Tartalomjegyzék

Mutatók (pointer-ek)

Mi is egy mutató?

A mutató (angolul pointer) típusú adat egy másik adat címét tárolja, vagyis azt, hogy hol van tárolva az adat a számítógép memóriájában. (Ez tulajdonképpen csak egy szám, ami azonosítja a memória egyik "http://wiki.math.bme.hurekeszét"http://wiki.math.bme.hu.) (Els? kép innen.)

Minden mutatót ugyanolyan bels? ábrázolással tárolunk (hiszen mindegyik egy cím egy memóriaterületre, egész pontosan a terület "http://wiki.math.bme.hukezd?pontjára"http://wiki.math.bme.hu), mégis van "http://wiki.math.bme.hutípusa"http://wiki.math.bme.hu, ami hivatkozott objektum típusát határozza meg. Ebb?l tudja majd a program futás közben, a mutató által mutatott objektum használatakor, hogy "http://wiki.math.bme.humekkora"http://wiki.math.bme.hu az objektum maga, vagyis mekkora az az adatterület a memóriában, ami a mutatott változóhoz tartozik.

Mire jók a mutatók?

Els? ránézésre a mutatók csak bonyolítják a dolgokat. Programozóként miért kell tudnunk egy változó címét? A Python jól el tudta rejteni el?lünk azt, hogy a változók hogyan és hol is léteznek fizikailag a memóriában (s?t még a változók típusát is elrejti nagyjából), és ez eléggé kényelmes volt.

A C egy alacsonyabb szint? nyelv, itt "http://wiki.math.bme.hugépközelibb"http://wiki.math.bme.hu eszközökkel kell dolgoznunk. Ez egyrészt kényelmetlen lehet, másrészt viszont lehet?séget ad olyan optimalizálásokra, olyan hatékony programok írására, ami egy magasabb szint? nyelven nehézkes vagy lehetetlen lenne. (Pl: bitenkénti m?veletek, bool vektor tárolása)

Konkrétabban mire jók a mutatók?

- komplex adatstruktúrák kialakítására (láncolt listák, fák)
- tömbök címzésére
- függvénynek paraméterként átadható: csak a cím másolódik, ez egyrészt hatékonyabb egy nagy adatstruktúra esetén, másrészt így a mutatott objektumot megváltoztathatja a függvény
- akár több mutatónk is lehet ugyanarra a memóriacímre: adatok sorbarendezése többféle sorrendben, adat-többszörözés nélkül

Hogy néz ki egy mutató C-ben?

Két operátort kell ismernünk a mutatók használatához (mindkett?t a változó neve *elé* kell írni amire vonatkoztanni akarjuk, és lehet köztük szóköz is):

- Egy létez? változó címét a "http://wiki.math.bme.hu&"http://wiki.math.bme.hu operátorral kérhetjük el.
- A "http://wiki.math.bme.hu*"http://wiki.math.bme.hu operátorral kérhetjük el a dolgot/objektumot amire egy mutató mutat

Egy egyszer? példa: a "http://wiki.math.bme.huszam"http://wiki.math.bme.hu egész típusú változó értékét a mutatóján keresztül állítjuk be. (Emlékeztet?: az értékadás operátora az "http://wiki.math.bme.hu="http://wiki.math.bme.hu, és ez mindig a bal oldali dolognak ad új értéket, a bal oldali változó új értéke a jobb oldali kifejezés kiértékelésének eredménye lesz.)

```
/* egy szám deklarálása */
int szam;
/* létrehozzuk a mutatót, a típusa "olyan mutató ami int típusra mutat"  */
int *szam_ptr;

/* itt még nincsenek "összekötve" a fenti változók,
nincs is értékük (ill. a lokális int-nem 0 lesz az értéke),
a lényeg hogy a memóriában lefoglalódott nekik a hely*/

/* most értéket adunk a mutatónknak, azt mondjuk mutasson
a "szam"-ra vagyis a "szam" címével tesszük egyenl?vé */
szam_ptr = &szam;

/* végül a "szam" értékét beállítjuk 3-ra a mutatót használva */
*szam_ptr = 3;
```

Nem kötelez?, de ajánlott, hogy a mutató típusú változóknak a neve is utaljon erre, vagyis végz?djön a neve "http://wiki.math.bme.hu_ptr"http://wiki.math.bme.hu-re, vagy kezd?djön "http://wiki.math.bme.hup_"http://wiki.math.bme.hu-vel (mindegy melyiket választod, de utána - legalább egy programon belül - mindig csak azt a jelölést használd amit választottál!). Ez segít a kód megértésében, csökkenti a hibák valószín?ségét.

Figyelem! Így NE!

```
int szam;
int *szam_ptr;

/* ha kihagyjuk ezt a lépést:*/
/* szam_ptr = &szam; */

/* és így írunk arra a memóriaterületre amire a szam_ptr mutat... */
*szam_ptr = 3;
```

... akkor nem tudjuk hogy mi is fog történni, a szam_ptr értéke véletlenszer?an akármi lehet, és mi erre a random helyre írunk! Véletlenül felülírhatunk más változókat, vagy akár magát a futó programkódot! Nehéz megtalálni egy ilyen hibát, ugyanis nem lesz determinisztikus a programunk m?ködése, lehet hogy legtöbbször teljesen jól le fog futni, néha azonban kiszámíthatatlan hibák fognak történni.

Ennek elkerülésére egyrészt ha lehet a mutatókat rögtön a létrehozáskor definiáljuk is, pl:

```
int szam;
int * szam_ptr = & szam;
```

vagy definiáljuk a mutatót a NULL értékkel, ami azt jelenti hogy ez a mutató még nem mutat sehova:

```
int *szam_ptr = NULL;
```

Tömbök címzése, "http://wiki.math.bme.hupointer aritmetika"http://wiki.math.bme.hu

Mint tudjuk, a C-beli tömbök csak azonos típusú elemeket tartalmazhatnak, amik értelemszer?en fix méret? helyet foglalnak el a memóriában. Azt is érdemes tudni, hogy a tömbök elemei mindig sorban és közvetlenül egymás után lesznek a elérhet?ek a memóriában. (Ezért ha nagyon nagy tömböt akarunk lefoglalni az problémás lehet, hiszan akkora helyet "http://wiki.math.bme.huegyben"http://wiki.math.bme.hu kell találni a gép memóriájában.)

Egy tömb elemei tehát sorban vannak. Ez lehet?vé teszi hogy az egyes elemek címeit (pointereit) összehasonlítsuk (aminek nagyobb a címe az hátrébb van a tömbben), vagy akár m?veleteket végezzünk velük (összeadás, szorzás).

Így hozhatunk létre két mutatót amik egy tömb 4. és 8. elemére mutatnak:

```
double *d_ptr1, *d_ptr2;
double a[10];
       =d&apt3];
       =d&apt7];
```

Mutatók összehasonlítása

Csak olyan mutatókat hasonlítsunk össze, amik ugyanannak a tömbnek az elemeire mutatnak! A használható operátorok:

```
==  !=  >  <  >=  <=
```

Például a fenti kódot kiegészíthetjük így, hogy a kés?bbi elemet írja ki:

```
double *d_ptr1, *d_ptr2;
double a[10];
       =d&apt3];
       =d&apt7];
if (d_ptr2 > d_ptr1) {
printf("%lf\n", *d_ptr2);  /* a mutatott értéket írjuk ki ! */
} else {
printf("%lf\n", *d_ptr1);
}
```

Mutatók összeadása, kivonása

(Képek itt.)

A pointer aritmetika arra ad lehet?séget, hogy egyszer?en léptessük a mutatóinkat egy tömbön belül. Ha egyet hozzáadunk a mutatóhoz, az a következ? tömbbeli elemre fog mutatni. Továbbírjuk az eredeti kódot, d_ptr2 az ötödik elemre fog mutatni a tömbben:

```
double *d_ptr1, *d_ptr2;
double a[10];
       =d&apt3];
       =d&apt7];
double *d_ptr3 = d_ptr1 + 1;
/* és ezzel akár értéket is adhatunk a[4]-nek vagyis az 5. elemnek: */
*d_ptr3 = 11;
```

```
/* a rákövetkez? (hatodik) elemet is beállítjuk, a d_ptr1-et és összeadást használva: */
*(d_ptr1+2) = 45;
```

Az utolsó sorban muszáj a zárójeleket kitenni, mert a * operátor "http://wiki.math.bme.huer?sebben köt"http://wiki.math.bme.hu a változónévhez mint az összeadás operátora (az operátorok er?sségi sorrendjér?l, vagyis a precedenciákról kés?bb tanulunk részletesebben).

Ha van két mutatónk amik ugyanannak a tömbnek az elemeire mutatnak, akkor a különbségük megmondja hogy hány elem van köztük a tömbben. Például:

```
double *d_ptr1, *d_ptr2;
double a[10];
       =d&p[t1r];
       =d&p[t2r];
printf("%d \n", (d_ptr2 - d_ptr1));
/* kiírja hogy 8 */
```

Többdimenziós tömbök

Létrehozhatunk többdimenziós tömböket is, pl egy 4x3-asat:

```
int size1 = 4;
  int size2 = 3;
  int a[size1][size2];
```

Ezt valahogy így képzelhetjük el:

```
                +-----+-----+-----+
       a[0]  ---> | a00 | a01 | a02 |
                +-----+-----+-----+
                +-----+-----+-----+
       a[1]  ---> | a10 | a11 | a12 |
                +-----+-----+-----+
                +-----+-----+-----+
       a[2]  ---> | a20 | a21 | a22 |
                +-----+-----+-----+
                +-----+-----+-----+
       a[3]  ---> | a30 | a31 | a32 |
                +-----+-----+-----+
```

De a memóriában ez is lineárisan foglalhat helyet, összesen *size1 * size2 * sizeof(int)* méret? helyet fog elfoglalni.

Legkényelmesebb úgy használni hogy a címzés is két(vagy több)szint?:

```
int size1 = 4;
  int size2 = 3;
  int a[size1][size2];

  a[0][0] = 2; /* a fenti ábrán az "a00" elemnek adunk értéket */
  a[3][2] = 5; /* a fenti ábrán az "a32" elemnek adunk értéket */
```

Végül egy kicsit hosszab kód, amiben a kétdimenziós tömb elemein végigmegyünk, és kétféleképpen is elérjük az egyes elemeket:

- el?ször a szögletes zárójeles indexeléssel
- másodszor pedig a tömb kezd? elemének címéb?l számolt mutatókkal

```
#include <stdio.h>
int main(void) {
    int size1 = 4;
    int size2 = 3;
    int a[size1][size2];
int *ptr;
    int i, j;
    int e;  /* mutató eltolását számoljuk benne  */
    ptr = &a[0][0];     /* mutató a00-ra */
    printf("\n\n");

    for (i = 0; i < size1; i++) {
      for (j = 0; j < size2; j++) {
        e      = i*size2 + j;
        printf("a[%d][%d] = %d \t\t", i, j, a[i][j]);
        printf("ptr + %d = %d\n", e, *(ptr + e));
      }
    }
    return 0;
}
```

Ha lefordítjuk és futtatjuk, valami ilyesmi lesz a kimenet:

```
 a[0][0] = 134513112            ptr + 0 = 134513112
 a[0][1] = -1081288636          ptr + 1 = -1081288636
 a[0][2] = -1216907324          ptr + 2 = -1216907324
 a[1][0] = 0            ptr + 3 = 0
 a[1][1] = -1217024512          ptr + 4 = -1217024512
 a[1][2] = 1            ptr + 5 = 1
 a[2][0] = 0            ptr + 6 = 0
 a[2][1] = 1            ptr + 7 = 1
 a[2][2] = 0            ptr + 8 = 0
 a[3][0] = 0            ptr + 9 = 0
 a[3][1] = 0            ptr + 10 = 0
 a[3][2] = 0            ptr + 11 = 0
```

Ezen a példán látszik, hogy a lefoglalt tömbök elemei nincsenek "http://wiki.math.bme.hukitakarítva"http://wiki.math.bme.hu, memóriaszemét van bennük. Az értékek betöltésér?l vagy 0-ra inicializálásról nekünk kell gondoskodnunk.

De az is látszik, hogy a kétféle címzési mód ugyanazt eredményezte.

Mutató átadása függvénynek

Azzal hogy egy változó címét adjuk át, lehet?séget kap a függvény, hogy a változó értékét módosítsa. Ezzel a "http://wiki.math.bme.hutrükkel"http://wiki.math.bme.hu az is megoldható, hogy egyszerre több kiszámított értéket is "http://wiki.math.bme.huelkérjünk"http://wiki.math.bme.hu a függvényt?l: egy-egy változó-címet adunk át minden olyan dologhoz, amit szeretnénk hogy a függvény kitöltsön.

Egy kis példa a*pluszminusz* függvény a kapott *a* és *b* egészek összegét és különbségét is kiszámolja, az egyiket visszaadja ahagyományos módon a *return* paranccsal, a másikat egy kapott címre írja:

```
#include <stdio.h>

/* visszaadja a es b különbségét, és a *sum -ba irja az összegüket */
int pluszminusz(int a, int b, int* sum_ptr) {
    /* sum_ptr egy mutató -> (*sum_ptr) egy int változó */
    *sum_ptr = a + b;   /* precedencia:  (*sum) = (a+b) */
    return a - b;
}
```

```
int main() {
    int x = 22;
    int y = 9;
    int kulonbseg, osszeg;
    int * osszeg_ptr = &osszeg;  /* fontos hogy értéket kapjon a mutató miel?tt használjuk,
                                    egy létez? változó cimét! (különben segmantation fault) */
    kulonbseg = pluszminusz(x, y, osszeg_ptr);
    printf("Kulonbseg: %d \t Osszeg: %d \n ", kulonbseg, *osszeg_ptr );
    return 0;
}
```

Források és további olvasnivalók

- http://oreilly.com/catalog/pcp3/chapter/ch13.html
- http://www.classle.net/book/simple-c-programming-pointers
- http://www.hit.bme.hu/~vitez/Progalap1/2011osz/Ea/ea06.pdf
- http://www.eet.bme.hu/publications/e_books/progr/cpp/node52.html
- http://www.ibiblio.org/pub/languages/fortran/append-c.html

Ellen?rz? kérdések

- Az alábbiak közül melyik egy érvényes mutató-deklaráció?

```
A. int x;
B. int &x;
C. ptr x;
D. int *x;
```

- Az alábbiak közül melyik kifejezés jelenti azt hogy "http://wiki.math.bme.hua *b* változó címe a memóriában"http://wiki.math.bme.hu ?

```
A. *b;
B. b;
C. &b;
D. address(b);
```

- Melyik kifejezés adja meg azt az értéket amely azon a memóriacímen van tárolva ahová a *p* mutató mutat?

```
A. p;
B. val(p);
C. *p;
D. &p;
```