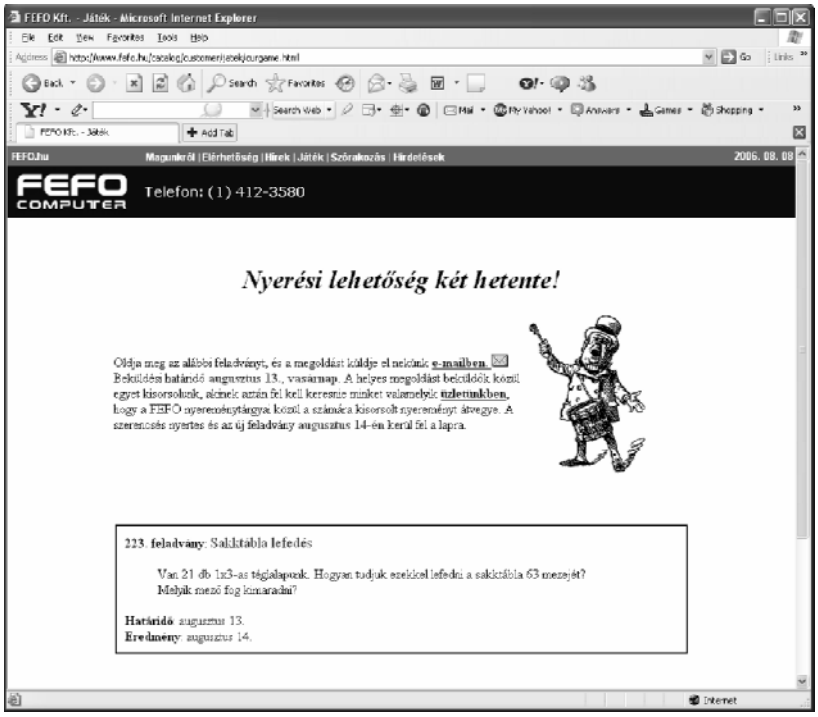


Kellemes és hasznos szórakozás mindenki számára, aki kedveli a logikai fejtörőket. Jó böngészést!

firkácska

Érdekes informatika feladatok

XV. rész

Tömbök tárolása (1.)

A *tömbök* olyan lineáris adatszerkezetek, amelyek elemeket tartalmaznak és az egyes elemekre indexszel lehet hivatkozni. Az elemek általában azonos típusúak, de egyes programozási nyelvek megengedik különböző típusú elemeket tartalmazó tömbök használatát is.

Az egydimenziós tömböket *vektorok*nak, a kétdimenziós tömböket *mátrixok*nak nevezzük. Léteznek többdimenziós tömbök is (*n-dimenziós tömbök*).

Programozási nyelvek elemzésekor a tömbök esetén a következő kérdésekre keressük a választ:

- Mi lehet az indexe?
- Mi lehet az eleme?

- Csak ugyanolyan típusú elemei lehetnek?
- Van-e indextúlcsordulás-ellenőrzés?
- Van-e kezdőérték-adás?
- Van-e egyben értékadás?
- Vannak-e dinamikus tömbök?
- Vannak-e konstans tömbök?
- Mikor dől el a mérete, a helyfoglalása?
- Van-e többdimenziós tömb?
- Van-e altömb (szelet) képzés?
- Vannak-e speciális tömbök?

Általában minden egyes programozási nyelv külön szintaktikai elemeket vezet be tömbök deklarálására.

A Pascal nyelv esetén n-dimenziós tömböt az

array[alsóhatár$_1$..felsőhatár$_1$, …, alsóhatár$_n$..felsőhatár$_n$] of elemtípus;

szintaktikai konstrukcióval lehet deklarálni. Amennyiben ez a konstrukció egy *type* cikkelyben van jelen típust deklaráltunk, ha egy *var* cikkelyben van jelen, akkor ilyen típusú változót deklaráltunk.

Megemlítendő az, hogy amennyiben egy *var* cikkelyben tömbváltozót deklarálunk, a változó számára a Pascal egy névtelen típust hoz létre. Ha több tömbváltozót deklarálunk, akkor az ugyanabban a sorban deklarált változók számára ugyanazt a névtelen típust hozza létre a fordítóprogram, a különböző sorokban deklarált változókhoz pedig különböző névtelen típust hoz létre.

Például az alábbi deklaráció esetén

```
var
  a, b: array[1..10] of integer;
  c: array[1..10] of integer;
```

az *a* és a *b* vektorok ugyanolyan típusúak lesznek (így megengedhető lesz pl. az *a := b* értékadás), a *c* vektor pedig különböző típusú lesz, hiába egyezik meg szintaktikailag a deklaráció az *a* és a *b* tömbök deklarációjával (ebben az esetben egy *c := a* értékadás például szemantikai hibát eredményez).

A névtelen típusok használatát úgy küszöbölhetjük ki, hogy típusként deklaráljuk a tömböt, majd ilyen típusú változókat használunk:

```
type
  TVektor = array[1..10] of integer;
var
  a, b: TVektor;
  c: TVektor;
```

Érdekes, hogy hogyan oldja meg a *C* a tömbök kezelését. A tömbváltozó egy mutató, amely a tömb első elemére mutat. Így szoros kapcsolat jön létre a tömbök és a mutatók között, azzal a megkötéssel, hogy míg egy mutató értékét meg lehet változtatni, a tömb mindig az első elemére mutat, tehát a tömböt egy konstans pointernek tekinthetjük. Például legyen p egy mutató és t egy tömb. A $p = t$ értékadás helyes, és eredményeként a p mutató is a t tömbre (pontosabban a tömb első elemére fog mutatni), viszont a $t = p$ értékadás már helytelen, mert a t konstans mutató.

A másik nagy különbség az a C és a Pascal között, hogy a C úgynevezett zéróindexű tömböket használ, a tömböknél nem kell megadni alsó és felső indexhatárt, csak

az elemszámot, és a tömbök indextartománya 0 és a megadott elemszám - 1 között fog mozogni.

Mátrixok vektorrá alakítása

Gyakran szükségünk lehet arra, hogy többdimenziós tömböket egydimenziós tömbökké bontsunk le. A következő példa mátrixok vektorokká alakítását mutatja be:

```
var
   m: array[1..3, 1..3] of integer;
   v: array[1..9] of integer;
   i, j: byte;

begin
  {Beolvassuk a matrixot}
  for i := 1 to 3 do
   for j := 1 to 3 do
    begin
      write('m[', i, ', ', j, '] = ');
      readln(m[i, j]);
    end;
  {Kiirjuk a matrixot}
  for i := 1 to 3 do
   begin
     for j := 1 to 3 do
      write(m[i, j]:5);
     writeln;
   end;
  {Atalakitjuk vektorra}
  for i := 1 to 9 do
   v[i] := m[(i-1) div 3 + 1, (i-1) mod 3 + 1];
  for i := 1 to 9 do
   write(v[i]:5);
  writeln;
  {Masodik modszer az atalakitasra}
  for i := 1 to 3 do
   for j := 1 to 3 do
    v[(i-1)*3+j] := m[i, j];
  for i := 1 to 9 do
   write(v[i]:5);
  writeln;
end.
```

Dinamikus tömbök

Tömböket is kezelhetünk dinamikusan. Pascalban ekkor csak a tömb „tartóelemét" kell hogy deklaráljuk. Ez a „tartóelem" egy absztrakt tömb: `array[1..1] of típus;` Erre definiáljuk dinamikusan a mutatót. A tömbre ezután nem a megszokott `nev[index]` konstrukcióval, hanem ennek dinamikus megfelelőjével: `nev^[index]` hivatkozunk. Az adatok kezelése a statikus tömbökéhez hasonlít. Ha több dimenziós tömböket akarunk létrehozni, akkor nekünk kell megírnunk az indexelés menetét, a konverziót egydimenziós tömbbé. Például ha egy kétdimenziós tömböt (mátrixot) hozunk létre és a szokásos *sor, oszlop* módszerrel szeretnénk indexelni, akkor a következő index konverziós műveletet kell végrehajtanunk:

*MemóriaIndex := (sor - 1) * OszlopokSzáma + oszlop;*

Példaprogram: A következő program egy komplex dinamikus mátrix- és vektorkezelő osztályt implementál. A tömbök adatzónája `PDataObject` típusú dinamikus, absztrakt objektumok. Ezt a típust kell felülírni a programban.

A unit:

```
unit GenMat;

interface

type
  PDataObject = ^TDataObject;
  PDataArray = ^TDataArray;
  TDataArray = array[1..1] of PDataObject;

 PDataMatrix = ^TDataMatrix;
 TDataMatrix = object
  Block: PDataArray;
  Line, Column: word;
  Size: word;
  constructor Init(n, m: integer);
  destructor Done; virtual;
  function CalcInd(i, j: integer): integer; virtual;
  procedure SetM(i, j: word; value: PDataObject); virtual;
  function GetM(i, j: word): PDataObject; virtual;
 end;

 PDataVector = ^TDataVector;
 TDataVector = object(TDataMatrix)
  constructor Init(n: integer);
  procedure SetV(i: word; value: PDataObject); virtual;
  function GetV(i: word): PDataObject; virtual;
 end;

 TDataObject = object
  constructor Init;
  destructor Done; virtual;
 end;

implementation

{****  TDataObject  ***}
constructor TDataObject.Init;
begin
end;

destructor TDataObject.Done;
begin
end;

{****  TDataMatrix  ****}
constructor TDataMatrix.Init;
begin
   Line := n;
   Column := m;
   Size :=  Line * Column * SizeOf(pointer);
   GetMem(Block, Size);
   if Block = nil then Fail;
end;

destructor TDataMatrix.Done;
var i, j: word;
```

```
begin
  for i := 1 to Line do
   for j := 1 to Column do
    if Block^[CalcInd(i, j)] <> nil then
dispose(Block^[CalcInd(i, j)], Done);
   FreeMem(Block, Size);
end;

function TDataMatrix.CalcInd;
begin
  CalcInd := (i- 1) * Column + j;
end;

procedure TDataMatrix.SetM;
begin
  Block^[CalcInd(i, j)] := value;
end;

function  TDataMatrix.GetM;
begin
  GetM := Block^[CalcInd(i, j)];
end;

{****  TDataVector  ****}
constructor TDataVector.Init;
begin
  inherited Init(1, n);
end;

procedure TDataVector.SetV;
begin
  Block^[i] := value;
end;

function TDataVector.GetV;
begin
  GetV := Block^[i];
end;

{* Main *}
end.
```

A program:

```
program Matrixok;
uses GenMat;

type
 PData = ^TData;
 TData = object(TDataObject)
  Data: word;
  constructor Init(wData: word);
  function GetData: word; virtual;
 end;

{*******  TData  *******}
constructor TData.Init;
begin
  inherited Init;
  Data := wData;
end;
```

```
function TData.GetData;
begin
  GetData := Data;
end;

{* Main *}
var
   m: PDataMatrix;
   v: PDataVector;
   i, j: integer;
   d: PData;

begin
  m := new(PDataMatrix, Init(3, 2));
  for i := 1 to 3 do
   for j := 1 to 2 do
    begin
      d := new(PData, Init(i+j));
      m^.SetM(i, j, d);
    end;
  v := new(PDataVector, Init(5));
  for i := 1 to 5 do
   begin
     d := new(PData, Init(i));
     v^.SetV(i, d);
   end;
  for i := 1 to 3 do
   begin
    for j := 1 to 2 do write(PData(m^.GetM(i, j))^.
GetData:4);
    writeln;
   end;
  writeln;
  for i := 1 to 5 do write(PData(v^.GetV(i))^.GetData:4);
  writeln;
  dispose(m, Done);
  dispose(v, Done);
  readln;
end.
```

*Delphi*ben például a `Variant` típus segítségével egész számokkal indexelhető dinamikus tömböket is létre lehet hozni. A tömbök elemei tetszőleges típusúak – akár tömbök is – lehetnek:

```
var
  a: Variant;
begin
  a := VarArrayCreate([0, 4], varVariant);
  a[0] := 1;
  a[1] := 1234.5678;
  a[2] := 'szöveg';
  a[3] := true;
  a[4] := VarArrayOf([1, 10, 100, 1000]);
  writeln(a[2]);    {szöveg}
  writeln(a[4][2]); {100}
end;
```

Létezik erre azonban egy elegánsabb módszer is: *Delphi*ben dinamikus tömböket is használhatunk, deklarációkor nem kell megadni a tartomány határait. Az ilyen tömbök

méretét a `SetLength()` eljárással állíthatjuk be dinamikusan, a tömb elemei a megadott típusúak kell hogy legyenek.

Például:

```
var
  a: array of integer;
begin
  SetLength(a, 1);
  a[0] := 1;
  ...
end;
```

Mátrixok ábrázolása listákkal

Mátrixokat ábrázolhatunk listák segítségével is. Ez különösen az úgynevezett *ritka mátrix*ok esetén igen hasznos. *Ritka mátrixok*nak olyan mátrixokat nevezünk, amelyek nagyon sok zérós elemet tartalmaznak. Nagyon nagy adatoknál nem éri meg, hogy a memóriában ezeket mátrix alakjában ábrázoljuk, hisz a sok zérót feleslegesen tároljuk. Egy ilyen adatstruktúrát egyszerűbb listák segítségével ábrázolni.

Vegyük a következő példát:

A mátrix:

$$\begin{pmatrix} 10 & 0 & 0 & 0 & 0 \\ 0 & 6 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 \end{pmatrix}$$

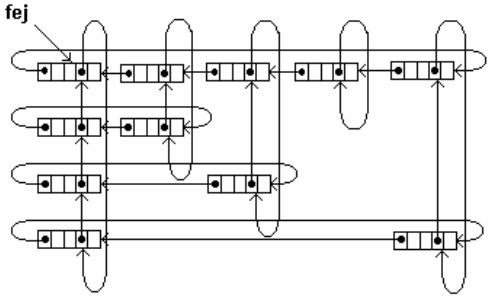
A mátrix minden elemét így ábrázoljuk:

balra	*sor*	*oszlop*	*fel*	*érték*

Ahol:

- *sor*: az elem sora
- *oszlop*: az elem oszlopa
- *érték*: az elem értéke
- *fel*: mutató a felső elemre
- *balra*: mutató a balra lévő elemre

A létrehozott lista így néz ki:



A *fej* a lista elejére mutat. A listába bevettünk még egy fej sort és egy fej oszlopot, így könnyebb a megfelelő indexek azonosítása. A listába új elemet úgy helyezünk be, hogy megkeressük a fejoszlopban az elem sorát, majd addig megyünk végig a soron, míg az

elem oszlopszámánál nagyobb oszlopot találunk. Így megkapjuk, hogy mi elé kell beillesszük az elemet. Ha ennek az oszlopszáma megegyezik az elemével, akkor csak az adatzónáját cseréljük ki, ha nem, akkor eléje kötjük az elemünket. Így járunk el a felfelé mutató bekötésekor is. A listában egy elemet hasonlóan keresünk meg. Ha a megfelelő sor és oszlopszámú elem nincs a listában, akkor ez zéró.

Példaprogram: A következő példaprogram ritka mátrixokat kezelő osztályt implementál:

```
program RitkaMatrixok;
type
 PElem = ^TElem;
 TElem = record
  Row, Column: integer;
  Data: real;
  Up, Left: PElem;
 end;

 PMatrix = ^TMatrix;
 TMatrix = object
  Block: PElem;
  Row, Column: integer;
  RHeap: pointer;
  constructor Init(n, m: integer);
  destructor Done; virtual;
  procedure SetM(i, j: integer; value: real); virtual;
  function GetM(i, j: integer): real; virtual;
 end;

{*******  TMatrix  *******}
constructor TMatrix.Init;
var i: integer;
      tmp, p: PElem;
begin
  Row := n;
  Column := m;
  Mark(RHeap);
  new(Block);
  Block^.Row := 0;
  Block^.Column := 0;
  Block^.Data := 0;
  Block^.Left := Block;
  Block^.Up := Block;
  p := Block;
  for i := 1 to Column do
   begin
    new(tmp);
    tmp^.Data := 0;
    tmp^.Row := 0;
    tmp^.Column := i;
    tmp^.Left := p;
    tmp^.Up := tmp;
    Block^.Left := tmp;
    p := tmp;
   end;
  p := Block;
  for i := 1 to Row do
   begin
    new(tmp);
    tmp^.Data := 0;
```

```
    tmp^.Row := i;
    tmp^.Column := 0;
    tmp^.Left := tmp;
    tmp^.Up := p;
    Block^.Up := tmp;
    p := tmp;
   end;
end;

destructor TMatrix.Done;
begin
  Release(RHeap);
end;

procedure TMatrix.SetM;
var tmp, p, q: PElem;
begin
  if (i in [1..Row]) and (j in [1..Column]) then
   begin
    p := Block^.Up;
    while p^.Row <> i do p := p^.Up;
    q := p;
    while q^.Left^.Column > j do q := q^.Left;
    if q^.Left^.Column = j then q^.Left^.Data := value
    else
     begin
      new(tmp);
      tmp^.Row := i;
      tmp^.Column := j;
      tmp^.Data := value;
      tmp^.Left := q^.Left;
      q^.Left := tmp;
      p := Block^.Left;
      while p^.Column <> j do p := p^.Left;
      q := p;
      while q^.Up^.Row > i do q := q^.Up;
      tmp^.Up := q^.Up;
      q^.Up := tmp;
     end;
   end;
end;

function TMatrix.GetM;
var p, q : PElem;
begin
  GetM := 0;
  if (i in [1..Row]) and (j in [1..Column]) then
   begin
    p := Block^.Up;
    while p^.Row <> i do p := p^.Up;
    q := p^.Left;
    while q^.Column > j do q := q^.Left;
    if q^.Column = j then GetM := q^.Data;
   end;
end;

var
   i, j: integer;
   m: PMatrix;
begin
   m := new(PMatrix, Init(4, 3));
```

```
    m^.SetM(1, 3, 10);
    m^.SetM(1, 3, 20);
    m^.SetM(1, 2, 30);
    m^.SetM(4, 3, 7.5);
    m^.SetM(3, 2, 3.5);
    for i := 1 to 4 do
     begin
      for j := 1 to 3 do
       write(m^.GetM(i, j):6:2);
      writeln;
     end;
    dispose(m, Done);
    readln;
end.
```

Kovács Lehel István

Alfa-fizikusok versenye

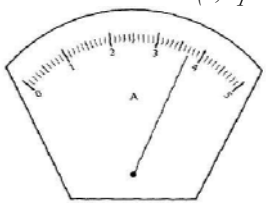
2002-2003.

VIII. osztály – V. forduló – döntő

1. Az alábbi távolságok különböző köröknek a sugarai.
Válaszd ki a legnagyobb és a legkisebb kört a felsoroltak közül! *(2 pont)*
r_1| = 5,5 cm; r_2 = 0,55 m; r_3 = 55 mm; r_4 = 5,5 m; r_5 = 5,5-10^3cm
r_6 = 0,055 m; r_7 = 5,5 dm; r_8 = 0,0055 km; r_9 = 0,55 mm.

2. Hány köbcentiméterrel több, mint 100 l? *(3 pont)*
V_1 = 500 dm^3; V_2 = 4,5 hl; V_3 = 0,5 m^3;
V_4 = 150000 cm^3; V_5 = 0,3 m^3; V_6 = 14 hl.

3. Olvasd le az ampermérő által mutatott értéket, ha a műszer méréshatára *(1,5 pont)*
a). 0,5 A; b). 2,5 A
a) ..... A b) ..... A



4. Írd be a megfelelő relációjeleket! *(1,5 pont)*

$U_1<U_2$	$U_1=U_2$	$U_1=U_2$
$R_1=R_2$	$R_1>R_2$	$R_1=R_2$
I_1 I_2	I_1 I_2	I_1 I_2

5. Az ábrán látható elrendezésben a tolóellenállás harmadrészénél áll a csúszka. Mozdítsuk el a jelzett irányba a csúszkát úgy, hogy most a másik végétől legyen harmadrésznyire. Hogyan változnak a mérőműszerek által felvett értékek? A tolóellenállás teljes