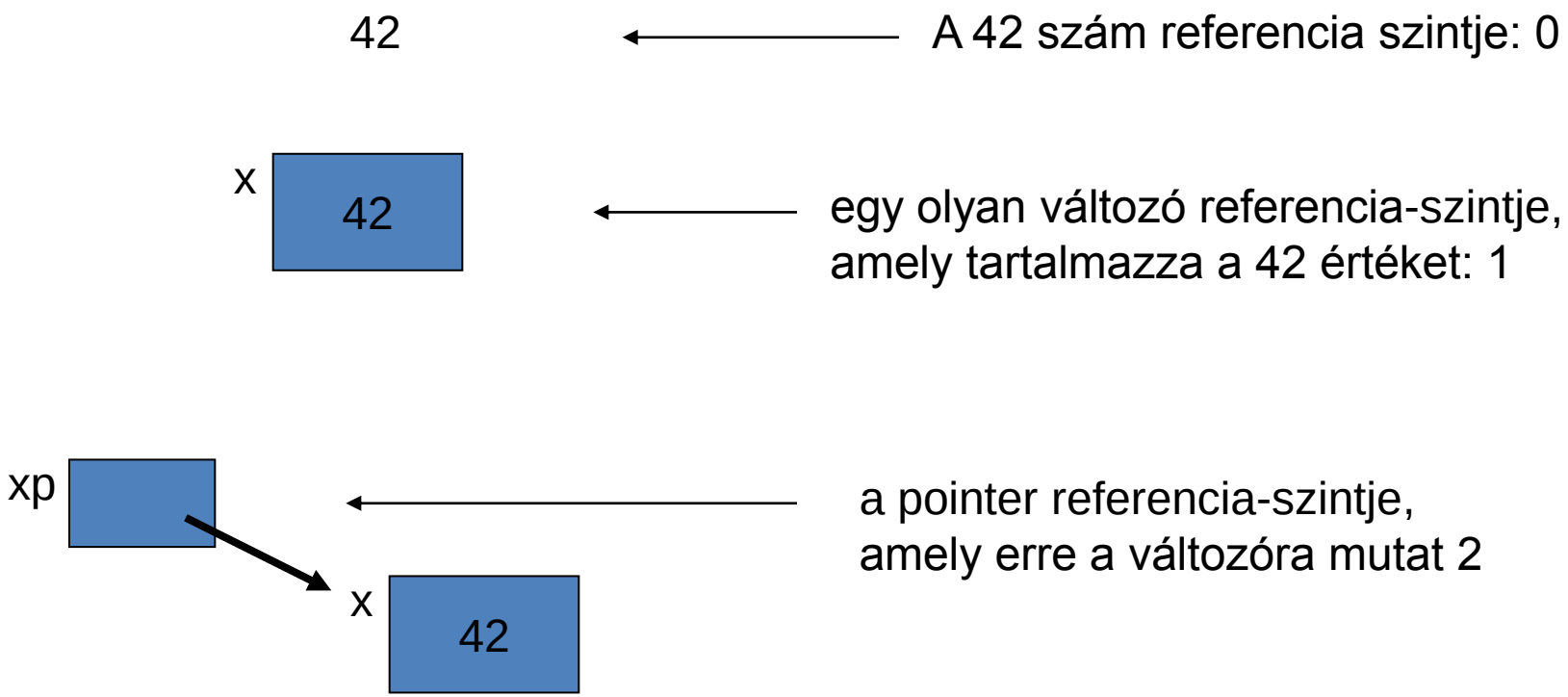
Beépített adattípusok, változók, kifejezések (folyt.)

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

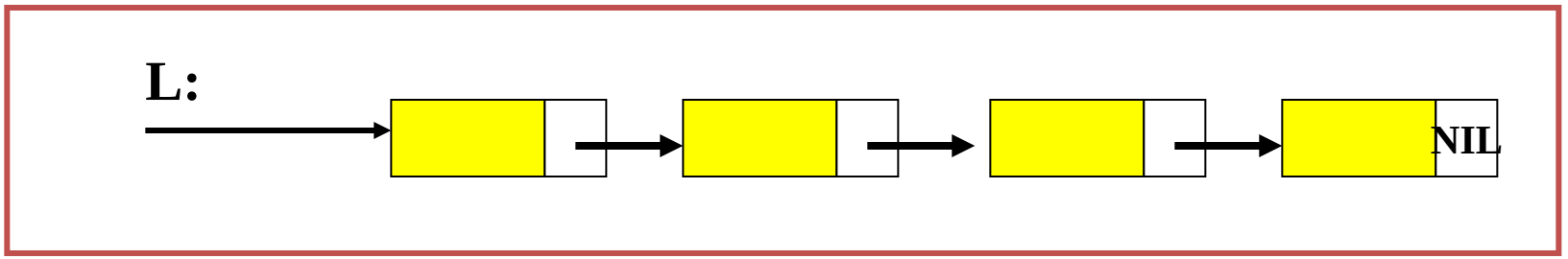
Mire kellenek a mutatók?

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

itt vigyázni kell az osztott használatra!
– jó, ha van read-only elérési lehetőség is

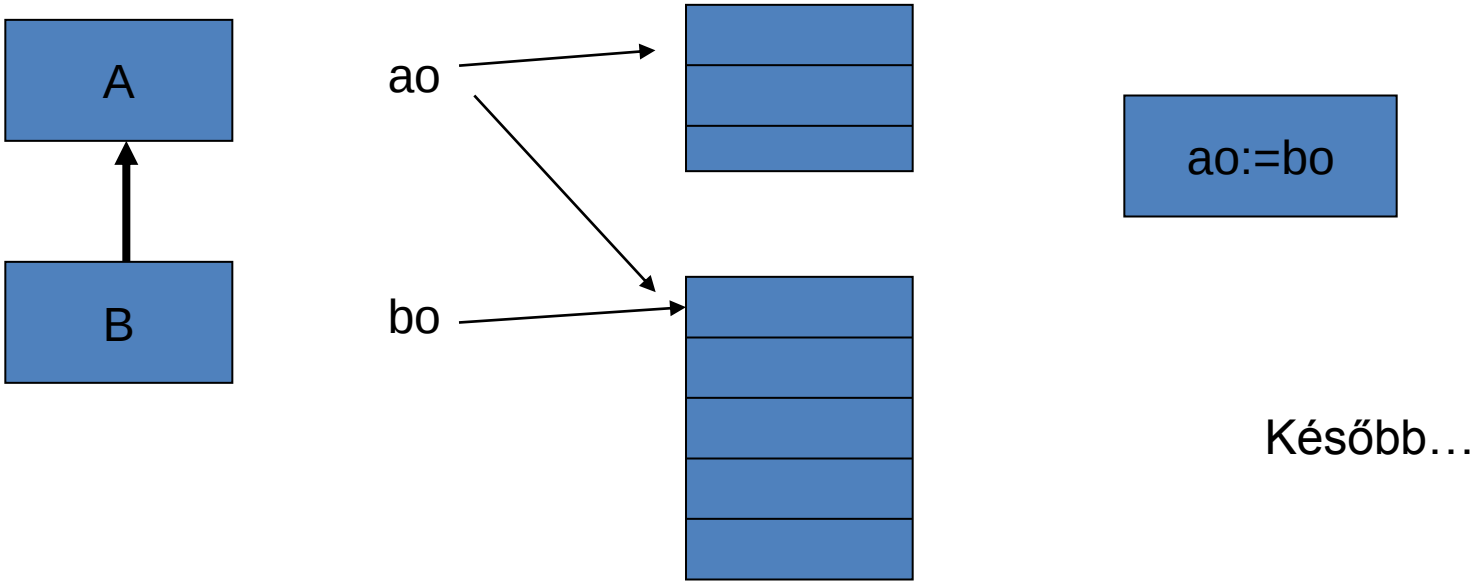
Mire kellenek a mutatók?

- dinamikus adatszerkezetek építéséhez



Mire kellenek a mutatók/referenciák?

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



A szokásos műveletek:

- **értékadás -** pointerek között, hivatkozott objektumok között (copy, clone, deep-copy, deep-clone!)
- **egyenlőség vizsgálat -** ha két ugyanolyan típusú pointer ugyanarra az adatszerkezetre mutat (ez is több szinten lehet)
- **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>
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.:

```
int 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; ADA-ban,

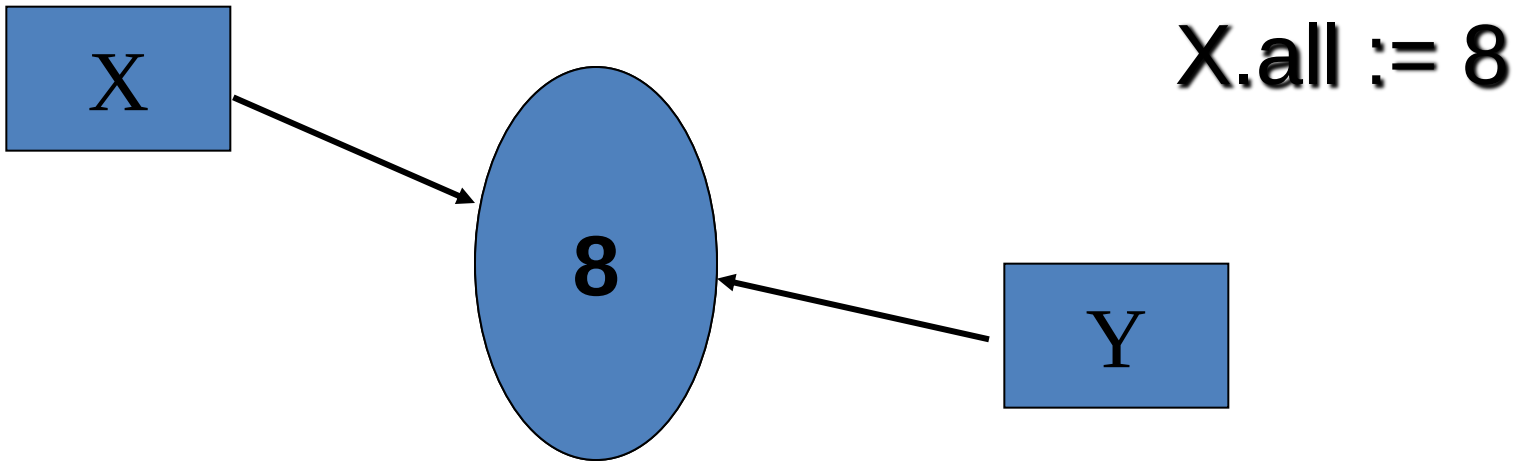
  int * x; C-ben)

A programozási nyelvek között a lehetséges különbségek:

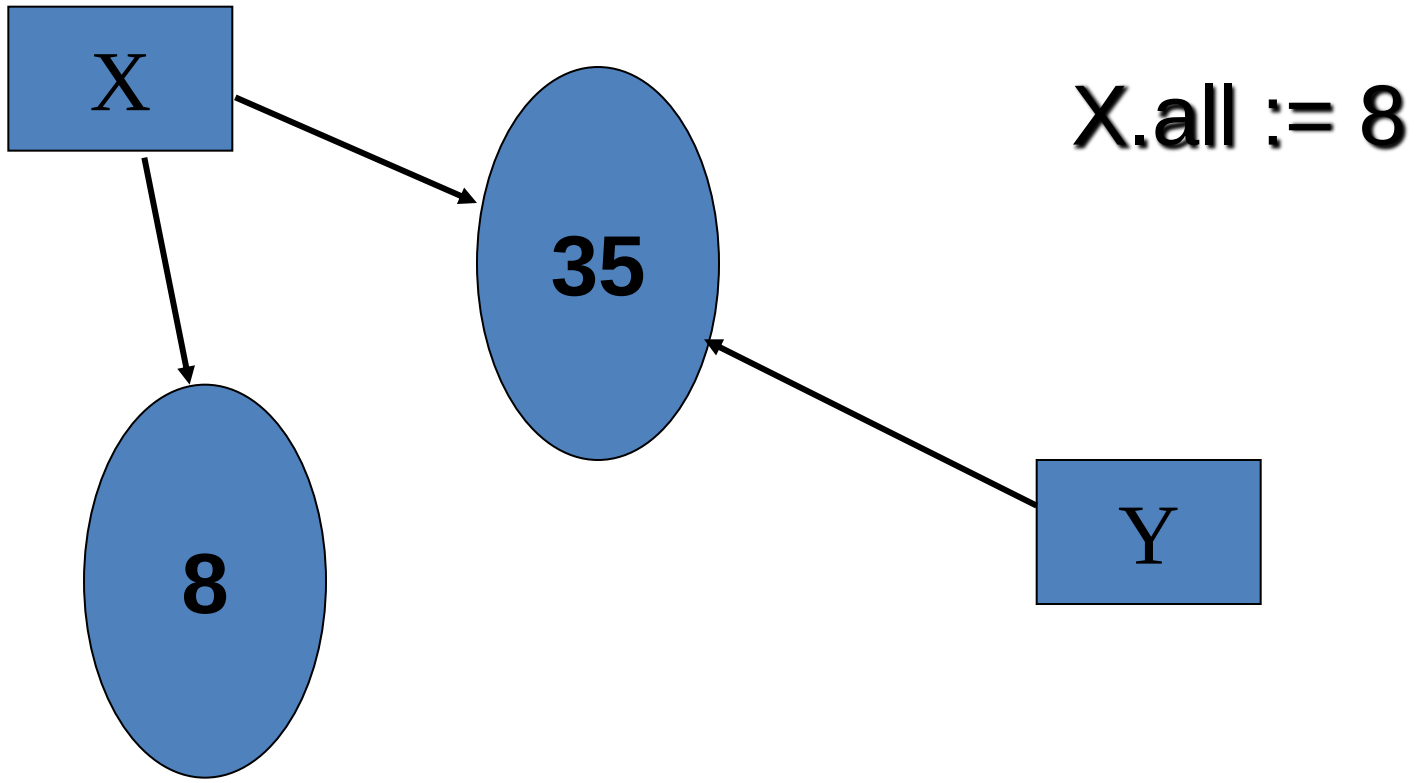
- Milyen biztonságosan kezelhető a “csellengő” pointerek problémája?
- Mi a megengedett műveletek halmaza?
- Kapnak a pointer változók kezdeti (üres) értéket a deklarációnál?
- Lehetséges-e ugyanazt az adatot két (vagy több) pointeren keresztül is változtatni/elérni?

- **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).

- mutable –változtatható:



- immutable – nem változtatható:



- **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 <typn>= <value desc>; type myint=integer;
  - C++

    typedef <value desc> <typn>; typedef int myint;
  - ADA

subtype <typn> is <typ1>; type <typn> is new <typ1c>;
subtype Int is Integer; type My_Int is new Integer;

  - CLU cluster .. később
  - Java, Eiffel, C# class … később

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, *nem* igazi sorozat típus.

  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 -- de általában az elemek ugyanahhoz a típushoz tartoznak, vagy egy adott típus lehetséges leszármazottai is lehetnek.

- **ADA:**
  - **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.**

– Az index típusa tetszőleges diszkrét típus lehet, az elemek típusa tetszőleges típus
– 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);

– Három előredefiniált string típus:

```
type String is array
        (Positive range <>) of Character;
type Wide_String is array
    (Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of
  Wide_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.**
  - **Nem tudja a méretét!**
  - **STL vector template osztály!**

- **Java:**

  "Java arrays are objects, are dynamically created and may be assigned to variables of type Object."

  ```
  int [] ai; // egészek tömbje
  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) Inicializálásra: { }

  - **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 az 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 $\Rightarrow$ 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.

Rekord típusok

A direkt szorzat és az unió típuskonstrukciókat számos nyelvben az ún. rekord típusok segítségével valósítják meg.

Direkt szorzat

- Ha adott két típus, S és T, direkt szorzatukat S x T jelöli.
  - $S \times T = \{(x,y) \mid x \in S;\ y \in T\}$
  - Ez általánosítható több halmazra is: $S_1 \times S_2 \times \ldots \times S_n$
- A COBOL, Pascal, Ada stb. rekordjait, az Algol68,
  C, C++ struktúráit a direkt szorzat terminusaival érthetjük meg.

```
record
  <name_1> : <type_1>;
  <name_2> : <type_2>;
  ….
  <name_k> : <type_k>;
end
```

- A komponenseket a rekord (struktúra) mezőinek hívják.
- Az alapművelet a komponens kiválasztás.

Lehet-e paramétere a típusnak?

Van-e kezdő értékadás a mezőkre?

Van-e teljes rekordra vonatkozó értékadás?

Van-e rekord konstans?

Hogyan működik a kiválasztás művelet?

Pascal

type rektipnev **= record**
mnev$_1$ **:** tipus$_1$**;**
…
mnev$_n$ **:** tipus$_n$**;**
end;

- változó deklarálása:
  rek: rektipnev;
- hivatkozás ponttal: rek.mnev$_1$
- csak mezőnkénti értékadás lehetséges

- Speciális utasítás, aminek a segítségével a rekord mezőire közvetlenül tudunk hivatkozni:

```
type Date= record
    Year : Integer;
    Month : 1..12;
    Day : 1..31;
end;
var R1, R2 : Date;
begin
  R1 := R2; {értékadás megengedett} …
  with R1 do begin
    Year := 2011; Month:=9; Day:=29;
    Year := R2.Year;
  end;
```

C++

struct strnev **{**

tipus$_1$ mnev$_1$**;**

…

típus$_n$ mnev$_n$**;**

};

- változó deklarálása:
  strnev x;
- hivatkozás ponttal: x.mnev$_1$

- A tömbökre használt jelölés alkalmazható struktúrákra is. pl.:

```
struct address{
  long number;
  char* street;
  char* town;
  int zip;
}
address a = {1, „Korkeakoulunkatu", „Tampere", 33720}
```

- Az inicializálásra a konstruktorok (később…) jobban használhatóak.
- Értékadás megengedett, de az egyenlőségvizsgálat nem előre definiált. A felhasználó definiálhat operátorokat rá (később…).

Ada

```
type Complex is record
        Re: Float;
        Im: Float;
    end record;
```

- változó deklarálása:
  C: Complex;
- a komponenseire
  C.Re, vagy C.Im segítségével
  hivatkozhatunk.
- megengedett az értékadás is:

```
C1, C2: Complex;
C1 := C2;
```

A rekord diszkriminánsa(i)

- a típus paramétere
- több diszkriminánsa is lehet egy típusnak
- a rekord diszkrimináns diszkrét típusú

```
type Szöveg( Hossz : Natural ) is  record
    Érték: String(1 .. Hossz) := (others=>' ');
    Pozíció: Natural := 0;
   end record ;
```

Bizonyos programozási nyelvekben –
pl. SmallTalk, Eiffel, Java - **nincs** rekord típus, a tervezők az osztályok használatát javasolják helyette.

C#: a rekord (struct) érték típus, az osztály (class) referencia típus.

Uniók és variáns rekordok

- Az unió típusértékhalmaza az unió komponensei típusértékhalmazának az uniója.
- Pl. bútor: szék vagy asztal vagy szekrény…

  színe, anyaga – van mindegyiknek

  egyéb speciális jellemzők –külön-külön
- A variáns rekordok olyan direktszorzatok, ahol a direktszorzat egy – az utolsó - komponense unió.

„Választó” típusműveletek

- A programozási nyelvek a megbízhatóság különböző szintjén támogatják ezt az adatszerkezetet.
- Az unió típusnak van egy speciális, *tag*-nek nevezett komponense, és egy kiválasztási mechanizmusa, ami megadja a tag különböző értékeinek megfelelő alstruktúrákat. Ha a tag-et **tárolja** a rekord, és az alkomponensek elérhetősége ennek aktuális értékétől függ, akkor ez egy „megkülönböztetett” (*discriminated)* unió, különben ez egy „szabad” (*free)* unió.

Unió (Variáns rekord):

- Meg lehet-e állapítani, hogy a rekord melyik változat szerint lett kitöltve?
- Ki lehet-e olvasni a kitöltéstől eltérő változat szerint??

Szerkezete (gyakran):

case <tag-name> : <tag-type> of
    <const1> : (<fields1>);
    <const2> : (<fields2>);
    …
    <constv> : (<fieldsv>);

- A szabad uniók esetében a tag mezők használata opcionális, és a fordító nem ellenőrzi a kiválasztott mező és a tárolt érték konzisztenciáját. Ez a nyelv típusrendszerét megbízhatatlanná teszi.
- A megkülönböztetett unió esetében fontos kérdés, hogyan lehet új értéket adni a tag mezőnek.
  - A tag értéket beállítja a rekord létrehozásakor.
  - Míg a program képes kell legyen új értéket adni a rekord "normális" komponenseinek, a tag megváltoztatása a rekord szerkezet megváltozását vonja maga után!

Pascal:

```
type Listptr=^Listnode;
type Listnode=record
    Next: Listptr;
    case Tag: Boolean of
      False: (Data:char);
      True: (Down: Listptr)
  end;
var p, q: Listptr; ….
p^.Tag:=true;
p^.Down:=q;
p^.Tag:=false;
writeln(p^.Data);
```

{!!!!hiba!!!}

C++:

```
union  typname{
        typ1 field1;
          ...
          typm fieldm;
          };
union Fudge{
     int i;
     int* p;
};
```

```
int* cheat(int i){
     Fudge a;
      a.i=i;
      return a.p;
}
```

Jó ez??

Ada:

```
type Állapot is (Egyedülálló, Házas, Özvegy, Elvált);
subtype Név is String(1..25);
type Nem is (Nö, Férfi);
type Ember (Családi_Áll: Állapot := Egyedülálló) is
    record
        Neve: Név;
        Neme: Nem;
        Születési_Ideje: Dátum;
        Gyermekek_Száma: Natural;
        case Családi_Áll is
            when Házas        => Házastárs_Neve: Név;
            when Özvegy       => Házastárs_Halála: Dátum;
            when Elvált       => Válás_Dátuma: Dátum;
                                 Gyerekek_Gondozója: Boolean;
            when Egyedülálló => null ;
        end case;
    end record ;
```

Hugó: Ember(Házas);

Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Házastárs_Neve

Eleonóra: Ember(Egyedülálló);

Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma

Ödön: Ember(Özvegy);

Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Házastárs_Halála

Vendel: Ember(Elvált);

Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma, Válás_Dátuma, Gyerekek_Gondozója

Aladár: Ember; -- Egyedülálló (kezdetben!)

Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma

- **<u>Helytelen</u> (futási idejű hiba, altípus-megszorítás megsértése): Hugó.Válás_Dátuma, Aladár.Házastárs_Neve**

Megszorítatlan altípus használata

Aladár: Ember; -- alapért. Egyedülálló

Aladár := (Házas, ...);

Aladár := Elek;

Aladár := Hugó;

- A szerkezetét megváltoztathatjuk, diszkriminánsostul
- Csak úgy, ha az egész rekord értéket kap egy értékadásban

Bizonyos programozási nyelvekben –
pl. SmallTalk, Eiffel, Java, C# - **nincs** unió típus, a tervezők az osztályok és az **öröklődés** használatát javasolják helyette.

Halmaz

- Speciális iterált típus
  - Mi lehet az eleme?
  - Hány eleme lehet?
  - Megvannak-e a 'hagyományos' halmazműveletek?

Pascal

Alaphalmaz: diszkrét típus

Elemek száma: max. 256

Értékek sorszáma csak 0..255 között

```
type Small_Letters = set of 'a'..'z';
type Digits = set of '0'..'9';
type Day = (Monday, Tuesday, Wednesday, Thursday,
    Friday, Saturday, Sunday);
type Days = set of Day;
    var A, B : Days;
```

A:=[Monday, Wednesday] - halmazkonstruktor

üres halmaz: []

Műveletek:

- értékadás,
- halmazműveletek:
  1. eleme-teszt (`in`),
  2. az `=` , `<>`, részhalmaz reláció ('<=', '>=')
  3. (a '<' és '>' nem megengedett!)
  4. unió ('+'), differencia ('–').
  5. metszet ('*')

Ez a precedencia-sorrend is.

Mikor ekvivalens két típus? (név - struktúra)

x, y: array[0..9] of integer;
z: array[0..9] of integer;

?

- Strukturális ekvivalencia esetén:
  - A rekordok mezőnevei is figyelembe vannak véve, vagy csak a struktúrájuk?
  - Számít-e a rekordmezők sorrendje?
  - Tömböknél elég-e az indexek számosságának egyenlőnek lenni, vagy az indexhatároknak is egyezniük kell?

```
Dimensions = RECORD
     Breadth: REAL;
     Length: REAL;
     END;

Complex = RECORD
     RealPart: REAL;
     ImPart: REAL;
     END;

Size: Dimensions;
Root: Complex;
```

Értékül adhatóak egymásnak? Akarjuk??

Név szerinti ekvivalencia esetén:

- Deklarálhatók-e egy típushoz típusok, amelyekkel ekvivalens?
- Névtelen tömb- ill. rekordtípusok ekvivalensek-e valamivel?

Típuskonverziók:
Van-e, és hogyan működik ?

- az automatikus konverzió?
- az identitáskonverzió?
- a bővítő konverzió?
- a szűkítő konverzió?
- a toString konverzió?

Java:

- identitáskonverzió*:*
  - a `boolean` csak ez szabad
- bővítő konverzió *:*
  - `byte` to `short, int, long, float` or `double`
  - `short` to `int, long, float` or `double`
  - `int` to `long, float` or `double`
  - `long` to `float` or `double`
  - `float` to `double`

Java:

- szűkítő konverzió *:*
  - byte to char
  - short to byte or char
  - char to byte or short (miért is?)
  - int to byte, short or char
  - long to byte, short, char or int
  - float to byte, short, char, int or long
  - double to byte, short, char, int, long or float

Változók

- Változó =

(név, attribútumhalmaz, hely, érték)

- Láthatóság, Elérhetőség
- Hogyan definiálhatunk változókat és konstansokat?

Változók

	szintaxis	példa
Pascal	**var <identif>: <type>;**	**var i : integer;**
C++	**<type> <identif>[= <value>];**	**int i = 0;**
Java	**<type> <identif>[= <value>];**	**int i = 0;**
ADA	**<identif>: <type>[:= <value>];**	**I : Integer := 0;**
CLU	**<identif>: <type>[:= <value>];**	**i : int := 0;**
Eiffel	**<identif>: [expanded]<type>;**	**I : INTEGER;**

Konstansok

	szintaxis	példa
Pasc.	**const <name> = <value>;**	**const val = 3;**
C++	**#define <name> <value> (?) vagy** **const <type> <name> = <value>;**	**#define val 3** **const int val = 3;**
Java	**final <type> <name> = <value>;**	**final int val = 3;**
ADA	**<name> :** **constant <type>:= <value>;**	**Val :** **constant Integer := 3;**
C#	**const <type> <name> = <value>;**	**const double PI = 3.14159;**
Eiffel	**<name> : <type> is <value>;**	**A: INTEGER is 3;**

Kifejezések

- Prefix jelölés

  + a b

- Postfix jelölés

  a b +

- Infix jelölés

  a + b

- A műveletek precedencia szintjei

Pascal

zárójel:	**()**
függvényhívás	**fv(..)**
unáris operátorok	**not, @, ^, +,-**
multiplikatív operátorok	***, /, and, div, mod, shl, shr**
additív operátorok	**+, - or, xor**
relációk	**in, <, >, <=, >=, <>**
balról jobbra kiértékelés	

Ada95

zárójelek	**()**
legmagasabb prec. op.	****, abs, not**
mult. op.	***, /, mod, rem**
unáris add. op.	**+, –**
Bináris add. op.	**+, –, &**
relációs op.	**=, /=, <, <=, >, >=**
logikai op.	**and, or, xor, and then, or else**
balról jobbra kiértékelés	

C++

zárójelek	**()**
scope	**::**
selection, call, size	**. -> [] () sizeof**
postf, pref, compl, not, un. add. address of, deref, cre., destroy, cast	**++ --, ~ ! + -, & * new, delete ()**
member	**.* ->***
multipl. op.	*** / %**
binary add.	**+ -**
shift	**<< >>**
relációs	**< <= > >=**
egyenlőség	**== !=**
bitwise AND	**&**
bitwise excl. OR	**^**
bitwise OR	**|**
logical AND	**&&**
logical OR	**||**
cond. expr.	**? :**
értékadások	**= *= /= %= += -= >>= <<= &= ^= !=**
throw	**throw**
comma (sequence)	**,**

Java

postfix	**. [] () ++ --**
prefix	**++ -- ~ ! + -**
constr, cast	**new ()**
multipl. op.	*** / %**
binary add.	**+ -**
shift	**<< >> >>>**
relational	**< <= > >= instanceof**
equality	**== !=**
bitwise AND	**&**
bitwise excl. OR	**^**
bitwise OR	**|**
logical AND	**&&**
logical OR	**||**
cond. expr.	**? :**
értékadások	**= *= /= %= += -= >>= <<= >>>= &= ^= !=**