

A PROGAMOZÁS ALAPJAI 1

7. előadás

Vitéz András
egyetemi adjunktus
BME Híradástechnikai Tanszék
vitez@hit.bme.hu

2012. március 20.

Miről lesz ma szó?

- Dinamikus memóriakezelés
  - Dinamikus változók
  - Dinamikus tömbök
  - Dinamikus stringek
  - Többdimenziós dinamikus tömbök
- Program kapcsolata a külvilággal.
  - a main() függvény paraméterezése, visszatérési értéke
  - adatállományok, fájlkezelő függvények

Dinamikus változók

Mikor használjuk?

- Ha nem tudjuk előre (fordítási időben), hogy
  - egy változóra szükségünk lesz-e
  - mekkora adatvektorra lesz szükségünk
  - milyen struktúrájú adatszerkezetre lesz szükségünk
- Ha egy változóra csak a futási idő egy részében lesz szükségünk, de ez nem köthető egy függvényhez

Dinamikus változók

Hogyan használjuk?

- Helyet kell foglalni az újonnan létrehozandó változónak
- A memóriát az operációs rendszertől kell kérnünk
- Ha a kérés teljesíthető, a lefoglalt terület címét kapjuk vissza, amit egy alkalmas mutató típusú változóba töltünk
- A mutatón keresztül a területet változóként használhatjuk
- Ha már nincs rá szükség, a területet visszaadjuk
- Nem kell, hogy mindvégig ugyanaz a mutató mutasson rá, de legalább egy mutatónak rá kell mutatnia, mert ha nem, többé már nem tudjuk elérni (memória szivárgás)
- Ha már visszaadtuk, de még van mutató, amelyik mutat rá, azon át már nem szabad hivatkozni rá (fityegő hivatkozás)

Dinamikus változók

Dinamikus helyfoglalás:
Szabványos könyvtári függvényekkel
Adott méretű helyet foglal:

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

Például

```
int* a;
a=malloc(sizeof(int));
*a=9;
```


Dinamikus változók

Dinamikus helyfoglalás:
Szabványos könyvtári függvényekkel
Adott méretű helyet foglal:

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

Megjegyzések:
0, ha nincs elég memória
A lefoglalt terület inicializálatlan!
malloc(0) megengedett
casting szükséges lehet

Dinamikus változók

Dinamikus helyfoglalás:

Szabványos könyvtári függvényekkel

Adott darabszámú változónak foglal helyet,
és a lefoglalt területet nullával tölti fel:

void* calloc(size_t *num*, size_t *size*);

Például

```
int* a;
a=calloc(1,sizeof(int));
*a=9;
```


Dinamikus változók

Dinamikus helyfoglalás:

Szabványos könyvtári függvényekkel

Adott darabszámú változónak foglal helyet,
és a lefoglalt területet nullával tölti fel:

void* calloc(size_t *num*, size_t *size*);

Megjegyzések:

a malloc() függvényt hívja meg
0, ha nincs elég memória
calloc(0) megengedett
casting szükséges lehet

Dinamikus változók

Példa:

Ezt láttuk a gyakorlaton:

```
typedef struct{double x,y;}pont;
pont felezo(pont a,pont b){
   pont c;
   c.x=(a.x+b.x)/2;
   c.y=(a.y+b.y)/2;
   return c;
}
```

Struktúrát ad vissza.

Dinamikus változók

Példa:

Lehetne így is csinálni:

```
typedef struct{double x,y;}pont;
pont*felezo(pont a,pont b){
   pont*c;
   c=malloc(sizeof(pont));
   c->x=(a.x+b.x)/2;
   c->y=(a.y+b.y)/2;
   return c;
}
```

Dinamikus adat címét adja vissza.

Dinamikus változók

Példa:

Így nem szabad csinálni:

```
typedef struct{double x,y;}pont;
pont*felezo(pont a,pont b){
   pont c;
   c.x=(a.x+b.x)/2;
   c.y=(a.y+b.y)/2;
   return &c;
}
```

Már nem létező adat érvénytelen címét adja vissza!

Dinamikus tömbök

Dinamikus helyfoglalás tömbnek:

Szabványos könyvtári függvényekkel

void* malloc(size_t *size*);

Például

```
int* a;
a=malloc(6*sizeof(int));
a[4]=9;
```

Dinamikus tömbök

Dinamikus helyfoglalás tömbnek:
Szabványos könyvtári függvényekkel

void* calloc(size_t *num*, size_t *size*);

Például

```
int* a;
a=calloc(6,sizeof(int));
a[4]=9;
```


Dinamikus helyfoglalás

Lefoglalt terület felszabadítása:

void free(void* *memblock*);

Megjegyzések:
A terület méretét nem kell megadni
free(0) megengedett
Téves hívás nem okoz hibajelzést!

Dinamikus helyfoglalás

Lefoglalt terület méretének megváltoztatása:

void* realloc(void* *memblock*,size_t *size*);

Működése:
a korábban lefoglalt terület méretét
a megadott új méretre módosítja

visszatérési érték az új terület címe

Dinamikus helyfoglalás

Lefoglalt terület méretének megváltoztatása:

void* realloc(void* *memblock*,size_t *size*);

Működése:
az eredeti információt megőrzi
(az új méret erejéig)

ha szükséges, átmásolja
a többlet területet nem inicializálja

0, ha nincs elég memória,
de az eredeti terület megmarad

Dinamikus helyfoglalás

Lefoglalt terület méretének megváltoztatása:

void* realloc(void* *memblock*,size_t *size*);

Példa:

```
int*q;
q=malloc(3*sizeof(int));
q[0]=3;
q[1]=5;
q[2]=7;
q=realloc(q,5*sizeof(int));
q[3]=6;
q[4]=4;
```


Dinamikus helyfoglalás

Lefoglalt terület méretének megváltoztatása:

void* realloc(void* *memblock*,size_t *size*);

Megjegyzések:
Az eredeti méretre nincs szükség,
mert a kiadott terület elé felírja.

Ezért nem szabad a lefoglalt területnek
nem az elejét jelző pointerre free()-t hívni!

Dinamikus helyfoglalás

Lefoglalt terület méretének megváltoztatása:

void* realloc(void* *memblock*,size_t *size*);

Megjegyzések:

ha az első paraméter 0,
mintha malloc() hívása lenne

Például:

realloc(0,4*sizeof(double));
malloc(4*sizeof(double));

Dinamikus helyfoglalás

Lefoglalt terület méretének megváltoztatása:

void* realloc(void* *memblock*,size_t *size*);

Megjegyzések:

ha a második paraméter 0,
mintha free() hívása lenne

Például: a=calloc(6,sizeof(int));

realloc(a,0);
free(a);

Dinamikus stringek

String műveleteknél gyakran nem tudjuk előre a méretet:

Például

Egyesítsünk két stringet!
Van könyvtári függvény, ami hozzáfűz,
de mi egyesíteni szeretnénk

Megoldás:

kiszámítjuk a tárolási igényt
helyet foglalunk az egyesített stringnek
egymás után másoljuk a két kapott stringet
visszaadjuk az új stringre mutató pointert

Dinamikus stringek

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

char*egyesit (char*a,char*b){
  int i,j;
  char*s;
  i=strlen(a);
  j=strlen(b);
  s=malloc(i+j+1);
  strcpy(s,a);
  strcpy(s+i,b);
  return s;
}
```

KI:

```
lódarázs
varázsló
```

```
int main(){
   char s1[]="ló",s2[]="darázs";
   char *s3,*s4;
   s3=egyesit(s1,s2);
   s4=egyesit(s2,s1);
   s4[0]='v';
   printf("%s\n",s3);
   printf("%s\n",s4);
   return 0;
}
```


Többdimenziós dinamikus tömbök

Miért lehet rá szükségünk:

- Nem tudjuk előre (fordítási időben) a tömb méreteit
- A fordító nem tudja kalkulálni az elemek címét
- Ugyanúgy szeretnénk használni, ahogy megszoktuk

Többdimenziós dinamikus tömbök

Megvalósítás mutató tömb használatával:
2D példa:

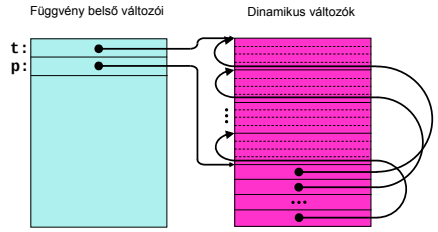
```
typedef . . . tombelem;
```

létrehozása:

```
tombelem**tomb2d(int n,int m){
   tombelem*t,**p;
   int i;
   t=malloc(n*m*sizeof(tombelem));
   p=malloc(n*sizeof(tombelem*));
   for(i=0;i<n;i++)
      p[i]=t+m*i;
   return p;
}
```

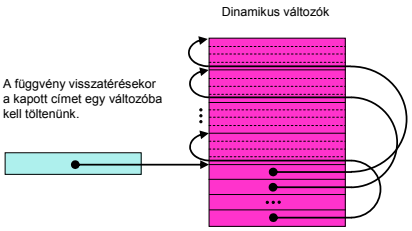
Többdimenziós dinamikus tömbök

Megvalósítás mutató tömb használatával:



Többdimenziós dinamikus tömbök

Megvalósítás mutató tömb használatával:



Többdimenziós dinamikus tömbök

Megvalósítás mutató tömb használatával:
2D példa:

megszüntetése:

```
void lebont2d(tombelem**p){
   free(p[0]);
   free(p);
}
```


Többdimenziós dinamikus tömbök

Megvalósítás mutató tömb használatával:
2D példa:

```
typedef double tombelem;
```

használata:

```
int i,j,s,o; tombelem**a;
scanf("%d%d",&s,&o);
a=tomb2d(s,o);
for(i=0;i<s;i++)for(j=0;j<o;j++)
   a[i][j]=i+1+0.1*(j+1);
for(i=0;i<s;i++){
   for(j=0;j<o;j++)printf("%5.1lf",a[i][j]);
   printf("\n");
}
lebont2d(a);
```


Többdimenziós dinamikus tömbök

Megvalósítás mutató tömb használatával:
2D példa:

BE:

```
4 7
```

KI:

```
1.1  1.2  1.3  1.4  1.5  1.6  1.7
2.1  2.2  2.3  2.4  2.5  2.6  2.7
3.1  3.2  3.3  3.4  3.5  3.6  3.7
4.1  4.2  4.3  4.4  4.5  4.6  4.7
```


A program kapcsolata a külvilággal

A main() függvény paraméterezése

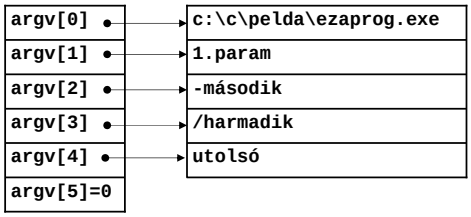
```
int main(int argc, char *argv[])
  {
    . . .
  }
```

A program kapcsolata a külvilággal

A main() függvény paraméterezése

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

argc=5



A program kapcsolata a külvilággal

A main() függvény visszatérési értéke

Program indítását az operációs rendszertől kérhetjük:

- parancssorból
- másik futó programból

A futás sikerétől függően más programok indítása válhat

- szükségessé
- feleslegessé

A visszatérési értéket „hibakódként” értelmezzük:

0 jelenti, hogy minden rendben volt
bármely más érték megegyezés kérdése

Adatállományok

Adatállomány:
(FILE)

- lineáris adatszerkezet
- szöveges vagy bináris adatok a memóriában vagy háttértárolón egymás után tárolva
- az elemek elérése az operációs rendszertől kérhető
- a hozzáférés szekvenciális vagy tetszőleges

Adatállományok

- az állománykezelés nem része a C nyelvnek
- szabványos könyvtári függvények segítségével valósítjuk meg
- a FILE olyan struktúra, amelynek belsejéhez nem férünk hozzá
- az operációs rendszer felé a kapcsolatot egy FILE* típusú változó biztosítja

Adatállományok

Szabványos stream-ek

- Szövegesek
- A program indításakor már meg vannak nyitva

stdin : szabványos bemenet
stdout : szabványos kimenet
stderr : szabványos hiba-kimenet

Adatállományok

Fájlkezelő függvények stdio.h

Fájl megnyitása: kapcsolat a operációs rendszerhez

```
FILE * fopen (char * filenév, char * mode);
```

mode két karakter

r: read
w: write
a: append
+: read&write
b: binary
t: text

Ha nem sikerül, 0 pointerrel tér vissza.
Mindig ellenőrizni kell!

Adatállományok

Fájlkezelő függvények stdio.h

Fájl lezárása: puffer kiírása, fájlrendszer bejegyzés

```
int fclose (FILE * fp);
```

0, ha sikeres
EOF, ha hiba történt

Az EOF egy makró, értéke tipikusan -1

Adatállományok

Fájlkezelő függvények stdio.h

Hiba vizsgálatára használható függvények

```
int feof (FILE * stream);
```
IGAZ, ha end-of-file jött.

```
int ferror (FILE *stream);
```
IGAZ, ha hiba történt,
hibakód: `extern int errno`

Adatállományok

Fájlkezelő függvények stdio.h

Bináris írás (memória tartalmat másol!)

```
size_t fwrite (const void * innen,
               size_t elemhossz,
               size_t elemszám,
               FILE   * stream);
```

- kiír az ***innen*** által mutatott területről
- ***elemszám*** darab
- ***elemhossz*** méretű (size_t) adatot a
- ***stream*** fájlba,
- visszaadja a sikeresen kiírt elemek számát, (nem byte-számot),

ami hiba esetén ***elemszám***-nál kevesebb.

Adatállományok

Fájlkezelő függvények stdio.h

Bináris olvasás (memória tartalmat másol!)

```
size_t fread (const void * ide,
              size_t elemhossz,
              size_t elemszám,
              FILE   * stream);
```

- beolvas az ***ide*** által mutatott területre
- ***elemszám*** darab
- ***elemhossz*** méretű (size_t) adatot a
- ***stream*** fájlból,
- visszaadja a sikeresen beolvasott elemek számát, (nem byte-számot),

ami ***elemszám***-nál kevesebb lehet, ha vége a fájlnak.

Adatállományok

Fájlkezelő függvények stdio.h

Formázott írás (text)

`printf(char*control,...);` : stdout-ra
`fprintf(FILE*fp,char*control,...);` : fájlba
`sprintf(char*sp,char*control,...);` : pufferbe

A control string karaktereit változtatás nélkül kiírja, kivéve a % jellel kezdődő nyomtatásvezérlő mezőket, ezeknek rendre egy-egy további paraméter kell a ... helyen. Ezek meglétét és helyességét a fordító ill. a függvény nem tudja ellenőrizni!

Adatállományok

Fájlkezelő függvények stdio.h

Formázott olvasás (text)

`scanf(char*control,...);` : stdin-ről
`fscanf(FILE*fp,char*control,...);` : fájlból
`sscanf(char*sp,char*control,...);` : stringből

A control string karaktereivel megegyező szöveget kell találnia, kivéve a % jellel kezdődő nyomtatásvezérlő mezőket, ezeknek rendre egy-egy további cím paraméter kell a ... helyen. Leáll, ha a szöveg nem egyezik, mindent beolvasott, vége az olvasott szövegnek, hibás az adat formátuma.

A sikeresen beolvasott adatok számát adja vissza.

A tárolás elnyomható.