C programozás

9 óra
Tömbök, mutatók, sztringek
memóriahasználat, kommunikáció az OP-rel

1. Tömbök

A C nyelvben szoros kapcsolat van a mutatók és a tömbök között. Ez indokolja, hogy a mutatókkal és a tömbökkel egyidejûleg foglalkozzunk. Valamennyi mûvelet, amely tömbindexeléssel végrehajtható, mutatók használatával éppúgy elvégezhetõ.

Az

int a [10]

deklaráció definiálja azt a tömböt, amelynek mérete 10, vagyis egy tíz, egymást követõ objektumból, az a[0], a[1], . . ., a[9] nevû elemekbõl álló blokkot határoz meg.

Az a[i] jelölésmód a tömbnek a kezdettõl számított i-edik pozícióját fejezi ki. Ha pa egészt megcímzõ mutató, amelyet

int *pa

deklarál, akkor a

pa =&a[0]

értékadás úgy állítja be pa-t, hogy az a nulladik elemére mutasson, vagyis pa az a[0] elem címét tartalmazza. Ekkor az

x = *pa

értékadás a[0] tartalmát x-be másolja.

Ha pa az a tömb adott elemére mutat, akkor definíció szerint pa + 1 a tömb következõ elemére mutat. Általában pa - i elemmel pa elé, pa + i pedig i elemmel pa mögé mutat. Így ha pa az a[0]-ra mutat, akkor

***(pa + 1)**

a[1] tartalmát szolgáltatja, pa+i az a[i] elem címe és *(pa+i) az a[i] elem tartalma.

Ezek a megjegyzések az a tömbben elhelyezkedõ változók típusától függetlenül mindig igazak. Az "adj 1-et a mutatóhoz" és ennek kiterjesztéseként az egész mutatóaritmetika alapdefiníciója, hogy a növekmény mértékegysége annak az objektumnak a tárbeli mérete, amire a mutató mutat. Így pa+i esetében a pa-hoz hozzáadás elõtt i azoknak az objektumoknak a méretével szorzódik, amire pa mutat.

Az indexelés és a mutatóaritmetika között láthatóan nagyon szoros kapcsolat van. Gyakorlatilag a tömbre való hivatkozást a fordító a tömb kezdetét megcímzõ mutatóvá alakítja át. Ennek hatására a tömb neve nem más, mint egy mutatókifejezés, amibõl számos hasznos dolog következik. Mivel a tömb neve ugyanaz, mint az illetõ tömb nulladik elemének címe, a

pa = &a[0]

értékadás úgy is írható, mint

pa = a

Legalábbis elsõ ránézésre még meglepõbb az a tény, hogy az a[i]-re történõ hivatkozás *(a+i)-ként is írható. a[i] kiértékelésekor a C fordító azonnal átalakítja ezt *(a+i)-vé; a két alak teljesen egyenértékû. Ha az ekvivalencia mindkét elemére alkalmazzuk az & operátort, akkor azonnal következik, hogy &a[i] és a+i szintén azonosak: a+i az a-t követõ i-edik elem címe. Az érem másik oldala viszont az, hogy ha pa mutató, akkor azt kifejezések indexelhetik: pa[i] azonos *(pa+i)-vel. Röviden, bármilyen tömb vagy indexkifejezés leírható, mint egy mutató plusz egy eltolás és viszont, akár egy utasításon belül is.

Van azonban egy fontos különbség a tömbnév és a mutató között, amire ügyelnünk kell. A mutató változó, így pa=a és pa++ értelmes mûveletek. A tömbnév azonban állandó, nem pedig változó: az olyan konstrukciók, mint a=pa vagy a++, vagy p=&a nem megengedettek!

Pointer-tömb rokonság != azonosság

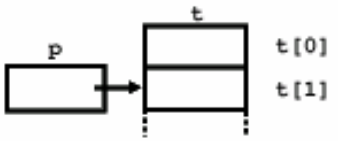
Pl.:

```
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.)

Több indexű tömbök

A több dimenziós tömbök igazából tömbök tömbjei, pl:

```
double dt [10] [20];        : 2-indexű, nem dt [10,20]
int t3 [5][44][10];         : 3-indexű (akárhány index lehet)
double * dp;
        ...
dt[2][3-z] = 34.56;         : ez sem írható így:  dt[2,3-z]
dp=dt[5];                   : dt 5. sorának címét teszi dp-be,
```

mert ha csak az első indexet adjuk meg, az eggyel kevesebb indexű tömböt ad; itt éppen 1-indexűt

A memóriában a több indexű tömb utolsó indexe változik a leggyorsabban, azaz pl. 2-indexű tömb sorfolytonosan tárolódik:

```
dt[0][0]  dt[0][1]   ...  dt[0][19]  dt[1][0]  dt[1][1]
...
```

így tehát pl. `dt[0][20] = dt[1][0]`

```
t3[0][0][0]  t3[0][0][1]  ...  t3[0][0][9]  t3[0][1][0]
t3[0][1][1]  ... t3[0][1][9]  t3[0][2][0] ...
t[30][2][9]
... t3 [0][43][9]  t3 [1][0][0]  ...  t3 [4][43][9]
```

Függvény paraméterénél az első indexhatár hagyható el, a többi kell az adott tömbelem címének meghatározásához:

```
void ff (int it [] [20][50],  ..... )
{
    if (it [x1][x2][x3] >= 0) ....
    ....
}
```

Pl. itt `it[x1][x2][x3]` **címe :**

```
(char*) it + sizeof(int) * ( 20*50*x1 + 50*x2 + x3)
```

Kezdeti érték megadása soronként saját `{}` között, vagy sorfolytonosan is lehet:

```
int t2d [5][4] = { {11,12,13,14},      : 0. sor
                   {21,22},            : 1. sor eleje, majd 0
                   {},                 :HIBA: nem lehet üres
                   {41,42,43} };       :  3. sor eleje, az
                                       összes többi elem 0

float f2d [2][3] = {1.1, 2.2, 3.3, 4.4};   :sorfolytonosan,
                 = {{1.1, 2.2, 3.3},       :mintha ez lenne
                    {4.4} };               :a többi 0.0f
```

Tömb eleme bármi lehet, kivéve függvény,

de lehet függvényre mutató pointer, pl.:

```
double f1 (int,char);
double f2 (int,char);                  :  két függvény

double (*fpt[10]) (int,char) = { f1, &f2 };
```

`fpt` :10 elemű, pointerekből álló tömb, melynek elemei `double` visszatérési értékű, `int` és `char` argumentum-típusú függvényekre mutathatnak, az első kettő kapott kezdeti értéket

Vegyük észre, hogy a függvény neve `()` nélkül a függvény címét jelenti, de ki is tehetjük az `&` cím-operátort.

Ugyanez másként:

```
typedef double f_t (int,char);          :függvény típus
f_t * fpt [10] = { f1, &f2 };           :pointerek tömbje
```

De: `double *fpt[10] (int,char) = { f1, f2 };`

HIBÁS: a `()` -nek, mint a függvényt jelentő operátornak magasabb a prioritása, mint a * -nak, azaz az indirekció jelének, így ez 10 elemű tömböt jelentene, melynek elemei olyan függvények (:ez a hibás), melyek `double`-re mutató pointert adnak vissza.

Dinamikus tárkezelés:

`malloc(), calloc(), realloc(), free()`

könyvtári függvényekkel.

```
#include <alloc.h>
#include <stdlib.h>
```
vagy kell, ezekben:

Memóriafoglalás:

```
void * malloc (size_t s);
```

- ha tud, lefoglal `s` byte-os területet (az un. heap-ben),
  - ennek címét adja vissza, ha sikerült
  - terület tartalma meghatározatlan.
- ha nem tud, `NULL`-t ad vissza ⇒ mindig meg kell vizsgálni!
  (de legalább nem hal meg a program).

`void *` : definiálatlan típusú adatra mutató pointer: olyan pointer típus, amelynek bármilyen típusú ptr. értékül adható, de `void *` csak `void*` -nak, és nem hivatkozható az általa mutatott adat sem (hiszen mérete nem ismert), csak típus-konverzióval = cast -olással. Pl.:

```
void * vp;
char * cp;

vp = cp;            : O.K.
cp = vp;            : TILOS, de:
cp = (char*) vp;    : O.K.

*vp = 1;            :TILOS, nem tudjuk, mekkora *vp
*((int*)vp) = 1;    :O.K., de nem túl stílusos
```

`malloc` használata pl.:

```
int *dp, meret, x;
   ....
meret= .....;
   ....
dp = (int*) malloc (meret * sizeof(int));
if (NULL == dp)  {          /* lehetne  !dp is */
     fprintf (stderr, "\n Nincs elég memória \n");
     exit(1);
}

for (x=0;  x<meret;  x++) {
  dp[x]=....;          :mintha dinamikus méretű "tömb" lenne
  ...
}
```

Másik memóriafoglaló fv.:

```
void *calloc (size_t nitems, size_t s);
```

- **`nitems * s` méretű területet foglal,**
- **ha sikerül, ezt nullázza,**
- **ha nem sikerül, NULL-t ad vissza.**

Memória felszabadítása:

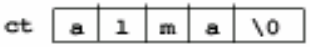
```
void free (void * block);
```

- **`malloc, calloc, realloc` által foglalt terület felszabadítása,**
- **csak az így foglalt terület elejét szabad neki átadni, közepét, stb. nem!**
- **méretet `malloc` (stb.) elteszi neki (célszrűen a terület elé), így azt ismeri, nem kell külön megadni `free`-nek**

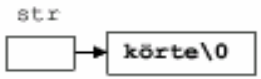
String típus

```
char ct [5] = {'a','l','m','a', '\0'};
            = "alma";
```
: ez nehézkes
: ez ugyanaz, de könnyebb

Memóriában a végét 0 értékű byte jelzi:

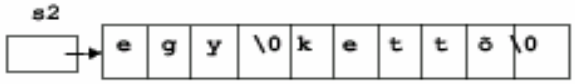



```
char *str="körte";
```



: 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:

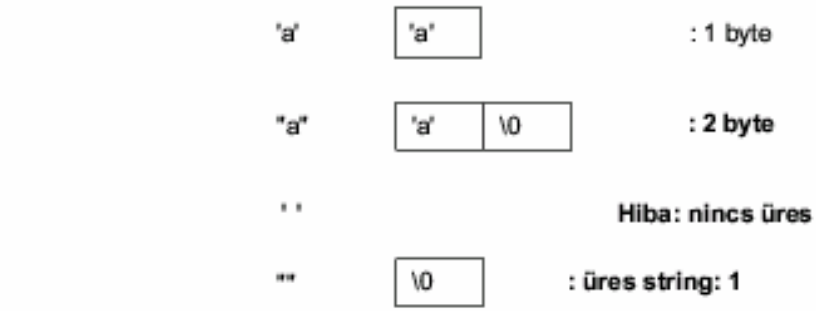
Tradícionális C változat:

```
              .... "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";
```

Nagy különbség van `char` és string között:



Kommunikáció az Operációs Rendszerrel

```
#include <stdio.h>

main()
{
char str[20]="-ls",command[20];

        printf("%s\n",str);
        sprintf (command, "ls %s",str);
      system (command);


}
```