Tartalomjegyzék

Dinamikus memóriakezelés

A pointerek használatának a legnagyobb el?nyét eddig nem használtuk ki, segítségükkel ugyanis dinamikusan kezelhetjük a memóriát. Ez azt jelenti, hogy a program futása során olyan méret? darabot foglalhatunk le a memóriából, aminek a méretét nem tudjuk el?re (vagyis a programkód írása során), majd a memóriát egy (vagy több) pointerrel manipulálhatjuk.

Ha csinálunk egy tömböt, annak konstans mérete van. A program elején definiáljuk (szögletes zárójelek között), az pedig a program végéig változatlan marad (ezért ilyenkor statikusnak nevezzük a memóriakezelést). De vannak esetek, amikor nem tudhatjuk el?re, hogy mekkora legyen a tömb mérete: például egy fájl olvasása során, vagy ha a felhasználótól várunk bemenetet aminek nem ismerjük a maximális mennyiségét.

Memóriafoglalás és felszabadítás

Az el?z? el?adáson volt szó róla, hogy egy pointert tökéletesen tudunk tömbként kezelni. Lesz egy mutatónk, ami tömbként viselkedik, és ennek a méretét futási id?ben ("http://wiki.math.bme.hudinamikusan"http://wiki.math.bme.hu) foglaljuk le. Memória dinamikus lefoglalására a *malloc()* függvényt használhatjuk (a neve a "http://wiki.math.bme.humemory allocation"http://wiki.math.bme.hu vagy "http://wiki.math.bme.humemory allocator"http://wiki.math.bme.hu -ból jön):

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

A *void* azt jelenti, hogy típus nélküli, tehát ha például egészre mutató pointernek akarunk helyet foglalni, akkor **cast**-olni kell azt. (részletesebben lentebb a "http://wiki.math.bme.huCasting"http://wiki.math.bme.hu résznél.)

A *malloc()* függvény lefoglal *size* darab byte memóriát a heap-b?l (a memória egy része, amib?l dinamikusan lehet memóriát igényelni a programoknak), és a lefoglalt terület címét adja vissza (vagyis egy mutatót). Ha nem sikerült a memória foglalás, NULL értékkel tér vissza (a "http://wiki.math.bme.husehova mutató mutató"http://wiki.math.bme.hu). A memóriafoglaló függvény párja a:

```
void free(void *p);
```

A *malloc()*-kal lefoglalt, *p* címen kezd?d? területet szabadítja fel.

Mekkora hely kell?

A *malloc()*-nak byte-ban kell megadni a lefoglalandó memória méretét. De honnan tudjuk hogy hány byte memóriában fognak elférni az adataink?

A *sizeof()* függvény meghatározza egy adott típus méretét byte-ban. Például *sizeof(int)* függvényhívás 4-et fog visszaadni, ha 32 bites egész számról van szó, a *sizeof(double)* pedig 8-at ha 64 biten tárolódnak a hosszú lebeg?pontos számok. A *sizeof()* m?ködni fog a felhasználó által definiált adattípusokra is (pl. struktúrákra).

Casting (szereposztás, kasztolás)

A "http://wiki.math.bme.hucasting"http://wiki.math.bme.hu jelentése "http://wiki.math.bme.huszereposztás"http://wiki.math.bme.hu, ugyanis a változó ideiglenesen "http://wiki.math.bme.hueljátssza"http://wiki.math.bme.hu hogy ? egy másik típusú változó. A lényege, hogy megmondjuk a fordítónak hogy az adott változót milyen típusúra konvertálja, és az új típussal használja a kifejezésben.

Néha a típuskonverzió automatikusan megtörténik, de ezt nem nevezzük kasztolásnak. Pl:

```
int i = 5;
    float f = i;
```

Ez a fenti példában nem gond (egy egészb?l törtszám lett), de ha fordítva tesszük akkor már információt veszítünk (a float törtrésze elvész).

A következ? kódban még mindig nincs explicit kasztolás, de az összehasonlításhoz (az if-ben) a fordítónak közös típusra kell konvertálnia az értékeket. Amire automatikusan konvertál az mindig a kisebb értékkészlet? lesz, itt az unsigned int:

```
#include <stdio.h>
int main() {
    float f = 5.7;
    int i = -f;
    unsigned int u = 3;
    if (u < i) {
        printf(" Az int erteke: %d\n", i);
    }
    return 0;
}
```

Most már tényleg explicit kasztolás példák következnek. A kasztoláshoz a kívánt típus nevét zárójelbe kell tenni annak a változónak a neve elé aminek a típusát meg szeretnénk változtatni. Valójában a változó nem fog megváltozni, egy következ? utasításnál ugyanolyan lesz mint eddig volt, tehát a kasztolás csak ideiglenes, abban a kifejezésben van csak hatása ahova odaírjuk.

```
float a = 5.25;
int b = (int)a;  /* Explicit kasztolás float-ból int-té. b értéke 5 lesz */

char c_a = 'A';
int x = (int)c_a;  /* x értéke 65 lesz: az 'A' karakter ASCII kódja */
char c_0 = '0';
int y = (int)c_0;  /* x értéke 48 lesz: a '0' karakter ASCII kódja */
```

Az 5. házihoz szükség lesz egy olyan függvényre, ami egy számot tartalmazó karakternek visszaadja az egész értékét, ezt most már meg tudjuk írni:

```
int atalakit(char c) {
    return (int)(c - '0');
}
```

Még egy, számolós példa:

```
int x=7, y=5 ;
float z;
z = x/y; /* z értéke 1 lesz, egész osztás */
```

Ha a 7/5 értéket pontosan szeretnénk megkapni float-ként, akkor kasztolni kell legalább az egyik változót a kifejezésben (a másik automatikusan fog konvertálódni, de biztos ami biztos kasztoljuk mindkett?t):

```
int x=7, y=5;
float z;
z = (float)x / (float)y;   /* z értéke 1.4 lesz */
```

Több példa: http://www.aui.ma/personal/~O.Iraqi/csc1401/casting.htm ASCII karakterkódok: http://en.wikipedia.org/wiki/ASCII

Dinamikus memóriakezelés - egyszer? példaprogram

Nézzünk egy programot, amiben futási id?ben adjuk meg, hogy hány számot szeretnénk tárolni, a többit megoldjuk malloc-val, és egy mutatóval:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n=0, i=0, x=0;
  int *tomb;    // a "tomb" nev? mutatónk még nem mutat sehova

  // bekérjük az elemek számát a felhasználótól (n-be):
  printf("elemszam:\n");
  scanf("%d", &n);

  // lefoglaljuk a helyet a dinamikus tömbünknek
  // a mutatót kasztolni kell mert a malloc void* -ot ad vissza
  tomb = (int *)malloc(sizeof(int) * n);   // az els? csillag "dereferencia", a második szorzás

  // bekérjük az értékeket sorban
  for(; i<n; ++i) {
     printf("%d: ", i+1);
     scanf("%d", &x);
     tomb[i] = x;   // a mutatót tömbként használjuk
  }

  // kiírjuk az értékeket
  for(i=0;i<n;++i) {
    printf("%d\t",tomb[i]);
  }

  // felszabadítjuk a lefoglalt memóriát !
  free(tomb);
  return(0);
}
```

Látszik, hogy ez a példa sem teljesen dinamikus, mivel itt is meg kell adni, hogy hány elem legyen. Ez ugye azért kell, mert mindenképp le kell foglalni azt az n*4byte-nyi helyet, miel?tt adatokat teszünk a memóriába. Tökéletesen dinamikus adatszerkezeteket láncolt listákkal lehet készíteni.

Láncolt listák

A láncolt lista (linked list) egy dinamikus adatszerkezet. A lista minden eleme tartalmaz valami adatot (akár többfélét, ill. akármilyen struktúra is lehet az adat), és egy mutatót a lista következ? elemére. A tömbbel ellentétben egy lista elemei nem feltétlenül lesznek szépen egymás után elrendezve a memóriában (a tömbös pointer aritmetika sem fog m?ködni), ezért kell a mutatókat eltárolni az elemekben, hogy megtaláljuk a következ? elemet.

kép

Ha mindkét irányba szeretnénk "http://wiki.math.bme.humászkálni"http://wiki.math.bme.hu a listán akkor egy elemhez két mutató is kell, az egyik az el?z?, a másik a következ? elemre mutasson (kétirányba láncolt lista, doubly linked lists).

A láncolt listák el?nyei a tömbökkel szemben

- a méretet nem kell el?re megadni, a program futása közben bármikor adhatunk hozzá új elemet (amíg van memória)
- beszúrhatunk egy elemet valahová középre is, ekkor nem kell az összes utánakövetkez? elemet "http://wiki.math.bme.hueltolni"http://wiki.math.bme.hu a memóriában (ami lassú lenne tömböknél)

A láncolt listák hátrányai a tömbökkel szemben

- nekünk kell kezelni az elemek elérését, ezért bonyolultabb
- nem alkalmazható pointer aritmetika: ha a 100. következ? elemet akarjuk elérni akkor végig kell menni a lista megfelel? szakaszán, 100 lépésben érem csak el (tömböknél csak 100-at kellett adni a mutatóhoz, ezzel egy lépésben elértük az elemet akármilyen messze volt)

Láncolt lista példa

Egy példaprogram (egyirányba láncolt lista, az elemek csak egy azonosító számot tartalmaznak a mutatón kívül):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// listaelem struktúra, benne mutató a köv. elemre
struct lista_elem_s {
  long azonosito;
  struct lista_elem_s * kovetkezo;
};

// rövidebb nevek
typedef struct lista_elem_s ELEM;
typedef struct lista_elem_s * ELEM_p;

// lista struktúra: elég a kezd?elem, aztán a "linkeket" követjük majd
```

```
struct lista_s {
  ELEM_p elso;
};

// rövidebb nevek
typedef struct lista_s LISTA;
typedef struct lista_s * LISTA_p;


ELEM_p lista_utolso_elem(LISTA lista);

ELEM_p lista_uj_elem(LISTA_p lista) {
    // uj elem lefoglalása, kasztolás
    ELEM_p uj_elem = (ELEM *)malloc(sizeof(struct lista_elem_s));
    // hibakezelés: sikerült-e a memóriafoglalás?
    if (uj_elem == NULL) {
        printf ("Nem sikerult uj elemnek helyet foglalni!\n");
        return NULL;
    }

    uj_elem->kovetkezo = NULL;
    uj_elem->azonosito = -1;

    // beállítani az el?z? elem (lista utolsója) mutatóját hogy az újra mutasson
    ELEM_p utolso = lista_utolso_elem(*lista);

    if (utolso != NULL) {
      utolso->kovetkezo = uj_elem;
    } else {
      lista->elso = uj_elem;
    }
    return uj_elem;
}

ELEM_p lista_utolso_elem(LISTA lista) {
    ELEM_p kovetkezo;
    ELEM_p aktualis = lista.elso;
    ELEM_p utolsoelotti = lista.elso;
    while(aktualis) {
        utolsoelotti = aktualis;
        kovetkezo = aktualis->kovetkezo;
        aktualis = kovetkezo;
    }
    return utolsoelotti;
}

void lista_kiir(LISTA_p lista) {
    ELEM_p aktualis = lista->elso;

    while( aktualis ){
        printf( "%ld\r\n", aktualis->azonosito);
        aktualis = aktualis->kovetkezo;
    }
}

void lista_felszabadit(LISTA_p lista) {
    ELEM_p kovetkezo;
    ELEM_p aktualis = lista->elso;
    while(aktualis) {
        kovetkezo = aktualis->kovetkezo;
        free(aktualis);
        aktualis = kovetkezo;
    }
    lista->elso = NULL;
```

```
}

long lista_elem_azonosito(LISTA lista, int sorszam) {
    int cnt = 0;
    ELEM_p aktualis = lista.elso;
    ELEM_p kovetkezo;
    while(aktualis && cnt < sorszam) {
        kovetkezo = aktualis->kovetkezo;
        aktualis = kovetkezo;
        cnt ++;
    }
    if (aktualis) {
        return aktualis->azonosito;
    } else {
        return -1;
    }
}

int main(){
    LISTA lista; lista.elso = NULL;

    ELEM_p a = lista_uj_elem(&lista);
    a->azonosito = 7;
    ELEM_p b = lista_uj_elem(&lista);
    b->azonosito = 6;
    ELEM_p c = lista_uj_elem(&lista);
    c->azonosito = 5;
    ELEM_p d = lista_uj_elem(&lista);
    d->azonosito = 4;
    ELEM_p e = lista_uj_elem(&lista);
    e->azonosito = 3;
    ELEM_p f = lista_uj_elem(&lista);
    f->azonosito = 2;
    ELEM_p g = lista_uj_elem(&lista);
    g->azonosito = 1;

    printf("A lista tartalma:\n\n");
    lista_kiir(&lista);

    printf("\nA lista %d szamu elemenek azonositoja: %ld\n", 2, lista_elem_azonosito(lista, 2));
    printf("\nA lista %d szamu elemenek azonositoja: %ld\n", 0, lista_elem_azonosito(lista, 0));
    printf("\nA lista %d szamu elemenek azonositoja: %ld\n", 6, lista_elem_azonosito(lista, 6));
    printf("\nA lista %d szamu elemenek azonositoja: %ld\n", 7, lista_elem_azonosito(lista, 7));

    lista_felszabadit(&lista);

    return 0;
}
```

Ellen?rz? kérdések

- Milyen esetekben van szükség dinamikus memóriafoglalásra?
- Melyik függvénnyel tudunk dinamikusan memóriát foglalni, és hogyan szabadíthatjuk fel a memóriaterületet ha már nincs rá szükségünk?
- Melyik függvény adja meg egy adott típusú változó memóriában elfoglalt méretét? Milyen egységekben mérve kapjuk meg az ereményt?
- Írj egy C kódsort, melyben egy "http://wiki.math.bme.huc"http://wiki.math.bme.hu nev?, karakter típusú változó értékét egy egész típusú "http://wiki.math.bme.hux"http://wiki.math.bme.hu változónak adod értékül, kasztolással!
- Vázlatosan rajzolj le egy 3 elem?, kétirányba láncolt listát!

Források, olvasnivalók

- Dinamikus memória: http://mernokinformatikus.blogspot.com/2012/02/c-programozas-10-dinamikus-memoria.html
- Kasztolás: http://www.aui.ma/personal/~O.Iraqi/csc1401/casting.htm
- Láncolt lista: http://www.cs.usfca.edu/~srollins/courses/cs112-f07/web/notes/linkedlists.html