Programozási nyelvek és módszerek

Pointerek

Pointer és referencia típusok

Egy pointer (és egy referencia) egy olyan objektum, amely megadja egy másik objektum címét a memóriában

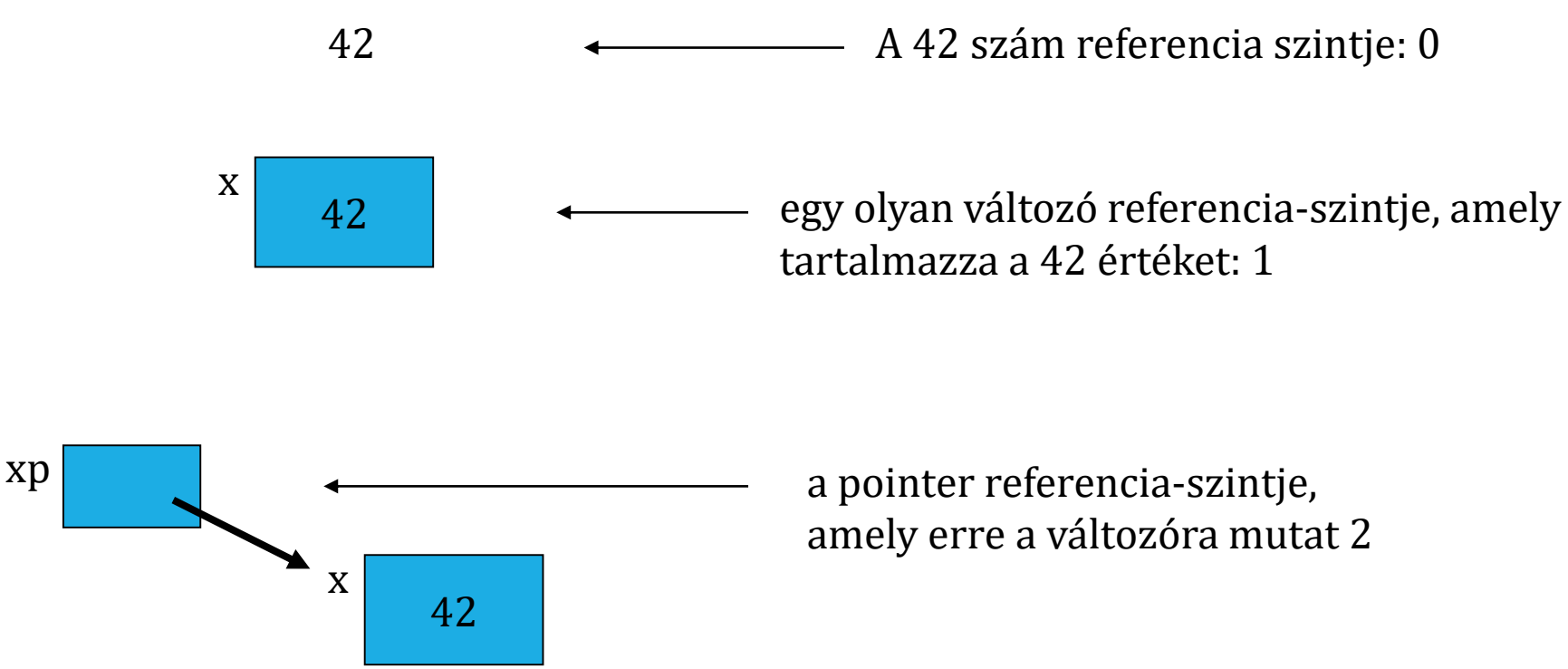
Egy pointer értéke egy memóriacím

- gépi nyelvekben az indirekt címzés lehetősége motiválta a pointerek létrehozását.

Vannak típusos és típus nélküli pointerek.

Referencia szint

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

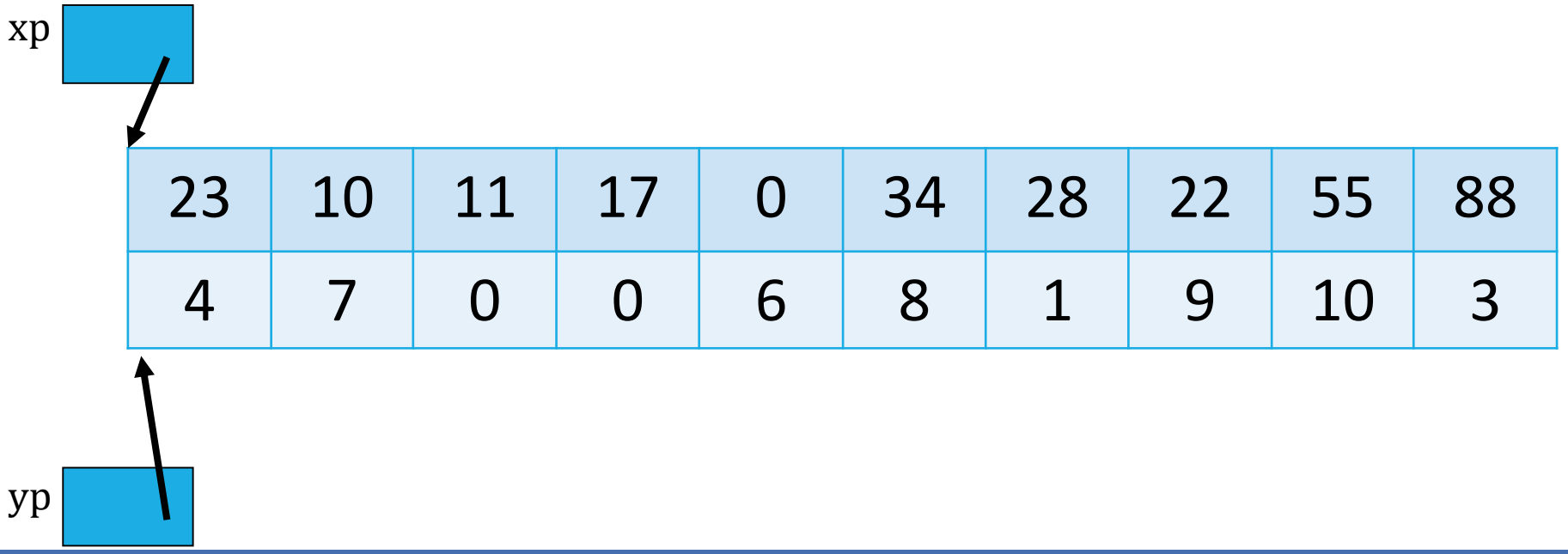
y

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



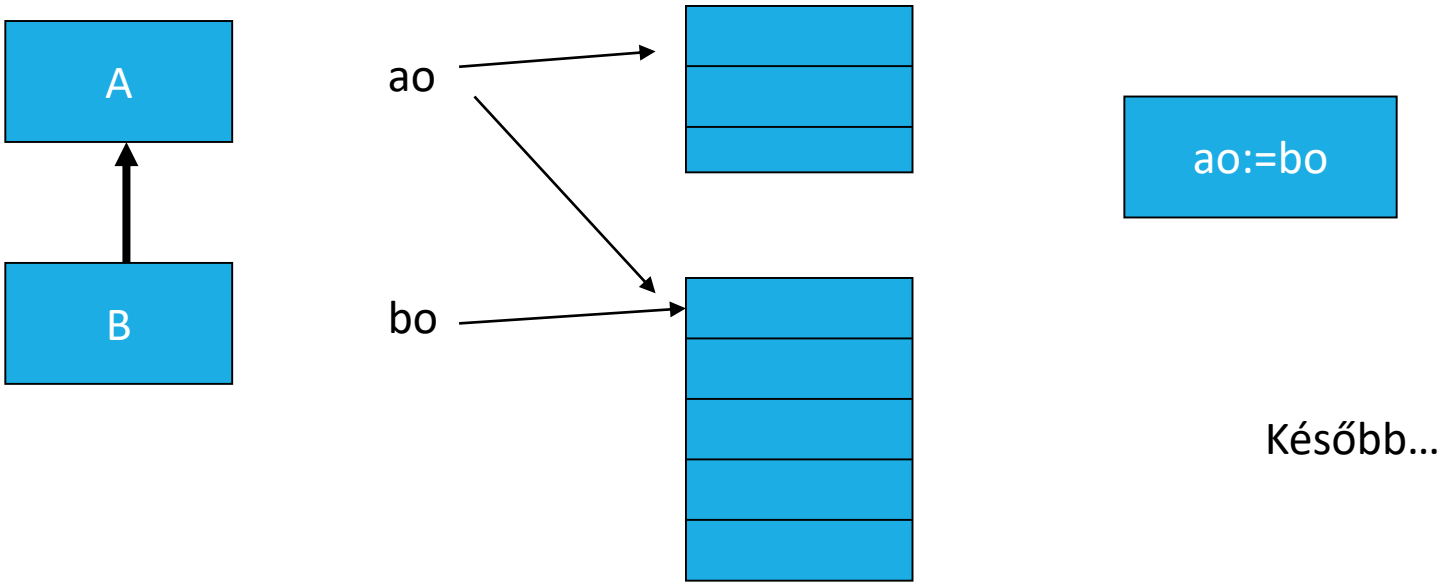
Mire kellenek a mutatók?

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 deallokálása - 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 << ".\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!

Pointerek az egyes nyelvekben

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?
- Milyen fajta konstans pointerek megengedettek?
  - 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?
  - `type PInteger is access to Integer;` (ADA-ban)
  - `int * x;` (C-ben)

Pointerek az egyes nyelvekben

Milyen biztonságosan kezelhető a „csellengő” pointerek problémája?

Mi a megengedett műveletek halmaza?

Kapnak a pointer változók kezdeti (üres) értéket a deklarációnál?

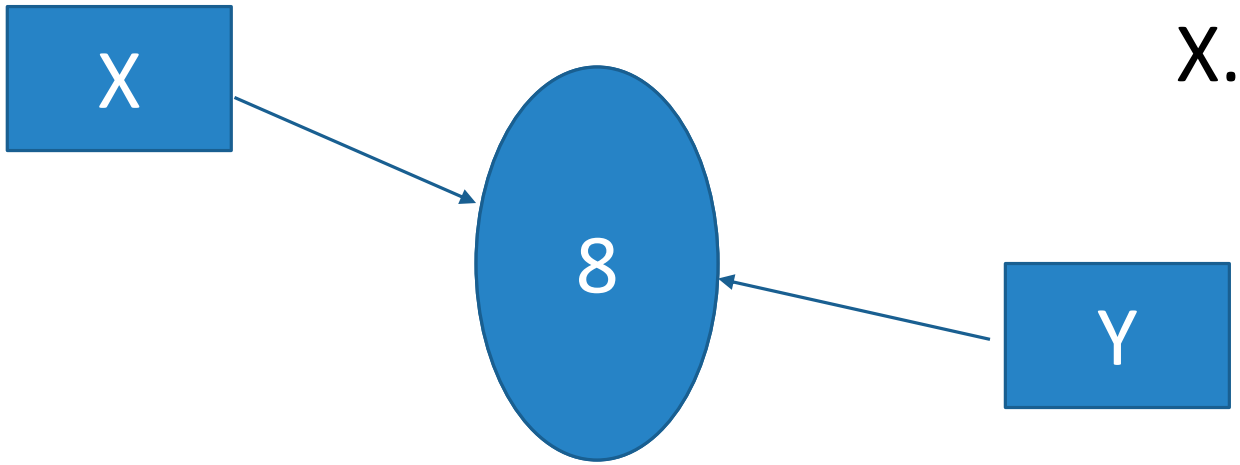
Lehetséges-e ugyanazt az adatot két (vagy több) pointeren keresztül is változtatni/elérni?

Pointerek

CLU

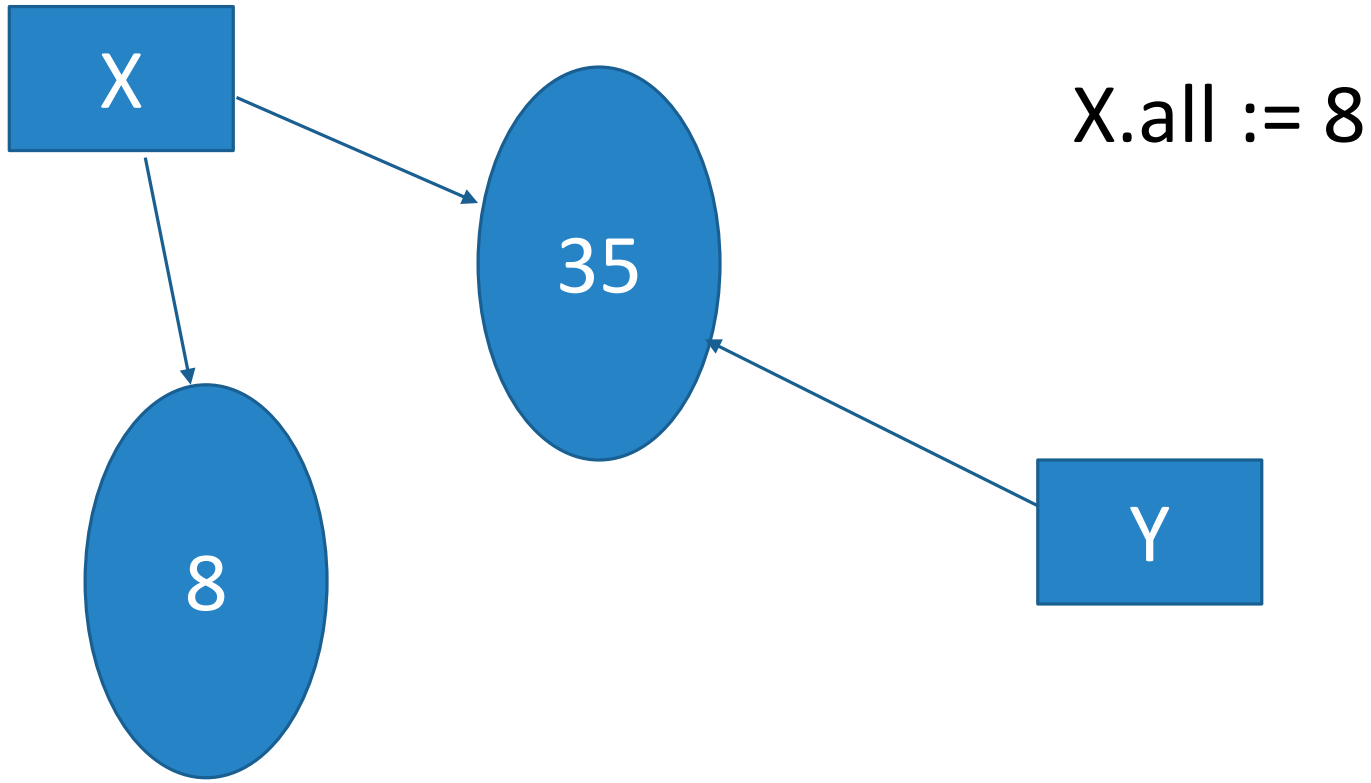
- A CLU-ban nincs hagyományos pointer típus.
- A program végrehajt műveleteket objektumokon
- Az objektumok mint egy univerzum részei léteznek, a program változói hivatkoznak ezekre az objektumokra
  - Garbage collection a felszabadításra
- A programban kétféle objektum lehet:
  - mindig ugyanaz az értéke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

Mutable – változtatható



X.all := 8

Immutable – nem megváltoztatható



Pointerek

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, amelynek
    - (char, char*) argumentumai vannak és egy egészet (int) ad vissza

Pointerek

C++

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

Pointerek

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, … veszélyesek lehetnek, vigyázzunk, ne keverjük össze a jelentését!
  - `pi++` a pointert inkrementálja, és a következő memóriacímre fog mutatni, ennek akkor van értelme, ha pi egy tömbre mutat, míg `r++` inkrementálja `i` értékét

Pointerek

Java

- Nincs hagyományos pointer típus
- A változókban kétféle érték tárolható
  - primitív értékek (egy numerikus típusból vagy egy logikai)
  - 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, '==' '!=' (referencia egyenlőségének vizsgálata) stb.

Pointerek

Eiffel

- Itt sincsenek hagyományos pointer típusok.
- A változókban kétféle érték tárolható - kiterjesztett értékek és referencia értékek.
- Ugyanarra az objektumra számos referencia hivatkozhat.

Pointerek

C#

- A referencia típusok objektumait kezelhetjük referenciákkal.
- Egy „unsafe” környezetben egy típus lehet pointer is, erre számos művelet megengedett (pl. a ++, -- is).

Adattípusok definiálása

Hogyan definiálhatunk új adattípusokat?

Példák

- 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;`
- Java, Eiffel, C#, Swift, Kotlin, stb.
  - `class`
  - `(struct)`
  - `enum`

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?

Indexelés

Az alapművelet az *indexelés*

- `A[i]`
- Elvárás, hogy az `A` tömb `i`. elemét gyorsan el kell tudni érni!

Általában az elemek ugyanahhoz a típushoz tartoznak

- vagy egy adott típus lehetséges leszármazottai is lehetnek
- nem mindig: Smalltalk esetén az elemek különböző típusúak is lehetnek

Tömbök

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);`
  - `B : Matr(1 .. 2, 1 .. 4);`
- Gyakran használják alprogramok paramétereként, sablonoknál.

Tömbök

ADA

- 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);`
- Lehetséges értékadás tömb „aggregátokkal” is:
  - `V:=(1..3=>1, others=>0);`

Tömbök

ADA

- Három előre definiált 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)`
    ```

Tömbök

```
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;
```

Tömbök

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!
    ```

Tömbök

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"};`

Tömbök

Java

- Egy tömbök tömbjében (ami a Java esetén gyakran több dimenziós tömbnek hívnak a dokumentációkban) 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` (`StringBuilder`) osztályokkal oldották meg.
  - Karakterek egy tömbje nem `String`!

Tömbök

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.

Tömbök

C#

- 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,}};
```

- 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.

Tömbök

C#

- Az `Array` osztály tulajdonságait megadó függvények:
  - `IsFixedSize` - rögzített hosszúságú-e
  - `IsReadOnly` - írásvédett-e
  - `IsSynchronized` - a tömb elérése kizárólagos-e (szálbiztos)
  - `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).

Tömbök

C#

- Az `Array` osztályban még számos szolgáltatás:
  - `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.

Tömbök

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

Asszociatív tömbök

Python

- Beépített „dictionary” típus, kulcs-érték párok tárolására
  - A kulcsnak „hashable” típusnak kell lenniük
  - A kulcsokat használunk az elemek elérésére.
- Példa
  - `tel = {'jack': 4098, 'sape': 4139}`
- Ezekre a változókra
  - a `list()` függvény a kulcsok listáját adja vissza,
  - a `del` utasítással lehet kulcs szerint törölni
    - `del tel['sape']`
  - for ciklussal végig lehet járni a tárolt kulcs-érték párokat
    - `for k, v in tel.items():`
  - az `in` kulcsszó megadja, hogy egy adott kulcs szerepel-e benne
    - `'jack' not in tel`

Asszociatív tömbök

A Java, a C++, az Eiffel szabványos osztálykönyvtárában is megtalálhatók

A .NET keretrendszer osztálykönyvtárában is

Egyéb nyelvek:

- PHP,
- Perl,
- Ruby,
- Lua,
- Pike,
- stb.

Rekord típusok

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

Ha adott két típus, S és T, direkt szorzatukat S × T jelöli.

- $S \times T = \{(x,y) | x \in S; y \in T\}$
- Ez általánosítható több halmazra is:
  $S_1 \times S_2 \times \ldots \times S_n$

A direkt szorzat terminusaival érthetjük meg:

- COBOL, Pascal, Ada stb. rekordok
- Algol68, C, C++, C# stb. struktúrák

Rekord típusok

```
record
  <name1> : <type1>;
  <name2> : <type2>;
  …
  <namek> : <typek>;
end
```

A komponenseket a rekord (struktúra) mezőinek hívják.

Az alapművelet a komponens kiválasztás.

Rekord típusok

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?

Rekord típusok

Pascal

```
type rektipnev = record
    mnev1 :  tipus1;
    ...
     mnevn :  tipusn;
end;
```

- változó deklarálása
  - `rek: rektipnev;`
- hivatkozás ponttal
  - `rek.mnev1`
- csak mezőnkénti értékadás lehetséges

Rekord típusok

Pascal

- 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;
```

Rekord típusok

C++

```
struct strnev {
        tipus1 mnev1;
        ...
        típusn mnevn;
};
```

- változó deklarálása
  - `strnev x;`
- hivatkozás ponttal
  - `x.mnev1`

Rekord típusok

C++

- 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 jobban használhatóak.
- Értékadás megengedett, de az egyenlőségvizsgálat nem előre definiált.
  - A felhasználó definiálhat operátorokat rá

Rekord típusok

ADA

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

- változó deklarálása
  - `C: Complex;`
- a komponenseire ponttal hivatkozhatunk
  - `C.Re`
    `C.Im`
- megengedett az értékadás is
  - `C1, C2: Complex;`
    `C1 := C2;`

Rekord típusok

ADA

- 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;
```

Rekord típusok

C#:

- a rekord (struct) érték típus
- az osztály (class) referencia típus

Bizonyos programozási nyelvekben nincs rekord típus, a tervezők az osztályok használatát javasolják helyette

- SmallTalk
- Eiffel
- Java
- …

Uniók és variáns rekordok

Az unió típusértékhalmaza az unió komponensei típusértékhalmazának az uniója.

- Például 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>);
```

Unió (Variáns rekord)

A szabad uniók esetében:

- A tag mezők használata opcionális,
- 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!

Példa

Pascal:

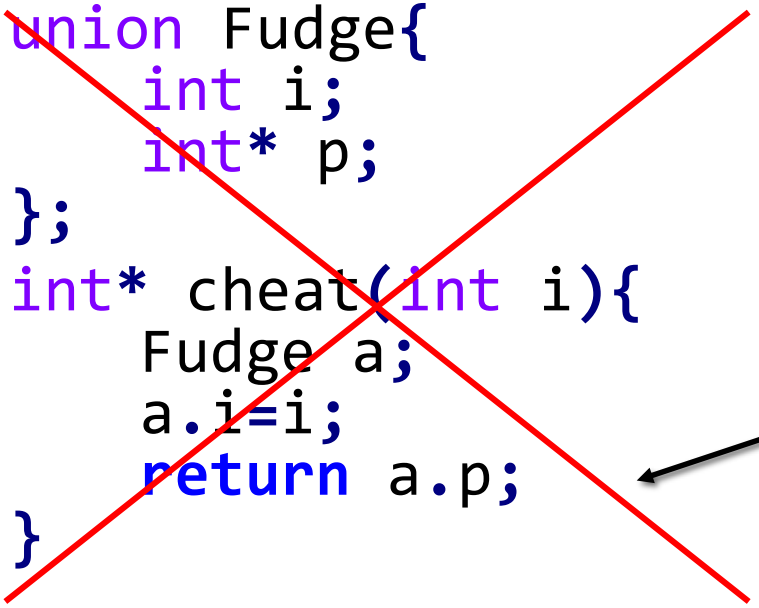
```
type Listptr=^Listnode;
type Listnode=record
       Next: Listptr;
       case Tag: Boolean of
           False: (Data:char);
                  True: (Down: Listptr)
    end;
var p, q: Listptr;
p^.Tag:=true;
p^.Down:=q;
p^.Tag:=false;
writeln(p^.Data);
```

HIBA

Példa

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??

Unió (Variáns rekord)

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;
```

Unió (Variáns rekord)

ADA

- `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`
  `Családi_Áll, Neve, Neme, Születési_Ideje, Gyermekek_Száma`
- Helytelen (futási idejű hiba, altípus-megszorítás megsértése):
  - `Hugó.Válás_Dátuma, Aladár.Házastárs_Neve`

Unió (Variáns rekord)

ADA

- 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

Unió (Variáns rekord)

Bizonyos programozási nyelvekben nincs unió típus, a tervezők az osztályok és az öröklődés használatát javasolják helyette:

- SmallTalk,
- Eiffel,
- Java,
- C#

Halmaz

Speciális iterált típus

- Mi lehet az eleme?
- Hány eleme lehet?
- Megvannak-e a „hagyományos” halmazműveletek?

Halmaz

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: [ ]

Halmaz

Pascal

- Műveletek:
  - értékadás,
  - halmazműveletek:
    - eleme-teszt (in),
    - az = , <>, részhalmaz reláció ('<=', '>=')
    - (a '<' és '>' nem megengedett!)
    - unió ('+'), differencia ('-').
    - metszet ('*')
- Ez a precedencia-sorrend is.

Mikor ekvivalens két típus?

```
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?

Mikor ekvivalens két típus?

Értékül adhatóak egymásnak? Akarjuk?

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


Complex = RECORD
          RealPart: REAL;
          ImPart: REAL;
    END;
Size: Dimensions;
Root: Complex;
```

Név szerinti ekvivalencia esetén

Deklarálhatók-e egy típushoz típusok, amelyekkel ekvivalens?

Névtelen tömb- illetve rekordtípusok ekvivalensek-e valamivel?

Típuskonverziók

Van-e, és hogyan működik?

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

Típuskonverziók

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`

Típuskonverziók

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;`
Swift	`var <identif>: [<type>][ = <value>]`	`var x: Integer = 10`

Konstansok

	Szintaxis	Példa
Pascal	`const <name> = <value>;`	`const val = 3;`
C++	`#define <name> <value>` `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;`
CLU	`const <type> <name> = <value>;`	`const double PI = 3.14159;`
Eiffel	`<name> : <type> is <value>;`	`A: INTEGER is 3;`
Swift	`let <identif>: [<type>] = <value>`	`let x: Integer = 10`

Kifejezések

Prefix jelölés

- + a b

Postfix jelölés

- a b +

Infix jelölés

- a + b

A műveletek precedencia szintjei fontosak

Precendica - Pascal

Leírás	Operátor
zárójelezés	
függvényhívás	
unáris operátor	not @
Multiplikációs operátor	* / div mod and shl shr as << >>
Additív operátor	+ - or xor
Relációs jel	= <> < > <= >= in is
Balról jobbra	

Precedencia – C++ 1.

Precedence	Operator	Description
1	::	Scope resolution
2	++	Suffix increment
	--	Suffix decrement
	()	Function call
	[]	Array subscripting
	.	Element selection by reference
	->	Element selection through pointer
	typeid()	
	const_cast	
	dynamic_cast	
	reinterpret_cast	
	static_cast	

Precedencia – C++ 2.

Precedence	Operator	Description
3	++	Prefix increment
	--	Prefix decrement
	+	Unary plus
	-	Unary minus
	!	Logical NOT
	~	Bitwise NOT
	(type)	Type cast
	*	Indirection (dereference)
	&	Address-of
	sizeof	Size-of
	new, new[]	Dynamic memory allocation
	delete, delete[]	Dynamic memory deallocation
4	.*	Pointer to member
	->*	Pointer to member
5	*	Multiplication
	/	Division
	%	Modulo (remainder)
6	+	Addition

Precedencia – C++ 3.

Precedence	Operator	Description
6	+	Addition
	-	Subtraction
7	<<	Bitwise left shift
	>>	Bitwise right shift
8	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
9	==	Equal to
	!=	Not equal to
10	&	Bitwise AND
11	^	Bitwise XOR (exclusive or)
12	|	Bitwise OR (inclusive or)
13	&&	Logical AND
14	||	Logical OR
15	?:	Ternary conditional (see ?:)

Precedencia – C++ 4.

Precedence	Operator	Description
6	+	Addition
16	=	Direct assignment
	+=	Assignment by sum
	-=	Assignment by difference
	*=	Assignment by product
	/=	Assignment by quotient
	%=	Assignment by remainder
	<<=	Assignment by bitwise left shift
	>>=	Assignment by bitwise right shift
	&=	Assignment by bitwise AND
	^=	Assignment by bitwise XOR
	|=	Assignment by bitwise OR
17	throw	Throw operator
18	,	Comma

Precedencia – Java

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Érdekesség

Algol 68

- A kétoperandusú operátorok prioritása általában felüldefiniálható a `prio` kulcsszóval:
  - `prio + = 3, * = 2;`
  - `print(6+3*5);  # (6+3)*5=45 lesz az eredmény`

Vezérlési szerkezetek

Egyszerű utasítások:

Értékadás – mit jelent?

- „változó” értékadásjel kifejezés

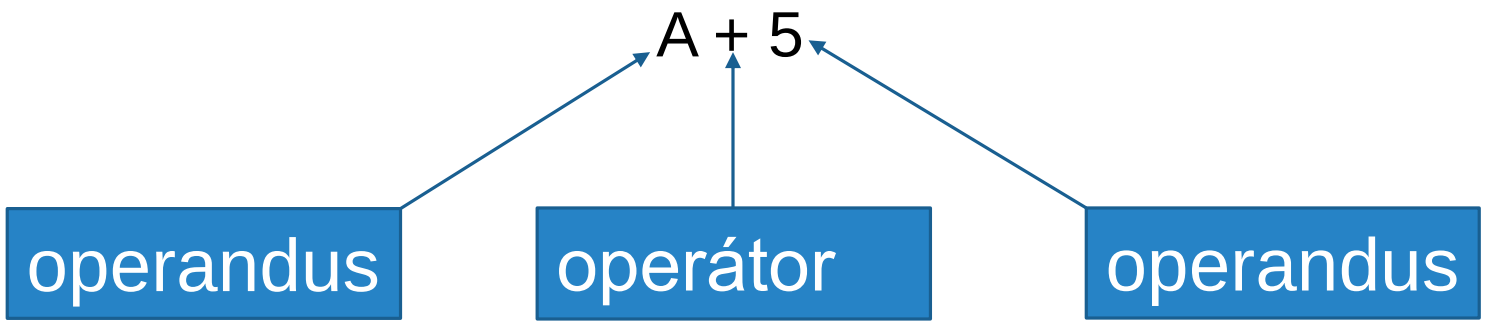
Helyette inkább:

- „balérték” értékadásjel „jobbérték”
  - (vagy fordítva?)
- Mit is jelent ez?
  - Értékadásjel
    - Szokásosan `:=` és `=`
    - De `->` és `<-`
  - Szemantikája
    - „megfelelő” típusú kifejezés kell!

Kifejezés

operandusokból és operátorokból áll

- minden operandus lehet egy újabb kifejezés
- kiértékelem és megkapom az értékét:



Kifejezés az értékadás?

Wulf (BLISS)

- „Of course, everything is”

Richie (C)

- „Yes, why not”

Wirth (Pascal)

- „No, only math-like things are expressions”

Két vonulat:

- Pascal, CLU, ADA, Eiffel, …
- C, C++, Java, …

Van-e többszörös értékadás?

Szemlélet – COBOL

Eredetileg

- `MOVE 23 TO A.`
- `MOVE B TO C.`
- `ADD A TO B GIVING C.`
- `SUBTRACT A FROM B GIVING C.`
- `MULTIPLY A BY B GIVING C.`
- `DIVIDE A BY B GIVING C.`

Újabban

- `COMPUTE A = ( A + B - C / ( A * B ) - A * B )`

Pascal

„balérték” := kifejezés;

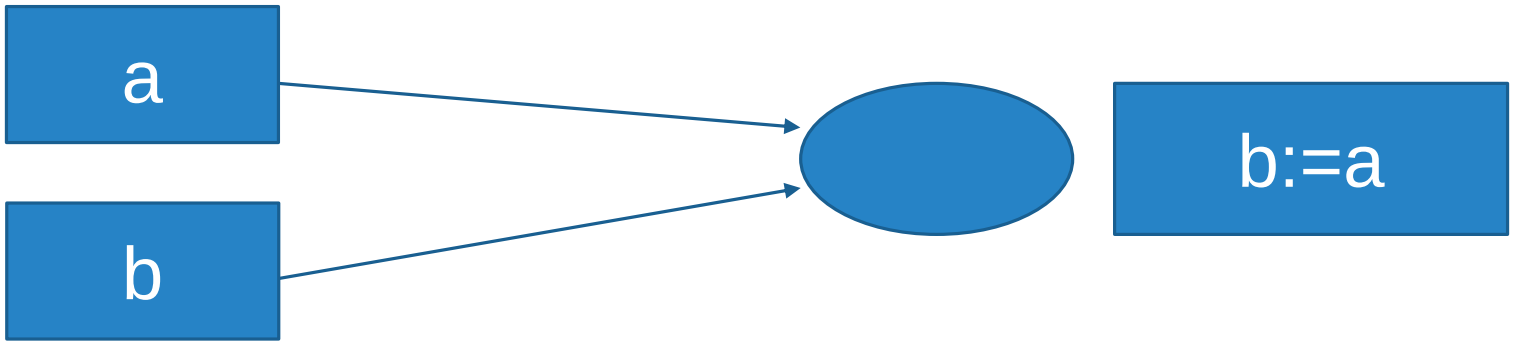
- A „balérték” és a kifejezés típusa megegyező, de legalább kompatibilis kell legyen.
  - `m: integer;`
  - `m:= m+1;`
- vagy:
  - `p := keres(gyoker, x);`

Többszörös értékadás nem megengedett.

CLU

referenciákat használ

- értékadás hatására két referencia hivatkozhat ugyanarra az objektumra



- Többszörös értékadás megengedett:
  - `x, y := y, x.`

C++

Számos értékadó operátor jobbról balra feldolgozva

- `A=B=C`
  - jelentése: `A=(B=C);`

Értékadó operátorok

- `= *= /= %= += -= >>= <<= &= ^=`

Például

- `y += g(x);`

Java

A C++ -hoz hasonló, számos értékadó operátor:

- `= *= /= %= += -= >>= <<= >>>= &= ^=`
- `E1 op= E2`
  - jelentése: `E1 = (T)((E1) op (E2))`,
  - ahol `T` az `E1` típusa.

Összetett értékadó operátoroknál mindkét operandus primitív típusú kell legyen

- kivéve: `+=`, ha a bal operandus `String` típusú),

Implicit cast előfordulhat!

- `short x=3; x+=4.6;` eredménye: `x ==7`!

`final`-nak deklarált változónak nem adható érték.

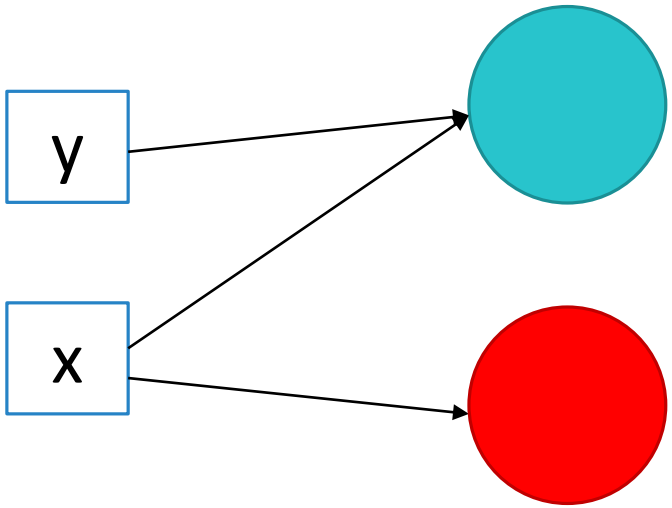
Alapvetően referenciákat használ

Eiffel

Az értékadás és a paraméterátadás szemantikája megegyezik.

- Ha x:TX, y:TY, akkor az x:=y eredménye TX és TY-tól függ
- Mindkettő lehet referencia vagy kiterjesztett típus.
  - Az Eiffel mechanizmusa a klónozás, referencia másolás szemantikáját segít megérteni

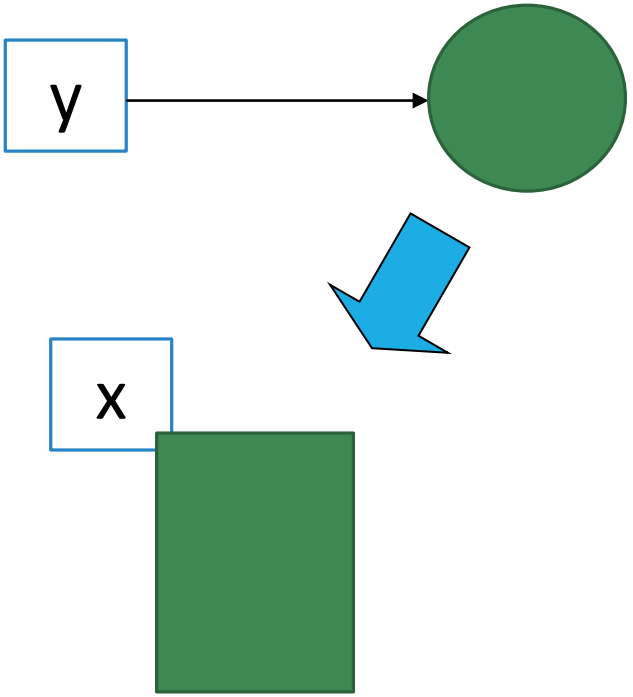
Mindkettő referencia



x:=y

referencia újrahozzárendelés

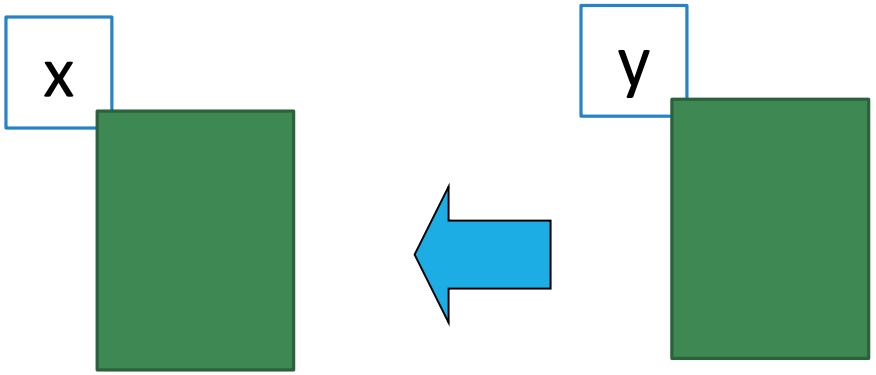
TX kiterjesztett, TY referencia



x:=y

másolás

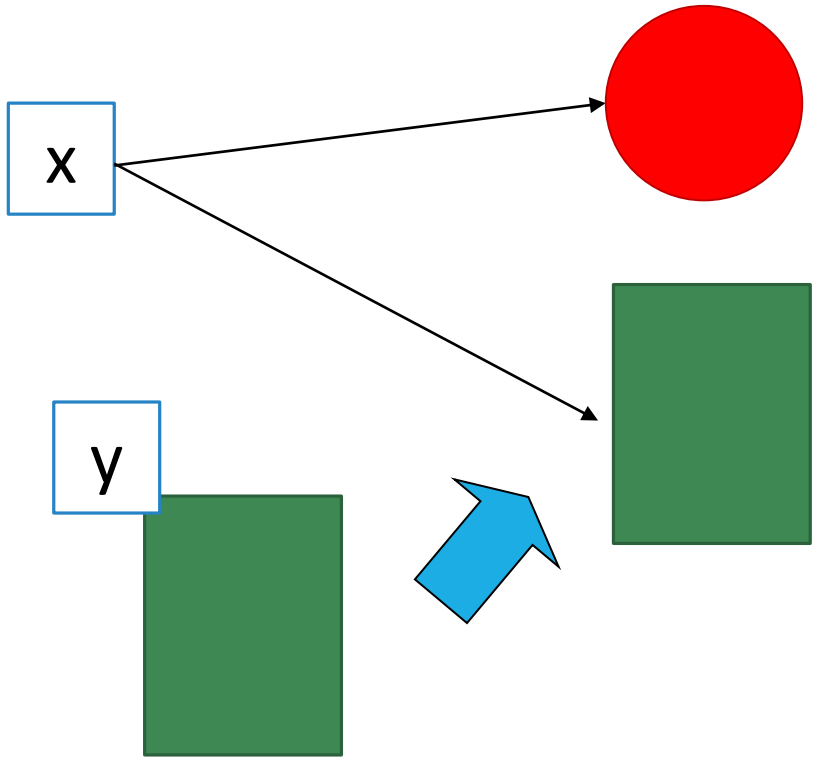
Mindkettő kiterjesztett



x:=y

másolás

TX referencia, TY kiterjesztett



x:=y

klónozás

Üres utasítás

Pascal
- megengedett (`case`)

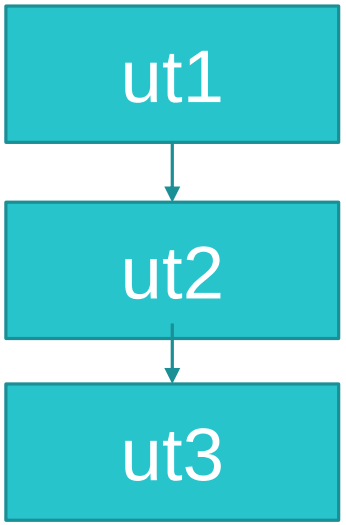
ADA
- null; (`case`)

C++, Java, stb.: „ ; ” használható
- Például ciklusnak lehet üres törzse

Eiffel: „ ; ” használható
- nincs valódi szerepe

Összetett utasítások

Szekvencia:



ut1; ut2; ut3;

Szekvencia

Lehet-e üres?

Terminátor vagy utasításelválasztó-e a „ ; ”?

Lehet-e blokkutasítást létrehozni?

Elhelyezhető-e a blokkutasításban deklaráció?

Szekvencia

Pascal, Delphi: a „ ; ” elválasztja az utasításokat:

- `begin ut1; ut2; ut3 end`

- üres utasítás is lehet:
  - `begin ut1; ut2; ut3; end`
- Az üres utasítás lehetősége miatt a „ ; ” beírása megváltoztathatja a program jelentését!

ADA: a „ ; ” lezárja az utasítást

C++, Java, Objective-C: a „ ; ” lezárja az utasítást

- de nem mindet, például a blokk utasítást nem

- Illetve , elválaszt kifejezéseket C++ esetén

Szekvencia

Python: az utasításokat az új sor jelek zárják le

- De a ; megengedett elválasztóként soron belül
  - `x = 5; print(x)`

Swift: a ; elválasztja az utasításokat

- A fordítóprogram elhelyezi

Blokk utasítások

Utasítások sorozatából alkothatunk egy összetett utasítást, blokkot.

Bizonyos nyelvekben lehet deklarációs része is.

Főleg azokban a nyelvekben fontos, ahol a feltételes és a ciklus utasítások csak egy utasítást tartalmazhatnak

- Pascal, C++, Java, …

Pascal: `begin utasítássorozat end`

ADA:

- `[declare deklarációk] begin utasítássorozat [kivételkezelő rész] end;`
- Például

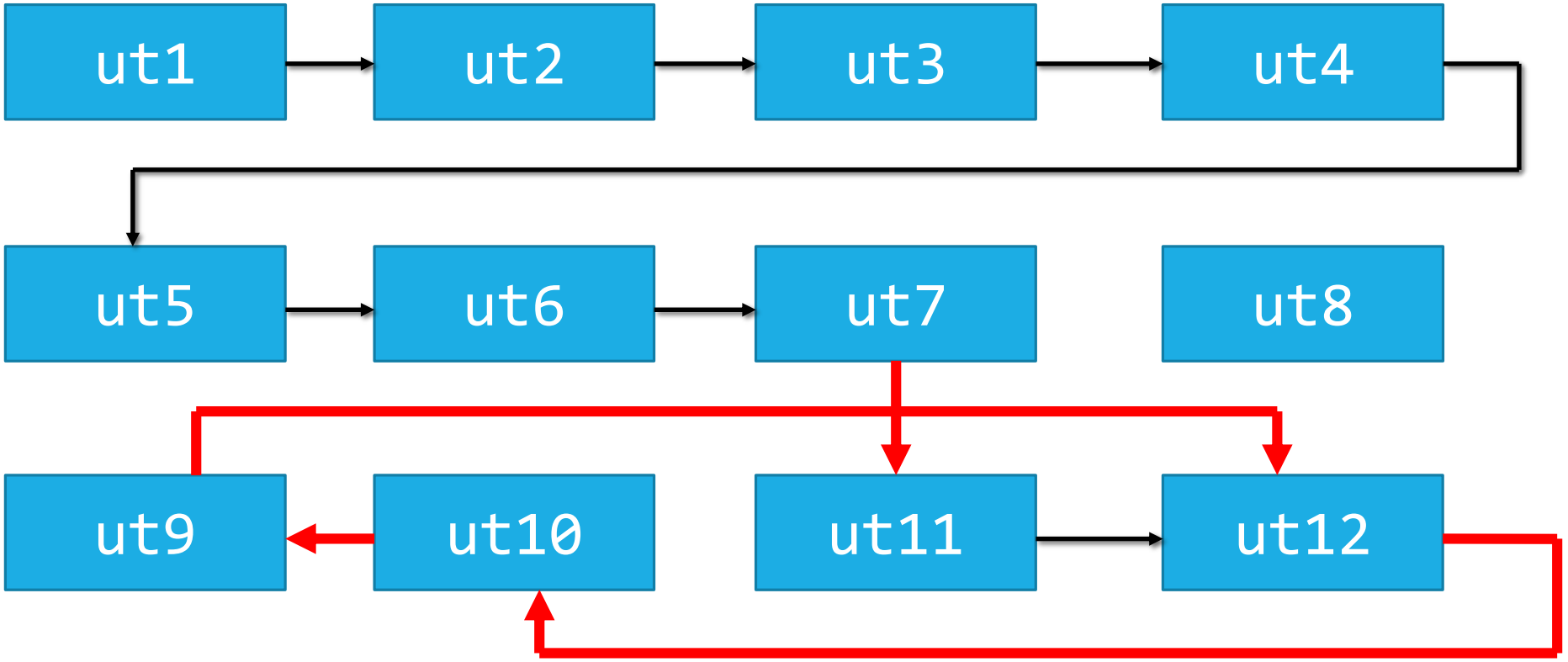
```
Csere:
  declare
  Temp : Integer;
    begin
      Temp := I; I := J; J := Temp;
    end Csere;
```

C++, Java, …

- “{“ és “}” között.

Feltétel nélküli vezérlésátadás

goto



Feltétel nélküli vezérlésátadás

javaslat: `goto` nélkül

Feltétel nélküli vezérlésátadás

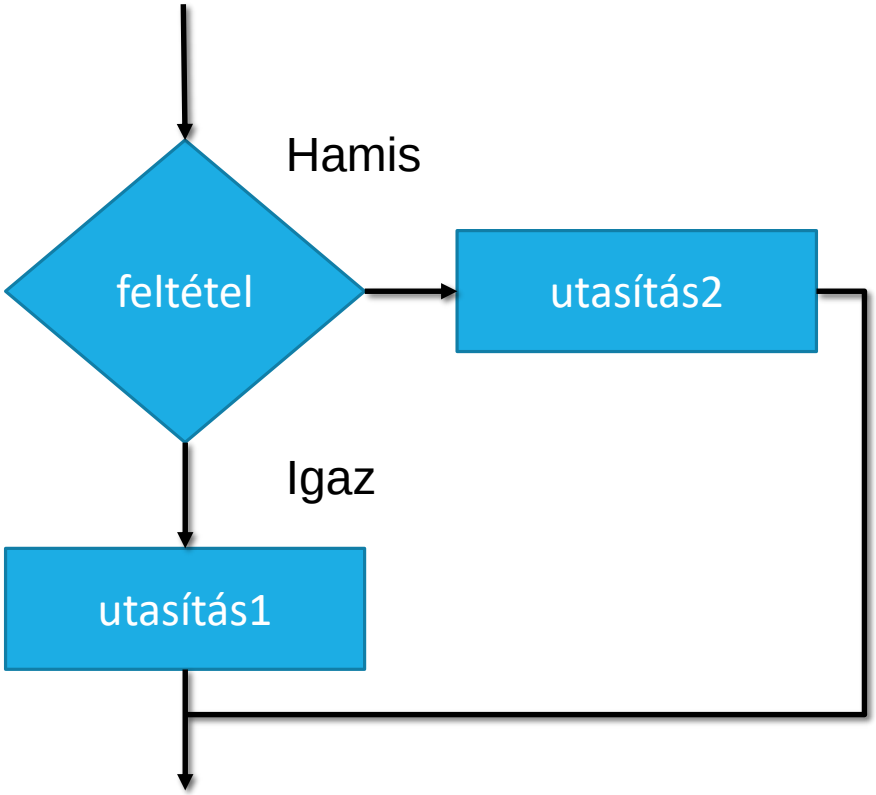
Ha mégis – Pascal, C stb.

```
címke:
   .... utasítások ...
   goto címke;
```

Nincs: Modula-3, Java, …

Elágazások:

- Feltétel igaz vagy hamis értékétől függően valamilyen utasítás(csoport) végrehajtása



Elágazások

```
if feltétel then ut1
```

```
if feltétel then ut1 else ut2
```

„csellengő” else kérdése

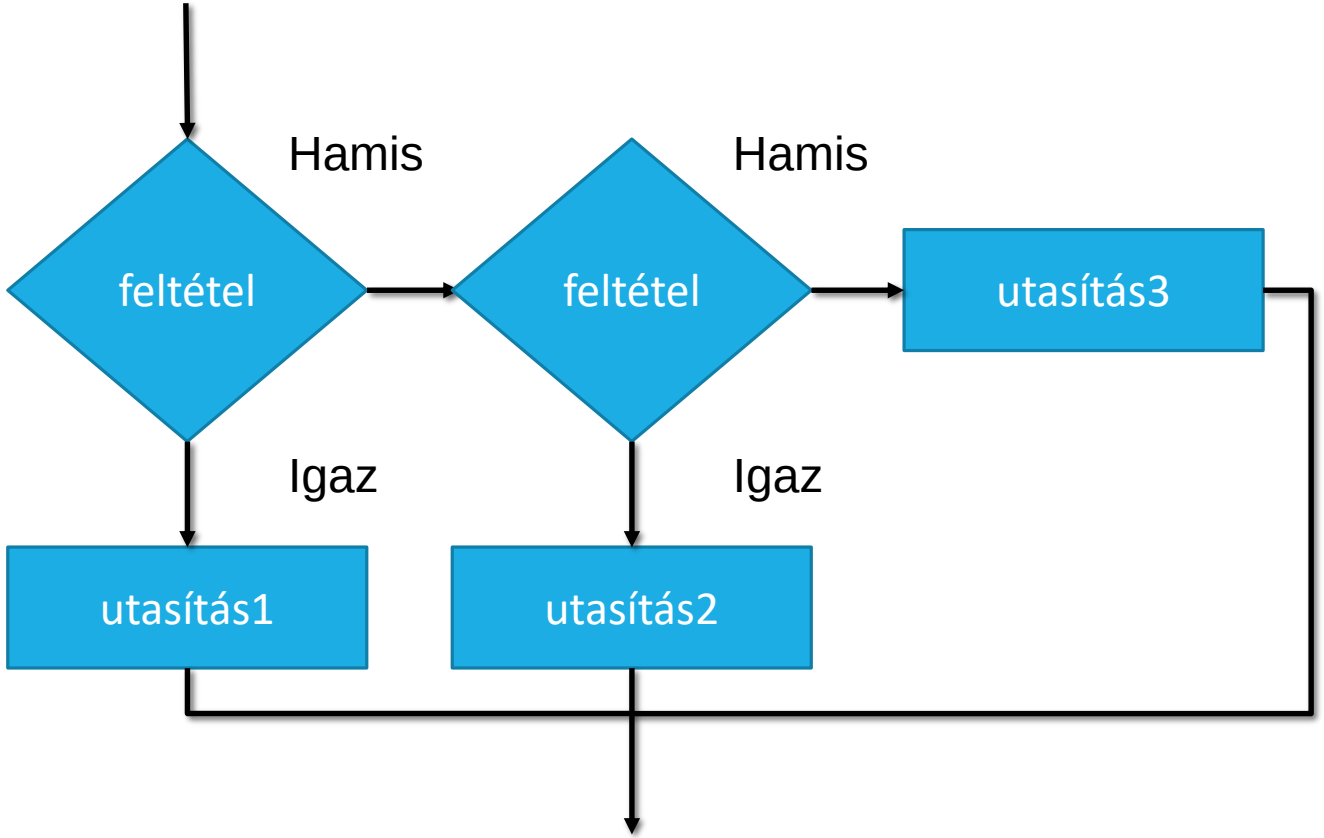
- `if felt1 then if felt2 then ut1 else ut2`
- Mit jelent?
  - `if felt1 then (if felt2 then ut1 else ut2)`
- Vagy?
  - `if felt1 then (if felt2 then ut1) else ut2`

Elágazások – csellengő else

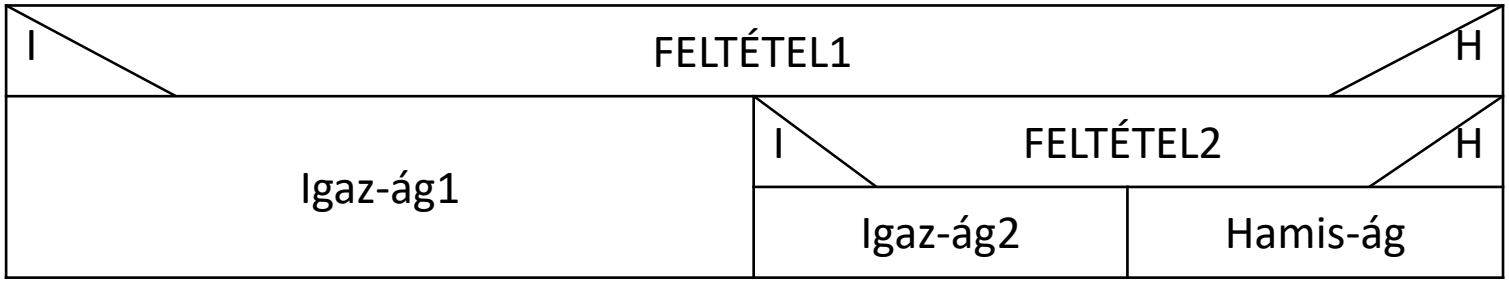
Különböző megoldások:

- blokk utasítás kell a beágyazásnál
- az `else` rész a legbelső `if`-hez tartozik
- explicit `end` konstrukció bevezetése

Többirányú elágazások



Többirányú elágazások



Többirányú elágazások

`elsif` lehetősége – óvatos módosítás kell!

```
if felt1 then s1;
elsif felt2 then s2;
elsif felt3 then s3;
else s4;
end if;
```

Pascal

```
if felt then S1 [else S2]
```

- több utasítás: blokk kell

„csellengő” else:

- legbelső if :

```
if felt1 then
  if felt2 then S1
           else S2
```

különben blokk:

```
if felt1 then
  begin
    if felt2 then S1
  end
else S2
```

Pascal

```
if (a>b)
  then writeln("a>b")
else
  if (a<b) then writeln("a<b")
           else writeln("a=b");
```

Pascal

Vigyázat!

- `if felt1 then ut1 else ut2`
  nem ugyanaz, mint:
- `if felt1 then ut1; else ut2`
  ez ekvivalens:
- `if felt1 then ut1; else ut2`
- ha
  - `if felt1 then ut1 else ut2` -ben
    S1 után beszúrunk egy „ ; ”-t, az hiba!

ADA

end if a végén, elsif megengedett:

```
if <expr>1 then
   <statm>1;
{elsif (<expr>2)then
   <statm>2;}
[else
   <statm>3; ]
end if;
```

```
if A>B then
   Put_Line("a>b");
elsif A<B then
   Put_Line("a<b");
else
   Put_Line("a=b");
end if;
```

C++

aritmetikai kifejezés kell, nem nulla: true.

„csellengő” else: legbelső if

- blokk kell különben

```
if (<expr>)
     <statm1>
[else
     <statm2>]
```

```
if (a>b) cout << "a>b";
else
  if (a<b) cout << "a<b";
  else cout << "a=b";
```

aritmetikai if kifejezés pl.:

```
if (a<=b) max=b;
else max=a;
```

inkább:

```
max=(a<=b)?b:a;
```

Java

Hasonló a C++-hoz

- kivévea kifejezés típusa boolean kell legyen

```
if (<expr>)
   <statm1>
[else
   <statm2>]
```

```
if (a>b)
  System.out.println("a>b");
else
  if (a<b)
    System.out.println("a<b");
  else
    System.out.println("a=b");
```

Eiffel

end zárja le az utasítást.

Megengedett az elseif használata.

```
if <expr>1 then
   <statm>1
{elseif <expr>2 then
   <statm>2 }
[else
   <statm>3 ]
end
```

```
if (a>b) then
   io.putstring("a>b");
elseif (a < b) then
   io.putstring("a < b")
else
   io.putstring("a = b");
end
```

Python

Csellengő else: a bekezdés számít!!

```
a, b = 1, 2
if a == 1:
    if b == 1:
        print ("a és b is 1")
else:
    print ("a nem 1")
```

Python

Csellengő else: a bekezdés számít!!

Így mást jelent:

```
a, b = 1, 2
if a == 1:
    if b == 1:
        print ("a és b is 1")
    else:
        print ("a nem 1")
```

if és értékadás

Ruby:

- `num += -1 if num < 0`

Scala:

- `val x = if (a > b) a else b`

Jobban olvasható, mint a `? :`

if és az értékadás

Kotlin

- További utasítások kerülhetnek bele

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

Esetkiválasztásos elágazások

Valamilyen kifejezés (szelektor) értékétől függően, általánosan:

kifejezés					
érték1	érték2	érték3	értékek	intervallum	egyéb
utasítások	utasítások	utasítások	utasítások	utasítások	utasítások

Esetkiválasztásos elágazás

Kérdések

- Mi lehet a szelektor típusa?
- Fel kell-e sorolni a szelektortípus minden lehetséges értékét?
- Mi történik, ha fel nem sorolt értéket vesz fel a szelektor?
- „Rácsorog”-e a vezérlés a következő kiválasztási ágakra is?
- Diszjunktnak kell-e lennie a kiválasztási értékeknek?
- Mi állhat a kiválasztási feltételben?
  - egy érték
  - értékek felsorolása
  - intervallum

„Case” utasítások

```
case expr of
  const1: st1;
  const2: st2;
 …
  constn: stn
end
```

„Case” utasítások

Sokszor igaz a case konstansokra:

- Tetszőleges sorrend,
- Nem feltétlenül egymás után,
- Több is vonatkozhat ugyanarra az alutasításra,
- Mind különböző kell legyen - különben, ha $const_i$ és $const_j$ egyenlőek, akkor melyiket választanánk, st_i -t vagy st_j-t?
- A kifejezés típusa diszkrét kell legyen (egész vagy felsorolási típ)
- Kell adni egy „others” ágat, hogy minden lehetőséget lefedjünk.

Pascal

A szelektor típusa lehet: integer, character, boolean vagy tetszőleges felsorolási vagy intervallum típus.

„else” megengedett, de nem kötelező (skip).

```
case var of
  val1: statm1;
  val2.: statm2;
...
  vali…valj : statmi;
[else statm]
end;
```

```
var Age: Byte;
case Age of
  0..13: Write('Child');
 14..23: Write('Young');
 24..65: Write('Adult');
 else    Write('Old');
end
```

ADA

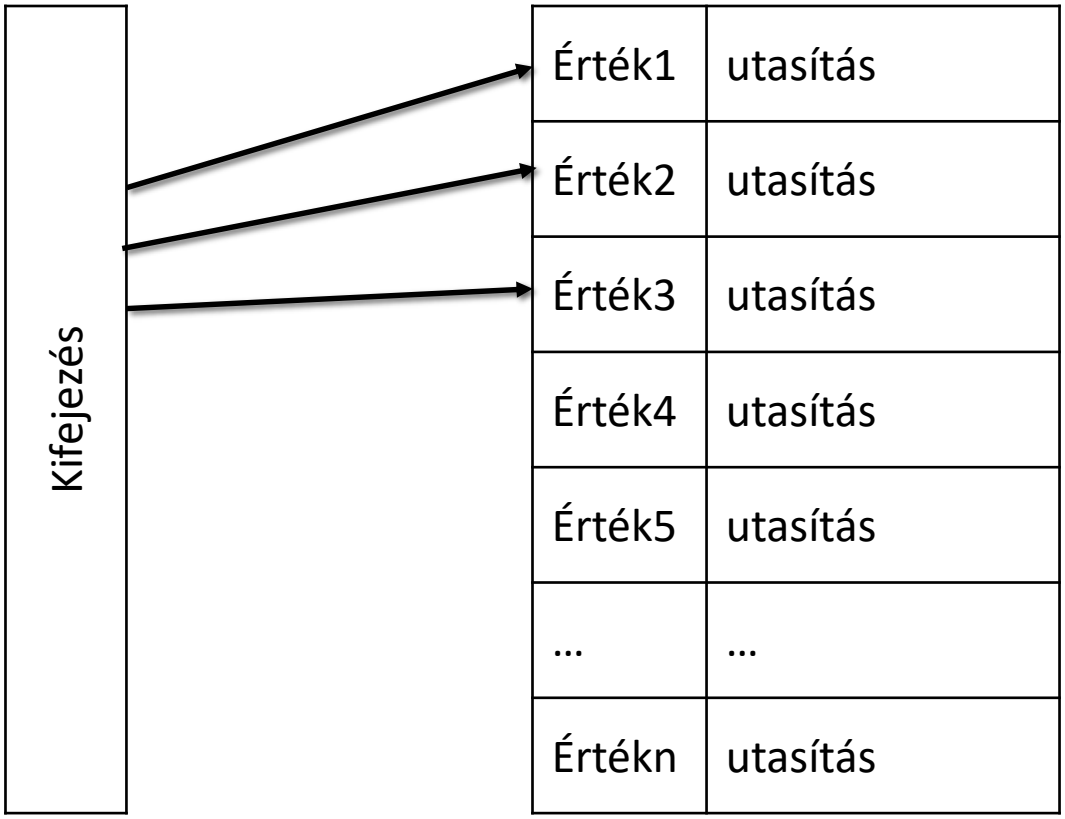
```
case expr is
 when choice1 => statms1;
 when choice2 => statms2; ...
 when others =>  statms;
end case;
```

others kötelező, ha nincs minden lefedve!

.. - intervallum,
| - vagy

```
case Today is
  when Monday => Initial_Balance;
  when Friday => Closing_Balance;
  when Tuesday..Thursday => Report(Today);
  when others => null;
end case;
```

C++


Kifejezés
Érték1
utasítás
Érték2
utasítás
Érték3
utasítás
Érték4
utasítás
Érték5
utasítás
…
…
Értékn
utasítás

break

C++

```
switch (intexpr) {
  case label1: statm1; break;
  case label2: statm2; break;
  ...
  default: statm;
}
```

kifejezés „integral” típusú

- „default:” címke lehetősége (nem kötelező).
- itt break kell, ha nem akarom folytatni!

C++

```
switch (val){
   case 1 : cout<<"case 1\n";
   case 2 : cout<<"case 2\n";
   default: cout<<"default case";
}
```

ha val=1, az eredmény:

```
case 1
case 2
default case
```

Java

Mint a C++:

```
switch (val){
 case 1 : System.out.println("case 1\n");
 case 2 : System.out.println("case 2\n"); break;
 default: System.out.println("default case");
}
```

ha val=1, az eredmény:

```
case 1
case 2
```

Java

Értékadás

```
final String exampleString =
 switch (val){
  case 1 : "case 1"; break;
  case 2 : "case 2"; break;
  default: "default"; break;
 }
```

ha val=1, az eredmény:

```
 case 1
```

Java

Értékadáshoz új szintaxis

```
final String exampleString =
 switch (val){
  case 1 -> "case 1";
  case 2 -> "case 1";
  default -> "default";
 }
```

Fontos változás, hogy itt nincsen átcsorgás!

ha val=1, az eredmény:

```
case 1
```

Kotlin

Hasonlóan a Java új szintaxishoz, de általánosan:

```
when (x) {
     1 -> print("x == 1")
     2 -> print("x == 2")
     else -> {
          print("x is neither 1 nor 2")
     }
}
```

Kotlin

Kifejezések és intervallumok használata

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}

when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is not in the range")
    else -> print("none of the above")
}
```

Eiffel

Pascal-szerű szemantika

A változó típusa INTEGER vagy CHARACTER

```
inspect var
  when expr1 then statmblock1
  when expr2 then statmblock2
  ...
  else statmblock
end
```

Exception lép fel, ha nem gondoskodtunk a megfelelő választék-elemről (null utasítás szükséges lehet).

Eiffel

```
inspect c
  when '0'..'9'                then n:= n + 1;
  when 'a'..'z','A'..'Z' then s:= s + 1;
  else  o:= o + 1;
end
```

C#

```
switch (kif)
{
    case ertek1 : utasítások… break;
    case ertek2 : utasítások… break;
    case ertek3 : utasítások… break; vagy:
                  goto case KONST/default;
…
}
```

szemantika

Ciklusként is működhet!!

SWIFT

Lehetséges az értékre rész-megszorítást adni

```
switch vegetable {
  case "celery":
    let vegetableComment = …
  case "cucumber", "watercress":
    let vegetableComment = …"
  case let x where x.hasSuffix("pepper"):
    let vegetableComment = …
  default:
    let vegetableComment = …
}
```

SWIFT

Tuple használata esetén, lehetséges figyelmen kívül hagyni az értékek egy részét:

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

Absztrakció, alprogramok

KÖVETKEZŐ HÉTEN