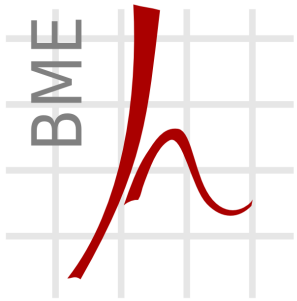
Vektoralgoritmusok, tömbök

A programozás alapjai I.



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

2020. szeptember 21.

Tartalom

1. fejezet

Az adatvektor

Az adatvektor fogalma

Az adatvektor

Azonos típusú adatok véges sorozata

- A sorrend számít
- Hozzáférés szerint lehet
  - Memóriában tárolt, adott számú adat
    - Sok helyet foglalhat, akkor használjuk, ha a feldolgozáshoz minden adat egyszerre szükséges
  - Program bemenetére sorosan érkező adatok
    - Mindig csak a következő elemhez férünk hozzá, de sokszor ez is elég a feldolgozáshoz

2. fejezet

A soros feldolgozás

A sorosan beérkező adatvektor

- Két lehetőség a darabszám meghatározására
  1. Először beolvassuk az adatszámot, majd az adatokat

4	renault	opel	kia	fiat

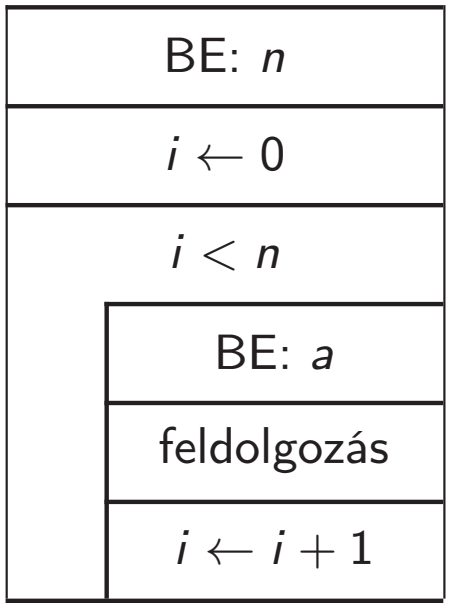
  2. Ciklusban beolvassuk és feldolgozzuk az adatokat, amíg egy előre megbeszélt (a többivel össze nem téveszthető) adatot nem kapunk

renault	opel	kia	fiat	vége

  Ez a végjeles sorozat

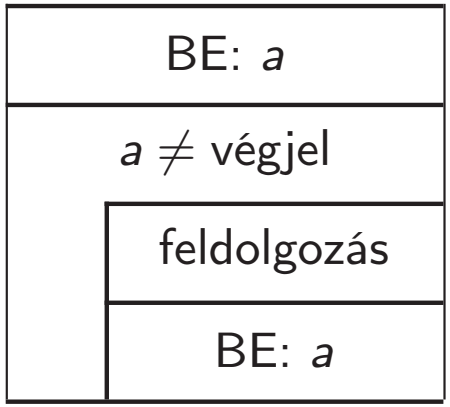
Adatvektor feldolgozása

Ismert méretű vektor



- A jelölések
  - n: adatok száma
  - a: beolvasott adat
  - i: ciklusváltozó

Végjeles vektor



- A jelölések
  - a: beolvasott adat

Kis kitérő

- Később részletesen lesz róluk szó, de addig is...

Néhány C-típus

`int` Egész értékek tárolására alkalmas típus, beolvasás és kiírás `%d` formátumkóddal

`double` Valós számok tárolására alkalmas típus, beolvasás `%lf`, kiírás `%f` formátumkóddal

`char` Szöveges karakterek tárolására alkalmas típus beolvasás és kiírás `%c` formátumkóddal

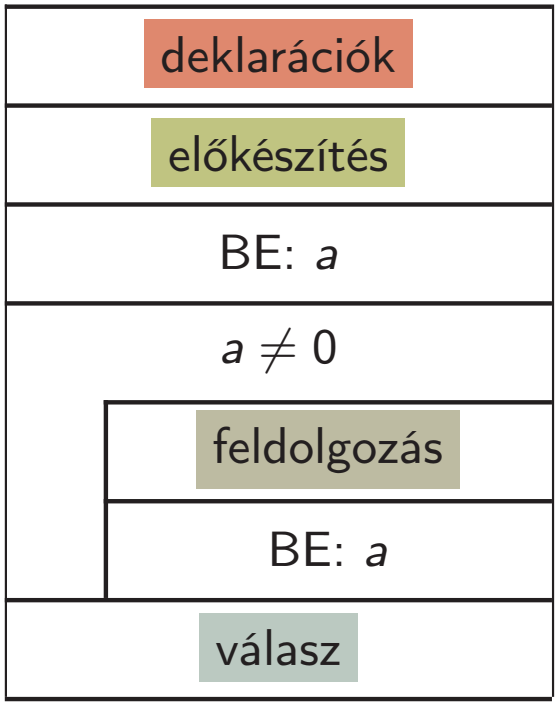
Néhány C-operátor

`==` (egyenlő) egyenlőségvizsgálat

`!=` (nem egyenlő) különbözőségvizsgálat

`&&` (logikai ÉS) konjunkció

A keretprogram végjeles vektor feldolgozásához



- A színes részeket kell kidolgoznunk, a többi mindig ugyanaz

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a;
6    /* deklarációk */
7    /* előkészítés */
8    scanf("%d", &a);
9    while (a != 0)
10   {
11     /* feldolgozás */
12     scanf("%d", &a);
13   }
14   /* válasz */
15   return 0;
16 }
```

Elemek összege

deklarációk
Felveszünk egy változót az összeg tárolására.

előkészítés
Kezdetben 0-ra állítjuk.

feldolgozás
Növeljük a beolvasott adattal.

válasz
Kiírjuk a kapott eredményt.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a;
6    int sum;
7    sum = 0;
8    scanf("%d", &a);
9    while (a != 0)
10   {
11     sum = sum + a;
12     scanf("%d", &a);
13   }
14   printf("%d", sum);
15   return 0;
16 }
```

link

Elemek szorzata

deklarációk
Felveszünk egy változót a szorzat tárolására.

előkészítés
Kezdetben 1-re állítjuk.

feldolgozás
Szorozzuk a beolvasott adattal.

válasz
Kiírjuk a kapott eredményt.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a;
6    int prod;
7    prod = 1;
8    scanf("%d", &a);
9    while (a != 0)
10   {
11     prod = prod * a;
12     scanf("%d", &a);
13   }
14   printf("%d", prod);
15   return 0;
16 }
```

link

Elemek átlaga

- Képezzük az elemek átlagát!
  - Végig számon kell tartanunk az elemek összegét és darabszámát.
  - Mindkettő kezdetben 0.
  - Az összeget a beolvasott adattal, a darabaszámot 1-gyel növeljük minden lépésben.
  - Végül kiírjuk az összeg és a darabszám hányadosát.
- Vigyázat! C-ben
  - 8/3=2 (egész osztás)
  - `8.0/3.0 = 8.0/3 = 8/3.0 = 2.6666...` (valós osztás)
  - ezért az összeget eleve valós számként tartjuk nyilván

Elemek átlaga

deklarációk

Felveszünk két változót az összeg és az elemszám tárolására.

előkészítés

Kezdetben az összeg és az elemszám is 0.

feldolgozás

Az összeget növeljük a beolvasott adattal, a darabszámot eggyel.

válasz

Kiírjuk az összeg és a darabszám hányadosát.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a;
6    double sum;
7    int n;
8    sum = 0.0;
9    n=0;
10   scanf("%d", &a);
11   while (a != 0)
12   {
13     sum = sum + a;
14     n = n+1;
15     scanf("%d", &a);
16   }
17   printf("%f", sum/n);
18   return 0;
19 }
```

link

Leszámlálás

- Képezzük egy bizonyos feltételnek megfelelő adatok darabszámát!
  - Végig számon kell tartanunk a megfelelő elemek darabszámát,
  - ami kezdetben 0,
  - és eggyel nő, ha megfelelő elem érkezik. (logikai vizsgálat)
  - Végül kiírjuk a darabszámot.
- Példánkban számláljuk össze a kétjegyű számokat!
- A megfelelő feltétel:

```
1  a >= 10 && a <= 99 /* && : logikai ÉS */
```

Leszámlálás

deklarációk

Felveszünk egy változót a darabszám tárolására.

előkészítés

Kezdetben 0-ra állítjuk.

feldolgozás

Ha az elem kétjegyű, növeljük a darabszámot.

válasz

Kiírjuk a darabszámot.

```c
#include <stdio.h>

int main()
{
  int a;
  int n;
  n=0;
  scanf("%d", &a);
  while (a != 0)
  {
    if (a>=10 && a<=99)
      n = n+1;
    scanf("%d", &a);
  }
  printf("%d", n);
  return 0;
}
```

link

Minimum meghatározása

Képezzük az elemek minimumát!

- Végig számon kell tartanunk a minimumot
- ~~Inicializáljuk 5000-rel (annál biztos csak kisebb jön)!~~

  Csak akkor tehetjük meg, ha a specifikációból ez következik!

Inkább módosítunk a szerkezeten:

- Először beolvassuk az első adatot, és a minimumot azzal inicializáljuk
- Ha a soron következő adat kisebb, mint a minimum, a minimumot átírjuk az új adatra
- Végül kiírjuk a minimumot

Elemek minimuma

deklarációk

Felveszünk egy változót a minimum tárolására.

előkészítés

az első `scanf` mögé került!

Kezdetben az első elem értékére állítjuk.

feldolgozás

Ha az elem kisebb, mint min, min $\leftarrow$ elem.

válasz

Kiírjuk a minimumot.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a;
6    int min;
7    scanf("%d", &a);
8    min=a;
9    while (a != 0)
10   {
11     if (a < min)
12       min = a;
13     scanf("%d", &a);
14   }
15   printf("%d", min);
16   return 0;
17 }
```

link

Elemek maximuma

deklarációk

Felveszünk egy változót a maximum tárolására.

előkészítés

Kezdetben az első elem értékére állítjuk.

feldolgozás

Ha az elem nagyobb, mint max, max ← elem.

válasz

Kiírjuk a maximumot.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a;
6    int max;
7    scanf("%d", &a);
8    max=a;
9    while (a != 0)
10   {
11     if (a > max)
12       max = a;
13     scanf("%d", &a);
14   }
15   printf("%d", max);
16   return 0;
17 }
```

link

Karakterfeldolgozás

Íljunk laccsolóploglamot,
mely a bemeneten érkező szöveget úgy írja ki a kimenetre, hogy az `'r'` betűket `'l'`-re cseréli.

- Változások
  - A program most karaktereket fog beolvasni, amíg van mit
  - Minden iterációban lesz válasz a kimeneten
  - Ennek értéke maga a beolvasott karakter vagy `'l'`, ha a karakter `'r'` volt.
- Figyeljünk a kis- és nagybetűkre is!

Karakterfeldolgozás

```c
#include <stdio.h>

int main()
{
  char a;
  while(scanf("%c", &a) == 1)/*scanf mint kifejezés*/
  {
    switch(a)
    {
    case 'R': printf("L"); break; /* ' " */
    case 'r': printf("l"); break;
    default: printf("%c", a);
    }
  }
  return 0;
}
```

link

A scanf mint kifejezés

- A scanf-nek van értéke. Megmondja, hogy hány dolgot sikerült beolvasnia.

```
1  db = scanf("%d%lf%d", &a, &b, &c);
```

`3 2.5 23`	db:3, a:3, b:2.5, c:23
`3 2.5 alma`	db:2, a:3, b:2.5, c:??
`3.8 2`	db:3, a:3, b:0.8, c:2
`alma 3 2`	db:0, a:??, b:??, c:??

Egy átalakítóprogram

- Egy szöveges fájl csupa fokban megadott szögértéket tartalmaz.
- Számítsuk át mindet radiánba, és az eredményt írjuk szöveges fájlba.

```c
#include <stdio.h>
int main()
{
  double d;
  while (scanf("%lf", &d) == 1)
    printf("%f ", d/180.0*3.141592); /* majdnem */
  return 0;
}
```

link

Használat:

```
radian.exe < fokok.txt > radianok.txt
```

GPS tracker

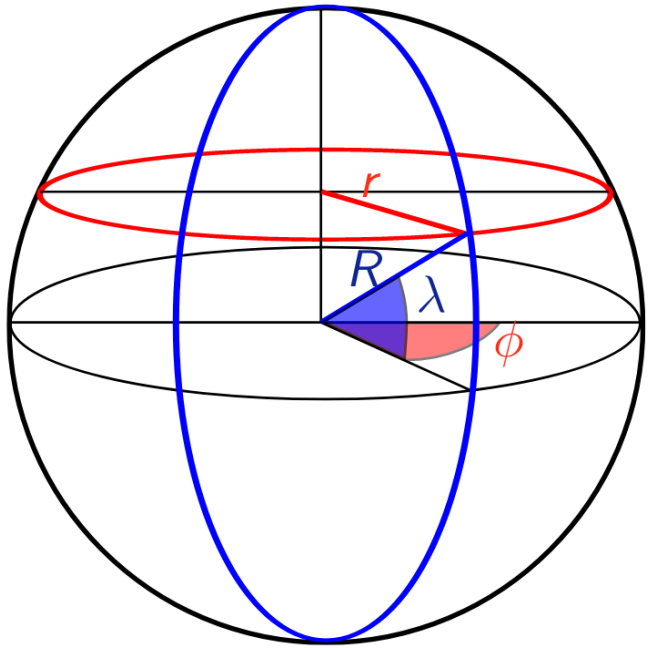
- hosszúsági (ϕ) és szélességi (λ) koordináták szöveges fájlban:
- adjuk meg a megtett út hosszát!
- Két közeli pont távolsága:

```
18.62493500     47.20635300
18.62624700     47.20693900
18.62618800     47.20711600
```

$$d \approx \sqrt{\Delta x^2 + \Delta y^2}$$

$$\Delta x = \Delta\phi \cdot r$$

$$\Delta y = \Delta\lambda \cdot R$$



ahol

- r: szélességi kör sugara
- R: hosszúsági kör sugara

- Valós számpárokat (ϕ, λ) olvasunk be, amíg lehet
- $(\Delta\phi, \Delta\lambda)$ számításához el kell tárolnunk az előző számpárt

GPS tracker

```c
int main()
{
  double R = 6371, r = 4328;
  double phi1, phi2, lam1, lam2, D = 0.0;
  scanf("%lf%lf", &phi1, &lam1);
  while (scanf("%lf%lf", &phi2, &lam2) == 2) {
    double dx = r * (phi2-phi1);
    double dy = R * (lam2-lam1);
    D = D + sqrt(dx*dx + dy*dy);
    phi1 = phi2;
    lam1 = lam2;
  }
  printf("A megtett tav: %f km\n", D);
  return 0;
}
```

link

Használat:

```
radian.exe < fokok.txt | gps.exe
```

Adatsor feldolgozás - kicsit másmilyen feladat

- Írjunk programot, mely összeszámolja, hogy a bemeneten érkező egész számok közül hány esik az átlag alá!
- Az átlag csak a teljes adatsor beolvasása után derül ki.
- Ezután újra végig kell járni ugyanazokat az elemeket, hogy a kisebbeket kigyűjthessük.
- Tárolnunk kell a beolvasott elemeket.
- Nyilván nem így:

```
1 int a, b, c, d, e, f, g, h, i;
2 scanf("%d%d%d%d%d", &a, &b, &c, &d... /* jaj jaj! */
```

- hanem úgy, hogy akármelyik elem egységes néven, indexelve (a_1, a_2, a_3, ... a_i) elérhető legyen.

3. fejezet

Tömbök

Tömbök

A tömb (adatvektor) fogalma

- lineáris adatszerkezet
- azonos típusú, véges számú adat a memóriában egymás után tárolva
- az elemek elérése indexeléssel, tetszőleges sorrendben lehetséges

a_0	a_1	a_2	$\ldots$	a_{n-1}

Tömbök szintaxisa

Tömb deklarációja

```
<elemtípus> <tömb azonosító> [<elemszám>];
```

```
1 /* 5 double értéket tároló, data nevű tömb */
2 double data[5];
```

- <elemszám> konstans kifejezés, fordítási időben (programíráskor) ismert!
- Vagyis nincs[1] olyan, hogy

```
1 int n = 5;
2 double data[n]; /* HIBÁS, n nem konstans, változó */
```

[1] A C99-szabvány már engedi, mi nem.

Tömbök szintaxisa

Tömbelemek elérése

`<tömb azonosító> [<elem index>]`

- n elemű tömb esetén indexelés 0-tól $n-1$-ig

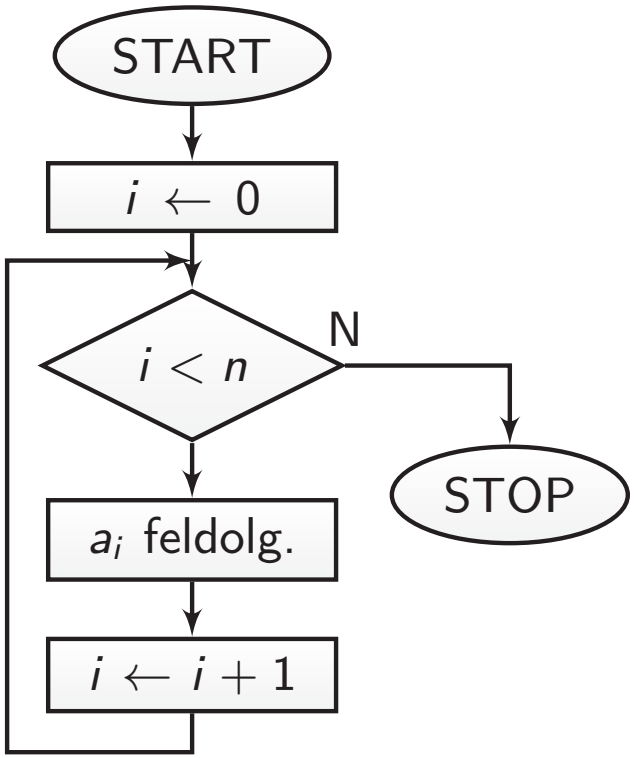
data[0]	data[1]	data[2]	...	data[n-1]

- `<elem index>` lehet nemkonstans kifejezés is, pont ez a lényeg!
- Tömbelemmel mindaz megtehető, ami különálló változóval

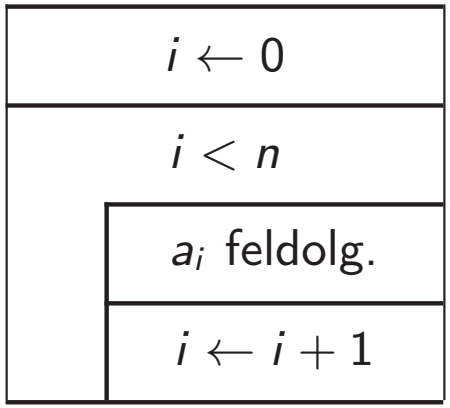
```
1  /* 5 double értéket tároló, data nevű tömb */
2  double data[5];
3
4  data[0] = 2.0;
5  data[1] = data[0];
6  data[i] = 3*data[2*q-1];
```

Tömb bejárása

- Bejárás: a tárolt adatok egymást követő feldolgozása



- A jelölések
  - n: konstans méret
  - a: a tömb
  - i: ciklusváltozó



- Ez egy `for` ciklus!

Tömb bejárása

- Bejárás megvalósítása célszerűen `for` ciklussal az alábbi módon:

```
1  double array[10];                /* 10 elemű tömb */
2  int i;                           /* ciklusváltozó */
3  for (i = 0; i < 10; i = i+1) /* i = 0,1,...,9 */
4  {
5    /* array[i] feldolgozása */
6  }
```

- Pl. Töltsünk fel egy tömböt beolvasott adatokkal

```
1  double array[10];
2  int i;
3  for (i = 0; i < 10; i = i+1)
4  {
5    scanf("%lf", &array[i]);
6  }
```

Tömb bejárása

- Határozzuk meg a tömbben tárolt elemek átlagát!

```
1 double mean = 0.0;
2 for (i = 0; i < 10; i = i+1)
3 {
4   mean = mean + array[i];
5 }
6 mean = mean / 10;
```

- Határozzuk meg az átlagnál kisebb elemek számát!

```
1 int n = 0;
2 for (i = 0; i < 10; i = i+1)
3 {
4   if (array[i] < mean)
5     n = n + 1;
6 }
```

Átlagnál kisebb elemek száma – Teljes program

```c
#include <stdio.h>

int main(void)
{
  /* deklarációk */
  double array[10];
  int i, n;
  double mean;

  /* tömb feltöltése */
  for (i=0; i<10; i=i+1)
    scanf("%lf", &array[i]);

  /* átlag számítása */
  mean = 0.0;
  for (i=0; i<10; i=i+1)
    mean = mean + array[i];
  mean = mean / 10;

  /* leszámlálás */
  n = 0;
  for (i=0; i<10; i=i+1)
  {
    if (array[i] < mean)
      n = n+1;
  }

  /* válasz */
  printf("%d", n);
  return 0;
}
```

link

Eldöntés

- Írjunk programot, mely eldönti, hogy igaz-e, hogy...
  - a vektor minden eleme rendelkezik adott tulajdonsággal
  - a vektor semelyik eleme sem rendelkezik adott tulajdonsággal
  - a vektornak van eleme, amely rendelkezik adott tulajdonsággal
  - a vektornak van eleme, amely nem rendelkezik adott tulajdonsággal

Eldöntés

- Igaz-e, hogy az n elemű data tömb minden eleme nagyobb, mint 10?

```
válasz ← IGAZ
Minden i-re 0-tól n-1-ig
  HA data[i] <= 10
    válasz ← HAMIS
KI: válasz
```

```
1  int answer = 1;
2  for (i=0; i<n; i=i+1)
3    if (data[i] <= 10)
4      answer = 0;
5  printf("%d", answer);
```

- C-ben nincs igazságérték típus, helyette `int`-et használunk
  - 0 → HAMIS
  - minden más → IGAZ
- És ha már az első (0. indexű) elemről kiderült, hogy ≤ 10?

Eldöntés

- hatékonyabb megoldás: csak addig vizsgálunk, míg ki nem derül az eredmény

```
válasz ← IGAZ
i ← 0
AMÍG i < n ÉS válasz IGAZ
  HA data[i] <= 10
    válasz ← HAMIS
  i ← i+1
KI: válasz
```

```
1  int answer = 1, i = 0;
2  while (i<n && answer==1)
3  {
4    if (data[i] <= 10)
5      answer = 0;
6    i = i+1;
7  }
8  printf("%d", answer);
```

Eldöntés

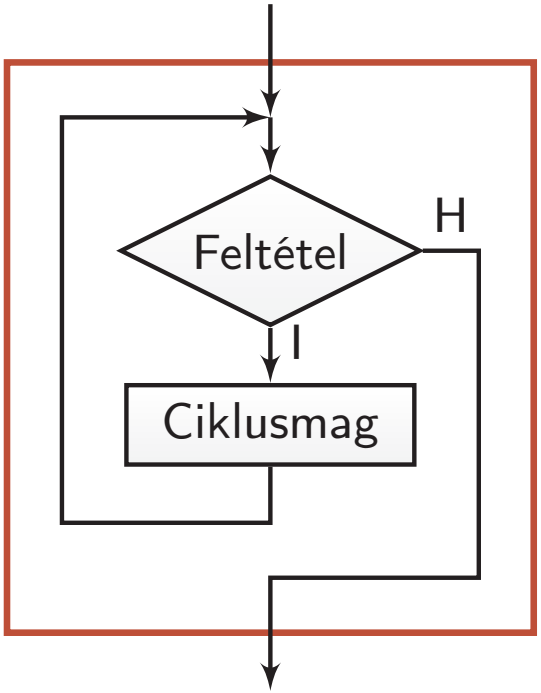
- ugyanaz másként, `answer` változó nélkül

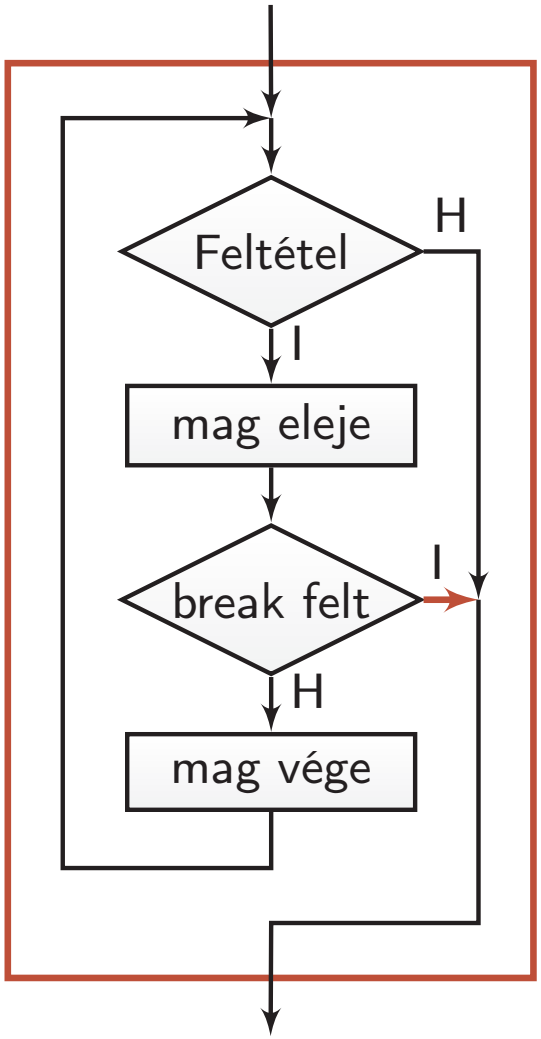
```
1  for (i=0; i<n; i=i+1)
2  {
3    if (data[i] <= 10)
4      break;
5  }
6  if (i == n) printf("Igaz."); /* végigértünk? */
7  else printf("Nem igaz.");
```

- A `break` utasítás megszakítja az őt közvetlenül tartalmazó ciklus (`for`, `while`, `do`) végrehajtását, és a ciklust követő utasításra ugrik

  nem strukturált elem

Elöltesztelő ciklus `break` nélkül és `break`-kel





Eldöntés

```
1  for (i=0; i<n; i=i+1)
2  {
3    if (data[i] <= 10)
4      break;
5  }
6  if (i == n) printf("Igaz."); /* végigértünk? */
7  else printf("Nem igaz.");
```

- Figyeljük meg, hogy
  - amikor a `break` kiugrik a `for` ciklusból, *i* növelése már nem történik meg, ezért a válasz akkor is helyes, ha csak az utolsó elemnél ugrunk ki.

Kezdeti értékek

- Ha egy tömböt a tanult módon deklarálunk, tartalma inicializálatlan, vagyis memóriaszemét.

```
1 int numbers[5]; /* véletlen tartalom, memóriaszemét */
```

Ez nem baj, csak feltöltés előtt ne használjuk a tömbelemeket.

- A skalár változókhoz hasonlóan tömböknél is lehetséges a kezdetiérték-adás:

```
1 int numbers[5] = {1, -2, -3, 2, 4};
```

- Ilyenkor (és csakis ilyenkor!) a méret meghatározását le is hagyhatjuk, hiszen kiderül a lista hosszából:

```
1 int numbers[] = {1, -2, -3, 2, 4};
```

- De ez is helyes:

```
1 int numbers[5] = {1, -2, -3 /* 0 , 0 */};
```

Leválogatás

- Gyűjtsük egy másik vektorba azokat az elemeket, melyek rendelkeznek egy bizonyos tulajdonsággal!
- Írjuk ki, hogy hány elemet másoltunk át!
- Legyen az egészeket tartalmazó forrástömb neve `data`, elemszáma 5.
- Legyen a céltömb neve `selected`, elemszámnak az 5 nyilván elégséges.
- Gyűjtsük külön a negatív elemeket!

Leválogatás

- A data tömböt egyszer be kell járni a már ismert módon.
- Jelölje *n*, hogy hány elemet másoltunk már át a selected tömbbe.
- *n* értéke kezdetben 0, minden másoláskor növeljük.

```
1  int data[5] = {-1, 2, 3, -4, -7};/* deklarációk */
2  int selected[5];
3  int i, n;
4  n = 0;                           /* előkészítés */
5  for (i = 0; i < 5; i=i+1)        /* bejárás */
6  {
7    if (data[i] < 0)               /* vizsgálat */
8    {
9      selected[n] = data[i];       /* másolás */
10     n = n+1;
11   }
12 }
13 printf("Negatívak száma: %d", n);/* válasz */
```

link

Leválogatás

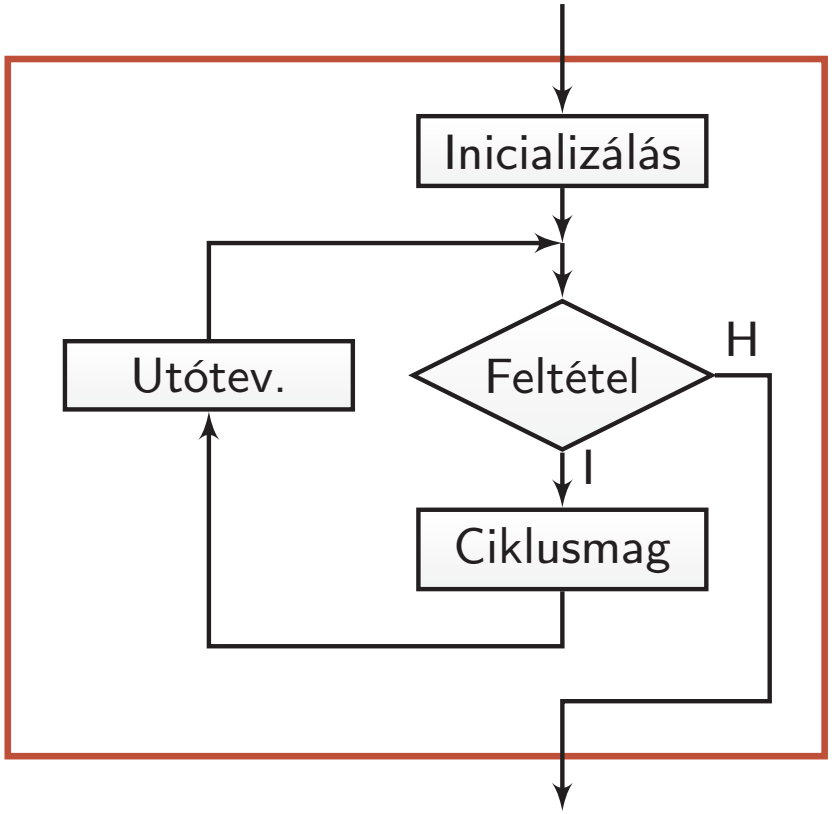
- Kicsit másként szervezett megoldás:

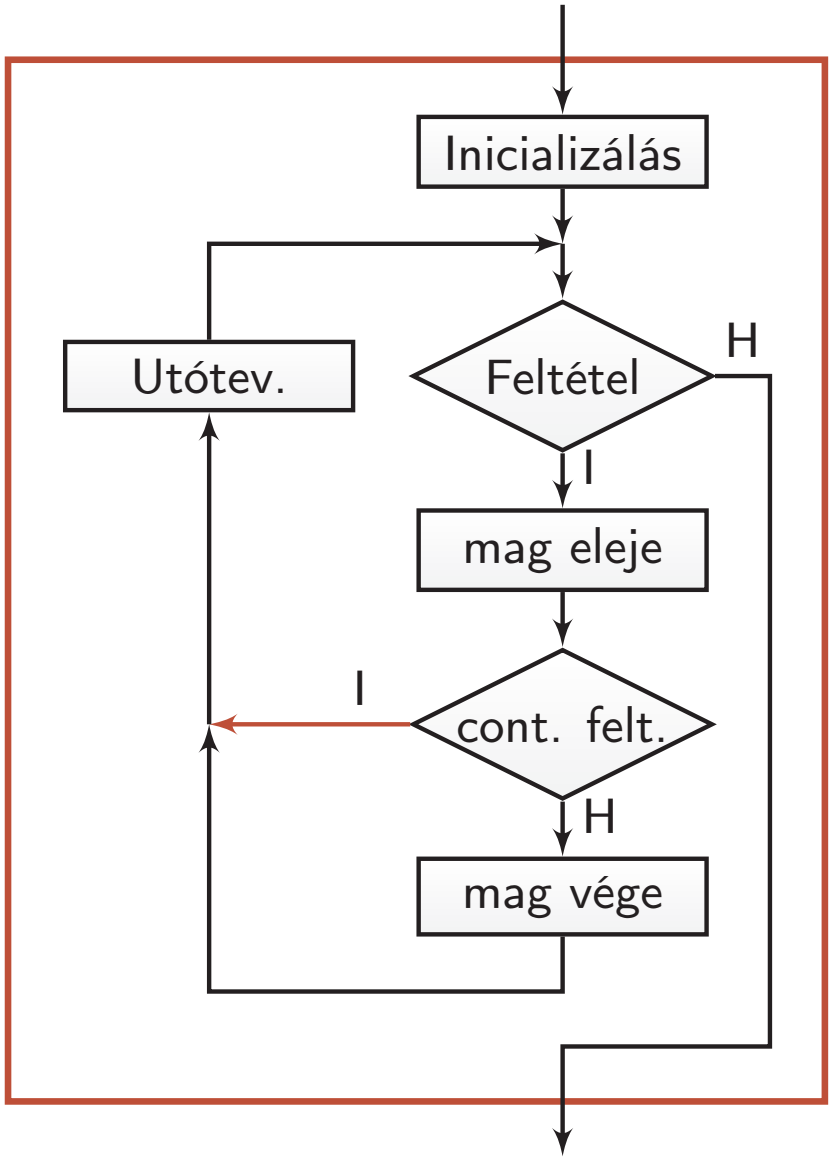
```
1  n = 0;                                  /* előkészítés */
2  for (i = 0; i < 5; i=i+1)               /* bejárás */
3  {
4    if (data[i] >= 0)                     /* vizsgálat */
5      continue;
6    selected[n] = data[i];                /* másolás */
7    n = n+1;
8  }
9  printf("Negatívak száma: %d", n);/* válasz */      link
```

- A `continue` utasítás megszakítja az őt közvetlenül tartalmazó `while`, `for`, `do` ciklus ciklusmagjának végrehajtását, és a következő iterációval folytatja a ciklust

  Ez sem strukturált elem
- Csak a ciklusmagot szakítja meg,
  `for` ciklusban alkalmazva az utótevékenységet végrehajtja

for ciklus continue nélkül és continue-val





Helyben szétválogatás

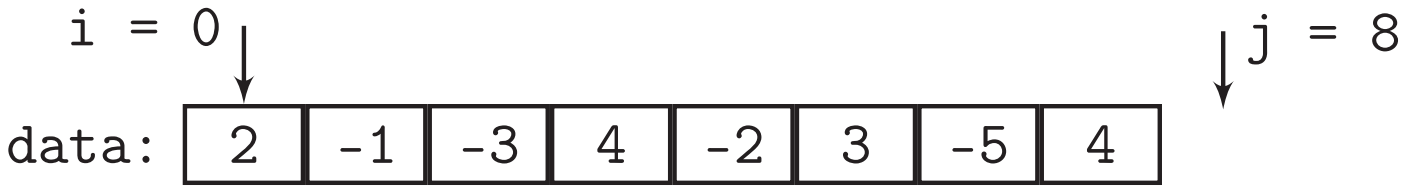
- Válasszuk ketté helyben a `data` tömb elemeit úgy, hogy a negatív elemek a tömb végébe kerüljenek!
- Írjuk ki, hogy hányadik pozícióban van az első negatív elem!

Helyben szétválogatás

- Az algoritmus

```
i ← 0;
j ← n;
AMÍG i < j
  HA data[i] >= 0
    i ← i+1;
  EGYÉBKÉNT
    j ← j-1;
    data[i] ↔ data[j]
KI: i
```

- Egy teszt $n = 8$ elemű vektorral



Helyben szétválogatás

- Az algoritmus

```
i ← 0;
j ← n;
AMÍG i < j
  HA data[i] >= 0
    i ← i+1;
  EGYÉBKÉNT
    j ← j-1;
    data[i] ↔ data[j]
KI: i
```

- Teljes? Véges? – bizonyítsuk be!
  - Minden iterációban i vagy j lép $\rightarrow$ véges, n lépés
  - i akkor lép, ha nemnegatívon áll, $\rightarrow$ i-től balra csak nemnegatívak
  - miután j lép, tartalmát negatívra cseréljük $\rightarrow$ j-től kezdve csak negatívak
  - Ha összeérnek, a tömb szét van válogatva

Helyben szétválogatás

- Akkor kódoljunk, nincs is annál jobb!

```
1  int i = 0, j = 8;
2  while (i < j)
3  {
4    if (data[i] >= 0)
5      i=i+1;
6    else
7    {
8      int xchg;
9      j=j-1;
10     xchg = data[i];     /* változó értékek cseréje */
11     data[i] = data[j]; /* nagyon gyakori fordulat */
12     data[j] = xchg;
13   }
14 }
15 printf("Az első negatív elem indexe: %d", i);
```

link

Köszönöm a figyelmet.