A C programozási nyelv

Készítette:
Burián Ágnes
főiskolai adjunktus

2005.

BMF
SZGTI

Tartalomjegyzék

A C nyelv kialakulása

A 60-as évektől kezdődik.

BCPL (Martin Richards)
B (Ken Thompson 1970.)
C (Dennis Ritchie)

Ken Thompson PDP – 7 számítógépre új operációs rendszert valósított meg: a UNIX-ot.
Kezdetben assembly nyelvet használt, de mivel az nehézkesnek bizonyult, menet közben kifejlesztette a B nyelvet, amelyen aztán meg is írta a UNIX operációs rendszert.
A C nyelv kialakulása erősen kötődik a UNIX-hoz: ezt az operációs rendszert az első assemblyben és B-ben írt változataitól eltekintve C-ben írták és írják.
A C az első magas szintű programnyelv, amelyen operációs rendszert lehet írni; a UNIX pedig az első operációs rendszer, amelyet magas szintű programnyelven írtak.
Újabban a C nyelv nem kötődik kizárólag a UNIX-hoz. Számos C fordító készült más operációs rendszerekhez.
Szabványosították 1983-tól → ANSI –>1987.

A C általános célú programnyelv: nem kötődik egyetlen speciális alkalmazási területhez sem.

A C középszintű nyelv: egyesíti a magas szintű nyelvek és az assembly nyelv elemeit.

Egy új programnyelv elsajátításának egyetlen módja, ha programokat írunk az adott nyelven.

A C programok szerkezete

A C program egy vagy több függvényből áll. A függvények általában meghatározott feladatot hajtanak végre. A függvények közül egy a main nevű kell legyen! A program végrehajtása a main aktiválásával kezdődik. A legegyszerűbb C program egy üres main függvény:

```
main ( )
{
}
```

A függvényeknek nem szükséges paraméteresnek lenni. A függvény jellemzője a visszatérési értéke. A void (üres) adattípust adjuk meg, ha nincs visszatérési értéke a függvénynek. Ha nem írunk típust, akkor int lesz, mert az int típus az alapértelmezett függvény típus.

```
void main ( )
        {
        }
```

Program, amely szöveget ír a képernyőre:

```
/*első. C */
# include <stdio.h>
void main ( )
{
        printf (″Tanuljuk a C nyelvet! \n”);
}
```

Mivel a C-ben nincsenek input-output utasítások, a szöveg kiírását egy függvény - a printf- meghívásával tudjuk elvégeztetni.
A printf függvény sohasem helyez el újsor karaktert automatikusan:

```
# include <stdio.h>
void main ( )
{
        printf ("Tanuljuk");
        printf ("a C nyelvet! ");
        printf ("\n”);
}
```

\n egyetlen karaktert jelent – escape karakter. A C escape jelsorozatot használ a nehezen előállítható, vagy láthatatlan karakterek jelölésére.
Ilyenek: \n , \t , \b , \", \\ , \f

```
/* a második . C */
# include <stdio.h>
 void main ( )
{
        printf (″Tanuljuk a \n \t C \n nyelvet! \n”);
}
```

Három egész szám összege:

```
/*szumma. C */
# include <stdio.h>
void main ( )
{
        printf ("összeg:2+4+5 = %d \n”, 2+4+5);
}
```

%d formátumelem = konverzió előírás (formátum specifikáció), jelentése decimális egész.

Ugyanez változókkal:

```
# include <stdio.h>
void main ( )
{       int a, b, c;
        int  sum;
        a=1, b=2; c=3;
        sum =a+b+c;
        printf ("összeg:a+b+c = %d \n", sum);
 }
```

A C-ben az utasításokat **;** zárja!
A C különbséget tesz a kis- és a nagybetűk között, az utasításokat és a C fenntartott szavakat kisbetűkkel kell írni!

A C változó típusai

C nyelv négy alapvető változótípust ismer:

char:	karakter – mérete a gép memóriájának alapegysége : 1 byte
int:	egész – 2 byte
float:	lebegőpontos változó – 4 byte
double:	kétszeres pontosságú lebegőpontos változó – 8 byte

Bármely egész jellegű változót elláthatunk az unsigned (előjel nélküli) típus módosító szócskával is. Ez módosítja a változónak az aritmetikai műveletekben játszott szerepét.

A lehetséges típusok:

típusnév	tartomány	méret
char	-128→127 $(-2^7 \rightarrow 2^7-1)$	1 byte
unsigned char	0→255 $(0 \rightarrow 2^8-1)$	1 byte
int	-32768→32767 $(-2^{15} \rightarrow 2^{15}-1)$	2 byte
unsigned int	0→65535 $(0 \rightarrow 2^{16}-1)$	2 byte
long	$-2^{31} \rightarrow 2^{31}-1$	4 byte
unsigned long	$0 \rightarrow 2^{32}-1$	4 byte
float	$10^{-38} \rightarrow 10^{38}$	4 byte
double	$10^{-307} \rightarrow 10^{307}$	8 byte

01111111 127
10000001 -127

void : üres

⇔ short

⇔ unsigned short

A változók neve: az angol ABC betűiből,
számjegyekből és _ karakterből állhat,
de számjeggyel nem kezdődhet.
A szabvány: kisbetű, kivétel a konstansok neve.

Számítsuk ki a kör területét programmal:

```
# include <stdio.h>
void main ( )
{
        float r, area;
        const float PI=3.14159;
        r=2.2;
        area=r*r*PI;
        printf ("Az r = %6.2f sugarú kör területe:%10.2f \n", r, area);
 }
```

A sugár kiírásának formátuma: 6 karakter mezőszélességben 2 tizedesjegy (%6.2f).

A char típusú változó karakterek és egész számok tárolására egyaránt alkalmas!

```
# include <stdio.h>
void main ( )
{
        char kar='A';          // Az A ASCII kódja: 65
        char egesz=48;         // 48: a 0 karakter ASCII kódja
        printf ("Karakterek: %c és %c\n",kar, egesz);   // A és 0 kerül kiírásra
        printf ("Egészek: %d és %d\n",kar, egesz);      // 65 és 48 kerül kiírásra
}
```

A C az aritmetikai műveletek eredményét egyáltalán nem ellenőrzi!
Nincs túlcsordulás kezelés!

A bekeretezett jelenség int típus esetén következik be:

```
-32768-1 = 32767
 32767+1=-32768
```

Előny: gyorsabbak a műveletek
Veszély: nem vesszük észre, ha túlcsordulás miatt helytelen az eredmény!

```
# include <stdio.h>
void main ( )
{
  unsigned int ua, ub, us;
  int  a, b, s;
  a=10000; b=10000;
  s = a+b;                    /* helyes eredmény */
  printf("a=%6d, b=%6d, s = %6d\n", a, b, s);
  a=20000; b=20000;
  s =a+b;                     /* túlcsordulás lesz */
  printf("a=%6d, b=%6d, s=%6d\n", a, b, s);

  ua=20000; ub=20000;
  us =ua+ub;                  /* ez jó lesz */
  printf("ua=%6u, ub=%6u, us=%6u\n", ua, ub, us);
  ua=40000; ub=40000;     /* ez is túlcsordul */
  printf("ua=%6u, ub=%6u, us=%6u\n", ua, ub, us);

  s=100000; us=100000;    // ez is érdekes
  printf("s=%6d, us=%6u, \n", s,  us);
}
```

Input - output a C-ben

Mivel a C-ben nincs I/O utasítás, ezért az input (beolvasás) és az output (kiírás) műveleteket könyvtári függvények használatával tudjuk elvégezni.

Formázott kiírás és beolvasás

Az eddigi példákban gyakran előfordult a printf függvény, ami formázott kiírást végez a standard outputra. (A standard output alapértelmezésben a képernyő.)

A kiírandó adatok formátum előírásánál használható opciók:

printf( )

%[-] [n[.m]] [l] x — formátum
long v. double
balra igazít tizedes jegyek száma
minimális mezőszélesség

A jobbra igazítás az alapértelmezett!

A formátum fajtái:

c	karakter	
d	előjeles egész	
u	előjelnélküli egész	csak előjel nélküli egészek esetén (unsigned)
o	oktális	
x	hexadecimális	
f	tizedestört	
e	exponenciális	
s	sztring	

ESC karakterek:

\n	újsor
\f	lapdobás
\r	kocsivissza
\t	vízszintes tabulálás
\v	függőleges tabulálás
\b	backspace
\”	”
\’	’
\\	\
\a	alarm (csengő)
\0	a 0 kódú karakter, ami nincs

A printf párja a scanf függvény, ami a standard inputról végez formázott bevitelt. (A standard intput alapértelmezésben a klaviatúra.)

scanf ( )

```
int i; float x, char name [50];
scanf (”%d  %f   %s ”, &i, &x, name);          // & címe operátor
```

A scanf függvénynél ugyanazok a formátum előírások szerepelhetnek, mint a printf-nél.
A scanf függvénynek a változó címét kell átadnunk, ahova tennie kell a beolvasott értéket a konverzió után a %d és a %f formátummal történő beolvasás esetén.
A három karakterláncot tetszőleges számú szóköz, tabulátor és újsor választhatja el!
A scanf pufferelt bevitelt végez!

Karakteres beolvasás és kiírás

getchar( ) -- egy karakter pufferelt bevitele a standard inputról,
putchar( ) -- egy karakter kivitele a standard outputra.
standard fájlok: logikai fájlok, logikai csatornák
stdin: standard input
stdout: standard output

A programok nem fizikai eszközről, hanem egy logikai fájlból olvasnak, ill. egy logikai fájlba írnak.
Rugalmasság, átirányíthatók: < stdin
> stdout
Egyetlen megszorítás: szekvenciális legyen az in ill. out!

A getchar() és a putchar() függvények használata:

```
char c:
        c = getchar( );
        putchar( c );
```

A C operátorai

Aritmetikai operátorok:

+ összeadás
- kivonás
* szorzás
/ osztás
% modulo = maradékképzés
++ inkrementálás } csak egész jellegű változók esetén
-- dekrementálás }

a = a+1 ⇔ a++ vagy ++a
a = a -1 ⇔ a-- vagy -- a
A műveleti sorrend miatt van különbség a kétféle írásmód között:

```
a = 10;          a = 10;
b = ++a;         b = a++;
b = 11 lesz      b =10 lesz
a  =11 lesz      a =11 lesz
```

Relációs operátorok

<	kisebb
< =	kisebb v. egyenlő
>	nagyobb
> =	nagyobb v. egyenlő
! =	nem egyenlő
= =	egyenlő

Logikai operátorok

&&	AND
| |	OR
!	NOT

Utasítások összekapcsolása – vessző operátor

```
int a, b, c;
      a = 2, b = 10, c=a+b;
```

Bitműveletek

&	bitenkénti	„és”
|	bitenkénti	„vagy” (megengedő)
^	bitenkénti	„kizáró vagy”
~	1-es komplemens	
!	logikai „nem”	$0 \longleftrightarrow \neq 0$
<<	balra léptetés	
>>	jobbra léptetés	

Hozzárendelési operátorok

A szokásos kétváltozós operátorok összekapcsolhatók az értékadást jelentő = jellel!
x - = 10; (x = x - 10)

A kifejezés:

változókból, konstansokból és aritmetikai operátorokból álló képlet.
Pl: double r, t;
t = r∗r∗3.14159;
A program kiszámítja az értékét.
Tágabb értelemben kifejezés az értékadás is.
változó1 = változó2 = kifejezés;
Keverhetjük a különböző típusú változókat és konstansokat egy kifejezésen belül.

A különböző típusok keveredésének 2 fajtája

- *A C nyelv mindig a baloldal típusára konvertálja a jobboldal típusát!*
- *De a jobboldalon szereplő kifejezés is tartalmazhat különböző típusokat. A kiértékelés során bármely elem a legerősebb típusra konvertálódik!*

A különböző típusok keveredésekor automatikus típus konverzió történik!
A C bármely lebegőpontos műveletet kétszeres pontossággal végez!!!

Balérték

A balérték, amely előfordulhat egy értékátadás bal oldalán, mint célterület.
Az egyváltozós aritmetikai operátorokat is csak balértékre alkalmazhatjuk.
Tehát helytelen a következő:
a+b=c; vagy (a+b)++;

A C nyelv utasítás készlete

Az elágazás utasítások:

if
if - else
switch - case

Az if utasítás

Egyágú elágazás, a program feltételes elágaztatására szolgál.

```
if  (kifejezés)    utasítás;
```
vagy:
```
if  (kifejezés)
{
   utasítások
   }
```

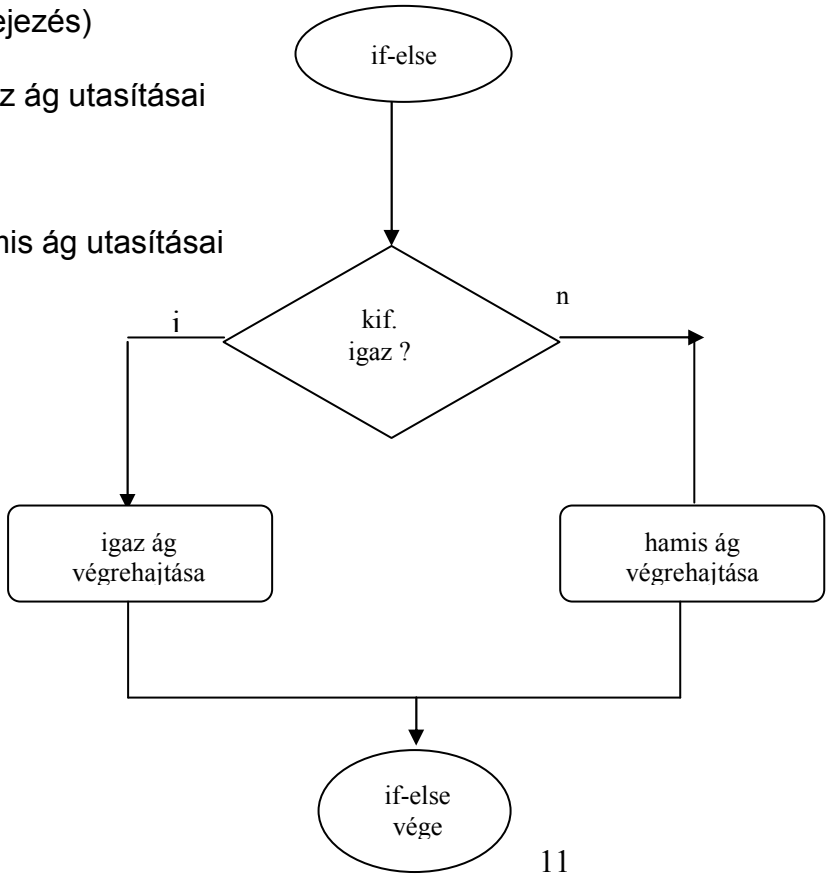
Pl.: **a** ne legyen kisebb, mint **b**! Ha kisebb, felcseréljük a tartalmukat.

```
   if (a < b)
    {
       t = a;       a = b;       b = t;
     }
```

Az if-else szerkezet

Kétágú elágazást megvalósító utasítás.

```
if  (kifejezés)
{
    igaz ág utasításai
  }
else
{
   hamis ág utasításai
  }
```



Tetszőleges mélységben ágyazhatunk egymásba if - else utasításokat ⟶ kapcsos zárójelek fontosak!

```
if  (kifejezés1)
    if  (kifejezés2)
        utasítás1

else

        utasítás2
// Itt az else a 2. if-hez tartozik.
```

```
if  (kifejezés1)
{   if  (kifejezés2)
    {
      utasítás1
    }
}
else
    {
      utasítás2
}
// Itt az else az 1. if-hez tartozik.
```

Feltételes kifejezés

Az **x** változóba betöltjük **a** és **b** minimumát először if - else utasítással:

```
if  (a < b)
    {    x = a;  }
else
    {    x = b;  }
```

Ugyanez megoldható feltételes kifejezéssel:

```
x = (a < b ? a : b);
```

Szintaxisa: kifejezés? kifejezés1 : kifejezés2
Értéke: ha kifejezés igaz, akkor kifejezés1, egyébként kifejezés2.
A feltételes kifejezés előnye az if-else szerkezetekkel szemben : nem utasítás, hanem kifejezés, ⟶ van értéke!
rövid, egy sorban elfér ⟶ makrók!

A switch – case szerkezet

Nézzünk egy kettőnél többágú tesztelést:

```
char c;
c =  getchar( );
switch(c)
{    case  'A':     printf("Alma");          /* A, programblokk */
                    break;
     case  'B':     printf("Banán");         /* B, programblokk */
                    break;
     case  'C':     printf("Citrom");        /* C, programblokk */
                    break;
     default :      printf ("Érvénytelen parancs!");
}
```

A program kiértékeli a switch (kapcsoló) utasítás melletti zárójelben álló kifejezést.
(Sorszámozott típus: egész vagy karakter!)

Majd sorban összehasonlítja a case (eset) utasítások mögött megadott konstansokkal (egész vagy kar.). Ezek címkézik a különböző esetekre előírt tevékenységeket. A program arra a pontra adja a vezérlést, ahol a konstans azonos (==) a kifejezés értékével.
A **break** utasítás a switch-case szerkezetből való kilépést eredményezi!
Ha nincs break, akkor a kövekező case-en folytatódik a program.
Default nem kötelező!
A fenti programrészlet *A, B* vagy *C* karakterek beolvasása esetén *Alma, Banán* vagy *Citrom* szöveget ír a képernyőre. Bármely más karakter esetén az *Érvénytelen parancs* jelenik meg.

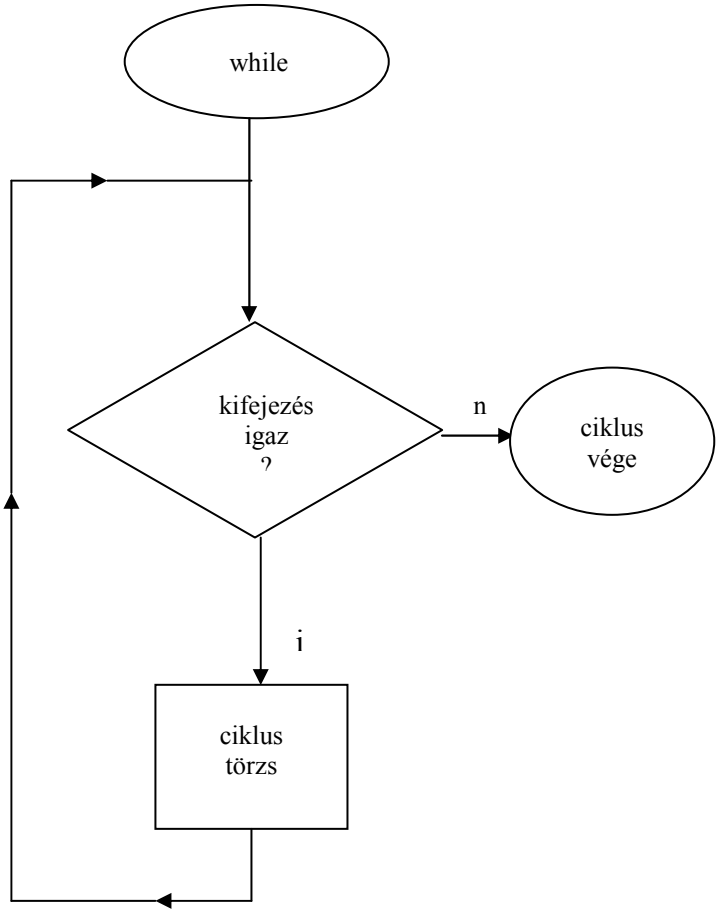
Ciklusszervezés

Három ciklusképző utasítás van a C-ben:

while for	}	elöltesztelő
do – while		hátultesztelő

while ciklus:

```
while (kifejezés)
{
    ciklustörzs
}
```



A számok összege 1-től 10-ig:

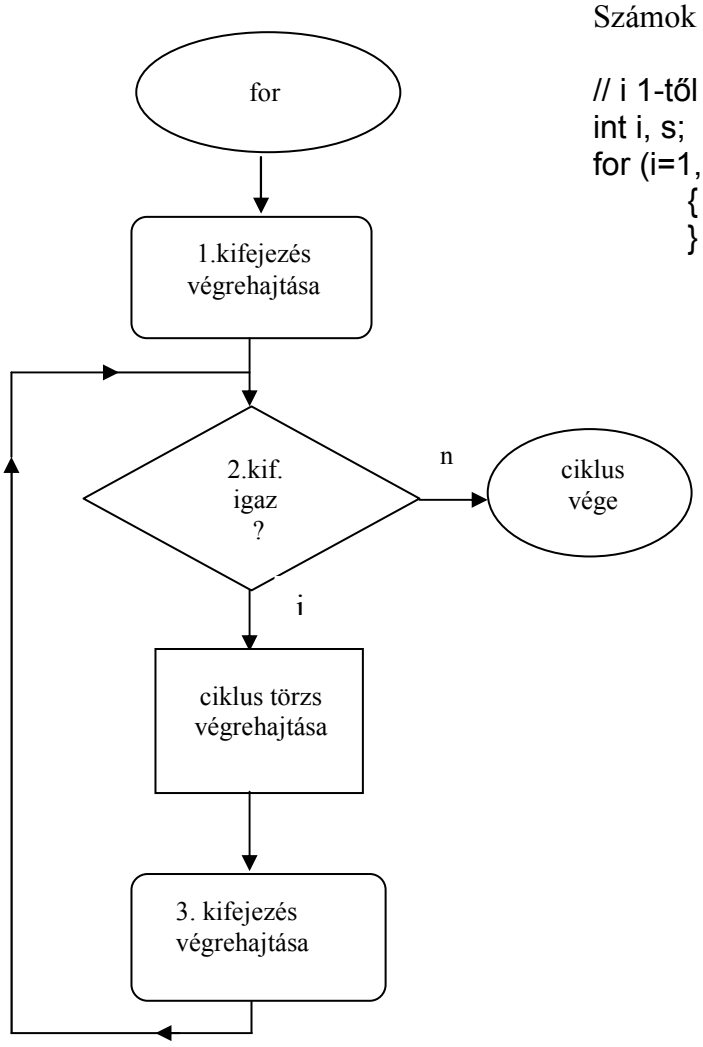
```
#include <stdio.h>
void main ( )
{ int a, sum;     a=1, sum=0;
  while (a<=10)                 // Az összegzést 1-től 10-ig végezzük!
      {  sum=sum+a;
         a=a+1;          //  vagy  a ++;
       }
   printf ("Az összeg 1-től 10-ig: %d", sum);   }
```

```
#include <stdio.h>
void main ( )
{  int a, sum;
   a=10, sum=0;       // Az összegzést 10-től 1-ig végezzük!
while (a)                          //  while (a!=0)
   {  sum=sum+a;
      a --;
    }
    printf ("Az összeg 1-től 10-ig: %d", sum);
}
```

Végtelen ciklus: while(1);

for ciklus:

```
for (1.fejezés; 2.kifejezés; 3.kifejezés)
        {
           ciklus törzs
         }
```



Számok összege 1-10-ig:

```
// i 1-től 10-ig nő, s-ben lesz az összeg
int i, s;
for (i=1, s=0; i<=10; i++)
          {  s=s+i;
          }
```

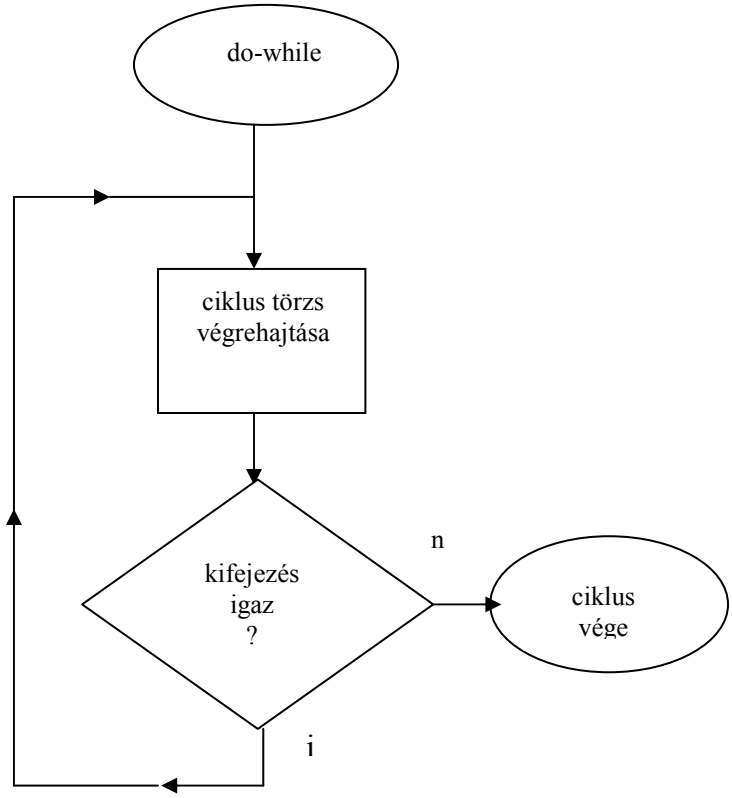
A while és a for teljesen egyenértékűek!
A while jellegzetes alkalmazási területe: olyan esetben, amikor az ismétlések száma előre nem látható.
A for ideális tömbök kezelésére.

Végtelen ciklus: for (; ;);

do - while ciklus:

Mivel hátul tesztelős ciklus, egyszer mindenképpen lefut!

```
do
  {
    ciklus törzs
  }
while (kifejezés);
```



Pl.: várakozás

```
main ( )
{   unsigned a;
    a=0;
    do
    {  a++;
     }  while (a);
// 65536-szor fut le a do-while ciklus
```

```
a=0;
while(a)
{  a++;
}
// Egyszer sem fut le a while ciklus!
```

A break utasítás

Az éppen aktuális ciklus elhagyására szolgál. Ha több ciklus van egymásba ágyazva, akkor csak egy szinttel kerülünk feljebb. (A *break* másik alkalmazási területe: *switch-case* elhagyása)

A continue utasítás

Az aktuális ciklus újabb iterációját indítja el. A ciklus fejrészére, pontosabban a ciklusmagot lezáró kapocsra (for!) ugrik.

A return utasítás

Visszatérés a hívott függvényből a hívóhoz érték visszaadásával vagy anélkül.

A goto utasítás

Feltétel nélküli vezérlés átadás. Használata csak akkor lehet indokolt, ha egymásba ágyazott ciklusok esetén az összes ciklusból ki akarunk lépni. (A break mindig csak egy szintet lép vissza.)
Kerüljük a használatát!

```
goto   címke;
  .
  .
  .
címke : utasítás
```

Függvények

Minden C program függvényekből áll, és minden függvény hívhat újabb függvényeket, elvileg tetszőleges mélységig. Önmagát is hívhatja bármely függvény: ez a rekurzió! A függvények teljesen egyenrangúak. A main( ) csak annyiban különleges, hogy ő a program belépési pontja. Minden C függvénynek tetszőleges számú bemenő paramétere lehet, de csak egy értéket adhat vissza. Ezt a visszatérési értéket értékadással vehetjük át.

```
c = getchar();          // A c változóba bekerül a beolvasott karakter ASCII kódja
```

Függvények : könyvtári
felhasználói (saját!).

Függvények definiálása

A függvények rendelkezhetnek típussal és paraméterekkel.
A függvény definiálása az, amikor a függvényt megírjuk. (Területfoglalás!)

Az ANSI szabvány szerinti írásmód:

```
fv_típus   fv_azonosító ( típus azonosító, típus azonosító, …)
{
   belső_ változók definiálása;
   függvény törzs
   return (visszatérési érték);
}
```

fv_típus: a függvény által visszaadott érték típusa, bármely alaptípus lehet;
fv_azonosító: a függvény neve;
paraméter deklarációk: a függvény neve utáni zárójelek között adjuk meg a függvény bemenő paramétereinek típusát és nevét (felsorolás vesszővel elválasztva). Ezek formális paraméterek!

Írjunk függvényt, amely 3 egész szám közül kiválasztja, és visszaadja a legkisebbet!

```
min3(int a, int b, int c)
{int i, j;
  i=a<b? a:b;
  j=i<c? i:c;
 return (j);          //vagy return j!
}
```

Az int típust nem kell kiírni, mert az az alapértelmezett:
min3(int a, int b, int c) ugyanazt jelenti, mint int min3(int a, int b, int c)

void függvény (void) típus és paraméter nélküli függvény, ilyen pl. : clrscr( )!
Ha a függvény típusnélküli, akkor nincs szükség a return utasításra!

A függvények deklarációja

A függvények sorrendje a programunkban tetszőleges lehet. Két módszer használatos:
- első a main( ),
- utolsó a main( ).

A fordító egymenetes, amikor az első függvényhívással találkozik, már tudnia kell, hogy a hívott függvény típusa mi, és hogy hány db és milyen típusú paramétere van.
Ha a függvényeket olyan sorrendben definiáljuk, hogy mire hívjuk őket, már ismeri a fordító, akkor minden rendben van.
(Először azokat kell definiálni, amelyek nem hívnak másik függvényt.)
De ha nem ilyen a sorrend pl. a main( ) van elől, akkor deklarálnunk kell a main( ) előtt a hívott függvényeket.
A függvény deklarációban a függvény típusát, azonosítóját és a paraméterek típusát adjuk meg.
Pl. a min 3 ( ) deklarációja v. prototípusa:
[int] min3 (int, int, int); vagy [int] min3(int a, int b, int c);
A következőkben azt az utat követjük, hogy a programok elején megadjuk a függvények deklarációját, ez ugyanis a függvények bármilyen sorrendje esetén biztonságos.
A fejlécfájlokban vannak a különböző könyvtári függvények deklarációi!

A függvények segítségével tudunk nagy programozási feladatokat megoldani.
Helyesen tervezett függvények esetében teljesen figyelmen kívül hagyhatjuk, hogyan keletkezik a függvény értéke (az eredmény), elegendő a feladat és az eredmény ismerete.
A C nyelv egyszerű, kényelmes és hatékony függvényhasználatot tesz lehetővé.

Paraméter átadás

A C-ben ***érték szerinti*** függvény argumentum átadás történik. Ez azt jelenti, hogy a hívott függvény az argumentumainak nem a címét, hanem az értékét kapja meg a veremben (stack). Tehát a hívott függvény nem tudja megváltoztatni a hívó függvény változóinak értékét, mert csak azok másolatát kapja meg a veremben.
Az érték szerinti hívás előny: tömörebbek a programok, kevesebb a segédváltozó és egyszerű rekurziót csinálni!
Ha meghívunk egy függvényt, és felsoroljuk a hívási paramétereket, egyúttal elhelyeztük ezeket az átadandó értékeket a tár egy erre szolgáló területén (STACK). Mikor a függvény belép, a paraméterek értéke a rendelkezésére áll, neki csupán azt kell megmondania, hogy hány db, és milyen típusú paramétert vár. Ez tehát deklaráció, mert már létező értékek nevét és típusát adja meg.
Így egy függvény paramétere tetszőleges kifejezés lehet (még egy függvényhívás is!)
Az argumentumok és belső változók csak arra az időre jönnek létre, amíg a függvény fut és elpusztulnak, ha a függvény kilép.
A STACK a memória egy speciálisan szervezett (li-fo = last in - first out) szegmense, dinamikus adattárolásra szolgál. Szubrutinok, függvények hívására és a visszatérés biztosítására találták ki. A stack lehetővé teszi tetszőleges számú és típusú paraméter megadását, és a rekurziót.
A fordítóprogram megvéd a veszélyektől.
A hívó program (függvény) a paraméterek után a visszatérési címet is a stack-re teszi.
A visszatérési cím a függvény hívása utáni utasítás címe!

A paraméterek az elhelyezés sorrendjében a visszatérési cím alatt vannak.

Példa függvényre:

Írjunk függvényt, amely hatványozást végez pozitív egész kitevővel, egész alappal!
A függvény az egész típusú alapot és az előjel nélküli kitevőt bemenő paraméterként kapja, a kiszámolt hatvány értéket hosszú egészként adja vissza a hívónak.
A főprogram 10-szer meghívja a hatvány függvényt változatlan alappal és 0-tól 9-ig változó kitevővel, majd a kapott hatványértéket kiírja a standard outputra.

```
#include <stdio.h>
long int hatvany (int alap, unsigned kit);
void main()
{  int a=13;              // alap
   unsigned x;            // kitevő
   long int aadx;
   for(x=0;x<10;x++)
      {  aadx=hatvany(a,x);
         printf("\n%d a(z)  %u hatványon: %ld",a,x,aadx);
      }
   }
long int hatvany (int alap, unsigned kit)
{ long ered=0;
   if (alap!=0)
      { ered=1;
        while(kit>0)
           {  ered*=alap;                    // ered=ered*alap;
              kit--;
           }
       }
   return ered;
}
```

A paraméterátadás megvalósításának két fontos kérdése:

- a paraméterek elhelyezési sorrendje,
- ki takarítja el a függvényből való kilépés során a stackről a feleslegessé vált paramétereket?

Egyenes sorrend: fv(a, b)

visszatérési cím
b értéke
a értéke

Az első paraméter van legtávolabb a visszatérési címtől

Fordított sorrend:

visszatérési cím
a értéke
b értéke

Az első paraméter van legközelebb a visszatérési címhez

Mindegyik eljárás logikus, jónak tűnik. Melyik lehet a jó megoldás?

A printf( ) tanulmányozása adja meg a választ: változó a paraméterszám, a printf csak a formátum sztringből tudja, hogy éppen hány. Csak úgy boldogul, ha a formátum sztring (az első paraméter) van legközelebb a visszatérési címhez!
Tehát fordított sorrend!
Ha a printf( )-nek kevesebb paramétert adunk, mint ahány formátum előírás van a formátum sztringben, akkor a függvény kivesz egy nem neki szánt értéket a stackből.
Ha pedig többet adunk, a felesleget nem használja fel.
A takarítást a hívó végezze: csak ő tudja, hogy hány paramétert adott!

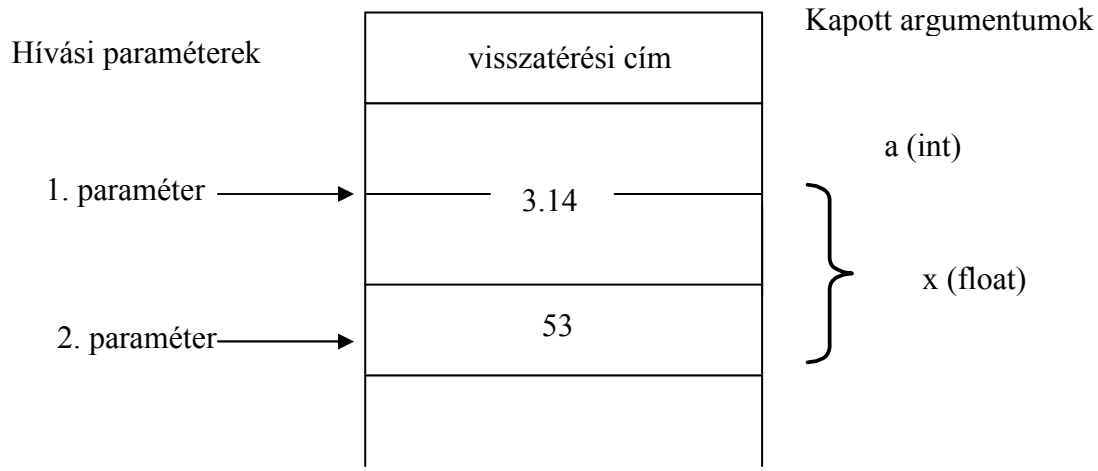
Paraméterek, argumentumok

Legyen egy függvény deklaráció a következő:

```
fgv (int a, float x);
  .
  .
  .
```

Helytelen meghívása egy másik függvényben:

```
float pi=3.14;
int i=50;
  fgv(pi, i+3);
```



Az ábrán látható, hogy az argumentumok várt sorrendjének nem felel meg a hívási paraméterek sorrendje.
A helyes meghívás: fgv(i+3,pi);

A paraméterek létrehozása és az argumentumok értelmezése független egymástól!
Egyeztetni kell a paraméterek és az argumentumok számát és típusát. (Sorrend!)
Súlyos és nehezen felderíthető hibákat okoz, ha a hívási paraméterek típusa és sorrendje nem felel meg a hívott függvény paraméter deklarációjának.
(Az újabb fordítók már a paraméterek számát ellenőrzik.)

A változók elhelyezése a C programokban

Változók definiálása történhet: valamely függvényen belül,
bármely függvényen kívül.

Lokális változók

A függvények saját, belső változói, amelyek csak a függvény meghívásakor jönnek létre, és kilépéskor megsemmisülnek. (Mivel a folyamat a C-ben automatikus, ezért automatikus változóknak is hívjuk őket.) Lokális változókra csak abban a függvényben vagy blokkban hivatkozhatunk, amelyben definiáltuk. Mivel a stack-ben vannak, kezdeti értékük meghatározhatatlan.

Globális változók

Az összes függvényre nézve külsők → értékük a függvény hívásoktól függetlenül fennmarad. Minden függvény név szerint elérheti őket. A permanens memória területen jönnek létre, kezdeti értékük 0.
Pl.:

```
int jel;
main ( )
{ jel = 100;
        .
        .
        f1 ( );
}
f1 ( )
{ int temp;
  temp = jel;
  f2 ( );
}
f2 ( )
{
  int jel;
   jel=10;
    .
    .
}
```

Példánkban van egy globális változó, neve jel. A main és az f1 függvények ezt a változót használják. Mivel az f2 függvénynek van egy jel nevű lokális változója, ez elfedi előle a globális változót, tehát az az ő számára elérhetetlen.

Statikus változók

A lokális változó csak a függvényben él.
A globálist mindig és mindenki név szerint elérheti.
A kettő közötti átmenet a statikus változó.

```
static int jel;
```

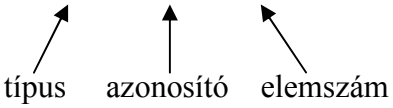
Ha belső változót (függvényen belül) statikusnak definiálunk, akkor permanens változó lesz, a függvény két hívása között megőrzi értékét. (De marad lokális, azaz csak az a függvény használhatja, amelyikben definiáltuk!)

Tömbök

Mint minden magas szintű nyelv, a C is ismeri a tömbök fogalmát.
A tömb: azonos tulajdonságú elemek rendezett halmaza, amelyben az elemszám rögzített.

Az elemtípus: alap,
összetett,
a felhasználó által deklarált.

Tömb definiálása: int tomb [10] ;



A tömb elemei: tomb[0], tomb[1], … tomb[9]

A tömb bármely elemére a tömb nevével, és az index megadásával hivatkozhatunk.
Az index tetszőleges egész jellegű (fixpontos) kifejezés!
A fenti példa egydimenziós tömböt definiál. Az egydimenziós tömb a vektor.
Kétdimenziós tömb: int tomb[10][20];
A kétdimenziós tömb a mátrix, az első méret a sorok (10), a második az oszlopok (20) száma.

Egydimenziós tömb általános definíciója:
típus név [dim];

Kétdimenziós tömb általános definíciója:
típus név [dim1] [dim2];

Több (i) dimenziós tömb általános definíciója:
típus név [dim1] [dim2] … [dimi];

Mivel a változók definiálásakor helyfoglalás történik, a tömbindexek mindegyikének fordítási időben ismertnek kell lenni!!!

A tömbelemek elhelyezése a memóriában

Bármilyen tömb esetében az elemek folyamatosan helyezkednek el a memóriában. Kétdimenziós tömbök esetében az első sor elemei után következnek a második sor elemei, …stb.

```
int     t [2] [3];      // Ennek a tömbnek 6 eleme van: 2*3
```

t [0] [0] t [0] [1] t [0] [2] t [1] [0] t [1] [1] t[1] [2]

A fordító csak akkor ellenőrzi, hogy átléptük-e a maximális indexet, ha konstansként adtuk meg, futtatás közben nincs ellenőrzés!
A C nem képes a tömbök globális kezelésére, azaz nincsenek olyan utasításai, amelyek a tömbökkel, mint egésszel végeznének műveletet. Minden műveletet nekünk kell megoldanunk a tömbök elemenkénti kezelésével!
Tömböt definiálhatunk külső és belső tömbként. Nagyobb tömböket célszerűbb külsőnek definiálni, hogy ne terheljük a stack-et!

Tömb feltöltése futásidőben

```
int  t [12];
  void    main()
  {    int     i;
       i = 0;
       while  (i < 12)
       {
           t [i] = i * i;
           i ++ ;
        }
   }
```

A for alkalmasabb, szemléletesebb a tömbkezelésre:

```
  int  t[12];
  void    main()
  {   int i;
       for  (i = 0;  i < 12;  i ++)
           t [i] = i * i;
   }
```

Határozott számú ismétlés, jobban látjuk a ciklus elejét és végét!

Kétdimenziós tömb feltöltése sorfolytonosan, majd az elemek kiírása a képernyőre mátrix alakban:

```
# include  < stdio.h >
# define   SORHOSSZ   6
# define   SORDB         8
int    tomb2 [SORDB] [SORHOSSZ];
void  main ( )
{
     int i, j;
     for (i = 0;  i < SORDB;  i ++)
     {
          for (j = 0;  j < SORHOSSZ;  j ++)
          {
              tomb2 [i] [j] = i + j;
           }
      }
      for (i = 0; i < SORDB; i ++)
      {  printf ("\n %d .␣ sor:\ t ", i);
         for (j = 0;  j< SORHOSSZ;  j ++)
         {
            printf ("%5d", tomb2 [i] [j]);
         }
       }
  }
```

A tömb minden sora külön sorba kerül ki a képernyőre!
Célszerű a tömbök méretét konstansként megadni, és a ciklusoknál ugyanezen konstansok szerepelnek az index felső határaként. Így elkerülhető a túlcímzés!

Tömbök előkészítése fordítási időben

Tömb előkészítés fordítási időben csak akkor lehetséges, ha statikus jellegű: külső vagy belső statikus. A C-ben az elő nem készített statikus területek minden bitje 0!

```
Pl.:    int  tomb [10 ] = { 1,2,3,4,5,6,7,8,9,10};
 vagy:  int  tomb [ ] = { 1,2,3,4,5,6,7,8,9,10};
```

Ilyenkor nem kell direkt módon megadni az elemszámot, a felsorolásból meg tudja állapítani azt a fordító.

```
int  tomb [2] [5]= { 1,2,3,4,5,6,7,8,9,10};
```

Itt a mátrix első sorában az elemek: 1 2 3 4 5
A második sorában pedig: 6 7 8 9 10

Többdimenziós tömbök esetében az első indexet nem kell megadni, de a többit igen:

```
int  tomb1[ ] [4] = {{1,2,3,4},
                     {10,20,30,40},
                     {100,200},
                     {-1, -2}      };          4 x 4

int  tomb2[ ] [4] = {1,2,3,4,
                    10,20,30,40,
                     100,200,                 3 x 4
                     -1, -2 };
```

Karaktertömbök, sztringek

Speciális karakteres tömb a sztring, amelyben nem kell tudni az elemszámot, mert a végét 0 érték ('\0' karakter) jelzi.

```
char str0[4] = {'a','b','c','d'};
```

Ez nem sztring, csak egy karakteres tömb, mert nincs záró 0!
Itt egyenként soroltuk fel a karakter konstansokat. Ilyenkor a fordító nem akarja a záró 0 értéket elhelyezni.

Ha nem aposztrófokat, hanem idézőjeleket használunk, akkor a fordító megpróbál sztringet létrehozni, azaz a záró 0 értéket elhelyezni az utolsó karakter után:

```
char str1[4] = "abcd";          // nem fér el a záró 0, ezért ez nem sztring!
char str2[5] = "abcd";          // elfér a záró 0, ezért ez sztring!
char str3[] = "abcd";           // elfér a záró 0, ezért ez sztring!
```

A következő feladatban egy, a fordítási időben előkészített sztringet kétféleképpen kiíratunk a printf függvénnyel: először sztringként, majd karakterenként.

```
# include  <stdio.h>
char  szoveg [ ] = "Ez egy string!\n";
void main ( )
{  printf ("%s", szoveg);              // A sztring kezdőcímét adjuk át a printf függvénynek
   int i;
   for (i=0; szoveg[i]; i++)                    // a feltétel :  szoveg[i]!=0
     {  printf ("%c", szoveg[i]);  }
   printf ("A sztring hossza: %d", i);
}
```

Példa:
Beolvasunk egy maximum 80 karakteres sort a klaviatúráról, a bevitel végét újsor karakter jelzi. Elhelyezzük a karaktereket egy tömbben, végére a záró 0-át is! (Sztring legyen!)
Először hátul tesztelős ciklussal oldjuk meg:

```
# include  <stdio.h>
char  buff [81];                              // Legyen hely a záró nullának is!
void  main ( )
{ int i=0;
   char c;
    do {    c= getchar ( );
            buff [i] = c;
            i++;
        }while(i<80 && c ! = '\n' );
    buff [i] = 0;                             //  vagy  buff [i] = '\0';
 }
```

Ebben a megoldásban az újsor karakter ('\n') is belekerül a tömbbe!

A feladat megoldása elöl tesztelős ciklussal:

```
# include  <stdio.h>
char  buff [81];                              // Legyen hely a záró nullának is!
void  main ( )
{ int i;
   char c;
   for (i = 0;  i<80 && (c = getchar ( ) )! = '\n' ; i++)
     {    buff [i] = c;              }
   buff [i] = 0;                              //  vagy  buff [i] = '\0';
 }
```

Ennek a megoldásnak az érdekessége, hogy a for ciklus feltételében történik meg a beolvasás. Ha újsor karaktert ('\n') olvastunk, a ciklusmag nem fut le, tehát az újsor karakter nem kerül bele a tömbbe.

Tömbök és függvények

A sztringekkel gyakran kell olyan feladatot megvalósítani, ami elég általános és önálló →
→ függvény a megoldás!
Ha a műveletet végző függvény név szerint hivatkozik egy globális tömbre, elveszti általánosságát.
Ezért a tömböt (vagy tömböket), paraméterként kell átadni!

Sztring másolás függvénnyel:

```
# include  <stdio.h>
void masol (char s1[ ], char s2[ ]);
//  a függvény deklarációja: a paraméterek deklarációjában a [ ] azt jelzi, hogy nem
//  egyetlen karakter, hanem egy karakter típusú tömb a paraméter
char tforras[ ] = "Ez a másolandó string! ";
char tcel [40];
void  main ( )
{    printf ("\nForras: %s\nCél: %s", tforras, tcel);
     masol (tforras, tcel);                          // a tömb neve a tömb kezdőcímét jelenti!
     printf ("\nForras: %s\nCél: %s", tforras, tcel);
}
void masol (char s1[ ], (char s2[ ])
{    int i ;
     for (i = 0; s1[i]!=0; i++)
          {   s2[i]= s1[i] ;    }                    //A ciklusmagban történik a másolás
     s2[i]=0;                                        // Mivel a 0 nem került át, pótoljuk!
}
```

A másoló függvény egy tömörebb C-szerűbb megoldása:

```
void masol (char s1[ ], (char s2[ ])
{    int i ;
     for (i = 0;  (s2[i]= s1[i])!=0; i++);
}
/*
Gyakorlatilag nincs is ciklusmag, mert a feltételben megtörténik a másolás.
Mivel a másolás után vizsgáljuk a karaktert, a záró 0 is átkerül!
*/
```

A másolás mindkét esetben megtörténik a célba.
Itt a paraméterek nem az eredeti értékek másolatai.

A tömb paraméterként való átadása

A függvényeknél (17. oldal) megtanultuk, hogy egyszerű változók esetében a paraméterátadás érték szerinti. A tömb paraméterként való átadása azonban, nem az elemeinek stack-beli másolatával (mint egyszerű változók esetében), hanem a **címével** (kezdőcím: az első elem címe) történik → indirekt hivatkozással lesznek az eredeti tömbelemek elérhetők a meghívott függvényben!
Tehát összefoglalva: bármilyen *tömb paraméterként történő átadása* a függvények között, *nem érték szerinti, hanem cím szerinti!*

Pointerek

A pointer olyan változó, ami valamilyen C objektum címét tartalmazza. A C-ben bárminek előállíthatjuk a címét, amennyiben van címe! (Változók, tömbök, struktúrák, függvények)

```
int a;  →  &a;
```

Kell egy speciális változó, amelybe bele tudjuk tenni valaminek a címét!

∗ → indirekció operátor. Pointer definiálása: int *p;

A pointerek használat előtt inicializálandók!

A pointerek használatba vétele:

- definiálunk olyan objektumot, amelyet pointerrel akarunk elérni,
- definiálunk ugyanolyan típusú pointert,
- az objektum címét betöltjük a pointerbe.

Példával:

```
int a, b;
int  ∗pnt;
     pnt=&a;
     b=∗pnt;        // a b változóba beírjuk a értékét indirekt hivatkozással.
```

A pointereket elsősorban tömbök kezelésénél használjuk, mert kényelmes.
C-ben a tömb neve a tömb kezdőcíme, azaz a tömb első elemének címe!

Pointerek alkalmazása tömbök esetén

Pl: külső karakteres tömb (sztring!) átmásolása két verzióban:

```
char s [ ]= "Ez a másolandó sztring!";
char t [40];
void main ( )
{    int i;
     i=0;
    while ((t[i]=s [i])!=0)   // a másolás a ciklus feltételében történik!
    {  i++; }
}
```

Pointerekkel:

```
char s [ ]= "Ez a másolandó sztring!";
char t [40];
void main ( )
{ char ∗p, ∗q;
    p=s; q=t;              // A pointerekbe beleírjuk a tömbök címét
    while ((∗q=∗p)!=0)
    { p++;  q++;  }
}
```

Pointerek alkalmazása egyszerű változók esetén

Legyen a csere( ) függvénynek két egész paramétere (egyszerű változók)!

```
void csere_1 (int a, int b);     // a függvény deklarációja
```

Feladata, felcserélni a paraméterként kapott változókat.

```
void csere_1 (int a, int b)
{    int tar;
     tar=a;    a=b;     b=tar; }
```

Az érték szerinti paraméterátadás miatt csak a stack-beli másolatokat cseréli fel!

A feladat pointerekkel megoldható:

```
void csere_2 (int *p, int *q)
{    int tar;
     tar=*p;  *p=*q;   *q=tar; }
```

Ebben az esetben a hívó a felcserélendő változók címét adja át!
Itt pointerek a függvényargumentumok!

A mutató argumentumokat gyakran alkalmazzák olyan függvényeknél is, amelyeknek egynél több értéket kell visszaadniuk.

A két cserélő függvényhez a főprogram:

```
void main ( )
{ int a, b;
   a=10; b=100;
   printf (" Csere előtt: a=%d, b =%d\n ",a, b);
   csere_1 (a,b);                                  // Nem történt meg a és b cseréje!
   printf (" csere_1 függvény után: a=%d, b =%d\n ",a, b);
   csere_2 (&a,&b);                                // Itt megtörtént!
   printf (" csere_2  függvény után: a=%d, b =%d\n ",a, b);
}
```

A csere_2() esetében nem a két változó értékének másolata kerül a stack-be, hanem a címe. Ezért a változók eredeti helyén történik meg a csere az indirekt címzés miatt!

Sztring másoló függvény pointeres megoldásokkal

Írjuk meg az strcopy() függvényt, amely a korábban vizsgált sztring másolást hajtja végre!
Nagyon tömör C-szerű megoldás:

```
void strcopy ( char s1[ ], char s2[ ])
{
  while((*s2++=*s1++)!=0);
    }
```

Itt nincs is ciklusmag, minden megtörténik a feltételben!

Részletesebben:

```
void strcopy ( char s1[ ], char s2[ ])
{
  while((*s2=*s1)!=0)
     {    s1++, s2++;   }
    }
```

Még részletesebben:

```
void strcopy ( char s1[ ], char s2[ ])
{
  while(*s1!=0)
       {  s2=s1;
          s1++,  s2++;  }
  *s2='\0';            }                    // Itt a ciklusban nem másolódik a záró 0!
```

A függvény törzse lehet ugyanez, akkor is ha a fejsor:

```
void strcopy ( char *s1, char *s2)
```

De mindkét esetben a függvény törzse lehet a következő is:

```
{ int i=0;
  while((s2[i]= s1[i])!=0)
    {  i++;   }
}
```

A függvény számára mindkét esetben az argumentumok címek, azaz pointerek.
Nem számít, ha tömbként deklaráljuk, és pointerként használjuk vagy fordítva.
Bár a tömb neve a tömb kezdőcímét jelenti, ő nem pointer, hanem egy címkonstans, ami csak a fordító tudatában létezik! Tehát nem végezhetünk vele műveletet!

```
char buff [40];
  * (buff+5)=0
    buff++  vagy  *buff++='a'              //    illegális!
```

Két sztringet összehasonlító függvény elemzése

A függvény döntse el, hogy az ABC szerint két sztring közül, melyik van előbb!

```
Pl.:      ABCDEF     és     ABED        //  itt az 1.
          ABCDEF     és     ABCD        //  itt a 2.
          ABCDEF     és     ABCDEF      //  ezek azonosak!
```

```
compare (char *s1,  char *s2)
{
  while (*s1 == *s2 && *s1)
  {       s1++;
          s2++;          }
  return (*s1 - *s2);
}
```

A ciklus addig fut, amíg a 2 sztringben azonosak a karakterek, és nem értük el a záró 0-át!
A visszaadott érték azonos sztringek esetén a két '\0' karakter különbsége, ami 0.
Különböző sztringek esetén az első eltérő karakterpár ASCII kódjának különbsége, azaz, ha az első kisebb (előbb van az ABC szerint), akkor negatív, egyébként pozitív.
Összefoglalva a lehetséges eseteket:
1. Ha a két sztring azonos, akkor 0 a visszaadott érték.
2. Ha az első sztring tartalmazza a másodikat, akkor pozitív.
3. Ha a második sztring tartalmazza az elsőt, akkor negatív.
4. Ha a sztringek valamely belső karakterükben eltérnek, akkor is jó lesz a válaszérték:
   (lehet negatív is és pozitív is!)
   a visszaadott érték < 0 akkor 1. < 2.
   a visszaadott érték > 0 akkor 1. > 2.

Az összehasonlítást elvégző függvény a C-beli sztringek szerkezetének szerencsés megválasztását igazolja!

A sizeof operátor

A sizeof operátor egy a memóriában elhelyezett objektum, vagy egy ismert változó típus méretét adja meg bájtokban. *Fordítási időben értékelődik ki!*

sizeof (int)
sizeof (float) } típus

sizeof (name) változó neve

Numerikus tömb átadása függvénynek

A sztring az egyetlen tömb típus, amelyet a kezdőcíme egyértelműen meghatároz! (A záró 0 miatt.)
Más típusú tömbök esetén az elemszámot is ismerni kell!

Írjunk függvényt numerikus tömb kezelésére!
Elkészítjük a buborék rendezés olyan változatát, amely azonnal észreveszi, ha a rendezés kész, és abbahagyja a további vizsgálatokat!

```
# include <stdio.h>
void  buborek (int a[ ], int m);
int t[ ] = {30, -12, 53, 0, 4, -2, 80, 512, -12, 8, 5, 10};
void main ( )
{   int mt, i;
    mt = sizeof (t) / sizeof (int);                    // a tömb elemszáma
    printf ("\n A tömb elemei rendezés előtt: \n");
    for (i=0; i < mt; i++)
         printf("%5d,", t [i]);
    printf("\b. ");       // az utolsó után . (pont) lesz a , (vessző) helyett
    buborek (t, mt);
    printf ("\n A tömb elemei rendezve: \n");
    for(i=0; i<mt; i++)
       printf("%5d,", t[i]);
    printf("\b. ");
}
```

```
void buborek(int a[ ], int m)                    // növekvő sorrend lesz.
{  int csere, i, t;
        do
         {  csere=0;
           for(i=0; i<m-1; i++)
           {    if(a[i] > a[i+1])
                { t=a[i]; a[i]=a[i+1]; a[i+1]=t;
                  csere=1; }
            }
           m - -;        // a legnagyobb a végén van!
         } while (csere);
}
```

Sehova mutató pointerek?

A pointerek alkalmazásának legnagyobb veszélye, hogy bármi a tartalma, mindig címez valahova. Tehát, ha előkészítetlen pointerrel címzünk, akkor BUMM!
Függvények is adhatnak vissza pointert (címet), ha ez NULL, akkor sikertelen volt a művelet.
A permanens változó kezdeti értéke 0, tehát ha az ilyen pointert nem inicializáljuk → *null pointer assignment* hiba üzenetet kapunk a program futtatásakor!

Címaritmetika (pointer aritmetika)

char ∗p; // a ++ operátor itt valóban 1-gyel növel
int típus esetén → 2-vel
float típus esetén → 4-gyel

A pointerekre nemcsak ++, -- operátor, hanem + és - is alkalmazható, de csak egész típusú értéket lehet hozzáadni és kivonni.
Sőt két azonos típusú pointert kivonhatunk egymásból. → a fizikai különbség a bájtban vett eltolás a két cím között.
A pointerhez elválaszthatatlanul hozzátartozik a típusa!

Egy tetszőleges típusú pointer, a ++ operátor hatására, a vele azonos típusú objektum típusának bájtokban számított hosszával nő!
(Tehát tömb esetén a következő elemre fog mutatni!)

A pointert nemcsak tömbök, hanem egyedi változók címének tárolására is használhatjuk.
Egyedi változók esetén nem végezhetünk címaritmetikát!

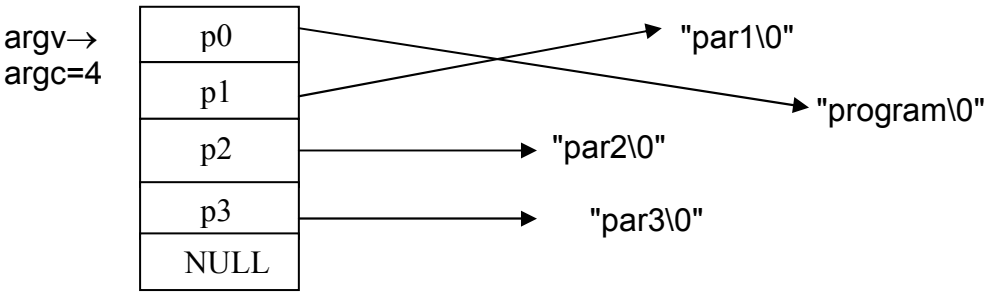
Pointerek és tömbök kapcsolata

Bármely tömböt lehet pointerrel kezelni és bármely pointer használható tömbök címzésére.
Ha egy tömböt függvénynek adunk át, akkor a függvény számára csak a pointer létezik.
De valójában a tömb és a pointer két teljesen különböző objektum.

Program argumentumok

A C nyelv és a UNIX operációs rendszer találkozása:
A parancssorban, a program neve után megadott argumentumokat a rendszer a program számára elérhetővé teszi. A parancssor szavakból áll. Az első szó a program neve, a továbbiak az argumentumok. A szavak egy-egy sztringben helyezkednek el a memóriában (nem is feltétlenül folytonosan), címeik egy karakteres pointertömbben vannak. Ennek a pointertömbnek a címét kapja meg a program belépésekor (kétszeresen indirekt pointer!).
Két argumentumot vehetünk át (opcionális az átvétel!).

A main ( ) argumentumainak szabványos neve: argc, argv (ajánlott nevek!)
argc egész ≥ 1 az argumentumok száma (a parancssor szavainak száma)
argv karakteres pointertömbre mutató pointer



\ program par1 par2 par3 //parancs-sor

Ha a parancssorban 3 paramétert adunk meg, akkor ez 4 szó, és minden szó egy különálló sztringben kerül eltárolásra a memóriában. Ezek címei (pointerek) egy tömbbe kerülnek, amelynek a címe lesz az *argv* paraméter. Tehát *argv* egy karakteres pointertömbre mutató pointer. Az *argc* a parancssori szavak számát adja meg, tehát értéke legalább 1.
Esetünkben az *argc* 4 lesz.

Három változatban megírjuk azt a programot, amely kiírja saját valódi argumentumait
(a program nevét nem!):

```
# include <stdio.h>                          //1.
void main (int argc, char *argv[ ])
{ int i;
  for(i=1; i<argc; i++)
     {  printf("\n %s", argv[i]); }
}
```

```
# include <stdio.h>                          //2.
void main (int argc, char **argv)
{   while (--argc)
    { printf("\n %s", * ++argv);}
}
```

```
# include <stdio.h>                          //3.
void main (int argc, char *argv[ ])
{   int i, j;
    for(i=1; i<argc; i++)
    {  for(j=0; argv[i][j]; j++)
           printf("%c", argv[i] [j]);
      printf("\n");
     }
}
```

Az utolsó változatban karakterenként történik a kiírás, a másik kettőben pedig sztringként.

Preprocesszor utasítások

Preprocesszor: előfordító vagy elő-feldolgozó. A C forrásprogramot előkészíti a compiler (fordító) számára, és végrehajtja a # -tel kezdődő utasításokat.

```
# include      - forrásnyelvi fájlok beemelése
# define       - konstans és makro deklaráció
# if
# else         } Feltételes fordítási blokkok vezérlése
# endif
```

Makró deklarálás

```
# define       quad(x)       x·x
```

Az " x·x " -et bemásolja a preprocesszor a program szövegébe a quad(x) helyett, közben elvégzi a paraméterhelyettesítést!

```
  z = quad(y);        //⇒ z = y·y;   lesz
  a = quad(25);       //⇒ a = 25·25;       lesz
! i = quad(a+3)       //⇒ i = a+3 * a+3   ez nem  a+3 négyzete, hanem 4*a+3!
```

Fontos a ZÁRÓJELEZÉS!

```
# define quad(x)     (x) * (x)
# define max (a,b)   ((a)>(b) ? (a):(b))
# define min (a,b)   ((a)<(b) ? (a):(b))
# define abs (a)     ((a)<(0) ? -(a):(a))
```

Az utolsó 3 makró példa a feltételes kifejezés használatára!

Struktúrák

Egy struktúra olyan memóriaterület, amely különböző típusú, összetartozó és egyedenként elérhető változók összességét tartalmazza.
A struktúrát valamely, a programunk által kezelt, objektum összes jellemző adatának összefoglalására használjuk fel.
A struktúra felhasználása is a definícióval kezdődik. De ez a felhasználó definiálta típus, tehát nemcsak a nevét, hanem belső felépítését is mi magunk szabjuk meg.

A struktúra használatba vételének lépései:

- a struktúra deklarációja (az új típus létrehozása): a struktúrának, mint változó típusnak nevet adunk, és megszabjuk a belső felépítését (felsoroljuk a tagjait);
- a struktúra definíciója: a tárban valahol helyet foglalunk a struktúra számára, azaz létrehozunk ebből az új típusból egy változót;
- a struktúra felhasználása: a tagjaira való hivatkozás.

A struktúra egységes egészként való felhasználása nem lehetséges!

Struktúra deklaráció

```
struct struktúra_cimke  {  típus változónév;
                           típus változónév;
                            .
                            .
                         };
```

Struktúra definíció

```
struct struktúra_cimke  struktúra_változó1, struktúra_változó2,…;
```

Összevonható a kettő:

```
struct struktúra_cimke  {
                           típus változónév;
                           típus változónév;
                            …
                         }struktúra_változó;
```

Pl.:
```
struct datum     { char nap;
                   char honap;
                   int ev;
                 } tegnap;

struct szemely  {  char nev [20];
                   char cs_nev [20];
                   struct datum sz_ido;
                   char sz_hely [20];
                 } valaki;
```

A *szemely* típusú struktúrának a harmadik tagja *datum* típusú struktúra.

Struktúra felhasználása

Hivatkozás a struktúra tagokra:

```
tegnap.ev=2005;
valaki.sz_ido.ev=1982;

printf ("Sz. hely: %.20s", valaki.sz_hely);
printf ("Sz. év: %d", valaki.sz_ido.ev);
```

A struktúrában szereplő tömböt elemenként is kezelhetjük:

```
int i;
for (i=0; valaki.sz_hely[i]!=0 && i<20; i++)
    { putchar (valaki.sz_hely[i]); }
```

A struktúra belsejében egyébként nem szokás tömböket definiálni, mert különösen karakteres tömböknél, fennáll a veszély, hogy esetleg kevés lesz a hely. A sztringeket másutt (pl. a dinamikus memóriában) helyezzük el, és a struktúrában csak a címeiket tároljuk.
A pointer fix memóriaterületet igényel!

```
struct személy { char *nev;
                 char *cs_nev;
                 struct datum sz_ido;
                 char *sz_hely;
               } valaki;
```

Struktúratömbök

```
struct szemely  oszt[30];    // egy 30 fős csoport adatainak tárolására
```

oszt egy 30 elemű tömb, amelynek elemei *szemely* típusú struktúrák.

```
int i=0;
oszt[i].cs_nev="Abai";
```

Az eddigiek szerint a struktúrákat nem tudjuk egységes egészként kezelni (pl. függvénynek átadni). Bonyolult és nehézkes a struktúratagok elérése.
oszt a struktúratömb kezdőcíme, nem pointer – csak cím-konstans!
Ha már deklaráltuk a *szemely* típusú struktúrát, a fordító ismeri ezt a típust. Így tudunk létrehozni olyan pointert is, amely alkalmas *szemely* típusú struktúra megcímzésére.
Az ilyen pointer típusa is természetesen *szemely* típusú struktúra lesz:

```
struct szemely *psza;
psza = & valaki;
```

Minden pointert inicializálni kell felhasználás előtt, tehát beleírjuk a *valaki* nevű *szemely* típusú struktúra kezdőcímét.

A struktúrapointer definiálása ugyanolyan, mint bármely más pointeré.
A struktúratagok elérése pointerrel:

```
psza → nev= "Lajos";
```

A struktúra tag neve egy offsetcímet jelent, a struktúra elejétől való eltolást bájtban.

valaki.nev	psza → nev
↓	↓
direkt hivatkozás	indirekt hivatkozás
a címek összeadását	futás közben történik
a fordító végzi el	a címszámítás

A struktúrapointer lehetővé teszi, hogy a struktúrákat a tömbökhöz hasonlóan a címük segítségével adjuk át egy függvénynek.
(Visszatérési érték csak egyszerű típus lehet, pointer is!)

A struktúra hossza: nem mindig egyezik meg a tagok összhosszával.
(Pl.: int és float típus páros címen kezdődik.)
sizeof (valaki) vagy sizeof(struct szemely)

Ha egy struktúra tömb kezdőcímét írom bele a struktúra pointerbe, akkor alkalmazható a cím aritmetika is. Ilyenkor a ++ operátor hatására a pointer tartalma a vele azonos típusú struktúra bájtban vett hosszával nő, azaz a pointer a tömb következő struktúrájára fog mutatni..

```
struct szemely  oszt[30];
struct szemely *pso;
pso=oszt;              //most a pso pointer az oszt tömb első struktúrájára mutat
pso++;                 //most a pso pointer az oszt tömb második struktúrájára mutat

pso->cs_nev="Baráth";
pso->nev="Gábor";
```

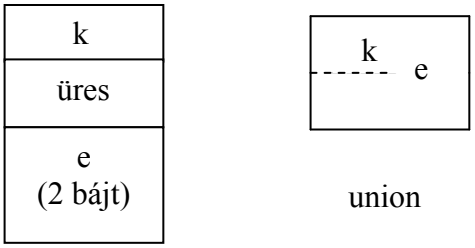
Beleírtuk az *oszt szemely típusú struktúra tömb* második elemébe a Baráth Gábor nevet.

Unionok

Olyan struktúra, amelyben a tagok nem egymás mellett vannak a tárban, hanem egymásra kerülnek. Így a union mérete azonos a legnagyobb méretű tagjáéval. Ha egy elemet módosítunk, akkor az összes többi is megváltozik.

```
union unev
{ char k;
  int e; };
union unev u;
```

Deklaráció és definíció ugyanaz, mint a struktúránál, de ha létrehozunk egy ugyanilyen szerkezetű struktúrát is, az eredményt a következő ábra szemlélteti:



struktúra

A uniont tárterület konverzióra használjuk!
Pl.: egy valós típusú adat bájtjait meg tudjuk nézni:

```
union ftipus { float f;
               char b[sizeof (float)];  } f_bajt;

f_bajt.f = 3.1415;
for (i=0; i < sizeof (float); i++)
  { printf ("%02x ␣"f__bajt.b[i]); }
```

vezető nullák
2 jegy hexadecimális

Fájlkezelés C-ben

A számítógépek háttértárain elhelyezkedő adatállományokat a C-ben tartalmuk alapján szöveges (text) és bináris fájlokra osztjuk.
A szöveges állományok általában olvasható információkat tartalmaznak, változó hosszúságú sorokból állnak és a sorokat CR/LF (kocsivissza/soremelés) zárja.
A bináris állományok bájtokból épülnek fel, a fájl egy egydimenziós bájt tömb.

Az állományok tartalmának eléréséhez a következő lépéseket kell megtenni:
- a fájl azonosító definiálása,
- az állomány megnyitása ellenőrzéssel,
- az állomány tartalmának feldolgozása fájlműveletek (olvasás, írás, pozicionálás, stb.) felhasználásával,
- a műveletek elvégzése után az állomány lezárása.

A nyitástól a zárásig valamennyi fájlműveletet könyvtári függvények meghívásával tudjuk elvégezni.

A C a fájlkezelés két szintjét támogatja:
- alacsony vagy egyes szintű fájlkezelés (UNIX szabvány) gépközeli;
- magas vagy kettes szintű fájlkezelés (C és UNIX szabvány) az egyesre épül.

A továbbiakban a magas (kettes) szintű fájlkezelésről lesz szó.
A magas szintű fájlkezelés az input/output műveleteket pufferelten hajtja végre. A rendszer biztosít egy 512 bájtos memória puffert, amelynek a kezelését is elvégzi. Tehát nem minden írás/olvasás jelent fizikai I/O műveletet.
A magas szintű fájlkezeléshez szükséges konstans-, típus- és függvénydeklarációk az **stdio.h** fejlécfájlban vannak. Többek között a **FILE típusú struktúra** deklarációja is.

Egy **FILE típusú struktúra** a megnyitott fájlra és a hozzárendelt memória pufferre vonatkozó információkat tárolja:
- fájl sorszám (file handle),
- FLAG bájt: a fájl utolsó művelet utáni állapotát (volt-e EOF vagy hiba?) jelzi,
- a fájlhoz rendelt memória puffer kezdőcíme,
- a pufferben még elérhető karakterek (bájtok) száma,
- pointer, amely a pufferben az utoljára elért elem mögé mutat.

A rendszer területén van egy ilyen 20 elemű FILE típusú struktúra tömb. Ebből következik, hogy egyidejűleg legfeljebb 20 fájl lehet nyitva. Sikeres fájlnyitás esetén az első szabad struktúrába belekerülnek a fájl és a hozzárendelt memória puffer adatai, és a programunk ennek a struktúrának a címét (memória pointer!) kapja meg, ezzel tudjuk azonosítani a fájlunkat a műveletek során.
Az első 5 struktúra a standard fájlok számára van fenntartva:
- stdin: standard input csatorna,
- stdout: standard output csatorna,
- stderr: standard hiba kimenet,
- stdaux: standard aszinkron adatátviteli csatorna,
- stdprn: standard nyomtató kimenet.

Az stdin és az stdout tehát nemcsak fogalom, hanem szimbólum is, a FILE típusú struktúratömb első két elemének a címe!

A fájl megnyitásakor meg kell adnunk a fájlhoz való hozzáférés módját, azaz, hogy mit akarunk a megnyitandó fájlban csinálni.

A szöveges és a bináris módú fájlfeldolgozás között csak annyi a különbség, hogy szöveges módban megtörténik a CR/LF karakterpár és az ENTER közötti konverzió.

Hozzáférési módok magas szintű fájlkezelésnél:

Szöveges (text)	Bináris	
"r" vagy "rt"	"rb"	Létező fájl megnyitása olvasásra.
"w" vagy "wt"	"wb"	Új fájl megnyitása írásra. Ha a fájl már létezik, tartalma elvész!
"a" vagy "at"	"ab"	Fájl megnyitása a végéhez való hozzáírásra (folytatás). Ha a fájl nem létezik, akkor létrejön.
"r+" vagy "rt+"	"rb+"	Létező fájl megnyitása írásra és olvasásra (update).
"w+" vagy "wt+"	"wb+"	Új fájl megnyitása írásra és olvasásra (update). Ha a fájl már létezik, tartalma elvész!
"a+" vagy "at+"	"ab+"	Fájl megnyitása a fájl végén végzett írásra és olvasásra (update). Ha a fájl nem létezik, akkor létrejön.

A magas szintű fájlkezelés legfontosabb függvényei:

fopen() megnyitás
fclose() lezárás

Olvasás	Írás	Adategység
fgetc()	fputc()	karakter
fgets()	fputs()	sztring
fscanf()	fprintf()	formázott
fread()	fwrite()	adatcsomagok (blokkok)

Egy megnyitott fájlhoz 3 alappointer tartozik:
- SP: Start Pointer értéke: 0 (a bájt tömb első elemének indexe)
- FP: File Pointer értéke: a pillanatnyi bájt pozíció
- EP: End Pointer értéke: a fájl hossza bájtban.

Az FP (File Pointer) módosítása és lekérdezése:
- fseek() FP állítása, pozicionálás,
- ftell() FP lekérdezése (bináris fájlok esetén addigi bájtszám),
- rewind() FP-t a fájl elejére állítja (0),
- feof() a fájl végének lekérdezése.

Egyéb fájlkezelő függvények:

- ferror() hiba lekérdezése,
- fflush() a puffer ürítése írás/olvasás esetén,
- setbuf() memória puffer kijelölése a fájlhoz,
- freopen() fájl átirányítása.

Bármilyen fájlművelet csak megnyitott fájlban végezhető el!
A fájlműveletek mindig az FP által megcímzett bájttal kezdődnek.

A pozicionálás (FP állítása) egy fájlban a 3 alappointer valamelyike szerinti lehet:

- Abszolút pozicionálás (SEEK_SET): SP-hez képest pozitív mértékű elmozdulás bájtban.
- Relatív pozicionálás (SEEK_END): EP-hez képest negatív mértékű elmozdulás bájtban.
- Relatív pozicionálás (SEEK_CUR): FP-hez képest pozitív vagy negatív mértékű elmozdulás bájtban.

A pozitív mértékű elmozdulás a fájl vége felé, a negatív mértékű pedig a fájl eleje felé történik.

Két egyszerű fájlkezeléses feladat

Az első szöveges, a második bináris feldolgozást valósít meg.

```
//Indítási paraméterként megadott szövegfájl listázása a standard outputra.
//24 soronként várakozás egy billentyű leütésére, majd képernyő törlés.

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
  main(int argc,char *argv[])
  {
     //fájl azonosító (FILE típusú struktúra pointer) definiálása:
     FILE *fp;
     char c;
     int sor=0;
     if (argc<2)  {  printf("Nincs paraméter!\n"); exit(1);  }

     // Fájl megnyitási kísérlet:

     fp=fopen(argv[1],"r");
     if (fp==NULL)   {  printf("Sikertelen a fájl megnyitása!\n");
                exit(1);  }

     //Olvasási ciklus EOF-ig:

     while(!feof(fp))
       {   c=fgetc(fp);
           fputc(c,stdout);
           if (c=='\n')  {  sor++;
                       if (sor==24) { sor=0;  getch();  clrscr(); }
        }  }

     fclose(fp);
     getch();
     exit(0);
  }
```

```
//Bináris módú fájlkezelés
//ODIJ típusú struktúrában tárolt adatok írása/olvasása,
//pozicionálás és FP lekérdezése.

#include <stdio.h>
#include <conio.h>
typedef struct odij { char nev[20];
                      char evf;
                      char szak;
                      float atlag;
                      unsigned penz;} ODIJ;

void kiir(ODIJ *strukt)
{  printf("\n %-20s",strukt->nev);
   printf("evf: %c  szak: %c\t",strukt->evf,strukt->szak);
   printf(" tlag:%5.2f\t”szt”nd˘j:%6u",strukt->atlag,strukt->penz);
 }

int main(void)
{
   FILE *stream;
   int rdb;
   long poz;
   ODIJ diak={"Gipsz Jakab",'1','V',3.4,5000};
   clrscr();
   if ((stream = fopen("MANI.DAT", "wb+"))== NULL)
   {  fprintf(stderr, "Sikertelen megnyitás!\n");
      return 1;
      }
do{
   fwrite(&diak, sizeof(ODIJ), 1, stream);
   diak.evf+=1;
   }while(diak.evf<'4');

   poz=ftell(stream);
   fprintf(stdout,"\n A fájl hossza bájtban: %ld\n",poz);
   rewind(stream);
     /* A fájl elejére pozicionálás másik módja:
   fseek(stream, SEEK_SET, 0);       */
   poz=ftell(stream);
   fprintf(stdout,"\n Az első rekord bájt pozíciója: %ld\n",poz);
   fprintf(stdout,"\n A rekordok:\n");
   while(!feof(stream))
   {
     rdb=fread(&diak,sizeof(ODIJ),1,stream);
     if(rdb==1)     kiir(&diak);
     else if (rdb!=0)  { fprintf(stderr,"Fájl olvasási hiba!");
                 return 1; }
      }
   fseek(stream,(long)-1*sizeof(ODIJ),SEEK_END);
   poz=ftell(stream);
   fprintf(stdout,"\n\n Az utolsó rekord bájt pozíciója: %ld\n",poz);
   fclose(stream);
   getch();
   return 0;
   }
```

Alacsony szintű fájlkezelés

Csak UNIX szabvány!
Az egyes szintű fájlkezelést elsősorban bináris adatok elérésére használjuk (nincs konverzió!).
Itt is megnyitáskor kell megadni, hogy bináris vagy szövegmódú-e a feldolgozás.
(Szövegmódú: ENTER $\leftrightarrow$ RC, LF konverzió)
A fájlkezelés során számos hiba léphet fel $\rightarrow$ mindig figyelni kell, hogy volt-e hiba, és meg kell vizsgálni mi volt az!

Az egyes szintű fájlkezelés legfontosabb függvényei

open ( )	létező fájl megnyitása
creat ( )	nem létező fájl létrehozása, a régi tartalmának törlése $\rightarrow$ újként való megnyitás
read ( )	olvasás
write ( )	írás
lseek ( )	a FP módosítása $\rightarrow$ a fájl tetszőleges bájtjának elérése
close ( )	a fájl lezárása

Standard fájlok kezelése

Olvasás	Írás	Adategység
getchar( )	putchar( )	karakter
gets()	puts()	sztring
scanf()	printf()	formázott

Sztring beolvasása a standard inputról:

Eddig korrekt sztring beolvasást csak karakterenkénti bevitellel tudtunk megvalósítani, mert a gets() függvénynek nem lehet megadni a karakter számot, a scanf() függvény pedig nemcsak enter esetén fejezi be a bevitelt, hanem szóköznél is. A megoldás az fgets() általános fájlkezelő függvénnyel valósul meg.

```
fgets(s,n,fp);
   char * s;        // a sztring kezdőcíme, ahová olvasunk
   int n;           // a karakterek maximális száma, beleértve a záró 0-t is
   FILE * fp;       // a fájl azonosító
```

```
fgets(s, 20, stdin);
```

Legfeljebb 19 karakter beolvasása a klaviatúráról ENTER-ig vagy EOF-ig az *s* című sztringbe.
Az fgets() függvény az újsor karaktert is elhelyezi és mindig a végére teszi a '\0'-t!

Algoritmusok megvalósítása C programmal

```
#include <stdio.h>
#include <conio.h>
long osszegzes (int a[], int n);
int lin_ker (int a[], int n);
int minimum (int a[], int n);
int szamlal (int a[], int n);
void buborek (int a[], int n);
void min_elv (int a[], int n);
void main()
{ int t[10], i, w, eredm;
  long summa;
  char rend;
  clrscr();
  printf("\nAlgoritmusok megvalósítása C programmal\n");
  printf("\n Kérek 10 egész számot!\n");
  for(i=0; i<10; i++)
    { printf("%d. elem :",i+1);
      fflush (stdin);
      w=scanf("%d",&t[i]);
      if (w==0)  i--;               }
  clrscr();
  printf("\nAlgoritmusok megvalósítása C programmal\n");
  printf("\nA tömb elemei:\n");
  for(i=0; i<10; i++)
    { printf("%6d",t[i]);  }

  summa=osszegzes(t,10);
  printf("\nAz elemek összege: %ld",summa);
  eredm=lin_ker(t,10);                              //párosakat keres!
  if (eredm==-1)       { printf("\nNincs páros!");  }
  else          { printf("\nAz első páros sorszáma: %d", eredm+1 );}
  eredm=minimum(t,10);
  printf ("\nA legkisebb érték: %d",eredm);
  eredm=szamlal(t,10);                              //nullára végződőket számlál!
  printf("\n%d db végződik nullára",eredm);

  do{
    printf("\n Válasszon rendezési módot!");
    printf("\n Buborék rendezés: 1, Min. elvű: 2  ");
    fflush (stdin);
    rend=getchar();
     } while (rend<'1' || rend>'2');
   if (rend=='1')  { buborek (t,10); }
   else    { min_elv(t,10); }

  printf("\nA tömb elemei növekvő sorrendben:\n");
  for(i=0; i<10; i++)
     { printf("%6d",t[i]); }
   getch ();
}                  // main() vége
```

```
long osszegzes(int a[],int n)
{ long s=0;
   int i=0;
  while (i<n)
   { s=s+a[i];  i++;}
  return s;
  }

int lin_ker(int a[], int n)                //páros
{ int i=0, sorszam=-1;
  while (i<n && a[i]%2!=0)
     { i=i+1; }
  if (i<n) { sorszam=i;}
  return sorszam;
   }

int minimum(int a[], int n)
{  int i, min;
   min=a[0];
   for(i=1; i<n; i++)
      { if (min>a[i])     { min=a[i]; }
      }
   return min;
 }

int szamlal(int a[], int n)                //nulla végűeket
{ int db, i;
  for (db=0,i=0; i<n; i++)
      { if (a[i]%10==0) { db++;  }
       }
  return db;
  }

void buborek(int a[], int n)            //növekvő
{ int j, w, csere;
   do { csere=0;
    for(j=0; j<n-1; j++)
          { if(a[j] > a[j+1] )  { w=a[j], a[j]=a[j+1], a[j+1]=w;
                      csere=1; }
       }
   }while(csere);      }

void min_elv(int a[], int n)
{ int i, j, w;
    for(i=0; i<n-1; i++)
      { for(j=i+1; j<n; j++)
          { if(a[i] > a[j] )
              {  w=a[i], a[i]=a[j], a[j]=w;}
          }
       }
 }
```