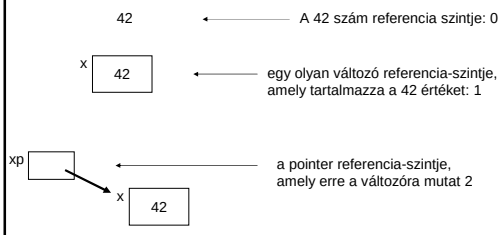
Pointer és referencia típusok

- Egy pointer (és egy referencia) egy olyan objektum, amely megadja egy másik objektum címét a memóriában.
- Egy pointer értéke egy **memóriacím** (gépi nyelvekben az indirekt címzés lehetősége motiválta a pointerek létrehozását).
- Vannak típusos és típus nélküli pointerek.

- Pointerek segítségével magasabb referencia-szinten hivatkozhatunk objektumainkra.



Mire kellenek a mutatók?

- hatékonyság - ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.

x

23	10	11	17	0	34	28	22	55	88
4	7	0	0	6	8	1	9	10	3

y

23	10	11	17	0	34	28	22	55	88
4	7	0	0	6	8	1	9	10	3

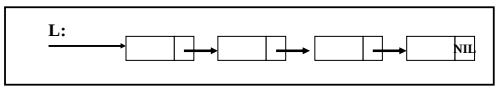
Mire kellenek a mutatók?

- hatékonyság -- ahelyett, hogy nagy adatszerkezeteket mozgatnánk a memóriában, sokkal hatékonyabb, ha az erre mutató pointert másoljuk, mozgatjuk.

itt vigyázni kell az osztott használatra!

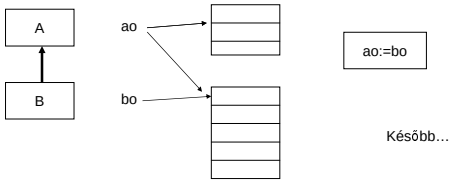
Mire kellenek a mutatók?

- dinamikus adatszerkezetek építéséhez



Mire kellenek a mutatók?

- objektumorientált funkciókhoz - a programozási nyelvekben a polimorfizmust akkor tudjuk támogatni, ha a változók objektumokra való referenciákat tartalmaznak.



A szokásos műveletek:

- **értékadás** - pointerek között
- **egyenlőség vizsgálat** - ha két ugyanolyan típusú pointer ugyanarra az adatszerkezetre mutat
- **dereferencing -** a *mutatott* objektum részére vagy egészére való hivatkozás
- **referencing -** egy objektum címe
- új objektum dinamikus **allokálása**
- egy objektum **deallokálás**a - explicit művelettel vagy implicit módon egy garbage collector-ral
- néha (pl. C, C++) **összeadás**, **kivonás** is megengedett

„Csellengő" pointerek:

„Csellengő" pointer**:** kísérlet olyan változó elérésére, ami már nem létezik.

```
#include <iostream.h>
int *r;
double *r2;
void f(){int v; r=&v;}
void g(){
double v; v=2.1; r2=&v;}

int main(){
  f(); g(); *r=3; *r2=1.2;
cout <<"dangling *r="<<*r;
cout << "\n *r2 " <<*r2;cout << ".\n"; …}
```

Compiler, builder:
0 error(s), 0 warning(s)
Az eredmény megjósolhatatlan!

„Csellengő" pointerek 2.:

```
void main(){
  int *j,*i;
  double *d;
   j=new int;
  *j=3;
   i=j;
   delete j;
   d=new double;
  *d=4.2;
   cout << *i;}
```

Compiler, builder:
0 error(s), 0 warning(s)
Az eredmény megjósolhatatlan!

A programozási nyelvek között a lehetséges különbségek:

- Csak konkrét típusra mutató pointerek megengedettek, vagy vannak típus nélküli pointerek is?
- Csak dinamikusan allokált objektumokra mutathat pointer, vagy "normál" változókra is?
- Lehetnek-e alprogramra mutató pointerek is?

A programozási nyelvek között a lehetséges különbségek:

- Milyen fajta konstans pointerek megengedettek? (Pl.: egy tömbnév C-ben egy konstans pointer a tömb objektum 0. elemére.)
- Kötelező a pointer típusoknak önálló nevet adni, vagy csak a mutatott típust kell megadni? (Pl.: type Ip is access to Integer; ADA95-ben
  int * x; C-ben)

A programozási nyelvek között a lehetséges különbségek:

- Milyen biztonságosan kezelhető a "csellengő" pointerek problémája?
- Mi a megengedett műveletek halmaza?
- Kapnak a pointer változók kezdeti (üres) értéket a deklarációnál?
- Lehetséges-e ugyanazt az adatot két (vagy több) pointeren keresztül is változtatni/elérni?

- **Pascal:**
  - Megengedettek a konkrét típusra mutató és a típus nélküli pointerek is:
    var p1 :^typename; p2:Pointer;
  - A típus nélküli pointerek Pointer típusúak, van néhány művelet, amely Pointer típusú eredményt ad vissza (@, Addr, Ptr).
  - Ne használjunk pointer változókat mielőtt értéket adtunk volna nekik! (Használjuk a Nil-t erre a célra.)

- **Pascal:**
  - Pointerek mutathatnak "normál" változókra is, nem csak dinamikusan allokált objektumokra:

```
var p: pointer; w: word;…. p:=@w;
```

(vigyázat, csellengő pointerek veszélye!)

  - Lehetnek alprogramra mutató pointerek is:

```
type proctyp =
  procedure(x: byte; var y: real);
procp = ^proctyp;
```

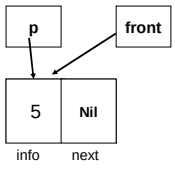
  - A Ptr(segment, offset) függvény egy konstans címet ad vissza. A nil pointer a Ptr(0,0).

- **Pascal:**
  - A megengedett műveletek halmaza:

Dereferencing	postfix ^
Dinamikus helyfoglalás	new
Deallocation	dispose
Értékadás	:=
Egyenlőség	=

Példa:

```
type link = ^cell;
    cell = record
info: integer;
next: link;
end;
var p, front: link;
….
new(p);
p^.info := 5;
p^.next := front;
front  := p; …stb.
```



- **CLU:**
  - A CLU-ban nincs hagyományos pointer típus. A program végrehajt műveleteket objektumokon. Az objektumok mint egy **univerzum** részei léteznek, a program változói hivatkoznak ezekre az objektumokra. Garbage collection a felszabadításra.
  - A programban kétféle objektum lehet:
    - mindig ugyanaz az értéke (immutable) és
    - változhat az értéke (mutable).

- **ADA95:**
  - Nem lehetséges a típus nélküli pointer, névtelen típusú sem, mindig kell egy konkrét típus:

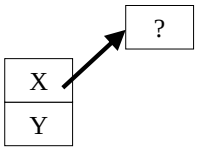
```
type Int_p is access  Integer;
```

Az `Int_p` típusú változók dinamikusan allokált egész objektumokra mutathatnak.

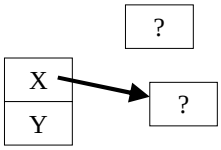
  - `null` érték - default kezdeti érték.

type P is access Integer;
X, Y: P;

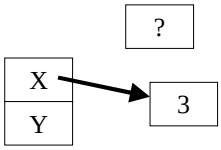
type P is access Integer;
X, Y: P;
begin
X := new Integer;



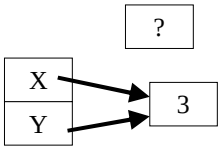
type P is access Integer;
X, Y: P;
begin
X := new Integer;
X := new Integer;



type P is access Integer;
X, Y: P;
begin
X := new Integer;
X := new Integer;
X.all := 3;

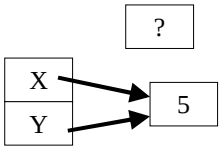


type P is access Integer;
X, Y: P;
begin
X := new Integer;
X := new Integer;
X.all := 3;
Y := X;



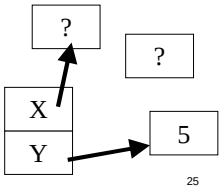
type P is access Integer;
X, Y: P;

X := new Integer;
X := new Integer;
X.all := 3;
Y := X;
X.all := 5;



type P is access Integer;
X, Y: P;

X := new Integer;
X := new Integer;
X.all := 3;
Y := X;
X.all := 5;
X := new Integer;….



- **ADA95:**

  mutathat „normál” változóra is

```
type Int_point is access all
  Integer;
K: aliased Integer;
I: Int_point := K'Access;
```

  – Az Access attribútum adja vissza a változó memóriacímét.

- **ADA95:**

  Lehetnek alprogramra mutató pointerek is:

```
type Int_Fn is access
  function(X: in Integer)
      return Integer;
F: array(1 .. 2) of Int_Fn;
function Sqr(X: in Integer)
  return Integer is
  begin return X*X;
end Sqr;
F(1) := Sqr'Access;
```


és ez lehet pl. egy alprogram hívás aktuális paramétere:

```
procedure Demo(Fcn: in Int_Fn) is
      X: Integer:=2;
   begin
     ...
    X := Fcn.all(X); ...
   end Demo;
Demo(F(1));
```

vagy hívhatjuk:

```
X:= F(I)(X); -- ez egy rövidítése az
  F(I).all(X)-nek!
```


- **ADA95:**
  - A megengedett műveletek halmaza:
  - Dereferencing: .comp-name .all
  - Dinamikus allokálás: new
  - Értékadás: :=
  - Egyenlőség: =
  - **Nincs** explicit eszköz a felszabadításra! A dinamikusan allokált objektum élettartama a **mutató típus** hatáskörétől függ.

Példa:

```
type Cell;    -- szükséges a nem teljes deklaráció!
  type Link is access Cell;
  type Cell is record
    Info: Integer;
    Next: Link;
  end record;
  P, Front: Link;
begin
  P:=new Cell;      -- allokálás
  P.Info := 5; --
  P.Next := Front;
  Front  := p; --referenciát másol
  Front:= new cell'(2,null); -- megengedett a
                             -- kezdeti érték adása
  Front.all:=P.all;-- a mutatott objektumot másolja
```

- **C++**:
  - A legtöbb `T` típusra, `T*` a megfelelő pointer típus:
    `int *p;`
  - A tömbökre és függvényekre mutató pointereknek kicsit bonyolultabb jelölése van:

```
int (*vp)[10]; // pointer 10 int tömbjére
int (*fp) (char, char*);
        // függvényre mutató pointer , melynek
        //(char, char*) argumentumai vannak és egy
          egészet (int) ad vissza
```


- **C++**:
  - A megengedett műveletek:

Dereferencing:	prefix *
Dinamikus allokálás	new
Deallokálás	delete
értékadás:	=
egyenlőség:	==
additív műveletek	+ -
increment, decrement	++ --
member ref.	.* ->*

- **C++**:
  - **van egy speciális operátor, az "address_of" '&', ennek segítségével adhatjuk értékül változók címét pointereknek:**

```
int i=10;
int *pi = &i; // a pi pointer az i változóra vonatkozik
int j=*pi; // j-t 10-re állítjuk
```

Az '&' operátorral létrehozhatjuk objektumok *referenciáit* is -- egy referencia úgy tekinthető mint egy konstans pointer, ami mindig automatikusan dereferenciát hajt végre:

```
int &r =i; // r és i ugyanarra vonatkozik
r=2; // i=2
```

Az increment, decrement stb. lehetőségek veszélyesek, vigyázzunk, ne keverjük össze a jelentését!

Pl.: `pi++` a pointert inkrementálja, és a következő memóriacímre fog mutatni, ennek akkor van értelme, ha `pi` egy tömbre mutat, míg `r++` inkrementálja `i` értékét.

- **Java:**
  - nincs hagyományos pointer típus. A változókban kétféle érték tárolható:
    - primitív értékek (egy numerikus típusból vagy egy logikai) és
    - referencia értékek. Az objektumokat (osztályok példányai vagy tömbök) referenciákkal kezeli.
  - Ugyanarra az objektumra számos referencia hivatkozhat.
  - Objektumok referenciáinak műveletei: mező elérés, metódus hívás, casting, string concatenation, instanceof, '==' '!=' (ref. egyenl.) stb.

- **Eiffel:**
  - Itt sincsenek hagyományos pointer típusok. A változókban kétféle érték tárolható - kiterjesztett értékek és referencia értékek. Ugyanarra az objektumra számos referencia hivatkozhat.
- C#:
  - A referencia típusok objektumait kezelhetjük referenciákkal.
  - Egy „unsafe" környezetben egy típus lehet pointer is, erre számos művelet megengedett (pl. a ++, -- is).

Hogyan definiálhatunk új adattípusokat?

- Példa
  - Pascal
    type <typen>= <value desc>;
    type myint=integer;
  - C++
    typedef <value desc> <typen>;
    typedef int myint;
  - Java class … később

Hogyan definiálhatunk új adattípusokat?

– ADA95

subtype <typen> is <typen1>;
type <typen> is new <typen1c>;
subtype Int is Integer;
type My_Int is new Integer;

– CLU

cluster .. később

– Eiffel

class … később

- Az értékhalmaz résztartományát is gyakran megadhatjuk. Pl. :

```
subtype Small_Int is Integer
  range 0..10;
```


Melyek a megengedett típus-konstrukciók?

- Iterált
  - egy kiinduló típusból
- Direkt szorzat
  - több kiinduló típusból
- Unió
  - több kiinduló típusból

Tömb típusok

- "Egy tömb egy olyan adatszerkezet, amely azonos típusú elemek sorozatait tartalmazza." Általában egy tömb egy *leképezés* egy folytonos diszkrét intervallumról elemek egy halmazára.

  Tömbnév(indexértékek) → elem
- A diszkrét intervallum elemeit hívjuk **index** értékeknek.
- Az elemek száma ebben az intervallumban definiálja a tömb **méretét**.

A legfontosabb kérdések:

– Milyen adattípusok lehetnek tömb típusok indextípusai?
– Mi lehet tömb típusok elemtípusa?
– Tartalmazzák-e a tömb típusok az indexhatárokat? És a tömb objektumok?
– Mikor dől el a mérete, a helyfoglalása?
– Van-e indextúlcsordulás-ellenőrzés?
– Van-e többdimenziós tömb? Van-e altömb (szelet) képzés? Van-e teljes tömbre vonatkozó értékadás? (kezdő értékadás?)Van-e tömbkonstans?
– Megváltoztatható-e egy tömb mérete? Rögzített méretű sorozat vagy nem?

- Az alapművelet az **indexelés** -- `A[i]`, az `A` tömb i. elemét gyorsan el kell tudni érni.
- Vannak programozási nyelvek, ahol az elemek különböző típusúak is lehetnek -- pl. SmallTalk, Clipper -- de általában az elemek ugyanahhoz a típushoz tartoznak, vagy egy adott típus lehetséges leszármazottai is lehetnek.

- **Pascal:**
  - Megengedett névvel rendelkező és névtelen tömb típusok használata.
  - Az index típusa egész, felsorolási vagy intervallum típus lehet, az elemek típusa tetszőleges típus.

```
type <array_type_name> =
  array[i1..j1] of <typen>;
```

  - A tömbök indexhatárait fordítási időben számítja ki, nem megengedett tömbtípus definíciójában változók használata.

- **Pascal:**
  - Többdimenziós tömbök is definiálhatók, akár tömbök tömbjeként, akár több index segítségével:

```
My_arr: array [1..5] of array
  [1..3] of word; vagy:
My_arr: array [1..5, 1..3] of word;
```

  - az elemeire mint `My_arr[i,j]` vagy mint `My_arr[i][j]`-re hivatkozhatunk, ahol i a sorindex, j az oszlopindex.
  - Sorfolytonosan tárolja a tömböket.

- **Pascal:**
  - Tömbkonstansok definiálhatók, itt az elemek típusa nem lehet fájl vagy mutató típus. Pl.:

```
const letters: array[1..5] of
      char=('a','b','c','d','e');
const letters2: array[1..5] of
                char=('abcde');
const t:array[1..2,1..2,1..3] of
  word = (((1,1,1),(2,2,2)),
          ((3,3,3),(4,4,4)));
```

  - Egy speciális beépített `String` típus kezeli a karakterek egydimenziós tömbjét.

- **CLU:**
  - **A beépített típuskonstrukciónak van *mutable* és *immutable* variánsa, konverziókkal. A *sorozat* típus nyelvi megvalósítása.**

Array (mutable):

  - **dinamikus, egy tömb mindkét végén tudja a hosszát változtatni létrehozása után.**
  - **egydimenziós, az index csak egész lehet, a több dimenziós tömb mindig tömbök tömbje.**
  - **egy tömbnek mindig van egy alsó és felső indexhatára, ez változhat a program végrehajtása során.**
  - **Pl. az `array[int]` típus egész elemeket tartalmazó tömbtípust definiál, az `array[array[int]]` típus elemei egészekből álló tömbök.**
  - **A tömbhatárok nem részei a típusnak!**

  - ***Létrehozás*:**

`new = proc () returns (array[T])` visszaad egy új, üres tömböt, alsó indexhatára 1, felső indexhatára 0.

  - **Egy nem üres tömb létrehozása:**

```
array[int]$[3 : 6 , 17 , 24]
   % alsó indexhatár 3, elemek: 6,17,24.
```

  - ***Műveletek:* `size, low, high, fetch, store, addh, addl, remh, reml`**

```
x := a[i] vagy:
x := array[int]$fetch(a,i)
a[i] := x vagy:  array[int]$store(a,i,x)
```

  - **"Szintaktikai cukor "…**
  - **`addh, addl` kiterjesztik a tömböt a felső/alsó pozíción**
  - **`remh, reml` csonkítják a tömböt és visszaadják a felső/alsó végéről eltávolított elemet.**

Sequence: az immutable tömb típus

- **A tömbökhöz hasonlóan egydimenziós, az index csak egész lehet.**
- **A tömböktől eltérően nem változtathatóak létrehozás után, és az alsó indexhatár mindig 1.**
- ***Létrehozás*:**

```
new = proc () returns (sequence[T])
```

vagy: `sequence[int]$[6, 17, 24, 100]`

- **az alsó határ nem szükséges, mert mindig 1.**
- ***Műveletek:* `size, low, high, fetch, replace, addh, addl, remh, reml, concat.`**
- **`addh, addl` kiterjesztik a sorozatot a felső/alsó pozíción,**
- **`remh, reml` csonkítják a sorozatot és visszaadják a felső/alsó végéről eltávolított elemet, de ahelyett, hogy módosítanák az eredetit, egy új sorozatot adnak vissza a megfelelő elemekkel.**
- **A tömbök `store` művelete helyett `replace` műveletük van.**
- **A `concat` (||) művelet „konkatenál", azaz egymás után ír 2 sorozatot.**

- **ADA95:**
  - Megengedett névvel rendelkező és névtelen tömb típusok használata. Az index típusa tetszőleges diszkrét típus lehet, az elemek típusa tetszőleges típus:

```
My_seq: array(Integer range 1 .. 6)
  of Integer;
```

```
Work : array(Napok) of Natural;--
```

feltéve, hogy a `Napok` típus egy már definiált felsorolási típus.

  - **Tömb típusok definiálhatók rögzített és megszorítás nélküli indexhatárokkal:**

```
type A is array(Integer range 2 .. 10)
                              of Boolean;
type Vect is array(Integer range <>)
                              of Integer;
type Matr is array(
       Integer range <>,Integer range <>)
                                 of Integer;
```

  - **A konkrét indexhatárokat az adott objektum deklarációjánál kell meghatározni:**

```
V : Vect(1 .. 30);
A : Matr(1 .. 2, 1 .. 4);
```

  - **Gyakran használják alprogramok paramétereként, sablonoknál.**

  - A definíciókat *futási időben* értékeli ki, az aktuális indexhatárok nem kell, hogy statikusak legyenek.
  - Tömbök szeletei is létrehozhatók: `V(2..12)`,
  - de: `V(1..1)` <-> `V(1)`!
  - Megengedett az értékadás azonos típusú tömbök között:

```
V(1..5):= V(2..6);
```

  - és tömb aggregátokkal is:

```
V:=(1..3=>1, others=>0);
```

  - A korlátozás nélküli egydimenziós tömbökre az '`&`' konkatenáció is elérhető.

  - Két előredefiniált string típus:

```
subtype Positive is Integer
    range 1 .. Integer'Last;
type String is array
    (Positive range <>) of Character;
type Wide_String is array
    (Positive range <>) of Wide_Character;
```

  - Vannak speciális attribútumai:

A'First/A'First(N),
A'Last/A'Last(N),
A'Range/A'Range(N),
A'Length/A'Length(N)


```
generic
  type Elem is private;
  type Index is (<>);
  type Vekt is array (Index range <>) of Elem;
  procedure Glinker (V: Vekt; E: Elem;
                     Found: out Boolean; Ind: out Index);

procedure Glinker (V: Vekt; E: Elem; Found: out Boolean;
                                     Ind: out Index) is
begin
  Ind:=V'First;
  Found :=V(V'First)=E;
  while not Found and Ind<V'Last loop
    Ind:=Index'Succ(Ind);
    Found:=V(Ind)=E;
  end loop;
end;
```

- **C++:**
  - **Egy `T` típusra, `T x[size]` a `T` típusú elemek size méretű tömbje. Az indexek `0` és `size-1` között.**

```
float v[3];  // 3 float tömbje
int a[2][5]; // 5 int két tömbje
```

  - **Kezdeti érték adható:**

```
char v[2][5] = {{'a','b','c','d','e'},
                {'0','1','2','3','4'}};
```

  - **A pointerek és tömbök szoros kapcsolatban vannak: egy tömbnév mindig a tömb nulladik elemére hivatkozik, így használható a pointeraritmetika tömbökre. Pl.:**

```
strcpy(s,t)
char *s,*t;
{
while ((*s++=*t++)!='\0');
} (K&R)
```

  - **Nem tudja a méretét.**

- **Java:**
  "Java arrays are objects, are dynamically created and may be assigned to variables of type Object."
  `int [] ai; //` **array of integers**
  `short [][] as1,as2; //` **as1 és as2 short-ok tömbjének tömbjei**
  – **ha a `'[]'` –t a változó neve után írjuk, akkor csak ez a változó lesz tömb:**
  `long l, al[]; //` **`l` egy `long` típusú változó,**
  `//` **`al long`** **típusú elemek tömbje**
  – **Tömb létrehozása:**
  `a = new int[20];`
  – **A tömb mérete lekérdezhető: `a.length`.**
  – **Kezdeti érték adható:**
  `String []colours ={"red","white","green"};`

– Egy többdimenziós tömbben az elemeknek lehet különböző mérete:

```
int[][] m = new int[3][];
for (int i=0; i<m.length; i++ ) {
  m[i] = new int[i+1];
  for (int j=0; j<m[i].length; j++)
    m[i][j] = 0;
 }
```

– A szövegek kezelését a String és StringBuffer osztályokkal oldották meg. (Karakterek egy tömbje nem `String`!)

- **C#**
  A tömb elemeinek számozása 0-val kezdődik, kétféleképpen lehet deklarálni: adott hosszúságú vagy dinamikus.
  A nyelvben a tömbök objektumok, a deklaráció után szükség van a tömb példányosítására (new) Inicializásra: { }
  – **Adott méretű tömb deklarálása:**
  int[] Tomb; Tomb = new int[3];
  – **Ugyanez a tömb inicializálva:**
  Tomb = new int[3] { 1,2,3 }
  – **Dinamikus tömb létrehozása inicializálással:**
  Tomb = new int[] { 1,2,3 }
  – **A deklarációval egybekötött inicializáció:**
  int[] Tomb = new int[3] { 1,2,3 }
  – **Ha egy tömböt nem inicializálunk , akkor a tömb elemei automatikusan inicializálódnak a elem típusának alapértelmezett inicializáló értékére.**

– A tömbök lehetőségei:
egydimenziós tömbök,
többdimenziósak vagy négyszögszerűek,
kesztyűszerűek (tömbök tömbjei)
kevert típusúak (az előzőekből)
– Példa egy kesztyűszerű dinamikus tömbre:
int[][] numArray = new int[][] { new int[] {1,3,5}, new int[] {2,4,6,8,10} };
– minden tömb típus a System.Array bázistípusból „származik”.
Az Array osztály egy absztrakt bázisosztály, de a CreateInstance metódusa létre tud hozni tömböket. Ez biztosítja a műveleteket a tömbök létrehozásához, módosításához, bennük való kereséshez illetve rendezésükhöz.

Az Array osztály tulajdonságait megadó függvények:
– IsFixedSize - rögzített hosszúságú-e
– IsReadOnly - írásvédett-e
– IsSynchronized - a tömb elérése kizárólagos-e (thread-safe)
– Length - a tömb elemeinek száma
– Rank - a tömb dimenzióinak száma
– SyncRoot - Visszatér egy objektummal, amit a tömb szinkronizált hozzáféréséhez használhatunk

Az Array osztályban még számos szolgáltatás:
- **BinarySearch - bináris keresés a tömbön**
- **Clear - minden elemet töröl a tömbből és az elemszámot 0-ra állítja**
- **Clone - másolatot készít**
- **Copy - egy tömb részét átmásolja egy másik tömbbe, végrehajtja az esedékes típuskényszerítést és csomagolást (boxing).**
- **CopyTo - átmásolja az elemeket egy egy-dimenziós tömbből egy másik egy-dimenziós tömbbe egy megadott indextől kezdve.**
- **CreateInstance - Létrehoz egy tömb példányt.**
- **GetEnumerator - Visszatér egy IEnumerator-ral a tömbhöz.**
- **GetLength - az elemek száma**
- **GetLowerBound - Megadja a tömb alsó korlátját.**
- **GetUpperBound - Megadja a tömb felső korlátját.**
- **GetValue - megadott indexű elem értéke.**
- **IndexOf - egy-dimenziós tömbben az első érték indexe.**
- **Initialize - Egy értéktípusú tömbben minden elemre meghívja az elemek alapértelmezett konstruktorát.**
- **LastIndexOf - Visszaadja az egy-dimenziós tömbben az utolsó érték indexét.**
- **Reverse - Megfordítja a tömb vagy tömbrészlet bejárási irányát.**
- **SetValue - A megadott elemet beteszi a megadott helyre a tömbben.**
- **Sort - A tömbön rendezést hajt végre.**

- **Eiffel:**
  - Az Eiffel tömbök az ARRAY[G] sablon osztály példányai.
  - A stringeket a STRING osztály objektumai valósítják meg.

Asszociatív tömbök

- Egy asszociatív tömb elemek egy rendezetlen halmaza, amelyet megegyező számú, kulcsnak nevezett értékek indexelnek.
- Ezeket a kulcsokat is tárolni kell ⇒ az elemek így (kulcs, érték) párok.

- **Perl:**

A hash skaláris adatok gyűjteménye, az indexek tetszőleges skalárok.
Ezek a kulcsok, amiket használunk az elemek elérésére.
A hash-eknek nincs sorrendjük.
A hash változók **%** jellel kezdődnek. A hivatkozás **{}**-lel történik.

```
%szinek = ( 'piros' => 0x00f,
            'kék' => 0x0f0,
            'zöld' => 0xf00 );
```

A hash változókra:
a **keys** függvény a kulcsok listáját adja vissza,
a **values** pedig az értékeket.
a **delete**-tel lehet kulcs szerint törölni,
az **each** függvény végigmegy a hash-en visszaadva a kulcs-érték párokat,
az **exists** függvény megadja, hogy egy adott kulcs szerepel-e a hash táblában.

Asszociatív tömbök

- A Java, a C++, az Eiffel szabványos osztálykönyvtárában is megtalálhatók
- A .NET keretrendszer osztálykönyvtárában is
- Egyéb nyelvek:
  - PHP,
  - Ruby,
  - Lua,
  - Pike,
  - stb.