkezdhetjük a gépelést.

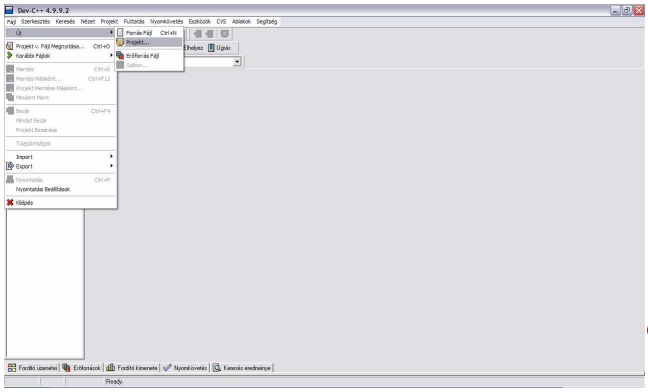
VC++ 2005-ben hasonlóképp a File menü New/Project… beállítását választjuk. Az ablakok kicsit mások.

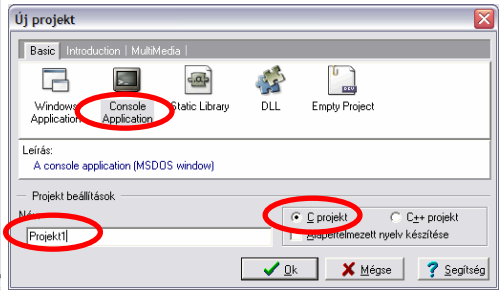




A Solution Explorer bal oldalon található.

DevC++





Itt is Fájl/Új/Project…, majd Console Application. A DevC++ nem üres szövegszerkesztőt ad, hanem a következő keretprogramot helyezi bele:

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

int main(int argc, char *argv[])
{

  system("PAUSE");
  return 0;
}
```

Ezt nyugodtan kitörölhetjük, és beírhatjuk a saját programunkat. A main fejléce különbözik a korábban látottaktól, ne törődjünk vele, a félév második felében magyarázatot fogunk rá kapni. A system függvény az stdlib.h fejléccel szerkesztődik be, és a paraméterként megadott programot futtatja. A PAUSE egy olyan program, amely kiírja, hogy „A folytatáshoz nyomjon meg egy billentyűt…”, és vár a billentyű lenyomására. Ennek a sornak az az értelme, hogy amikor a program véget ért, ne csukódjon be azonnal a konzol ablak, hanem meg lehessen nézni a végeredményt. A PAUSE csak Windowsban működik, UNIX-on nem!

1.2.3 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.

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

- Compile: lefordítja a .cpp fájlt, obj. fájlt hoz belőle létre.
- Build Solution/Build *proba2*: lefordítja azokat a forrásfájlokat, melyek módosultak a legutóbbi fordítás óta, majd a linker létrehozza az .exe-t. Egy Solutionön belül több program/library is lehet, a Build Solution az összeset lefordítja, a Build proba2 csak a kiválasztott proba2 projektet. Mi kizárólag egyprogramos Solutiont használunk, tehát a fentiek közül bármelyiket használhatjuk.
- Rebuild Solution/Rebuild *proba2*: 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 (ez utóbbi segítségével jelentősen felgyorsíthatjuk a programunkat, mivel kihasználja az MMX, SSE, SSE2, SSE3 utasításkészleteket, és további jó néhány optimalizációs eljárást is tartalmaz, melyeket a Microsoft fordítója nem). Az Intel fordítójának időlimites, amúgy teljes értékű próbaváltozata regisztráció után letölthető a vállalat honlapjáról.

Amennyiben nem írjuk át a beállításokban, a fordítás a programot tartalmazó mappa alkönyvtárában jön létre. Az alkönyvtár neve a fordítás módjától függ. A felső eszköztáron választhatunk Debug és Release közül, de magunk is létrehozhatunk fordítási profilt. Debug esetén ki vannak kapcsolva az optimalizációk, és, Release esetén kisebb és gyorsabb programot kapunk.

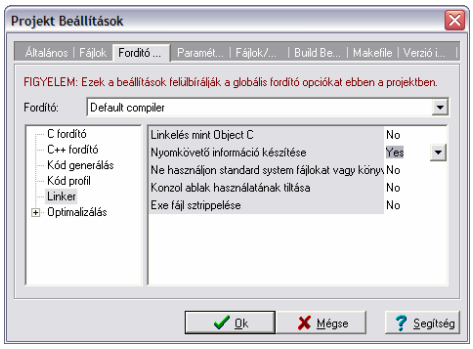
- Debug: minden optimalizáció ki van kapcsolva, a kódba kerülnek olyan részek, melyek lehetővé teszik a hibakeresést (töréspontok beépítése, soronkénti futtatás stb.). Emiatt ez így kapott exe nagy és lassú lesz.
- Release: a véglegesnek szánt változat, optimalizációval és a debuggolást segítő kódok nélkül: release állásban nem tudunk debuggolni, csak ha megváltoztatjuk a konfigurációt, de akkor már inkább a debug módot használjuk.

Ha bepillantunk a Debug és Release mappákba fordítás után, az .obj és az .exe fájlokon kívül számos egyebet is találunk itt. Ezekre a fájlokra a később nem lesz szükségünk, tehát nyugodtan törölhetők. Ha újrafordítjuk a programot, ismét létrejönnek.

A konfigurációkhoz tartozó beállításokat a Project menü Properties pontjában állíthatjuk be, de ugyanez a dialógusablak előhívható a Solution Explorerben a program nevén jobb egérgombot nyomva, és a Propertiest választva (Figyelem! Ne a Solutionön, hanem a program nevén kattintsunk, különben más Properties ablak jön elő!). Itt teljesen átszabhatjuk a

beállításokat, akár olyannyira, hogy Release üzemmódban legyen olyan, mint alapértelmezés szerint Debug módban, és fordítva

DevC++-ban a fordítás a futtatás menü megfelelő pontjaival érhető el. A hibakereső funkciók használatához a Projekt menü Projekt Beállítások pontját választva a dialógusablak Fordító… fülén kattintva válasszuk a Linkert, és a „Nyomkövető információ készítése tulajdonságot állítsuk Yes-re.

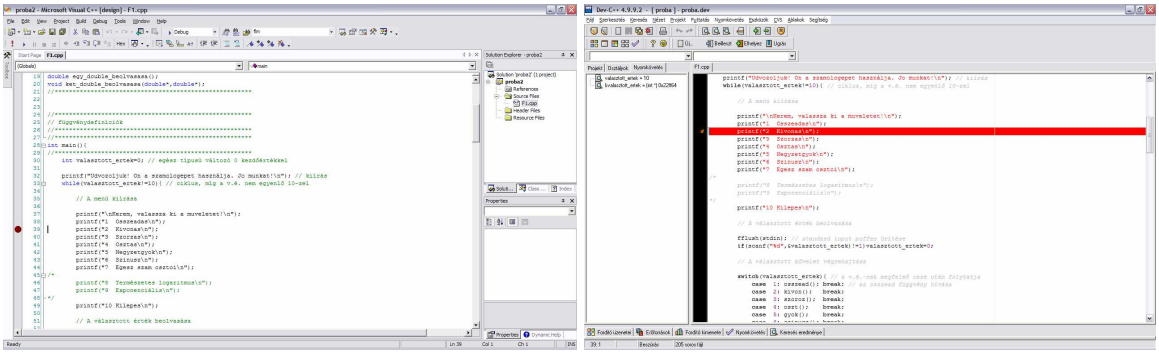


Amennyiben a programunk szintaktikai hibát tartalmazott, tehát valamit rosszul írtunk, esetleg lefelejtettünk egy zárójelet vagy pontosvesszőt, az erre vonatkozó hibaüzenetek a képernyő alján jelennek meg. A hibaüzenetre kattintva a kurzor arra a sorra ugrik, ahol a hibát elkövettük a fordító szerint, de előfordulhat, hogy a hiba az előző sorban volt, pl. ott hagytuk le a pontosvesszőt. **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.

Helyezzünk töréspontot a programba! A töréspont azt jelenti, hogy a program fut normálisan a töréspontig, és ott megáll. Ekkor meg tudjuk nézni a változók aktuális értékét, és soronként tovább tudjuk futtatni a programot (vagy normálisan is tovább futtathatjuk). Töréspontot legegyszerűbben úgy helyezhetünk a programba, ha a programkód melletti szürke (VC) ill. fekete (DevC) sávon kattintunk bal egérgombbal.



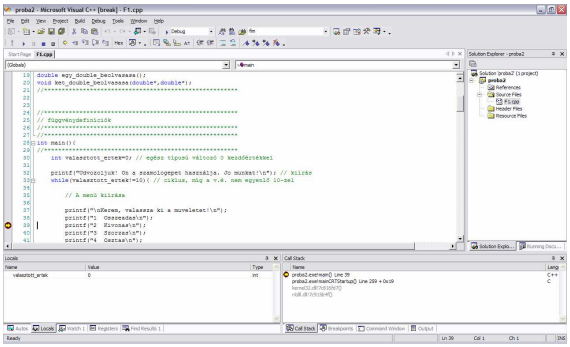
A töréspontot piros pötty, illetve piros sáv jelzi.

- VC++-ban a Debug menü Start pontját válasszuk, a program futása az első töréspontnál fog megállni. Ha nincs töréspont, úgy fut, mintha nem debug üzemmódban lennénk. (Ha nem akarunk hibát keresni, akkor célszerű inkább a Start Without Debugging menüpontot választani, mert ezt használva a program ablaka nem zárul be a futás végén, megnézhetjük a kiírt információkat.)
- DevC++-ban a Nyomkövetés menü Nyomkövetés pontjával kell indítani a programot, ha a Futtatás menü Futtatás pontját választjuk, akkor nem fog megállni a töréspontnál.

Visual C++ esetén a töréspontnál azt is megadhatjuk, hogy ne mindig álljon meg ott a program, hanem csak valahanyadik odaérés alkalmával, vagy pedig valamiféle feltétel teljesülése eseté. Ezek beállíthatóak a töréspontra jobb egérgombbal kattintva, a Breakpoint Properties… menüpontot választva.

Mindhárom fordító esetén lehetőség van a változók értékének nyomkövetésére. DevC++-nál a bal oldali eszköztár Nyomkövetés pontjára kattintva láthatjuk a változókat. Jobbklikk => Megfigyelés Hozzáadása módon bővíthetjük a listát (fenti ábra).

Visual C++-nál a hibakeresés megkezdése után tűnnek fel alul a debug ablakok. Ezek közül az Autos a rendszer által önkényesen kiválasztott változókat mutatja, a Locals az adott függvényben definiált összes változót, a Watch1 pedig az általunk megadott változókat. Ezen kívül, ha az egeret valamelyik változó fölé visszük, egy buborékban megjelenik a változó értéke. A jobb alsó ablak Call Stack füle még nagyon hasznos számunkra, mert ennek segítségével tudhatjuk meg, mely függvény hívta meg az aktuális függvényt, azt melyik hívta, stb.



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

Töréspont helyett a Run to Cursor (Futtatás a Kurzorig) parancsot is használhatjuk adott helyen történő megállásra.

Ha megállta a program, többféleképpen is továbbmehetünk. A soronkénti léptetésre két lehetőség is van:

- Step Over (Léptetés): végrehajt egy sornyi programot. Ha a sorban függvényhívás történik, a függvényt is végrehajtja.
- Step Into (Léptetés(beugrással)): elkezd végrehajtani egy sor programot. Ha a sorban függvényhívás történik, beugrik a függvénybe, és a következő Step Over vagy Step Into utasításokkal a függvényen belül lépegethetünk.
- Step Out (csak VC++): végrehajtja az adott függvényt, és visszaugrik a hívás helyére

A programok szállítása:

Visual C++ esetén 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 a korábban 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.

DevC++ esetén a projektek sokkal kisebb helyet foglalnak, mint Visual C++ esetén, így ott nem gond, ha mindent viszünk. Természetesen elég a .h és .c(pp) fájlokat vinni, ha szűkös a hely.

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 Elmélet

Egy bonyolultabb program részeiként ismerkedhetünk meg a fejezetben tárgyalt elmélettel.

```
//*******************************************************
#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 megfelelő 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");
}
```

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\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);
```

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: 1.2345×10^6=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].

A számítási pontatlanság ellenőrzésére egy újabb példa:

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

//*******************************************************
void main(){
//*******************************************************
        float a,b,c,x1=1.0,x2=1.0,d,y1,y2;
        int i;
        for(i=0;i<20;i++,x2*=10.0){
                a=1.0;
                b=-(x1+x2);
                c=x1*x2;
                d=b*b-4*a*c;
                y1=(-b+sqrt(d))/(2*a);
                y2=(-b-sqrt(d))/(2*a);
                printf("x1=%g\nx2=%g\ny1=%g\ny2=%g\n\n",x1,x2,y1,y2);
        }
}
```

Ez a program az $(x-1)(x-10^n)=0$ egyenlet gyökeit számolja ki. Előbb $ax^2+bx+c=0$, azaz $x^2-(1+10^n)+10^n=0$ egyenletté alakítja, majd ennek gyökeit veszi az $y_{1/2}=(-b\pm\sqrt{b^2-4ac})/2a$ képlet segítségével. Ha jól számolna, akkor a gyökök 1 és 10^n volnának, de nem ezt tapasztaljuk. n=4-nél már láthatóvá válik egy kis hiba, 1 helyett 1.00003 a gyök, és a hiba folyamatosan nő: n=8-nál már 2,00294-et kapunk 1 helyett, n=19-nél pedig 7.01818e+010-et. A 10^n gyök mindig jó. Ha a float-ot double-re cseréljük, javul a pontosság, de a hiba továbbra is fennáll: n=16-ig jó, de n=17-nél 1 helyett hirtelen 0-t kapunk, és az eredmény ennyi is marad.

A hiba oka a -b-sqrt(d) kivonás. Itt ugyanis két nagy számot vonunk ki egymásból, melyek között kicsi a különbség.

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”):

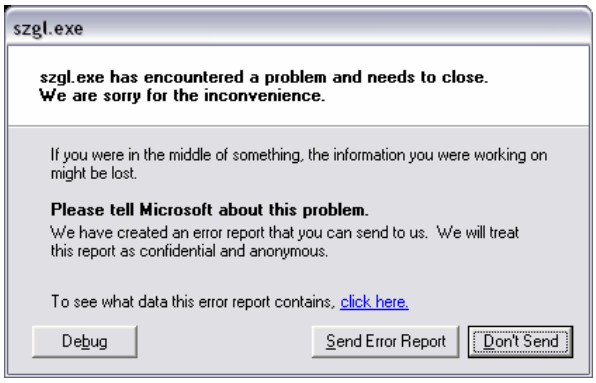
’J’	’o’	’ ’	’m’	’u’	’n’	’k’	’a’	’t’	’!’	’\n’	’\0’

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]);
```

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 Elmélet

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. Enum és az állapotgép

4.1 Példák

4.1.1 Mi az az állapotgép?

Előfordulnak olyan feladatok, amikor nagyobb mennyiségű adatot kell feldolgoznunk, de nincs szükség az adatok tárolására, viszont a korábban beérkezett adatok befolyásolják, hogy az adott lépésben mit kell tennünk az aktuális adattal.

Ilyen például, amikor egy ismeretlen hosszúságú szöveg érkezik hozzák, és ebből a szövegből valamit ki kell szűrnünk, vagy valamit ki kell cserélnünk másvalamire. A fejezet későbbi részeiben több konkrét példát is látni fogunk erre.

A szövegfeldolgozásnál is jobban igényli az állapotgépes megvalósítást sok valós idejű algoritmus, például egy mérőműszert vezérlő szoftver. Például az a feladat, hogy a szoftver állítsa be a termosztát hőmérsékletét, várja meg, míg stabilizálódik a hőmérséklet, adjon a mérendő eszközre egységugrás jelet, és másodpercenként n darab mérést végezve rögzítse a választ mindaddig, míg a kimeneten be nem áll a stabil állapot, ezután ismételje meg ugyanezt más hőmérsékleten.

A fenti esetben két bizonytalan tényező is van: mennyi idő alatt áll be a hőmérséklet, és mennyi idő alatt áll be a válasz?

A következő állapotokat vehetjük fel:

A – Kezdeti állapot, parancs kiadása új hőmérséklet beállítására (csak egy lépésnyit vagyunk ebben az állapotban)

B – Figyeljük a hőmérsékletet, míg stabil nem lesz. Mindaddig ebben az állapotban maradunk, amíg a hőmérséklet változik. A stabilitást például úgy definiáljuk, hogy a legutolsó száz mérés eredményét eltároljuk, és ha nincs 0,1 Celsiusnál nagyobb eltérés a legalacsonyabb és legmagasabb hőmérséklet között, akkor stabilnak tekintjük

C – Ráadja az egységugrás gerjesztést (csak egy lépésnyit vagyunk ebben az állapotban)

D – Addig maradunk ebben az állapotban, míg a kimeneti jel nem stabilizálódik, közben fájlban rögzítjük a mérési eredményeket.

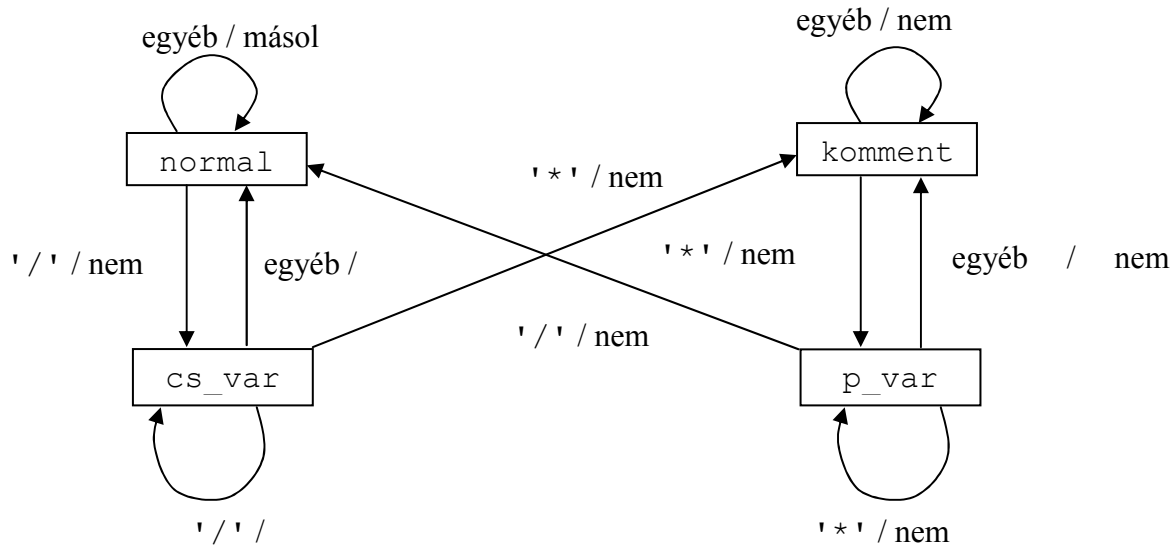
A fenti állapotgépet egy ütemező hívja mondjuk másodpercenként tízszer, vagy amilyen sűrűn akarjuk mérni az értékeket. Több hőmérséklet esetén az egész állapotgép egy ciklus belsejében lesz, ami minden menetben más beállítandó hőmérsékletet és mérési eredményeket tároló fájlnevet biztosít.

A továbbiakban ennél egyszerűbb példákat vizsgálunk.

4.1.2 Poppe András állapotgépes példája: megjegyzések szűrése

Feladat: a szabványos bemeneten érkező szöveget úgy továbbítsuk a szabványos kimenetre, hogy eltávolítjuk belőle a /**/ stílusú megjegyzéseket.

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;
                }
        }
}
```

4.1.3 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);
                }
        }
    }
```

4.2 Feladatok

1. Készítsen szabványos ANSI C programot, amely bekér a felhasználótól egy 1 és 6 közötti egész értéket, majd a szabványos bemenet tartalmát úgy írja át a szabványos kimenetre, hogy a megadott egész értéknél rövidebb szavakat a szó hosszának megfelelő csillag (*) sorozattal helyettesíti. Szavaknak tekintjük a betűkből álló sorozatokat. Szót határol minden nem betű.
2. Írjon C programot, mely a standard bemenetről érkező szövegben minden kisbetűt nagybetűre cserél, az eredményt a standard kimenetre (képernyő) írja. A []-ek között álló szöveget változatlan formában írja ki (itt tehát nem alakítja a kisbetűket naggyá), a szöveg végét EOF jelzi. A zárójelek egymásba ágyazhatók, tehát pl.: be: ”abc de14 x[un1[z3]k2]rs5a” => ki: ”ABC DE14 X[un1[z3]k2]RS5A”
3. Írjon C programot, mely a standard bemenetről érkező szöveget úgy írja a standard kimenetre, hogy a magánhangzóval kezdődő szavakban a magánhangzókat cseréli nagybetűre, a mássalhangzóval kezdődő szavakban pedig a mássalhangzókat cseréli nagybetűre.
4. Írjon C programot, mely a szabványos bemenetről érkező C kódot úgy írja a szabványos kimenetre, hogy a stringeket üres stringekkel helyettesíti. Például: `printf(”Hello\n”);` => `printf(””);`

a. A C kódban nincs megjegyzés, és nincs olyan string, mely \”-lel megadott idézőjelet tartalmaz.
b. A C kódban van /**/ stílusú megjegyzés, a megjegyzésben szereplő stringeket hagyja eredeti formában!
c. A C kódban /**/ és // stílusú megjegyzés is van, , a megjegyzésben szereplő stringeket hagyja eredeti formában!
d. A C kódban nincs megjegyzés, de van olyan string, mely \”-lel megadott idézőjelet tartalmaz. Pl.: `printf(”Hello \”sárkány\”\n”);` => `printf(””);`
e. A C kódra vonatkozóan semmiféle korlátozással nem élünk.

5. Többdimenziós tömbök, pointerek

5.1 Elmélet

Ezúttal is több kisebb programot fogunk megnézni.

5.1.1 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.

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

5.1.2 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\n",i);
        }


        // Számoljuk meg kézzel, második módszer:

        {
                char *p=s;
                while(*p++!='\0');
                printf("2. A karakterek szama: %d\n",p-s-1);

                // írjuk még rövidebben:

                p=s;
                while(*p++);
                printf("3. A karakterek szama: %d\n",p-s-1);
        }

}


//*********************************************************
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, karakterméretben. 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 mutató pointert (egy pointerből nem lehet eldönteni, hogy az egy darab értékre mutat, vagy egy tömbre, mert az egy darab érték olyan, mint egy egyelemű tömb). 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\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 '\0' is benne van a tömbben, tehát kell neki a hely, azonban ezt nem számítjuk hozzá a string hosszához.

```
{       char *p=s;
        while(*p++!='\0');
        printf("2. A karakterek szama: %d\n",p-s-1);

        // írjuk még rövidebben:

        p=s;
        while(*p++);
        printf("3. A karakterek szama: %d\n",p-s-1);
}
```

Most nézzünk néhány „C-szerűbb" megvalósítást. Ezek használatát mindig alaposan fontoljuk meg, mert nagymértékben ronthatják a kód olvashatóságát, futási sebességük pedig azonos az index operátorossal (ez az **index operátor**: []).

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. A posztinkremens ++ precedenciája nagyobb, mint a *-é, ezért nem kell zárójelbe tenni, ez a kiértékelés sorrendje: *(p++).

Miután megnéztük, a 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 lépni 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).

A végeredményből azért kell még egyet kivonni, mert a p pointer a '\0' utáni memóriacímen áll meg a ciklus lefutását követően a posztinkremens ++ miatt.

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 <u>mindig</u> 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.

5.2 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!

6. Rendezés, keresés

6.1 Elmélet

6.1.1 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.

6.1.2 Qsort

A C rendelkezik egy beépített rendező függvénnyel, a qsorttal. A q a quickre utal. Előbb megnézzük a beépített qsort függvény használatát, majd mi magunk írunk qsort függvényt.

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.

6.1.3 Saját QuickSort

A quicksort algoritmus a következőképpen működik:

1. Keressük meg a tömb középső elemét! Mivel ez nem megy az elemek végigolvasása nélkül, amire nem akarunk időt vesztegetni, válasszuk ehelyett mondjuk a két szélső elemet, és vegyük az átlagukat!
2. Rendezzük át úgy az elemeket, hogy a „középső”-nél kisebbek a tömb aljára kerüljenek, a nagyobbak pedig a tetejére.
3. Vegyük a középsőnél kisebb elemek résztömbjét, meg a nagyobb elemek résztömbjét, és ezeken ismételjük meg ismét az 1-2-3 lépéseket. A folyamat akkor ér véget, ha egy résztömbben max. egy elem marad.

Pl.:

3	9	1	7	5	2	4	6	8

Középső elem: (3+8)/2=5

3	4	1	2	5	7	9	6	8

Alsó blokk középsője: (3+5)/2=4, felső blokk középsője: (7+8)/2=7

3	4	1	2	5	7	6	9	8

A négy részből egy egyelemű, azt tehát nem bántjuk, a másik három középsője: 2, 6, 8

1	2	3	4	5	6	7	8	9

Már csak két blokk van alul, ahol egynél több elemünk van. Itt 1 és 3 lesz a középső elem. A sorrend természetesen nem változik.

A megvalósítás úgy történik, hogy miután kettéválasztottuk az elemeket, a két résztömbre meghívjuk ismét a quicksort függvényt, azaz saját magát. Ha egy függvény magát hívja, azt rekurziónak nevezzük.

```
void qs(double * t,int meret){
        if(meret<2)return;              // ha 1 vagy 0 tömbelem van, kész
        int left=0,right=meret-1;       // a két szélén kezdjük
        double med=(t[right]+t[left])/2;  // a középső elem kiválasztása
        do{
                while(t[left]<med)left++;// átlépjük a középsőnél kisebb elemeket
                while(t[right]>med)right--;// átlépjük a középsőnél nagyobb elemeket
                if(right>=left){ // Ha nem ment el egymás mellett a kép "pointer", felcseréljük az elemeket
                        double temp=t[left];t[left]=t[right];t[right]=temp;
                        left++;
                        right--;
                }
        }while(right>=left);     // Ha elment egymás mellett a két "pointer", kész a szétválogatás
        qs(t,right+1);           // Az alsó résztömbre meghívjuk a quicksortot
        qs(t+left,meret-left);   // A felső résztömbre meghívjuk a quicksortot
}
```

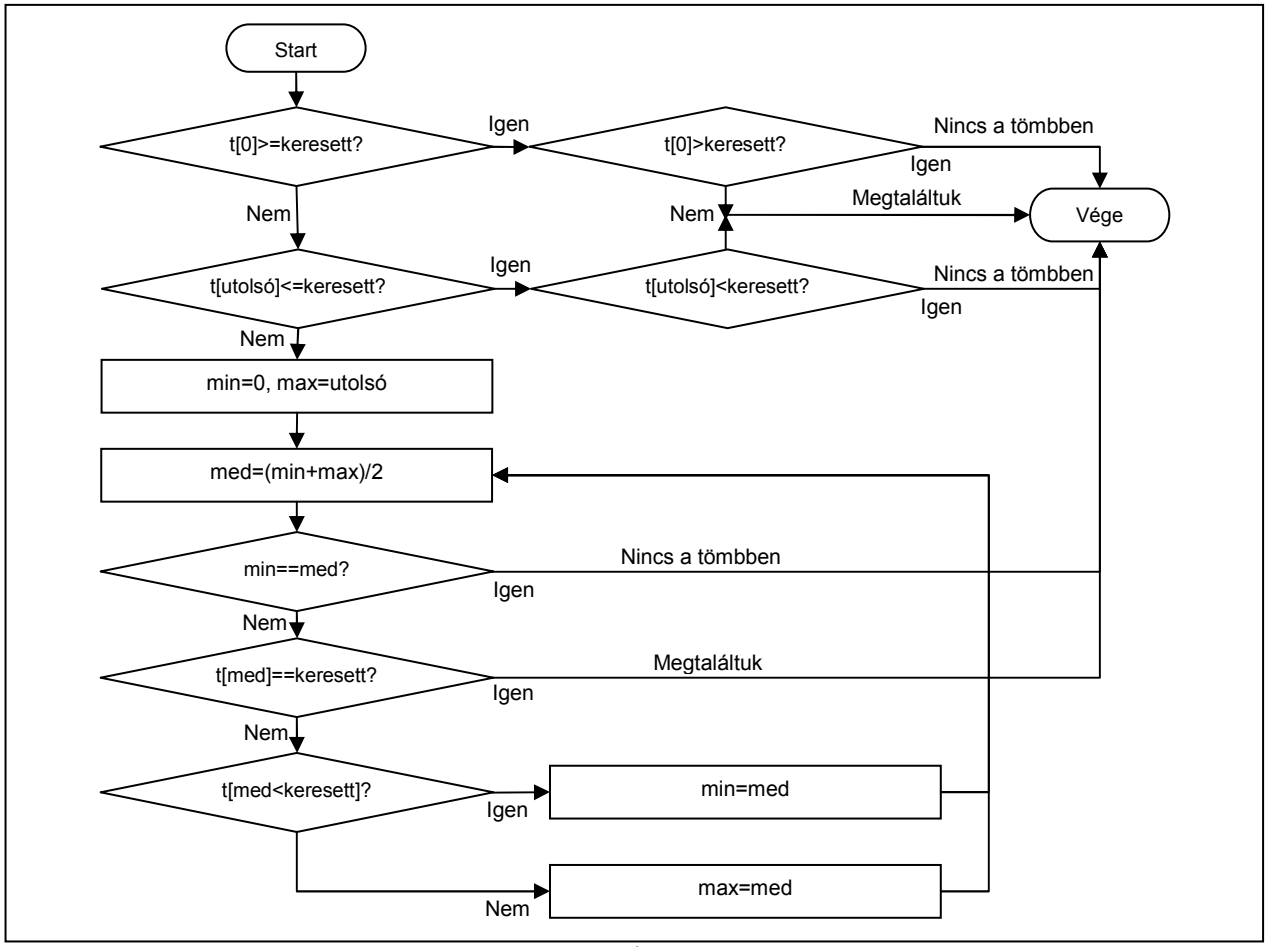
Ha ennél általánosabb megvalósításra van szükség, a „középső” elem lehet a résztömb első eleme is, a `right>=left` helyett pedig összehasonlító függvényt alkalmazhatunk.

A quicksort algoritmus jellegzetessége, hogy láncolt listán is alkalmazható.

6.1.4 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].

6.1.5 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.

6.2 Feladatok

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.

7. Függvénypointerek, rekurzió

7.1 Elmélet

7.1.1 Függvénypointerek

A függvények kódja a memóriában található, csakúgy, mint a változók, tehát meg lehet mondani azt a memóriacímet, ahol az adott függvény kezdődik. A C nyelvben definiálhatók olyan pointerek, melyek függvények címeit tárolják. Ezekből a pointerekből tömböt is szervezhetünk, vagy átadhatjuk függvénynek paraméterként.

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

void main(){
        double (*t[4])(double),d;
        int n;

        t[0]=sin;
        t[1]=cos;
        t[2]=exp;
        t[3]=tan;
        printf("Kerek egy valos szamot: ");
        scanf("%lg",&d);
        printf("Mit szamoljak?\n1. sin\n2. cos\n3. exp\n4. tan\n");
        scanf("%d",&n);
        if(n<1||n>4)return;
        printf("Eredmeny: %g\n",t[n-1](d));
}
```

A fenti példában létrehozunk egy olyan tömböt, melynek elemei double visszatérési értékű, egy double paraméterrel rendelkező függvények címei. Ezután feltöltjük a tömböt négy függvénnyel, melyek a math.h-ban találhatók. Természetesen saját függvényeket is használhatnánk. Az utolsó sorban meghívjuk a tömb n-1-edik elemét.

A következő példa azt mutatja be, hogyan írhatunk függvényt, amely tetszőleges, egyváltozós (valós) függvény integrálját számítja ki téglalap formulával. A `teglalap` függvény `fv` függvény integrálját számítja ki [`a`,`b`] intervallumon, az intervallumot `n` egyenlő részre osztja.

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

double teglalap(double a,double b,int n,double (*fv)(double)){
        double dx=(b-a)/n,sum=0;
        int i;
        for(i=1;i<n;i++)sum+=fv(a+dx*i);
        return (sum+0.5*(fv(a)+fv(b)))*dx;
}


void main(){
        printf("I=%14.14g\n",teglalap(0,1,100,sin));
}
```

A `teglalap` függvény negyedik paramétere tehát egy olyan függvény címe, amelynek mind a paramétere, mind a visszatérési értéke double típusú. A paraméterként átadott függvényt úgy hívjuk meg, mintha egyszerű függvény volna, nem pointer.

A függvénynek átadott függvénypointer használatát korábban láthattuk a `qsort` esetében. A `qsort` paraméterként kapta az összehasonlító függvényt.

7.1.2 Rekurzió

A rekurzióval már találkoztunk korábban, a quicksort algoritmusnál. Két formája van: az önrekurzió, amikor egy függvény önmagát hívja, és a kölcsönös rekurzió, amikor két függvény felváltva hívogatja egymást.

```
#include <stdio.h>

int faktorialis(int n){
        int v;
        if(n<2)return 1;
        v= n*faktorialis(n-1);
        return v;
}

void main(){
        printf("7 faktorialisa=%d\n",faktorialis(7));
}
```

A fenti példában a faktoriálist rekurzív módon számítjuk ki. Az n!=n*(n-1)!. Itt is, mint minden rekurziót tartalmazó programnál arra kell figyelni, hogy a rekurzió egyszer biztosan véget érjen. Jelen esetben ez akkor következik be, ha n értéke 2 alá csökken, ekkor ugyanis a függvény nem fogja magát hívni.

Hogyan működik a fenti kód? Rekurzió esetén ehelyett a következőt kell elképzelnünk:

```
#include <stdio.h>

int faktorialis1(int n1){
        return 1;
}

int faktorialis2(int n2){
        int v2;
        if(n2<2)return 1;
        v2= n2*faktorialis1(n2-1);
        return v2;
}

int faktorialis3(int n3){
        int v3;
        if(n3<2)return 1;
        v3= n3*faktorialis2(n3-1);
        return v3;
}

int faktorialis4(int n4){
        int v4;
        if(n4<2)return 1;
        v4= n4*faktorialis3(n4-1);
        return v4;
}

int faktorialis5(int n5){
        int v5;
        if(n5<2)return 1;
        v5= n5*faktorialis4(n5-1);
        return v5;
}

int faktorialis6(int n6){
        int v6;
        if(n6<2)return 1;
        v6= n6*faktorialis5(n6-1);
        return v6;
}
```

```
int faktorialis7(int n7){
	int v7;
	if(n7<2)return 1;
	v7= n7*faktorialis6(n7-1);
	return v7;
}

void main(){
	printf("7 faktorialisa=%d\n",faktorialis7(7));
}
```

Azaz amikor egy rekurzív függvény meghívja magát, az új meghívásnál új, azonos nevű változók jönnek létre, tehát nem rontódnak el a korábbi változók értékei, de nem is használhatjuk a korábbi változók értékeit! A rekurzió használatának egyik gátja pont az, hogy minden lépésben új változók jönnek létre a verem memóriában, ami túl sok változó, vagy túl nagy hívási mélység esetén betelhet.

7.1.3 Integrálszámítás adaptív finomítással

Az alábbi példa függvénypointert és rekurziót egyaránt tartalmaz.

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

double teglalap(double a,double b,int n,double (*fv)(double)){
	double dx=(b-a)/n,sum=0;
	int i;
	for(i=1;i<n;i++)sum+=fv(a+dx*i);
	return (sum+0.5*(fv(a)+fv(b)))*dx;
}

double finomitas(double a,double b,double hiba,double (*fv)(double),double elozo,int szint){
	double kozep=(a+b)*0.5;
	double bal=teglalap(a,kozep,16,fv);
	double jobb=teglalap(kozep,b,16,fv);
	double uj=bal+jobb;

	if(szint>20)return uj; // ha túl nagy lenne a rekurzió mélysége, nem megyünk tovább
	if(fabs((uj-elozo)/uj)<hiba)return uj; // ha kész
	hiba*=0.5;
	return finomitas(a,kozep,hiba,fv,bal,szint+1)+finomitas(kozep,b,hiba,fv,jobb,szint+1);

}

double integral(double a,double b,double hiba,double (*fv)(double)){
	return finomitas(a,b,hiba,fv,teglalap(a,b,16,fv),1);
}

double pipi(double x){
	return 4.0/(1.0+x*x);
}

void main(){
	printf("I=%14.14g\n",integral(0,1,1e-8,pipi));
}
```

Az `integral` függvény számítja ki `fv` függvény integrálját [a,b] intervallumon. Az integrálásnál használt x irányú felosztás ebben az esetben nem egyenletes, mint a korábbi, téglalap formulát használó esetben, hanem ahol a függvény hullámosabb, ott sűrűbb a felosztás, ahol pedig egyenletesebb, ott ritkább.

Ezt úgy érjük el, hogy első lépésben kiszámítjuk az integrál közelítő összegét kevés (16) pontra a téglalap formulával (még az `integral` függvényben), majd (már a `finomitas` függvényben) vesszük az intervallum alsó és felső felét, és kiszámítjuk azok integrálját is,

ugyancsak 16-16 osztóponttal. A két félre kapott integrál összege egy pontosabb közelítést, ad, mert kétszer annyi osztópontot használtunk.

Megnézzük, hogy az új és az előző integrálközelítő összeg mennyire tér el egymástól. Ha az eltérés a `hiba` változóban magadottnál kisebb, akkor nem számolunk tovább. Ha nagyobb, akkor a két oldalon külön-külön számoltatunk egy pontosabb eredményt. Ha az egyik oldalon a következő lépesben elég pontos eredményt kapunk, a másik oldalon viszont nem, akkor a két oldalon eltérő felbontáshoz fogunk jutni.

A rekurzív hívás tehát megáll,ha elértük a kívánt pontosságot. Előfordulhat, hogy ezt sosem tudjuk elérni (számítási hibák miatt), ezért bevezettünk egy korlátozást: ha 20-nál nagyobb mélységre lenne szükség, akkor is leáll a folyamat.

7.1.4 Buborék rendezés tetszőleges elemeket tartalmazó tömbre

A példát Nagy Gergely készítette. A `bubble` függvény paraméterezése megegyezik a C szabványos függvényei között található `qsort` függvényével.

```
/*
A BUBORÉK RENDEZÉS ÁLTALÁNOS TÖMBÖKRE (BUBBLE()) A QSORT() MINTÁJÁRA,
ÁLTALÁONS TÖMBÖKET KIÍRATÓ FÜGGVÉNY (PRINT_ARRAY(), PRINT_E()),
ALACSONY SZINTŰ MEMÓRIA-KEZELÉS (CHANGE())
*/

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

typedef enum {false, true} bool;

struct a {
    int b;
    double c;
};

int sgn(double a) {
    if (a < 0) return -1;
    else if (a == 0) return 0;
    return 1;
}

int cmp(const void *a, const void *b) {
    return sgn(((struct a*)a)->c - ((struct a*)b)->c);
}

/*
A bubble() által használt belső függvény, amely két (void *) pointer
által mutatott s méretű területen lévő adatokat kicserél (bájtonként).
Ez egy módja annak, hogy egy területhez alacsony szinten, típusoktól függetlenül
hozzáférjünk.
*/
void change(void *a, void *b, unsigned s) {
    unsigned i;
    char temp;

    for (i = 0; i < s; i++) {
        temp = *((char *)(a + i));
        *((char *)(a + i)) = *((char *)(b + i));
        *((char *)(b + i)) = temp;
    }
}

/*
Hívása: bubble(tomb, elemszam, elemmeret, hasonlito_fv);

tomb: a rendezendő tömbre mutató pointer
elemszam: a tömb elemeinek a száma
elemmeret: a tömb egy elemének a mérete (bájtokban)
hasonlito_fv: egy függvény, ami meghatározza két elem viszonyát
```

```
A felhasználónak meg kell írnia a hasonlító függvényt, ami (const void *)
mutatókat kap a két elemre, és a visszatérési értéke:

-1: a < b
 0: a == b
 1: a > b

 ha a híváskor: my_compare(a, b);
*/
void bubble(void *t, unsigned n, unsigned s, int (*cmp)(const void*, const void*)) {
    bool changed;
    unsigned i;

    do {
        changed = false;
        for (i = 0; i < n - 1; i++) {
            if (1 == cmp(t + i * s, t + (i + 1) * s)) {
                change(t + i * s, t + (i + 1) * s, s);
                changed = true;
            }
        }
    } while (changed);
}

void print_e(const void *e) {
    printf("%d\t%lf\n", ((struct a *)e)->b, ((struct a *)e)->c);
}

/*
Hívása: print_array(tomb, elemszam, elemmeret, kiiro_fv)

tomb: a rendezendő tömbre mutató pointer
elemszam: a tömb elemeinek a száma
elemmeret: a tömb egy elemének a mérete (byte-okban)
kiiro_fv: függvény, amely képes a tömb egy elemét kiírni

A felhasználónak meg kell írnia egy függvényt, ami (const void *)
pointert kap a kiírandó elemre, és azt megjeleníti a képernyőn
*/
void print_array(const void *t, int n, int s, void (*pe)(const void *)) {
    int i;
    for (i = 0; i < n; i++) {
        pe(t + i * s);
    }
}

int main(void)
{
        struct a t[] = {{12, 3.4}, {4, 8.7}, {123, -46.12}, {-78, 3.14}};
        unsigned s = sizeof(struct a), n = sizeof(t) / s;

    printf("Tomb rendezese a valos tipusu mezo (c) szerint:\n");
    printf("-----------------------------------------------\n\n");

    printf("Rendezes elott:\n");
    print_array(t, n, s, print_e);

    bubble(t, n, s, cmp);

    printf("\n\n\nRendezes utan:\n");
    print_array(t, n, s, print_e);

        return 0;
}
```

7.2 Feladatok

1. Készítse el a tetszőleges tömböt rendezni képes quicksort függvényt az 1.4. pontban látott buborék algoritmushoz hasonlóan!
2.

*