Vektorok, mutatók

Kormányos Andor

Komplex Rendszerek Fizikája Tanszék

2019. szeptember 23.

Tömbök

Gyakori feladat, hogy azonos típusú elemekből álló adathalmazt akarunk létrehozni és rajta műveleteket végrehajtani.

A **tömb** (**array**) olyan objektumok halmaza

- azonos típusúak
- memóriában folytonosan helyezkednek el
- az egyes elemek elérése egy elemsorszám (index) segítségével történik

Egydimenziós tömb: vektor

Tömbök

Gyakori feladat, hogy azonos típusú elemekből álló adathalmazt akarunk létrehozni és rajta műveleteket végrehajtani.

A **tömb** (**array**) olyan objektumok halmaza

- azonos típusúak
- memóriában folytonosan helyezkednek el
- az egyes elemek elérése egy elemsorszám (index) segítségével történik

Egydimenziós tömb: vektor

Hasonlóan a változókhoz, a tömböket is deklarálni kell:

```
típus vektornév[méret];
```

ahol a `méret` fordító által kiszámítható konstans kell legyen

Példa vektorok használatára

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define limit 5
5
6   int main()
7   {   double szamok[limit]={2.1, 4.3, 0.5, 11.2, 13};
8       double atlag = 0.0;
9       int i;
10
11      for (i=0; i< limit; i++)
12      {
13      atlag += szamok[i];
14      }
15
16      atlag = atlag/limit;
17
18      printf("Az atlaga =  %f\n", atlag);
19
20      return 0;
21  }
```

- Definiálhatunk **makrókat**, ezek az előfordítás során behelyettesítésre kerülnek a program megfelelő helyére

Példa vektorok használatára

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define limit 5
5
6   int main()
7   {   double szamok[limit]={2.1, 4.3, 0.5, 11.2, 13};
8       double atlag = 0.0;
9       int i;
10
11      for (i=0; i< limit; i++)
12      {
13      atlag += szamok[i];
14      }
15
16      atlag = atlag/limit;
17
18      printf("Az atlaga =  %f\n", atlag);
19
20      return 0;
21  }
```

- Definiálhatunk **makrókat**, ezek az előfordítás során behelyettesítésre kerülnek a program megfelelő helyére
- deklaráljuk **szamok** nevű, 5 elemű vektort és feltöltjük számokkal

Példa vektorok használatára

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define limit 5
5
6   int main()
7   {   double szamok[limit]={2.1, 4.3, 0.5, 11.2, 13};
8       double atlag = 0.0;
9       int i;
10
11      for (i=0; i< limit; i++)
12      {
13      atlag += szamok[i];
14      }
15
16      atlag = atlag/limit;
17
18      printf("Az atlaga =  %f\n", atlag);
19
20      return 0;
21  }
```

- Definiálhatunk **makrókat**, ezek az előfordítás során behelyettesítésre kerülnek a program megfelelő helyére
- deklaráljuk **szamok** nevű, 5 elemű vektort és feltöltjük számokkal
- `for` ciklussal végigvesszük a vektor minden elemét

Példa vektorok használatára

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define limit 5
5
6   int main()
7   {   double szamok[limit]={2.1, 4.3, 0.5, 11.2, 13};
8       double atlag = 0.0;
9       int i;
10
11      for (i=0; i< limit; i++)
12      {
13      atlag += szamok[i];
14      }
15
16      atlag = atlag/limit;
17
18      printf("Az atlaga =  %f\n", atlag);
19
20      return 0;
21  }
```

- Definiálhatunk **makrókat**, ezek az előfordítás során behelyettesítésre kerülnek a program megfelelő helyére
- deklaráljuk **szamok** nevű, 5 elemű vektort és feltöltjük számokkal
- `for` ciklussal végigvesszük a vektor minden elemét
- összeadjuk a **szamok** elemeit és tároljuk a **atlag** változóban

Példa vektorok használatára

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define limit 5
5
6   int main()
7   {   double szamok[limit]={2.1, 4.3, 0.5, 11.2, 13};
8       double atlag = 0.0;
9       int i;
10
11      for (i=0; i< limit; i++)
12      {
13      atlag += szamok[i];
14      }
15
16      atlag = atlag/limit;
17
18      printf("Az atlaga =  %f\n", atlag);
19
20      return 0;
21  }
```

- Definiálhatunk **makrókat**, ezek az előfordítás során behelyettesítésre kerülnek a program megfelelő helyére
- deklaráljuk **szamok** nevű, 5 elemű vektort és feltöltjük számokkal
- `for` ciklussal végigvesszük a vektor minden elemét
- összeadjuk a **szamok** elemeit és tároljuk a **atlag** változóban
- átlag számolás és kiírás

A számítógép memóriája

A processzor (elvileg) egységes memóriát lát

- a bájtok külön-külön címezhetők $\Rightarrow$ memóriacímek
- 0-tól kezdődően a memória méretéig
- legfeljebb 4 GB (32 bites OS) vagy 256 TB (64 bites OS, 48 bites címbusz) memóriacím

A program szempontjából a memória több dologra is való:

- ebben található a programkód, amit a processzor végrehajt
- itt található a **verem (stack)**, ami pl a függvények paramétereinek átadására való
- a fennmaradó, szabadon **foglalható** memória a **halomterület (heap)**
- a stack-re és a heap-re később még bővebben visszatérünk

A memóriát az operációs rendszertől foglaljuk

Eddig egészen egyszerű változókat használtunk

- Egy változóban egyetlen számot tároltunk *valahol* a memóriában
- Valójában ezek a változók a **veremben** tárolódnak
- A verem mérete általában véges[1], spórolni kell vele

További problémák, amikre megoldás kellene

- Ha használunk egy memóriaterületet, akkor arról az operációs rendszernek tudnia kell
- Ha a vermen kívüli memóriára van szükség, az **le kell foglalni**
- Egyszerre sok számnak is kellhet memória (pl egy mátrixnak)

[1] 64 bites Linuxon alapértelmezésben 8MB

Mutatók (pointer) a C nyelvben

A memóriacímek 0-tól kezdődő egész számok

- 64 bites rendszereken ezek 8 bájtot foglalnak el

Mutatók (Pointerek) a C-ben

- olyan változó, ami egy memóriacímet tartalmaz
- de van egy típusa is, mégpedig az, hogy az adott címen *milyen típusú* adat található
- alapesetben a stack-en foglal helyet

Mutatók (pointer) a C nyelvben

Egy egyszerű példa:

```
1  int main() {
2    double a = 5.0;
3    double *dp;
4  // a * operatorral deklaraljuk a double tipusu dp pointert
5
6   dp= &a;  // az & operatorral beallitjuk, hogy dp mely
7           // memoriacimre mutasson
8
9   printf("%p\n", dp);    // eredmeny: 0060FF00
10  printf("%f\n", *dp);   // eredmeny: 5
11
12  int b = *dp;  //adott memoriacimre mutato pointer
13                //erteke kiolvashato
14
15   return 0;
16 }
```

Mutatók (pointer) a C nyelvben

A pointer által mutatott memóriacím felülírható:

```
1  double  a = 5;
2  double *dp = &a;
3
4  *dp = 6;
5  // az a valtozo  erteke most 6 lesz
```

Mutatók (pointer) a C nyelvben

A pointer által mutatott memóriacím felülírható:

```
1  double  a = 5;
2  double *dp = &a;
3
4  *dp = 6;
5  // az a valtozo  erteke most 6 lesz
```

Figyelem!
Ez nem működik:

```
1  double *dp;
2
3  *dp = 5;
```

mert a *dp* a deklarálás után nem mutat egy konkrét memóriacímre

Mutatók (pointer) a C nyelvben

A pointer által mutatott memóriacím felülírható:

```
1  double  a = 5;
2  double *dp = &a;
3
4  *dp = 6;
5  // az a valtozo  erteke most 6 lesz
```

Figyelem!
Ez nem működik:

```
1  double *dp;
2
3  *dp = 5;
```

mert a *dp* a deklarálás után nem mutat egy konkrét memóriacímre

Ez sem működik:

```
1  double a = 5;
2  double *dp;
3
4  *dp = a;
```

Mutatók és vektorok

A C-ben a mutatók és a vektorok között szoros kapcsolat van. Minden művelet, ami egy tömb indexelésével elvégezhető, megoldható mutatókkal is.

Definiáljunk egy 10 elemű vektort:

```
double a[10];
```

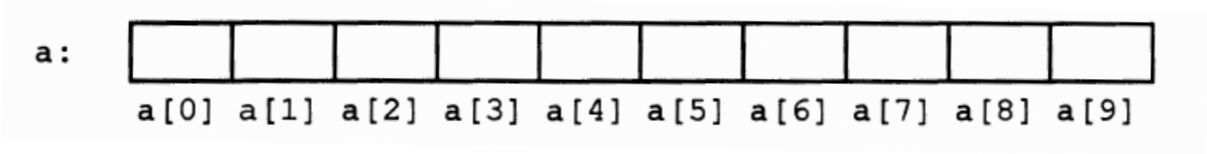
Mutatók és vektorok

A C-ben a mutatók és a vektorok között szoros kapcsolat van. Minden művelet, ami egy tömb indexelésével elvégezhető, megoldható mutatókkal is.

Definiáljunk egy 10 elemű vektort:

```
double a[10];
```

A vektor elemei egy adott címtől kezdve folytonosan helyezkednek el a memóriában



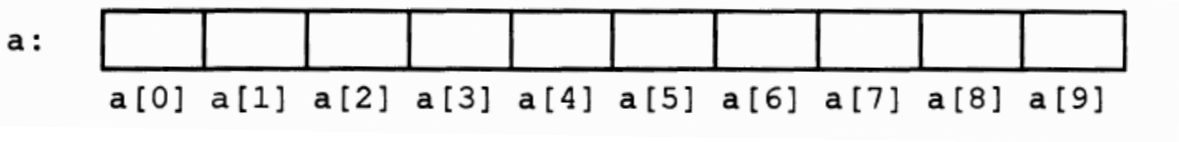
Mutatók és vektorok

A C-ben a mutatók és a vektorok között szoros kapcsolat van. Minden művelet, ami egy tömb indexelésével elvégezhető, megoldható mutatókkal is.

Definiáljunk egy 10 elemű vektort:

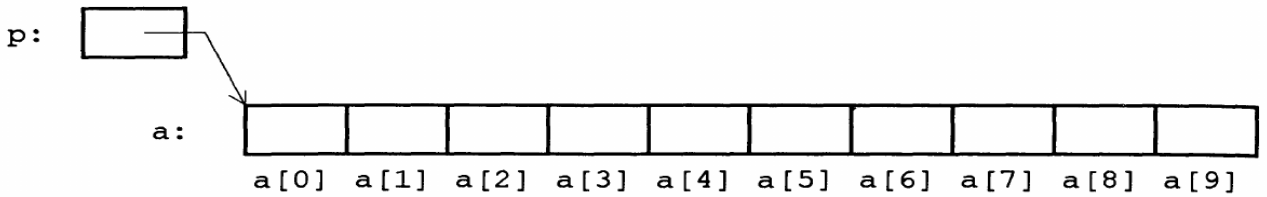
```
double a[10];
```

A vektor elemei egy adott címtől kezdve folytonosan helyezkednek el a memóriában



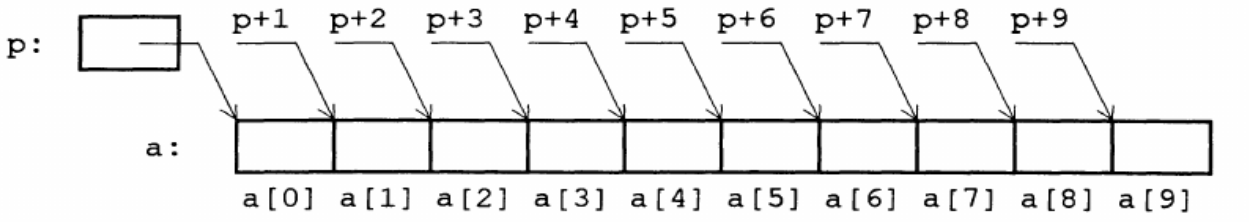
Definiálhatunk egy pointert és ráállíthatjuk az **a** vektor 0-ik elemének címére:

```
1  double a[10];
2  double *dp;
3
4  dp = &a[0];
5  // dp most az a tomb 0-ik elemere mutat
```



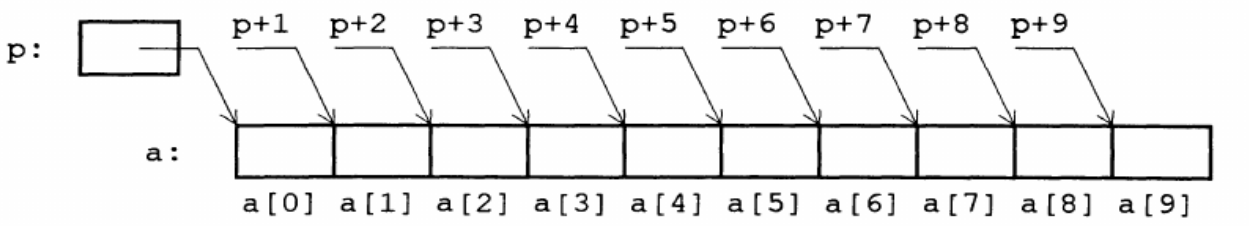
Mutatók és vektorok

A mutató tulajdonságai szerint ha a p= &a[0] paranccsal az **a** vektor 0-ik elemének címére állítottuk a *p* mutatót, akkor a p+i kifejezéssel az **a** vektor *i*-ik elemének címére mutatunk



Mutatók és vektorok

A mutató tulajdonságai szerint ha a p= &a[0] paranccsal az **a** vektor 0-ik elemének címére állítottuk a *p* mutatót, akkor a p+i kifejezéssel az **a** vektor *i*-ik elemének címére mutatunk



A tömb neve és 0-ik indexű elemének a címe igazából szinonímák, ezért a

```
1  double a[10];
2  double *dp;
3
4  dp = &a[0];
```

mellett a

```
1  double a[10];
2  double *dp;
3
4  dp = a;
5  // dp most az a tomb 0-ik elemere mutat
```

is helyes.

Fontos: a *dp* mutató egy változó, értéke változtatható, az **a** vektor viszont egy konstans tömb.

```
1  double a[10];
2  double *dp;
3
4  p = &a[0];
5
6   *(dp++)=5;
7  // helyes, a mutato most a[1] cimere mutat es erteke 5
8
9  (*dp)++;
10 // helyes, a mutato altal kijelolt memoriacimen levo
11 // erteket megnoveltuk
12
13 dp[5]=dp[5]+1;
14 // helyes, a vektor i-ik  elemere
15 // p[i]-vel is hivatkozhatunk
16
17 a++;
18 // helytelen, hibauzenetet kapunk
```

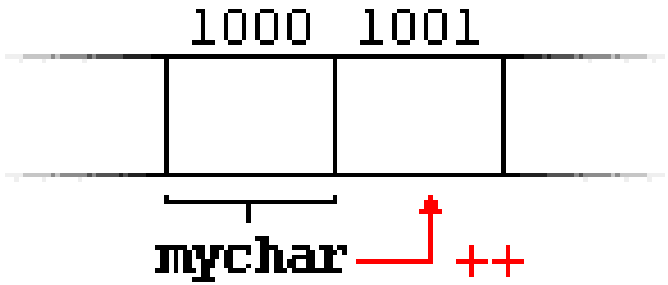
Mutatók és vektorok

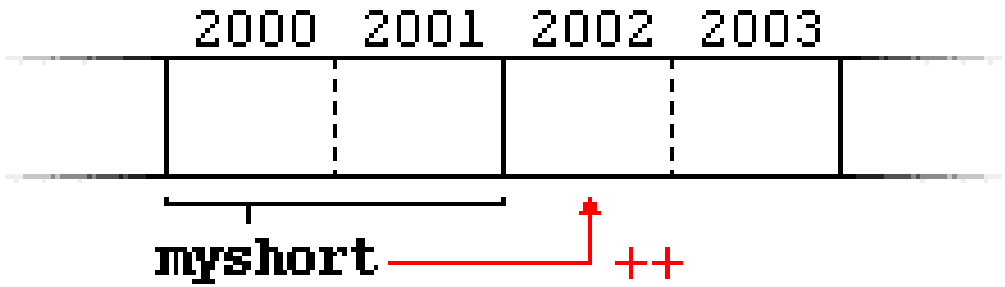
Miután a p pointert ráállítottuk egy **a** vektor első elemének a címére, a következők egyenértékűek:

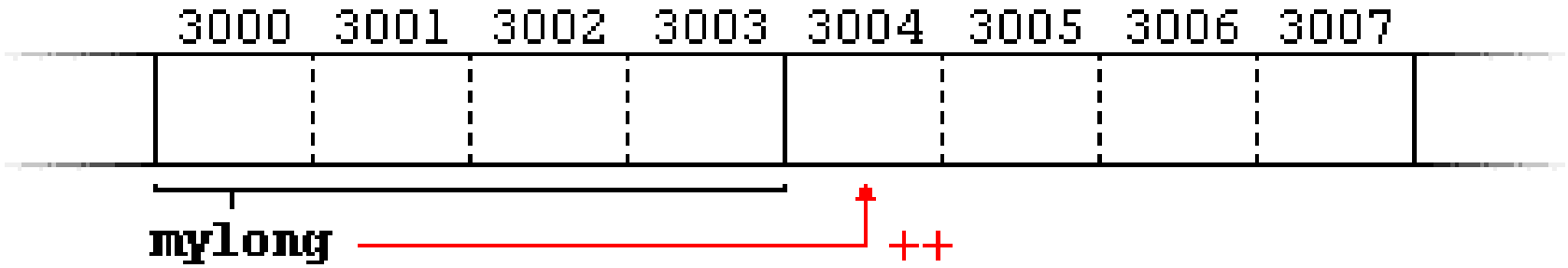
`*(p + i)`	$\Leftrightarrow$	`p[i]`	az i-ik elem értéke
`p + i`	$\Leftrightarrow$	`&p[i]`	az i-ik elem címe

A pointer és a mutatott típus mérete

A pointer memóriacímre mutat, de tisztában van a mutatott típus méretével. Ezt figyelembe veszi, amikor a következő elem memóriacímét kérdezzük le:







Példák mutatók használatára

A következő példákat tekintjük:

- változók cím szerinti átadása függvénynek
- egy szám beolvasása file-ból
- vektor beolvasása file-ból

Változók cím szerinti átadása függvénynek

Miért lehet szükség egy változó memóriacímére?

- C-ben, amikor paramétert adunk át egy függvénynek, annak az **értéke** adódik át egy lokális másolatnak
- a függvény ezt a másolatot tudja átírni, az eredetit nem
- sokszor el akarjuk érni, hogy a függvények bele tudjanak írni a változókba

Ötlet:

- a változó értéke helyett adjuk át a változóra mutató memóriacímet
- ekkor a függvény be tudja írni az új értéket a változónak megfelelő memóriaterületre

Változó átadása függvénynek *cím szerint*

Egy változó címét általában függvénynek kell átadni

```
1  void increment(int a, int *b, int *c) {
2    *b = a + 5;
3    *c = a + 10;
4  }
5
6  int main() {
7    int a = 25;
8    int b, c;
9    increment(a, &b, &c);
10   printf("%d %d %d\n", a, b, c);  // eredmeny: 25 30 35
11   return 0;
12 }
```

Ez arra is jó, ha több “visszatérési értéke” van egy függvénynek.

Változó átadása függvénynek *cím szerint*

Egy változó címét általában függvénynek kell átadni

```
1  void increment(int a, int *b, int *c) {
2    *b = a + 5;
3    *c = a + 10;
4  }
5
6  int main() {
7    int a = 25;
8    int b, c;
9    increment(a, &b, &c);
10   printf("%d %d %d\n", a, b, c);  // eredmeny: 25 30 35
11   return 0;
12 }
```

Ez arra is jó, ha több "visszatérési értéke" van egy függvénynek.

Feladat

Írjunk egy `swap` függvényt, amely pl egy tömb két elemét fel tudja cserélni! Hogyan néz ki a main() függvényben ennek a `swap` függvénynek a hívása?

Adatok beolvasása fájlból

Itt most szöveges fájlokkal foglalkozunk

- ezekből is csak számokat olvasunk be
- a számok 10-es számrendben vannak tárolva

Egy fájl olvasásához vagy írásához

- a fájlt meg kell nyitni
- át kell tudni adni az olvasó/író függvénynek
- a végén a fájlt be kell zárni

A fájlra egy pointeren keresztül tudunk hivatkozni

- ennek a típusa mindig `FILE*`

Példa: egyetlen szám beolvasása fájlból

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5    double a;
6    FILE* f = fopen(argv[1], "r");
7    fscanf(f, "%lf", &a);
8    fclose(f);
9    printf("%f\n", a);
10   return 0;
11 }
```

- A beolvasandó szám az a változóba kerül

Példa: egyetlen szám beolvasása fájlból

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5    double a;
6    FILE* f = fopen(argv[1], "r");
7    fscanf(f, "%lf", &a);
8    fclose(f);
9    printf("%f\n", a);
10   return 0;
11 }
```

- A beolvasandó szám az `a` változóba kerül
- Megnyitjuk a fájlt az `fopen` függvénnyel
  - a fájlnév az első parancssori paraméter
  - az `"r"` jelentése: olvasásra

Példa: egyetlen szám beolvasása fájlból

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5    double a;
6    FILE* f = fopen(argv[1], "r");
7    fscanf(f, "%lf", &a);
8    fclose(f);
9    printf("%f\n", a);
10   return 0;
11 }
```

- A beolvasandó szám az `a` változóba kerül
- Megnyitjuk a fájlt az `fopen` függvénnyel
  - a fájlnév az első parancssori paraméter
  - az `"r"` jelentése: olvasásra
- Fájlból olvasás

Példa: egyetlen szám beolvasása fájlból

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5    double a;
6    FILE* f = fopen(argv[1], "r");
7    fscanf(f, "%lf", &a);
8    fclose(f);
9    printf("%f\n", a);
10   return 0;
11 }
```

- A beolvasandó szám az `a` változóba kerül
- Megnyitjuk a fájlt az `fopen` függvénnyel
  - a fájlnév az első parancssori paraméter
  - az `"r"` jelentése: olvasásra
- Fájlból olvasás
- Bezárjuk a fájlt

Példa: egyetlen szám beolvasása fájlból

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5    double a;
6    FILE* f = fopen(argv[1], "r");
7    fscanf(f, "%lf", &a);
8    fclose(f);
9    printf("%f\n", a);
10   return 0;
11 }
```

Az `fscanf` függvény működése

- az első paraméter a fájl, amiből olvasunk
- a második egy formátumstring
  - picit más, mint a `printf`
  - double: `%lf`
- Az utolsó paraméter egy **pointer** arra a memóriára, ahova a beolvasott számot tárolni szeretnénk
- az `fscanf` visszatérési értékét most nem használjuk
  - azt adja vissza, hogy hány számot sikerült beolvasni

Dinamikus helyfoglalású vektorok

- ha több memóriára van szükségünk (nagy tömb), akkor a halomterületről (heap) kell foglalni
- memóriafoglalás: `malloc`, `calloc` és `realloc` függvényekkel
- a lefoglalt terület kezdetére **mutatókkal** (**pointer**) hivatkozunk
- miután elvégeztük a feladatot, a memóriaterületet felszabadítjuk

Előkészület: void * általános pointer

A void * a pointerek egy univerzális típusa.

- olyankor használjuk, amikor a mutatott adat típusa mindegy
- például memóriafoglaláskor
- nem használható rá a mutató feloldása operátor:

```
1  double d = 1.1234;
2  void *p = &d;        // ez megy
3  *p = 5.0;              // hiba
```

Előkészület: void * általános pointer

A void * a pointerek egy univerzális típusa.

- olyankor használjuk, amikor a mutatott adat típusa mindegy
- például memóriafoglaláskor
- nem használható rá a mutató feloldása operátor:

```
1  double d = 1.1234;
2  void *p = &d;        // ez megy
3  *p = 5.0;              // hiba
```

Akkor mégis mikor?

- például ilyen pointert ad vissza a `malloc` függvény
- ilyet vár a `free` függvény is
- ilyet használ az `fread` és `fwrite`
- ezek univerzális függvények, tetszőleges típusú adattal tudnak dolgozni
- az adat típusa nem, de a mérete (hány bájt) számít

Vektor feltöltése számokkal

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       int n = 10;
7       double *dyn_v;
8
9       dyn_v = (double *)malloc(n * sizeof(double));
10
11      for (int i = 0; i < n; i ++) {
12          dyn_v[i] = i;
13      }
14
15      for (int i = 0; i < n; i ++) {
16          printf("%f\n", dyn_v[i]);
17      }
18
19      free(dyn_v);
20
21      return 0;
22  }
```

Mutató deklarálása

- ez a mutató `double` számra (vagy számokra) for mutatni
- nem foglaltunk még memóriát
- még nem mutat sehova

Vektor feltöltése számokkal

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

int main()
{
    int n = 10;
    double *dyn_v;

    dyn_v = (double *)malloc(n * sizeof(double));

    for (int i = 0; i < n; i ++) {
        dyn_v[i] = i;
    }

    for (int i = 0; i < n; i ++) {
        printf("%f\n", dyn_v[i]);
    }

    free(dyn_v);

    return 0;
}
```

Mutató deklarálása

- ez a mutató `double` számra (vagy számokra) for mutatni
- nem foglaltunk még memóriát
- még nem mutat sehova

Memóriafoglalás: `malloc`

- hány bájt memóriát szeretnénk
- elemek száma $\times$ egy elem mérete byte-ban
- a `malloc` függvény egy **void** típusú mutatót ad vissza
- `(double *)` az ún. **cast**-olás, jó programozási gyakorlat

Vektor feltöltése számokkal

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

int main()
{
    int n = 10;
    double *dyn_v;

    dyn_v = (double *)malloc(n * sizeof(double));

    for (int i = 0; i < n; i ++) {
        dyn_v[i] = i;
    }

    for (int i = 0; i < n; i ++) {
        printf("%f\n", dyn_v[i]);
    }

    free(dyn_v);

    return 0;
}
```

Egy egész tömböt foglaltunk

- elemek elérése: `dyn_v[i]`
- legelső: `dyn_v[0]`
- legutolsó: `dyn_v[n - 1]`

Vektor feltöltése számokkal

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       int n = 10;
7       double *dyn_v;
8
9       dyn_v = (double *)malloc(n * sizeof(double));
10
11      for (int i = 0; i < n; i ++) {
12          dyn_v[i] = i;
13      }
14
15      for (int i = 0; i < n; i ++) {
16          printf("%f\n", dyn_v[i]);
17      }
18
19      free(dyn_v);
20
21      return 0;
22  }
```

Egy egész tömböt foglaltunk

- elemek elérése: `dyn_v[i]`
- legelső: `dyn_v[0]`
- legutolsó: `dyn_v[n - 1]`

A végén a memóriát felszabadítani!

- a `free` függvény meghívásával

Mutató mutatója

Probléma:

- olyan függvényt szeretnénk írni, ami memóriát allokál egy vektornak
- a lefoglalt memória címét bele kellene tenni egy átadott pointerbe

Mutató mutatója

Probléma:

- olyan függvényt szeretnénk írni, ami memóriát allokál egy vektornak
- a lefoglalt memória címét bele kellene tenni egy átadott pointerbe

Emlékezzünk a `Változó átadása egy függvénynek cím szerint` fóliára:

```
1   void increment(int a, int *b) {
2   *b = a + 5;
3   }
4
5   int main() {
6     int a = 25;
7     int b;
8     increment(a, &b);
9     return 0;
10  }
```

Mutató mutatója

Az előbbiek alapján egy lehetséges megoldás egy vektornak memóriát allokáló függvényre:

```
1  void alloc_vec(int n, double **v) {
2    *v = malloc(n * sizeof(double));
3  }
4
5  int main()
6  {
7    int n = 25;
8    double *v;
9    alloc_vec(n, &v);
10   // additional steps here
11   free(v);
12   return 0;
13 }
```

- A double **v egy mutató mutatója (pointer to pointer) deklaráció
- a függvény egyik paraméterében kapjuk vissza a lefoglalt memória címét

Vektornak memóriát allokáló függvény

Egy másik lehetséges megoldás a vektornak memóriát allokáló függvényre: visszatérési értékként egy memóriacímet adó függvény

```
1
2   double *alloc_vec(int n) {
3     double *v = (double *)malloc(n * sizeof(double));
4     return v;
5   }
6
7   int main()
8   {
9     int n = 25;
10    double *v = alloc_vec(n);
11    // additional steps here
12    free(v);
13    return 0;
14  }
```

Vektor beolvasása fájlból

Összegezve:

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

double *alloc_vec(int n) {
  double *v = (double*)malloc(n * sizeof(double));
  if (v == 0) {
    printf("Not enough memory.\n");
    exit(-1);
  }
  return v;
}

void load_vec(FILE* f, double *v, int n) {
  for (int i = 0; i < n; i ++) {
    fscanf(f, "%lf", v + i);
  }
}

void print_vec(double *v, int n) {
  for (int i = 0; i < n; i ++) {
    printf("%f\n", *(v + i));
  }
}
```

```c
int main(int argc, char* argv[])
{
  if (argc < 3) {
    printf("Not enough arguments.");
    exit(-1);
  }

  int n = atoi(argv[1]);
  double *v = alloc_vec(n);

  FILE* f = fopen(argv[2], "r");
  if (f == 0) {
    printf("Cannot open file %s.", argv[1]);
    exit(-1);
  }
  load_vec(f, v, n);
  fclose(f);

  print_vec(v, n);

  free(v);
  return 0;
}
```

Második házi feladat

Lásd a gyakorlat honlapján!

Beküldési határidő: 2019. október 6, 23:59