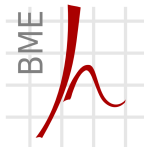


# Mutatók és tömbök. Sztringek. Keresés

## A programozás alapjai I.



Hálózati Rendszerek és Szolgáltatások Tanszék  
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2020. október 19.

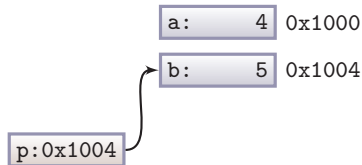
# Tartalom

- 1 Mutatók és tömbök
  - Mutatóaritmetika
  - Mutatók és tömbök
- 2 Sztringek

- Definíció
  - Kezelés
- 3 Keresés adatvektorban
    - Lineáris keresés
    - Logaritmikus keresés

## Mutatók – Ismétlés

```
1 int a, b;
2 int *p; /* int pointer */
3
4 a = 2;
5 b = 3;
6 p = &a; /* p a-ra mutat */
7 *p = 4; /* a = 4 */
8 p = &b; /* p b-re mutat */
9 *p = 5; /* b = 5 */
```



# 1. fejezet

## Mutatók és tömbök



# Megjegyzések:

- Miért jó, hogy különböző típusok címei különböző típusúak?
- Típus = értékészlet + műveletek
- Az értékészlet nyilván minden mutatóra ugyanaz (előjel nélküli egész címek)
- A műveletek eltérőek!
- Az indirekció (\*) operátor
  - `int` pointerből `int`-et
  - `char` pointerből `char`-t képez
- Egyéb műveletbeli különbségek a mutatóaritmetikában...

# Mutatóaritmetika

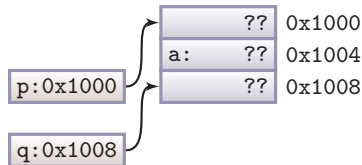
Ha  $p$  és  $q$  azonos típusú mutatók, akkor

kif.	típus	jelentés
$p+1$	mutató	a következő <u>elemre</u> mutat
$p-1$	mutató	az előző <u>elemre</u> mutat
$q-p$	egész szám	két cím közötti <u>elemek</u> számát adja meg

```

1 int a, *p, *q;
2
3 p = &a;
4 p = p-1;
5 q = p+2;
6 printf("%d", q-p);

```



2

- Mutatóaritmetikai műveleteknél a címeket nem bájtban, hanem a mutatott típus ábrázolási méretében mérjük<sup>1</sup>

<sup>1</sup>A példában feltételezzük, hogy `int` 4 bájtos



# Mutatóaritmetika

- A fenti példában a mutatóaritmetikának nincs sok értelme, hiszen nem tudhatjuk, mi áll az a változó előtt vagy mögött.
- A művelet ott nyer értelmet, ahol a memóriában egymást követő, azonos típusú változók helyezkednek el.
- Ezek a tömbök.

# Mutatók és tömbök

## ■ Tömb bejárása lehetséges mutatóaritmetika alkalmazásával

```

1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = &t[0];
5 for (i = 0; i < 5; ++i)
6     printf("%d ", *(p+i));
  
```

1 4 2 7 3

t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000



- Jelen példában  $*(p+i)$  megegyezik  $t[i]$ -vel, mert  $p$  a  $t$  tömb elejére mutat



# Mutatók és tömbök

- Mutatók tömbként kezelhetők, vagyis indexelhetők.

Definíció szerint

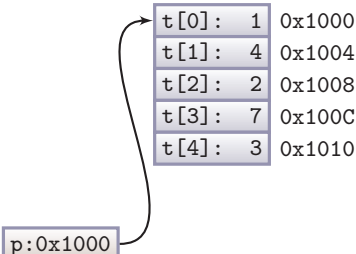
$p[i]$  azonos  $*(p+i)$ -vel

```
1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = &t[0];
5 for (i = 0; i < 5; ++i)
6     printf("%d ", p[i]);
```

1 4 2 7 3

t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000



- Jelen példában  $p[i]$  megegyezik  $t[i]$ -vel, mert  $p$  a  $t$  tömb elejére mutat

# Mutatók és tömbök

- Tömbök mutatóként kezelhetők.  
Tömb nevét írva a tömb kezdőcímét kapjuk meg, vagyis a `t` kifejezés értéke `&t[0]`

```


1 int t[5] = {1,4,2,7,3};
2 int *p, i;
3
4 p = t; /* &t[0] */
5 for (i = 0; i < 5; ++i)
6     printf("%d ", p[i]);

```

1 4 2 7 3

t[0]:	1	0x1000
t[1]:	4	0x1004
t[2]:	2	0x1008
t[3]:	7	0x100C
t[4]:	3	0x1010

p:0x1000



- A mutatóaritmetika tömbökre is működik:  
`t+i` azonos `&t[i]`-vel



# Mutatók és tömbök – összefoglalás

- Mutató kezelhető tömbként, tömb kezelhető mutatóként.
- Az index operátor csak egy jelölés  $a[i]$ -t a fordító mindig  $*(a+i)$ -vel helyettesíti, akkor is, ha  $a$  mutató, akkor is, ha  $a$  tömb.
- Különbségek:
  - A tömbelemeknek fenntartott tárhelyük van (változók). A mutatóhoz nem tartoznak foglalt elemek.
  - A tömb kezdőcíme konstans, nem változtatható. A mutató változó, a benne tárolt cím módosítható.

```
1 int array[5] = {1, 3, 2, 4, 7};
2 int *p = array;
3
4 /* az elemek p-n és a-n keresztül elérhetőek */
5 p[0] = 2;          array[0] = 2;
6 *p = 2;           *array = 2;
7
8 /* p változtatható   array nem */
9 p = p+1; /* jó */   array = array + 1; /* HIBA */
```



# Tömbök átadása függvénynek

- Határozzuk meg függvényel az array tömb első negatív elemét!
- Tömb átadása:
  - Első elem címe `double*`
  - Tömb mérete `typedef unsigned int size_t2`

```
1 double first_negative(double *array, size_t size)
2 {
3     size_t i;
4     for (i = 0; i < size; ++i) /* minden elemre */
5         if (array[i] < 0.0)
6             return array[i];
7
8     return 0; /* mind nemnegatív */
9 }
```

[link](#)

```
1 double myarray[3] = {3.0, 1.0, -2.0};
2 double neg = first_negative(myarray, 3);
```

[link](#)

<sup>2</sup>az `stdio.h` definiálja



# Tömbök átadása függvénynek

- Hogy a paraméterlistán elkülönüljön a tömb és a mutató, tömbök átvételekor alkalmazhatjuk a tömbös jelölést is.

```
1 double first_negative(double array[], size_t size)
2     /* (double *array, size_t size) */
3 {
4     ...
5 }
```

- Formális paraméterlistán `double a[]` azonos `double *a`-val.
- Formális paraméterlistán csak az üres `[]` jelölés használható, a méretet mindig külön paraméterként kell átadni!



# Tömbök átadása függvénynek

- Határozzuk meg függvényel az array tömb első negatív elemét!
- Visszatérési érték legyen a megtalált elem **címe**

```
1 double *first_negative(double *array, size_t size)
2 {
3     size_t i;
4     for (i = 0; i < size; ++i) /* minden elemre */
5         if (array[i] < 0.0)
6             return &array[i];
7
8     return NULL; /* mind nemnegatív */
9 }
```

[link](#)



# Nullpointer

- A nullpointer (NULL)
  - A 0x0000 memóriacímet tartalmazza
  - Megállapodás szerint „nem mutat sehova”

## 2. fejezet

# Sztringek



# Sztringek

- C-ben a szövegeket végjeles karaktertömbökben, ún. sztringekben (string, karakterfüzér) tároljuk.
- A végjel a 0-s ASCII-kódú `'\0'` nullkarakter.

'E'	'z'	' '	's'	'z'	'ö'	'v'	'e'	'g'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

# Sztringek definiálása karaktertömbként

## ■ Karaktertömb definiálása kezdetiérték-adással

```
1 char s[] = {'H', 'e', 'l', 'l', 'o', '\\0'};
```

## ■ Ugyanaz egyszerűbben

```
1 char s[] = "Hello"; /* s tömb (konst.cím 0x1000) */
```

'H'	0x1000	'D'	0x1000
'e'	0x1001	'e'	0x1001
'l'	0x1002	'l'	0x1002
'l'	0x1003	'l'	0x1003
'o'	0x1004	'a'	0x1004
'\\0'	0x1005	'\\0'	0x1005

## ■ s elemei indexeléssel vagy mutatóaritmetikával elérhetőek

```
1 *s = 'D'; /* s-et mutatóként kezelem */
2 s[4] = 'a'; /* s-et tömbként kezelem */
```

# Sztringek definiálása karaktertömbként

- Hosszabb sztringnek is helyet foglalhatunk későbbi felhasználás céljából

```
1 char s[10] = "Hello"; /* s tömb, (konst.cím 0x1000) */
```

'H'	0x1000	'H'	0x1000
'e'	0x1001	'e'	0x1001
'l'	0x1002	'l'	0x1002
'l'	0x1003	'l'	0x1003
'o'	0x1004	'o'	0x1004
'\0'	0x1005	'!'	0x1005
?	0x1006	'!'	0x1006
?	0x1007	'\0'	0x1007
?	0x1008	?	0x1008
?	0x1009	?	0x1009

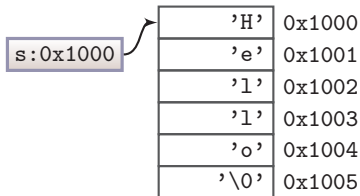
- Módosítás:

```
1 s[5] = s[6] = '!';
2 s[7] = '\0';          /* le kell zárni */
```

# Sztringek definiálása karaktermutatóként

- Konstans karaktertömb és rá mutató pointer definiálása kezdetiérték-adással

```
1 char *s = "Hello"; /* s mutató */
```



- Itt a karaktereknek az ún. statikus területen foglalunk helyet, és a sztring tartalma nem módosítható.
- s értéke viszont felülírható, de ez nem ajánlott, mert a sztringnek lefoglalt területet csak s-en keresztül érjük el.

# Megjegyzések

## ■ Karakter vagy szöveg?

```
1 char s[] = "A"; /* két bájt: {'A', '\0'} */
2 char c = 'A'; /* egy bájt: 'A' */
```

## ■ Üres szöveg van, üres karakter nincs

```
1 char s[] = ""; /* egy bájt: {'\0'} */
2 char c = ''; /* HIBA, ilyen nincs */
```

# Sztring beolvasása és kiírása

- sztringek kiírása-beolvasása a `%s` formátumkóddal

```
1 char s[100] = "Hello";  
2 printf("%s\n", s);  
3 printf("Adj meg egy max 99 karakter hosszú szót: ");  
4 scanf("%s", s);  
5 printf("%s\n", s);
```

Hello

Adj meg egy max 99 karakter hosszú szót: csodalampa  
csodalampa

- Miért nem kell a `printf` függvénynek átadni a méretet?
- Miért nem kell a `scanf` függvényben a `&`?

# Sztring beolvasása és kiírása

- A scanf csak az első whitespace karakterig olvas. Több szóból álló szöveg beolvasása fgets függvénnyel:

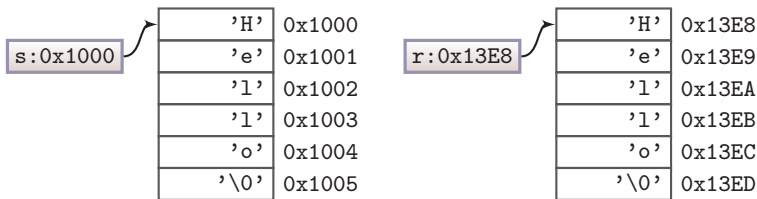
```
1 char s[100];  
2 printf("Adj meg max. 99 karakter hosszú szöveget: ");  
3 fgets(s, 100, stdin);  
4 printf("%s\n", s);
```

```
Adj meg egy max. 99 karakter hosszú szöveget: ez szöveg  
ez szöveg
```

## Sztringek – tipikus hibák

## ■ Tipikus hiba: sztringek összehasonlítása

```
1 char *s = "Hello";  
2 char *r = "Hello";  
3 if (s == r) /* mit hasonlítunk össze? */  
4 ...
```



## ■ Tömbös definíció esetén ugyanez a hiba



# Sztringfüggvények

- Sztringek összehasonlítása
- az eredmény
  - pozitív, ha s1 a névsorban s2 után áll
  - 0, ha megegyeznek
  - negatív, ha s1 a névsorban s2 előtt áll

```
1 int strcmp(char *s1, char *s2) /* mutatós jelölés */
2 {
3     while (*s1 != '\0' && *s1 == *s2)
4     {
5         s1++;
6         s2++;
7     }
8     return *s1 - *s2;
9 }
```

- Nem baj, hogy s1 és s2 megváltozott vizsgálat közben?
- Gondoljuk meg: A megoldásban kihasználtuk, hogy `\0` a 0 kódú karakter!

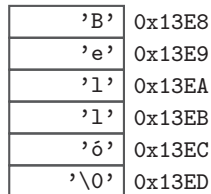
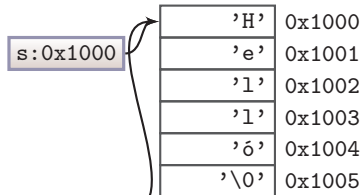
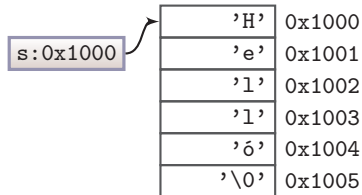
## Sztringek – tipikus hibák

## ■ Tipikus hiba: sztringek (képzelt) másolása

```

1 char *s = "Helló";
2 char *r = "Belló";
3 r = s; /* mit másolunk? */

```



# Egyéb sztringfüggvények

- `#include <string.h>`
  - `strlen` sztring hossza
  - `strcmp` sztringek összehasonlítása
  - `strcpy` sztring másolása
  - `strcat` sztring másik után fűzése
  - `strchr` karakter keresése sztringben
  - `strstr` sztring keresése sztringben
- a `strcpy` és `strcat` függvények ész nélkül másolnak, a felhasználónak kell gondoskodnia az eredménynek fenntartott helyről!

## 3. fejezet

# Keresés adatvektorban



# Vektoralgoritmusok

- Emlékeztető: eldöntési feladat
  - Van-e a vektornak olyan eleme, amely rendelkezik egy adott tulajdonsággal?
  
- Keresési feladat
  - Van-e a vektornak olyan eleme, amely rendelkezik egy adott tulajdonsággal?
  - Ha van, melyik az első ilyen?
  - tulajdonság: a tárolt elem valamelyik része (a keresés kulcsa) megegyezik egy konkrét értékkel.



# Lineáris keresés

- Az első elemtől kezdve egyesével vizsgáljuk az elemeket, amíg
  - a keresett elemet meg nem találjuk,
  - vagy ki nem derül, hogy nincs ilyen elem.
- A vektor elemtípusa
  - struktúra, amelynek egyik tagja a kulcs,
  - nagyon egyszerű esetben maga a kulcs típusa.

```
1 typedef int kulcs_tipus; /* pl. cikkszám */
2
3 typedef struct{
4     kulcs_tipus kulcs;
5     double ar;
6 } tombelem;
```



# Keresés függvénnyel

- Ha függvényként valósítjuk meg
  - milyen paramétereket adjunk át?
  - mi legyen a visszatérési érték?
- Visszaadhatjuk a megtalált elemet

```
1  tombelem lin_keres_elem(tombelem t[], int n,  
2                               kulcs_tipus kul)  
3  {  
4      int i;  
5      for(i=0; i<n; i++)  
6          if(t[i].kulcs == kul)  
7              return t[i];  
8      return t[0]; /* ajjaj */  
9  }
```

- kényelmes, de nem tudjuk, hol volt
- Mit adjunk vissza, ha nem találtunk megfelelőt?!

# Keresés hivatkozás visszaadásával

- A függvény visszaadhatja a megtalált elem indexét

```
1 int lin_keres_ind(tombelem t[], int n,  
2                 kulcs_tipus kul)  
3 {  
4     int i;  
5     for(i=0; i<n; i++)  
6         if(t[i].kulcs == kul)  
7             return i;  
8     return n;  
9 }
```

- Az elemet indexeléssel elérhetjük.
- Ha nem találtunk megfelelőt, visszaadhatunk
  - negatív indexet (pl. -1)
  - n-et, ilyen indexű elem már nincs
- Visszaadhatjuk a megtalált elem címét
  - Az elemet indirekcióval elérhetjük.
  - Ha nem találtunk megfelelőt, visszaadhatunk
    - null-pointert, ezt könnyű tesztelni is



# A lineáris keresés várható lépésszáma

Fontos, hogy olyan kulcs érték, amely nincs tárolva a tömbben, általában sokkal több létezik, mint olyan, amely tárolva van.

Ha a tömb mérete  $N$ , a várható lépésszám  $N$ .

2	-1	-3	4	-2	3	-5
---	----	----	---	----	---	----

Ha a tömb a kulcs szerint rendezett, a lépésszám csökkenthető  $N/2$ -re.

- a tárolt kulcsok megtalálásához átlagosan  $N/2$  lépés szükséges
- nem tárolt kulcsok keresésekor átlagosan  $N/2$  lépés után dől el, hogy nincsenek meg (meghaladtuk)

-5	-3	-2	-1	2	3	4
----	----	----	----	---	---	---



# Keresés rendezett tömbben

```
1 tombelem* linrend_keres(tombelem t[], int n,  
2                           kulcs_tipus kul)  
3 {  
4     int i;  
5     for(i=0; i < n; i++)  
6         if(t[i].kulcs >= kul)  
7             return t+i;  
8     return NULL;  
9 }
```

- ha megvan a keresett kulcsú elem, akkor hivatkozást adhatunk vissza az elemre
- ha nincs, akkor hivatkozást adhatunk vissza arra a tömbelemre, ahol lennie kéne
  - ez további vizsgálatot igényelhet a hívás helyén, de később még jól jöhet
- Ha a tömb rendezett, van még ennél is jobb módszer



# Egy régi ismerős feladat

```
1 int main() {
2     int a=1,f=127;
3     printf("Gondolj egy számra %d es %d kozott!\n",a,f);
4
5     while (1) {
6         int v, k = (a+f)/2;
7         printf("%d?\t", k);
8         scanf("%d", &v);
9         if(v==0)
10            break;
11        if(v>0)
12            a=k+1;
13        else
14            f=k-1;
15    }
16    return 0;
17 }
```

[link](#)

Számkitaláló játék adott intervallumon...



# Logaritmikus (bináris) keresés

- Ugyanígy, csak nem egy számot, hanem egy indexet keresve
- Minden egyes összehasonlító lépésben a keresési tartomány középső elemét vizsgáljuk
- A keresési tartomány minden egyes lépésben feleződik

```
1  int log_keres(tombelem t[], int n,  
2              kulcs_tipus kul) {  
3      int a=0, f=n-1, k;  
4      while(a<f) {  
5          k = (a+f)/2;  
6          if(kul == t[k].kulcs)  
7              return k;  
8          if(kul > t[k].kulcs)  
9              a=k+1;  
10         else  
11             f=k-1;  
12     }  
13     return kul <= t[k].kulcs ? k : k+1;  
14 }
```

[link](#)

Köszönöm a figyelmet.