Pointer, cím operátor

Alapfogalmak:

Pointer- indirekció operátora: `*`

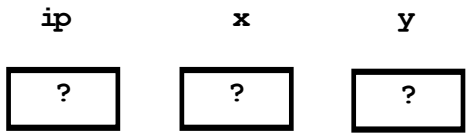
Cím meghatározásának operátora: `&` **(ampersand)**

Pl.:

```
int *ip, x, y;
```

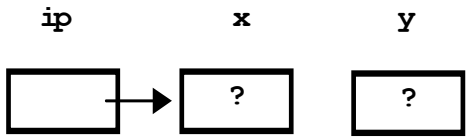
`ip` :`int`-re mutató pointer,

értéke még meghatározatlan!!!



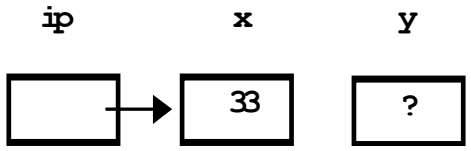

```
ip = &x;
```

`x` címét `ip`-be tölti




```
*ip = 33;
```

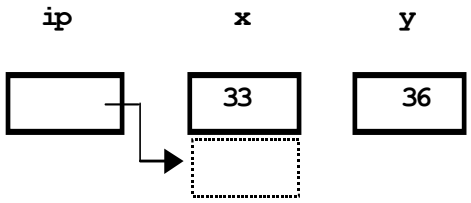
`*ip` értelme: az `ip` által mutatott egész, tehát ebbe (most: `x`-be) 33-at tesz




```
y = *ip++ +3;
```

Az `ip` által mutatott érték+3 -at `y`-ba teszi, majd `ip`-t egy int-nyivel megnöveli,

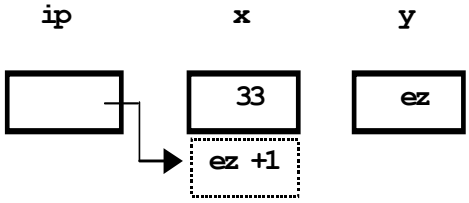
de NEM TUDJUK, MI VAN OTT !!!




```
y = (*ip)++;
```

Az `ip` által mutatott értéket `y`-ba teszi, majd az `ip` által mutatott értéket 1-el megnöveli,

de MIT NÖVELT MEG???



EZ TIPIKUS VÉGZETES HIBA: ha pointer rossz helyre mutat, attól még (bátran) végrehajtja a műveletet, és megváltoztat valamit, DE MIT?

Az inicializálás nem értékadás: az inicializálás a pointer értékét állítja be, nem a mutatott értéket (szerencsére, hiszen még nem is mutat sehova), bár a `*` operátor precedenciája nagyobb, mint a `=` precedenciája, de ez a `=` nem az értékadás, hanem az inicializálás jele. Pl.:

`double d1=0, d2, *dp=&d2;` : `dp` mutató `d2`-re mutat, de:

`*dp = &d2;` : HIBÁS, mert itt `=` az értékadást jelenti;
`*dp` típusa `double`, `&d2` típusa `double` pointere

A pointer-érték, ami nem mutat sehová: `NULL` (tradícionális C)
vagy: `0` (ANSI)
:minden pointer típussal kompatibilis, FALSE értékű

Tipp inicializálatlan pointerrel történő hivatkozás korai elcsípésére: MINDEN pointert NULL-ra inicializálunk: `char * cp = NULL;`
Így értelmes operációs rendszer, (tehát nem DOS) alatt a mutatott értékre hivatkozás megöli a programot: azonnal kiderül a hiba.

Pointer-tömb rokonság != azonosság

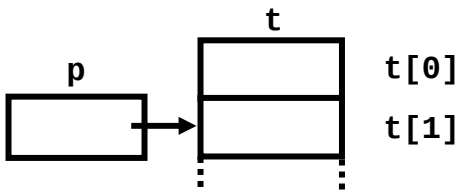
Pl.:

```
double  t[100],  *p;
```

`[ ]` prioritása nagyobb, mint `&` -é, ami most az értelmezést, nem pedig a végrehajtási sorrendet szabja meg:

`p = & t[0];` : `p` mutató a `t` tömb első elemére mutat: `*p ≡ t[0]`

`p++;` ≡ `p = p+1;` azaz `p`-t egy `double` -nyi értékkel növeli meg, `p` most `t[1]` -re mutat: `*p ≡ t[1]`



`p = &t[0] +6;` `p` most `t[6]` -ra mutat: `*p ≡ t[6]`

`p = &t[0];` `p` ismét `t` elejére mutat: `*p ≡ t[0]`

Ekkor:
```
*(p+0) ≡  t[0]
*(p+1) ≡  t[1]
*(p+x) ≡  t[x]        sőt a címeik is azonosak:
  p+x  ≡ &t[x]
```

A C nyelvben ezért logikus definíció: legyen a tömb nevének egy további jelentése:

tömb neve ≡ a tömb elsõ elemére mutató konstans pointer érték

A compiler is `t[x]` -et először `*(t+x)` alakra alakítja, majd ezt dolgozza fel:

`t[x]` ⇒ `*(t+x)` és így `&t[x]` ⇒ `t+x`

Ebből következik, hogy **p[x] ≡ *(p+x)** azaz pointert is használhatunk úgy, mintha tömb neve lenne, hiszen **p[x]** -et a compiler úgy tekinti, mintha az ***(p+x)** lenne.

Hasonlóan: pointert tömb elejére, vagy tetszőleges elemére az **&** cím-operátor és indexelés nélkül is beállíthatunk:

p = t; **p** most **t** elejére mutat, hiszen:

&t[0] ≡ &(*(t+0)) ≡ &(*t) ≡ &*t ≡ t

p = t+z; **p** most **t**-nek **z** indexű elemére mutat

További következmények:

⇒ a tömbhivatkozás pointer-hivatkozássá alakítása miatt nincs is lehetőség az indexhatárok ellenőrzésére !!! (No, azért lenne, de nem egyszer•.)

⇒ nincs tömb-értékadás, pl.:

```
char ct1[10], ct2[10];
```
: azonos típusúak

```
ct1=ct2;
```
: szintaktikai hiba: a **ct2** terület kezdetére mutató pointert másolná át **ct1**-be, és nem a terület tartalmát, ha **ct1** nem konstans pointer lenne.

(Tömb másolását pl. a **memcpy** ill. **memmove** függvénnyel végezhetjük.)

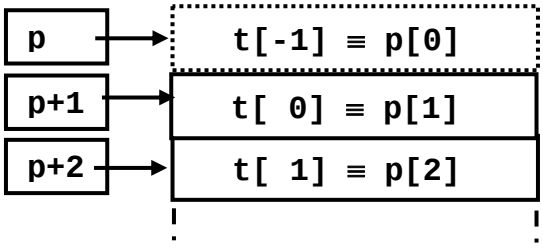
⇒ Ha két pointer ugyanazon tömb elemeire mutat, akkor a pointerek különbsége megadja a két elem indexének különbségét, pl.:

```
int t[100], *ip=t;
      ...                    : t feltöltése valamivel
while (*ip) ip++;            : első 0 megkeresése t-ben, ha nincs, nulla elem
                               t-ben, akkor is megkeresi ⇒ t után !!!
if (ip-t > 15) ....;         :legalább 16 indexű a 0 elem
```

Egy hasznos alkalmazás: ha nem 0-tól akarunk indexelni egy tömböt, hanem pl. 1-től, akkor pl.:

```
int  t [....],  *p = t-1;
```

: **p** mutató **t** első, azaz 0 indexű eleme *elé* mutat:



p[1] ≡ *(p+1) ≡ *((t-1)+1) ≡ *t ≡ t[0]

tehát **p** segítségével 1-től indexelhető **t** tartalma.

TIPIKUS, HALÁLOS HIBA: 

`double *p;`	:p tartalma meghatározatlan, tehát bárhová mutathat
`....`	:p nem kap értéket
`*p = 11;`	:oda teszi le az értéket, ahová **p** véletlenül mutat: adatokba? Stack-be? Gépi utasításokba? Operációs rendszerbe?
`p[5] = 1.1;`	:ez 5 **double**-el arrébb, de hova???
`p = NULL;`	:**p** mostmár nem mutat sehova
`*p = 1;`	:tipikusan a 0 címre teszi: IT vektorba? Első adatba?

Egy tanulságos példa: gyorsrendezés tömbbel és mutatóval:
File : **c-pelda.zip**-ben **QSort-px.c**

```
/*----- Quick-sort with indices -----*/

void QSort_x (int a[], int n)
{  int left, right, ref, w;

   for (left=0,  right=n-1, ref=a[n/2];
        left<=right;  )  {
     while (a[left] < ref)  left++;
     while (ref < a[right]) right--;
     if (left <= right) {
       w = a[left];  a[left++] = a[right];
       a[right--] = w;
     } /* if */
   } /* for */
   if (0 < right)   QSort_x (a, right+1);
   if (left < n-1)  QSort_x (&a[left], n-left);
} /* Qsort_x() */


/*----- Quick-sort with pointers -----*/

void QSort_p (int *a, int n)
{  int *left, *right, ref, w;

   for (left=a,  right=a+n-1, ref=*(a+n/2);
        left<=right;  )  {
     while (*left < ref)  left++;
     while (ref < *right) right--;
     if (left <= right) {    /* swap and update */
       w = *left;  *left++ = *right;  *right-- = w;
     } /* if */
   } /* for */
   if (a < right)     QSort_p (a, right-a+1);
   if (left < a+n-1)  QSort_p (left, a+n-left);
} /* Qsort_p() */
```

Results: Pentium 100 MHz, BC 3.1.

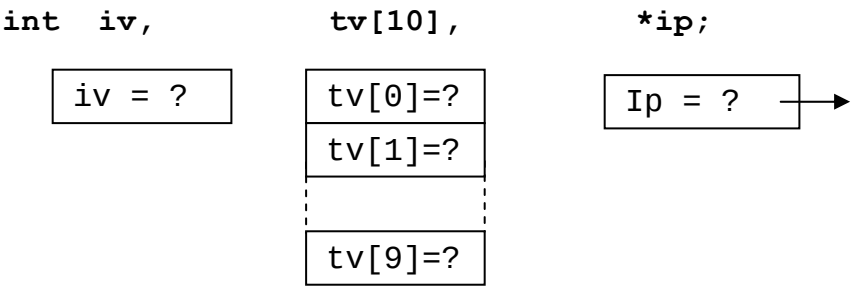
Memory model = Small:

```
Time for pointer-method = 1.706 [msec] =  100.0 %
Time for index  -method = 2.509 [msec] =  147.0 %
```

A lényeges különbség oka az, hogy az adott tömbelem eléréséhez először meg kell határozni annak címét; indexelés esetén ehhez az indexet meg kell szorozni egy tömbelem hosszával, majd ezt hozzá kell adni a tömb címéhez., elem-mutatók esetén pedig a szorzás és az összeadás nem kell.

Pointer ≠ tömb

A különbség lényege a helyfoglalás:



Pl. BC 3.1, Memory model = Huge:

```
sizeof(iv)  ≡ 2  ≡ sizeof(int)
sizeof(tv)  ≡ 20 ≡ 10 * sizeof(int)
sizeof(ip)  ≡ 4
sizeof(*ip) ≡ 2  ≡ sizeof(int)
```

`sizeof` operátor:

Megadja az adott objektum ≡ érték, illetve típus tárolásához szükséges memória méretét. Az eredmény típusa **`size_t`**, ami valamilyen előjel nélküli egész, tipikusan **`unsigned int`**, vagy **`unsigned long int`**.

`size_t` nem külön típus, definícója pl. **`<stddef.h>`** -ban van, pl.:

```
typedef unsigned size_t;
```

Típushoz zárójellel kell használni, objektumhoz nem kell zárójel, de lehet. Pl.:

`sizeof(char)` ez mindig 1

`sizeof(int*)` **`int`**-re mutató pointer mérete (a különböző típusokra mutató pointerek mérete ugyanaz szokott lenni)

```
int v;
```

`sizeof v` ≡ **`sizeof(int)`** (A változó neve köré nem kell a zárójel, de lehet)

```
short x;
```

`sizeof(x+1)` ≡ **`sizeof(int)`** mert bármilyen kifejezésben **`short`** típusú értéket először **`int`** –é alakítja

Nem függvény, hanem operátor, értékét a C program fordítása során a fordítóprogram határozza meg, így argumentuma nem hajtódik végre,

⇒ annak mellékhatásai sem történhetnek meg. Pl.:

```
meret = sizeof ( a=b++ )
```

`meret` megváltozik, **`a`**, **`b`** nem.

Függvények alapjai

- tagolásra, struktúráltság növelésére
- külső, könyvtári függvények használatára

Definíció ≠ Deklaráció

Függvénydefiníció alakja:

típus név (formális paraméterek)
blokk

Pl.:

```
double  LegkisebbTombben  (double t[],  int n)
{  int x;
   double min;

   for (min = t[0], x=1;   x < n;   x++)
      if (t[x] < min)   min = t[x];
   return min;                  /* visszatérés és érték megadása */
} /* LegkisebbTombben () */
```

Hívása pl.:

```
main ()                     /* ez is egy speciális függvény ... */
{    ...
   #define tombmeret 2000
   double dt [tombmeret], mm;
   int x;
   ...
   x= ....;
   mm = LegkisebbTombben (dt,x*2-1) +3;   /* fv. hívása */
   ...
}
```

Függvénydeklaráció alakja:

típus név **(** formális paraméterek **) ;**

azaz csak a függvényfejet adjuk meg, utána pontosvessző.
Ezt nevezik fv. prototípusnak is. Pl.:

```
double Egyik (int p1,  double p2);
```

:megadja azokat az információkat, melyek a függvény hívásához kellenek: függvény neve, visszatérési típusa, paraméterek száma és típusa, de nem adja meg a függvény által végrehajtandó utasításokat.

Felhasználása: külön fordított függvények hívási módjának megadására és "forward"-olásra, azaz pl. közvetett rekurzióhoz:

```
double Egyik (int p1, double p2);
```
; van a végén: deklaráció

```
void Masik (char *p1, unsigned t[])
{ ....
  if (Egyik( .....) >0) ....
                        ....
} /* Masik() */
```
: definíció, mert blokk követi:
: **Egyik** hívása

```
double Egyik (int p1,  double p2)
{  ....
   Masik ( ..... );
   ....
} /* Egyik() */
```
: **Egyik** definíciója
: **Masik** hívása

C-ben csak értékparaméter van!

Visszaadott érték:

- függvény visszatérési értékével
- globális, azaz file-szintű változóval
- pointer formális-, cím aktuális argumentum segítségével:

Pl. tömbben legkisebb és legnagyobb elem meghatározása:
nincs visszatérési értéke:

```
void  MinMax (double t[], int n, double *minp, double *maxp)
{  int x;

   *minp = *maxp = t[0];  /* első elem értéke a minp és maxp
                             által mutatott helyekre kerül */
   for (x=1;  x<n;  x++)  {
      double tx=t[x];           /* blokk lokális változója */
      if (tx < *minp) *minp=tx;  else
      if (tx > *maxp) *maxp=tx;
   } /* for x */
   return;                      /* itt elhagyható */
}
```

Hívása pl.:

```
ff (....)                          :egy másik függvény, MinMax után
{
   double dd [....], min, max;
   ...
   MinMax (dd, 100, &min, &max);   :dd, min és max címét adja át,
                                    az eredmény min-be és max-ba kerül
   ...
} /* ff() */
```

`MinMax()` végrehajtása alatt:

Megjegyzés: a pointer-tömb rokonság miatt a függvény feje lehetne ilyen is:

```
void  MinMax (double *t, int n, double *minp, double *maxp)
```

és ettől függetlenül akár ez után, akár az előző, **`double t[ ]`** -t tartalmazó függvényfej után a függvény törzse lehetne akár ilyen, akár az előző is:

```
{  int x;
   *minp = *maxp = *t++;       /* t pointert növeljük */
   for (x=1;  x<n;  x++, t++)
      if (*t < *minp)  *minp=*t;  else
      if (*t > *maxp)  *maxp=*t;
}
```

Bár az előző függvényfejben a **`t`** paraméter tömb, azaz konstans pointer lenne, ha ez függvény paramétere, mindenképpen sima pointerként viselkedik, azaz megváltoztatható! Ennek az az oka, hogy erre is vonatkozik: az (érték)paraméter értéke a függvényben megváltoztatható, kivéve, ha konstansnak (**`const`**) definiáljuk)..

Ugyanez az “elcserélt fejek” igaz az előzőleg látott Quick-sort-ra is.

Mutató mutatója ≡ Mutatóra mutató mutató

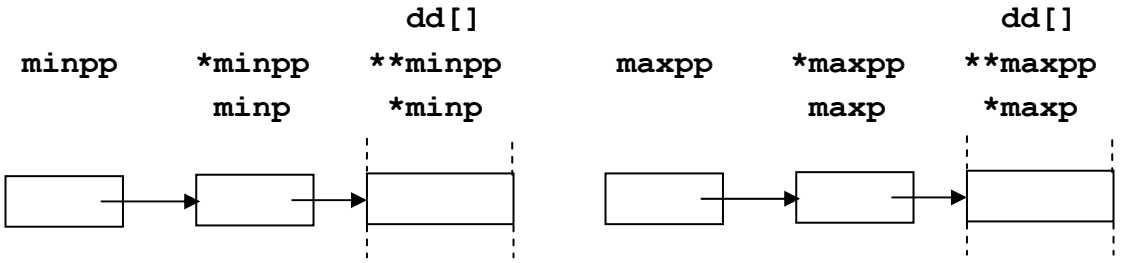
(pointer pointere ≡ pointerre pointoló pointer)

Feladat: tömbben legkisebb és legnagyobb elem CÍMÉNEK meghatározása:

Először a hívását lássuk: azt akarjuk, hogy a függvény hívása után **minp** ill. **maxp** a **dd** tömb legkisebb ill. legnagyobb elemére mutasson:

```
ff2 (....)
{
   double dd [....], * minp, * maxp;
   ...
   MinMaxCim (dd, 100, & minp, & maxp);
                /*: minp és maxp címét adjuk át, hogy az eredmény
                   minp-be és maxp-be kerülhessen */
...
} /* ff2() */


void  MinMaxCim (double t[], int n,     /* mint előbb */
                 double ** minpp, double ** maxpp)
                    /* pointereket akarunk beállítani, ezért
                       pointerre mutató  pointerek a paraméterek */
{  int x;
   *minpp = *maxpp = t;   /* első elem címe a minpp és maxpp
                             által mutatott helyre kerül */
   for (x=1;  x<n;  x++)  {
      if (t[x] < **minpp) *minpp=t+x;     /* a címet tesszük el */
      else
         if (t[x] > **maxpp) *maxpp=t+x;
   } /* for x */
} /* MinMaxCim() */
```



Függvények további szabályai

A visszatérési érték alapértelmezése int, lehet pointer, de nem lehet függvény vagy tömb. Pl.:

```
ff (float f) ....   ≡  int ff (float f) ....
```

Függvény csak file-szinten definiálható, blokkban lokális függvény nem definiálható, de deklarálkató, pl.:

```
int f2 (double p) {                    : file szintű definíció, benne:

   void f3 (float fp);                 : lokális fv. Deklaráció: OK.

   int Negyzet (int x)
      {  return x*x; }                 : HIBA: fv. definíció csak file szinten lehet
   ....
   f3 (3.2*f1("szöveg"));              : f3 és f1 hívása
}

char *f4 (void)  {                     : paraméter nélküli fv.
   ....
   f3(4.1F);                           : HIBA: f3 itt nem ismert
   ....
}

void f3 (float fp) { ....}             : f3 definíciója
```

Egymást követő, azonos típusú formális paraméterek típusát külön-külön ki kell írni, pl.:

```
void fx (int p1, p2)          HIBÁS!!!
void fx (int p1, int p2)      Helyes
```

Paraméter nélküli fv. definiíciója ill deklarációja pl.:

```
char * f4 (void) ....
```
: paraméter nélküli fv.

Ennek hívása:

```
.... f4() ....
```
: ki kell írni: **()**

Deklarációban a paraméternevek elhagyhatóak. pl.:

```
double Egyik (int, double);
```
: paraméter név itt nem fontos, de definícióban igen

Tradicionális C-ben a definíció alakja pl. (ANSI-ben is jó elvileg, de nem ajánlott):

```
double Egyik (p1, p2, p3)          : csak paraméternevek
  double p2;                       : a sorrend mindegy
  int p1;                          : elhagyható, akkor int
{    .... }
```

Deklaráció alakja csak prototípus lehet:

```
void f5 (int p1, char p2);
```

Rossz: `void f5 (p1,p2);  int p1; char p2;`

Függvényekről lesz majd még: Változó ill. definiálatlan számú és típusú paraméterek

Változók inicializálása

Definíciójuknál kezdeti érték is megadható, alakja pl.:

```
int iv1=1, iv2=5*1024, iv3, iv4=-1;
double dv1, dv2=3.1415, dv3=0.0;
```

Hogy mi lehet a kezdeti érték, az függ a tárolási osztálytól:

Tárolási osztály	Kezdeti érték
static **extern**	konstans kifejezés
automatic **register**	tetszőleges kifejezés

1-indexű tömb kezdeti értéke: **{** és **}** között, vesszővel elválasztva:

```
int t[10]   = {3,4,5,6,7,8,9};          : a többi eleme 0 lesz
double d[3] = {1.1, 2.2, 3.3, 4.4};     : HIBA: túl sok érték
float f[]   = {1.4, 2.3, 3.5, 4.1};     : 4 elemű lett a tömb
char ct[4]  = {'a', ,'b','c'};    : HIBA: nem szabad  közben kihagyni
```

(Többindexű: lásd később.)

`f` elemszáma pl. így vehető elő a későbbiekben:

a) Gyengébb megoldás: **`sizeof(f) / sizeof(float)`**

A gond akkor van, he megváltozik az elemtípusa, pl. double-ra, és persze, elfelejtjük a fenti kifejezést is módosítani

b) Jobb megoldás: **`sizeof(f) / sizeof(*f)`**

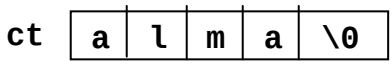
String típus

```
char ct [5] = {'a','l','m','a', '\0'};
            = "alma";
```

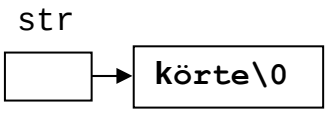
: ez nehézkes

: ez ugyanaz, de könnyebb

Memóriában a végét 0 értékű byte jelzi:



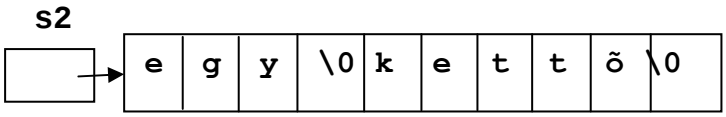

```
char *str="körte";
```



: 5+1=6 byte, végén 0;
az **str** pointer-változó a **k** betűre mutat

```
char *s2="egy\0kettő";
```

: 3+1+5+1 byte, de minden string-kezelő függvény csak 3+1 byte-osnak tekinti




```
char s[] = "autom. hossz";
```

: 12+1 elemű tömb lett

String-konstans így folytatható új sorban:

Tradícionális C változat:

```
              .... "eleje itt van, de ha túl hosszú lenne,\
folytatás a sor elején";
```

ANSI C ezt is ismeri:

```
     "eleje "           /* itt komment is lehet */
    /* itt is */    "folytatás bárhol";
```

Makró argumentuma is stringbe illeszthető:

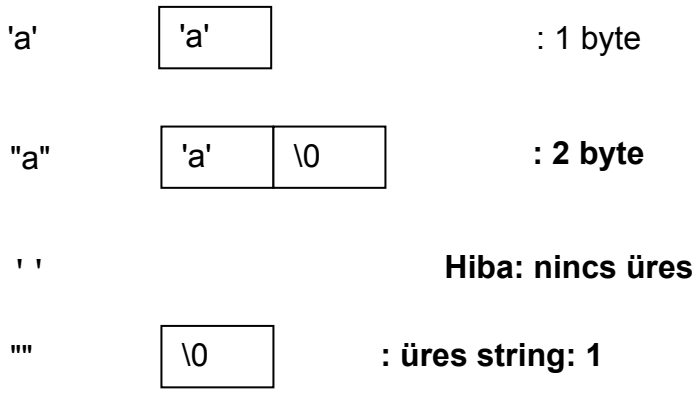
```
#define m(p1)     fun ( "Ez marad p1,"                    \
          " ide aktuális p1 kerül: ", p1,                 \
          " ide stringként helyettesítődik be:", #p1,     \
          " ezzel egyesíti is:" #p1 );

m (abcd)
```

Kifejtve, stringeket egyesítve:

```
fun ( "Ez marad p1, ide aktuális p1 kerül: ", abcd,
      " ide stringként helyettesítődik be:", "abcd",
      " ezzel egyesíti is: abcd" );
```

Nagy különbség van **`char`** és string között:



Néhány gyakran használt könyvtári string-kezelő függvény működése
Megtalálhatóak **<string.h>** -ban:

```
int strcmp (char *op1,  char *op2)    /* összehasonlítás */
  {  while (*op1 && *op1 == *op2)  { op1++;  op2++; }
     return *op1-*op2;
  }
```

Hogy minek erre külön függvény?

```
     char s1[100],  s2[200];
```

```
     if (s1 == s2) ...
```

: címeket hasonlít össze !!!

```
char * strcpy (char *dest, char *source)  /* másolás */
  {  char *d=dest;
     while (*d++ = *source++);    /* BÁTRAN másol */
     return dest;
  }
```

```
size_t strlen (char *str)          /* string hossza */
  {  char *s =str;
     while (*s++);       /* amíg 0 byte-ot nem talál */
     return s-str-1;     /* karakterek száma \0 előtt */
  }
```

```
char *strchr (char *str,  char ch)  /* ch címe str-ben */
  {  while (*str && *str != ch)  str++;
     if (*str == ch)  return str;   /* megvan */
     return NULL;                   /* nincs meg */
  }
```

HF: String másolása, átlapoltakat is helyesen kezeli:

```
char * strmove (char * dest,  char * src);
```

Program paraméterek

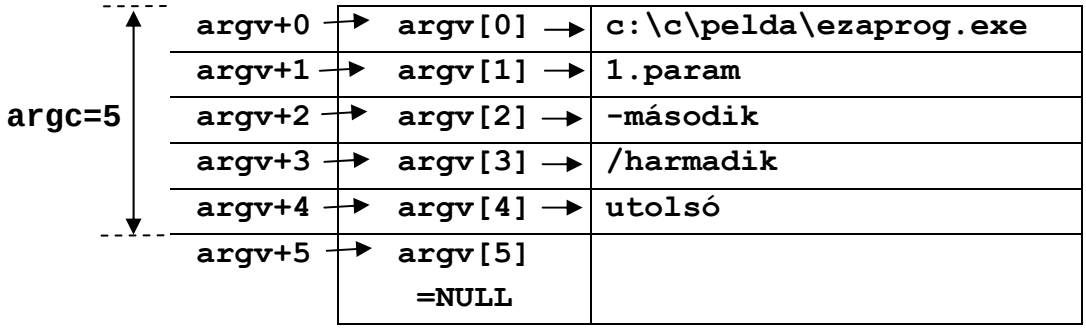
A program, azaz a **`main`** függvény argumentumaiként a program neve, paraméterei és azok száma megkaphatóak a következő formában.
Pl. ha egy programot így indítunk:

```
c:\c\pelda> ezaprog 1.param -második /harmadik utolsó <bemf.dat
```

akkor a paraméterekhez így férhetünk hozzá:

```
int main ( int    argc,        : program-argumentumok száma
           char * argv [] )    : argumentum-stringekre mutató
                                 pointerek tömbje
{    ....
```

Itt használhatóak a program argumentumok:



A szabványos input és a szabványos output átirányítása:

Pl.:

```
prog   <bemfile.dat   >kimf.ere
```

Ez NEM program-paraméter.

Lásd még pl. a **`ProgArgs.c`** mintaprogramot!