

Változók élettartama

- A program végrehajtási idejének egy szakasza
- Amíg a változó számára lefoglalt tárhely a változóé
- Kapcsolódó fogalmak
 - Hatókör
 - Memóriára való leképezés

Hatókör és élettartam

- Sok esetben az élettartam az az idő, amíg a változó hatókörében vagyunk
 - Globális változó: az egész program végrehajtása alatt létezik
 - Lokális változó: csak a definiáló blokk végrehajtása alatt létezik
- Nem mindig így van
 - „Dinamikusan” lefoglalt változók
 - C/C++ static változók, Java zárványok (inner)

Dinamikusan lefoglalt változók

- Allokátorral lefoglalt tárterület
- Mutatók, referenciák (egy későbbi előadás)
- Lásd még: memóriára való leképezés

Ha a hatókör kisebb az élettartamnál

```
int sum ( int p )  
{  
    static int s = 0;  
    s += p;  
    return s;  
}
```

```
void f ( void )  
{  
    cout << sum(10) << endl;  
    cout << sum(3) << endl;  
}
```

Deklaráció kiértékelése

- Statikus (fordítás közben)
 - Rugalmatlan
 - C, C++
 - `int t[10];`
- Dinamikus (futás közben)
 - pl. Ada
 - A blokk utasítás szerepe

A blokk utasítás egyik haszna

```
procedure A is
    N: Integer;
begin
    Put("Hány adat lesz?"); Get(N);
    declare
        T: array (1..N) of Integer;
    begin
        -- beolvasás és feldolgozás
    end;
end;
```

Egy másik haszon

- Ha egy nagy tárigényű változót csak rövid ideig akarok használni
- Egy blokk utasítás lokális változója
- Ada: kivételkezelés

Változók leképzése a memóriára

- Statikus
 - A fordító a tárgykódban lefoglal neki helyet
- Automatikus
 - Futás közben a végrehajtási vermen jön létre és szűnik meg
- Dinamikus
 - Allokátorral foglaljuk le, és pl. deallokátorral szabadítjuk fel (vagy a szemétgyűjtés...)

Statikus változók

- Az élettartamuk a teljes program végrehajtása
- Fordítási időben tárterület rendelhető hozzájuk
- Tipikusan a globális változók
 - A hatókörhöz igazodó élettartam
- De ilyenek a C/C++ static változók is
- Egy futtatható program: kód + adat

Egy program a memóriában

- Futtatás közben a program által használt tár felépítése:
 - kód
 - (statikus) adatok
 - végrehajtási verem
 - dinamikus tárterület (heap)

Dinamikus változók

- Dinamikus tárterület
 - Ahonnan a programozó allokátorral tud memóriát foglalni
 - Explicit felszabadítás vagy szemétdgyűjtés
- Mutatók és referenciák
- Utasítás hatására jön létre (és esetleg szabadul fel) a változó
 - Statikus és automatikus: deklaráció hatására

Végrehajtási verem

- execution stack
- Az alprogramhívások tárolására
- Az éppen végrehajtás alatt álló alprogramokról aktivációs rekordok
- A verem teteje: melyik alprogramban van az aktuálisan végrehajtott utasítás
- A verem alja: a főprogram
- Egy alprogram nem érhet véget, amíg az általa hívott alprogramok véget nem értek
- Dinamikus (hívási) lánc

Aktivációs rekord

- Activation record, stack frame
- Egy alprogram meghívásakor bekerül egy aktivációs rekord a verembe
- Az alprogram befejeződésekor kikerül az aktivációs rekord a veremből
- Rekurzív alprogram: több aktivációs rekord
- Az aktivációs rekord tartalma:
paraméterek, lokális változók, egyebek
 - Blokkszerkezetes statikus hatókörű nyelvek esetén:
tartalmazó alprogram

Automatikus változók

- A végrehajtási veremben
- A blokkok (alprogramok, blokk utasítások) lokális változói
 - ha nem static...
- Automatikusan jönnek létre és szűnnek meg a blokk végrehajtásakor
 - A hatókörhöz igazodó élettartam
- Rekurzió: több példány is lehet belőlük

Kommunikáció programegységek között

- Nonlokális és globális változók
 - Általában nem szerencsés, nem javasolt
 - Néha hasznos
 - Rövidebb a paraméterlista
 - Hatékonyabb lehet a kód
 - Blokkok, hatókör, blokkszerkezetes nyelvek
- Paraméterek

Alprogram nonlokális változói

```
procedure Rendez ( T: in out Tömb ) is
  function Max_Hely ( T: Tömb ) return Index is
    Mh: Index := T'First;
  begin
    for I in T'Range loop
      if T(Mh) < T(I) then Mh := I; end if;
    end loop;
    return Mh;
  end;
begin
  for I in reverse T'Range loop
    Mh := Max_Hely( T(T'First .. I) );
    Felcserél( T(I), T(Mh) );
  end loop;
end Rendez;
```


Információcsere taszkok között

- Nonlokális (globális) változókon keresztül
 - Nem szeretjük...
 - Biztonságosabbá tehető különféle pragmak segítségével (Atomic és Volatile)
- Randevúval
 - Ezt fogjuk sokat gyakorolni
 - Aszimmetrikus, szinkron, pont-pont, kétirányú kommunikációt tesz lehetővé
- Védett egységek használatával

Nonlokális változón keresztül

procedure P is

 N: Natural := 100;

 task type T;

 task body T is

 begin

 ... if N > 0 then N := N-1; ... end if; ...

 end T;

 A, B: T;

begin ... end P;



legfeljebb N-szer
szabadna

Kivételek

- A végrehajtási verem kiürítése
 - stack trace
- Vezérlésátadás kivételes esetek kezelésénél
- Kivétel: eltérés a megszokottól, az átlagostól
 - Programhiba (dinamikus szemantikai hiba)
pl. tömb túlindexelése
 - Speciális eset jelzése
- Kiváltódás, terjedés, lekezelés, definiálás, kiváltás

Egy program a memóriában

Futtatás közben a program által használt tár felépítése:

- kód
- (statikus) adatok
- végrehajtási verem
- dinamikus tárterület (heap)

Változók leképzése a memóriára

- Statikus
 - A fordító a tárgykódban lefoglal neki helyet
- Automatikus
 - Futás közben a végrehajtási vermen jön létre és szűnik meg
- Dinamikus

Dinamikus változók

- Dinamikus tárterület
 - Ahonnan a programozó allokátorral tud memóriát foglalni
 - Explicit felszabadítás vagy szemétygyűjtés
- Mutatók és referenciák
- Utasítás hatására jön létre (és esetleg szabadul fel) a változó
 - Statikus és automatikus: deklaráció hatására

Típusosztályok az Adában

elemi típusok

skalár típusok

diszkrét típusok

felsorolási

egész (előjeles, ill. moduló)

valós típusok (fix- és lebegőpontos)

mutató típusok

összetett típusok

tömb, rekord stb.

Mutató típusok

- A dinamikus változók használatához
- Ada: access típusok

```
type P is access Integer;
```

```
X: P;
```

- Dinamikus változó létrehozása

```
X := new Integer;
```

- Dinamikus változó elérése a mutatón keresztül

```
X.all := 3;
```

- Dinamikus változó megszüntetése...később...

Mutató típusok definiálása

type P is access Integer;

type Q is access Character;

type R is access Integer;

- Meg kell adni, hogy mire mutató mutatók vannak a típusban: gyűjtőtípus
- P, Q és R különböző típusok

Mutatók a C++ nyelvben

- Nincsen önálló „mutató típus” fogalom
- Mutató típusú változók

```
int *x;
```
- Nem lehet két különböző „int-re mutató mutató” típust definiálni
- A Javában csak implicit módon jelennek meg a mutatók

Mutató, ami sehova sem mutat

- Nullpointer
- Az Adában: a null érték
- Minden mutató típusnak típusértéke

```
type P is access Integer;
```

```
X: P := null;
```

```
Y: P;    -- implicit módon null-ra inicializálódik
```

Hivatkozás null-on keresztül

```
type P is access Integer;
```

```
X: P := null;
```

```
N: Integer := X.all;
```



Constraint_Error
futási idejű hiba

Indirekció

- A mutatókkal indirekt módon érjük el a változóinkat. (mutató \approx memóriacím)

```
X := new Integer;
```

- Az X változó egy újonnan (a heap-en) létrehozott változóra mutat: X.all

```
X.all := 1;
```

```
X.all := X.all + 1;
```



balérték

Indirekció (1)

```
type P is access Integer;
```

```
X, Y: P;
```

```
X := new Integer;
```

```
X := new Integer;
```

```
X.all := 3;
```

```
Y := X;
```

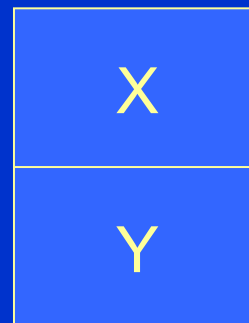
```
X.all := 5;
```

```
X := new Integer;
```

Indirekció (2)

```
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;
```



Indirekció (3)

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

```
X := new Integer;
```

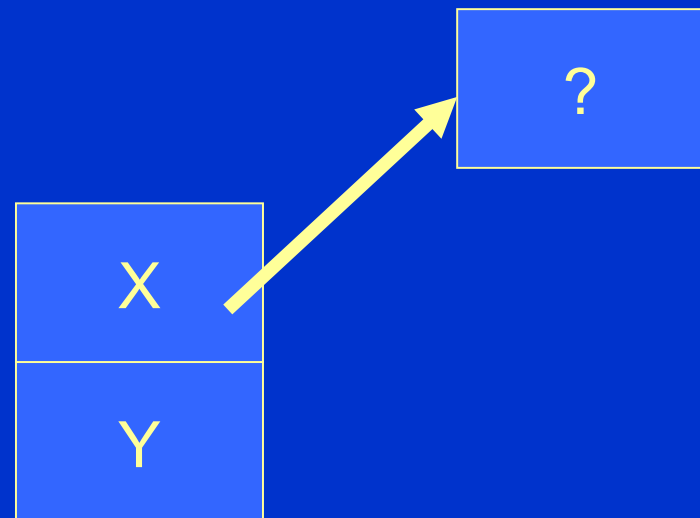
```
X := new Integer;
```

```
X.all := 3;
```

```
Y := X;
```

```
X.all := 5;
```

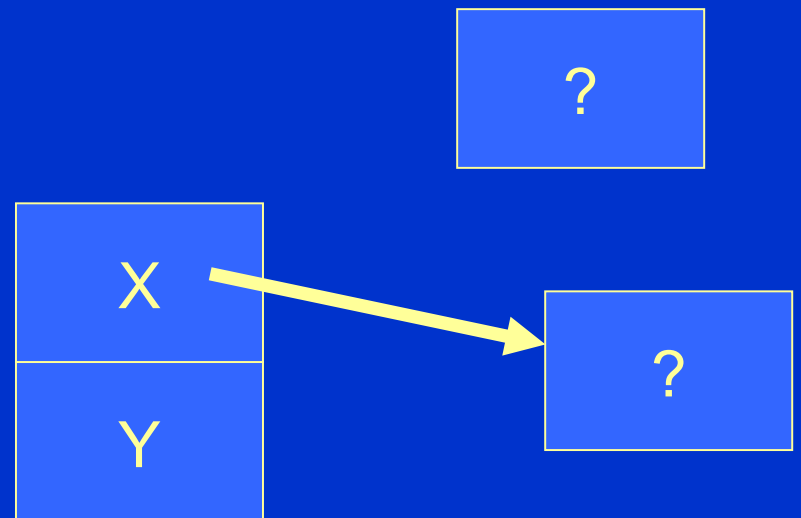
```
X := new Integer;
```



Indirekció (4)

```
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;
```



Indirekció (5)

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

```
X := new Integer;
```

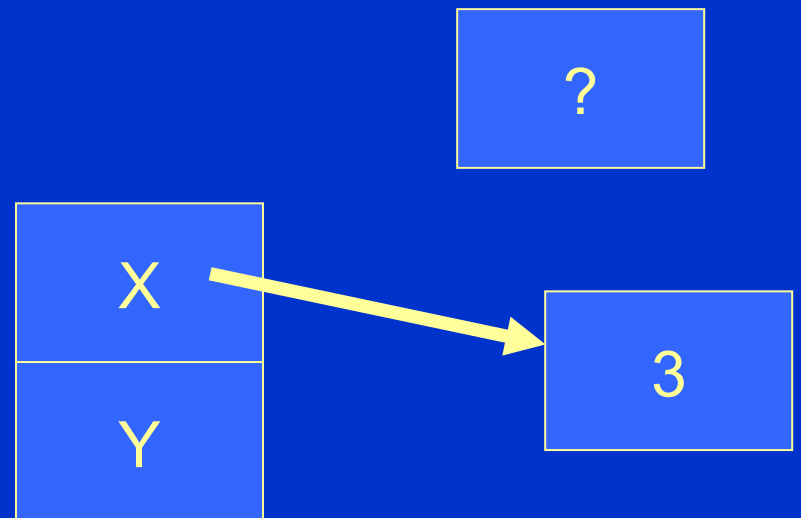
```
X := new Integer;
```

```
X.all := 3;
```

```
Y := X;
```

```
X.all := 5;
```

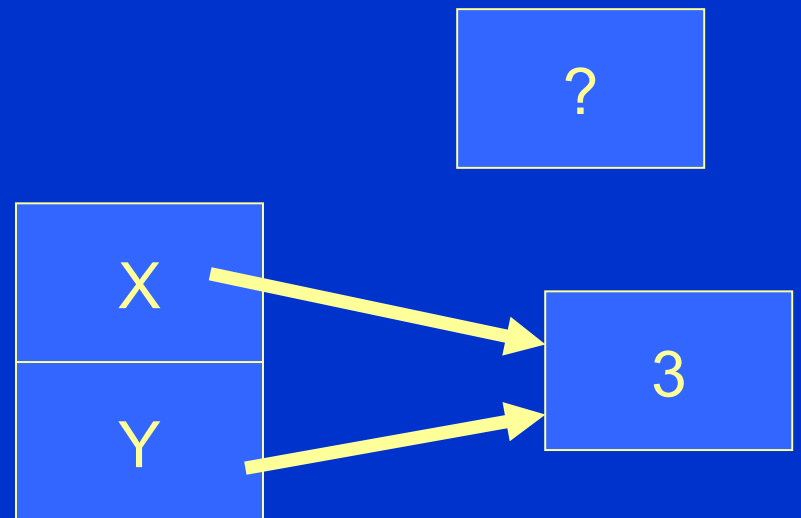
```
X := new Integer;
```



Indirekció (6)

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

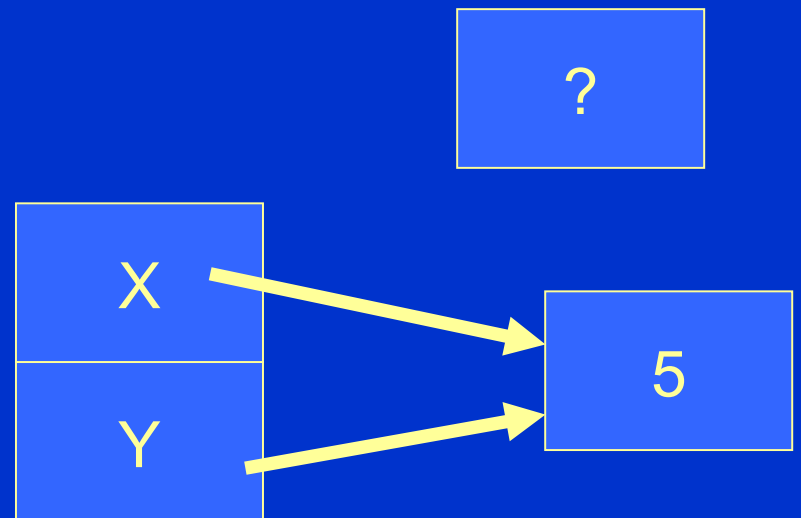
```
X := new Integer;  
X := new Integer;  
X.all := 3;  
Y := X;  
X.all := 5;  
X := new Integer;
```



Indirekció (7)

```
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;
```



Indirekció (8)

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

```
X := new Integer;
```

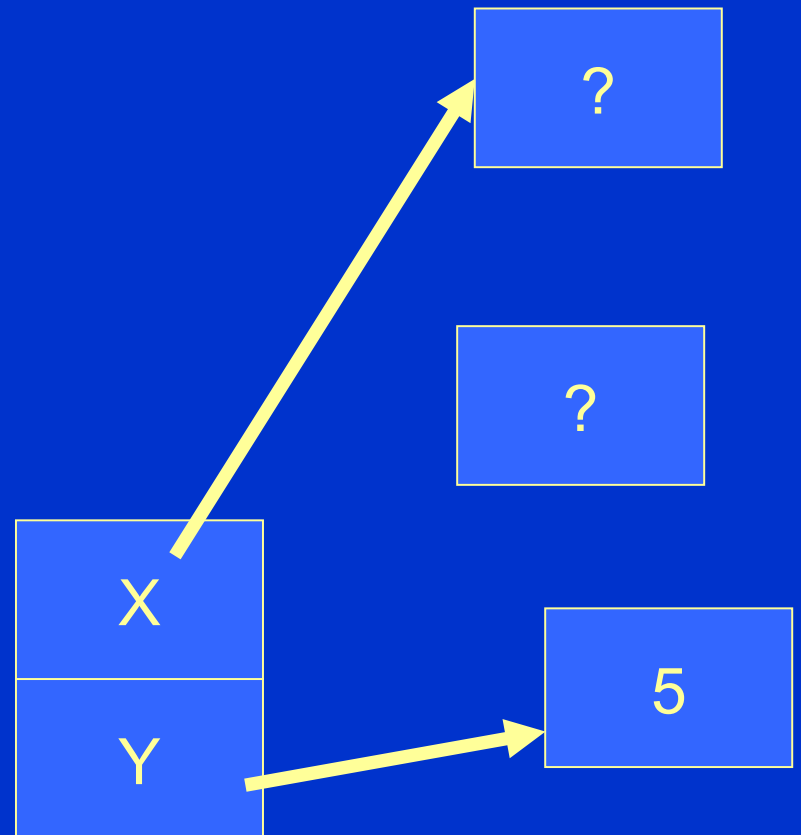
```
X := new Integer;
```

```
X.all := 3;
```

```
Y := X;
```

```
X.all := 5;
```

```
X := new Integer;
```



Alias kialakulása

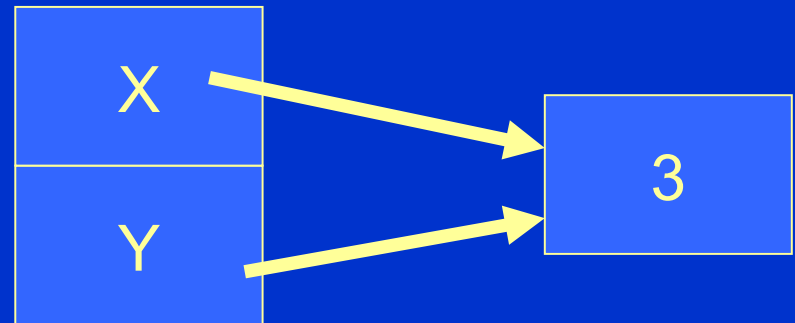
- Ha ugyanazt a változót többféleképpen is elértem
- Például cím szerinti paraméterátadásnál
- Például mutatók használatával

```
X := new Integer;
```

```
X.all := 3;
```

```
Y := X;
```

- Veszélyes, de hasznos
 - C++ referenciák



A dinamikus memóriakezelés baja

- Mikor szabadítsunk fel egy változót?
- Dinamikus változó élettartama
- Az alias-ok miatt bonyolult
- Ha felszabadítom: nem hivatkozik még rá valaki egy másik néven?
- Ha nem szabadítom fel: felszabadítja más?

Ha felszabadítom

```
int *x = new int;  
int *y = identitás(x);  
delete x;  
*y = 1;           // illegális memóriahivatkozás
```

```
int *identitás( int *p )  
{  
    return p;  
}
```


Ha nem szabadítom fel

```
int *x = new int;  
x = másolat(x);  
// memory leak
```

```
int *másolat( int *p )  
{  
    int *q = new int;  
    *q = *p;  
    return q;  
    // return new int(*p);  
}
```

Megoldások

- Legyen ügyes a programozó
- Legyen szemétygyűjtés
- Használjunk automatikus változókat
 - Ott a hatókörhöz kapcsolódik az élettartam
 - C++: automatikus objektum destruktora

Szemétygyűjtés

- Garbage collection
- Ne a programozónak kelljen megszüntetnie a nem használt dinamikus változókat
- A futtató rendszer megteszi helyette
- Nő a nyelv biztonságossága
- A hatékonyság picit csökken
(a memóriaigény és a futásiidő-igény is nő)
- Megéri (kiforrott szemétygyűjtési algoritmusok)
- LISP (1959), Ada, Java, modern nyelvek

Felszabadítás az Adában

- Szemétygyűjtéssel (alapértelmezett)
 - A típusrendszer az alapja
 - Egészen más, mint például a Javában
- Explicit felszabadítással
 - `Ada.Unchecked_Deallocation`
 - A hatékonyság növelése érdekében
 - Van, amikor csak így lehet

Szemétygyűjtés az Adában

- A dinamikus változó felszabadul, amikor a létrehozásához használt mutató típus megszűnik
- Ekkor már nem férek hozzá a változóhoz
 - alias segítségével sem,
 - csak ha nagyon trükközök
- Tehát biztonságos a felszabadítás

„Automatikus” mutató típus esetén

procedure A is

 type P is access Integer;

 X: P := new Integer;

begin

 X.all := 3;

 ...

end A;



P megszűnik,
és X.all is

A mutató és a mutatott objektum élettartama más hatókörhöz kötött

```
procedure A is
  type P is access Integer;
  X: P;
begin
  declare
    Y: P := new Integer;
  begin
    Y.all := 3;
    X := Y;
  end;
  ...
end A;
```

Y megszűnik, de
Y.all még nem

„Statikus” mutató típus

```
package A is  
    type P is access Integer;  
    ...  
end A;
```

Ha az A egy
könyvtári egység,

```
X: P := new Integer;
```

az X.all a program
végéig létezik

A programozó szabadít fel

- Ha a mutató típus a program végéig létezik, nincs szemétgyűjtés
- A programozó kézbe veheti a felszabadítást
 - Nem csak ilyenkor veheti kézbe...
- `Ada.Unchecked_Deallocation` sablon
 - ez felel meg a C++ `delete`-jének

Ada.Unchecked_Deallocation

```
with Ada.Unchecked_Deallocation;
procedure A is
    type P is access Integer;
    procedure Free is new
        Ada.Unchecked_Deallocation(Integer,P);
    X: P := new Integer;
    Y: P := X;
begin
    Free(X);
    Y.all := 1;           -- definiálatlan viselkedés
end A;
```

Mire használjuk a dinamikus változókat?

- Láncolt adatszerkezetekhez
 - Ha a méret vagy a szerkezet (sokat) változik futás közben (beszűrő, törlő műveletek)
 - Ha nem kell az adatelemeket "közvetlenül" elérni (indexelés helyett csak "sorban")
 - Listák, fák, gráfok, sorozat típusok
- Változó, vagy ismeretlen méretű adatok kezelésére
 - A gyűjtőtípus ilyenkor egy paraméteres típus

Dinamikus méretű objektumok

- A gyűjtőtípus lehet
 - diszkriminációs rekord
 - határozatlan méretű tömb
- Megszorítás megadása: legkésőbb allokáláskor
- Az allokált objektum mérete nem változtatható meg

```
type PString is access String;
```

```
X: PString;      -- akármilyen hosszú szövegre mutathat
```

```
X := new String(1..Méret);
```

```
X := new String ' ("Alma");      -- allokálás + inicializálás
```

```
X := new String ' (S);          -- másolat az S stringről
```

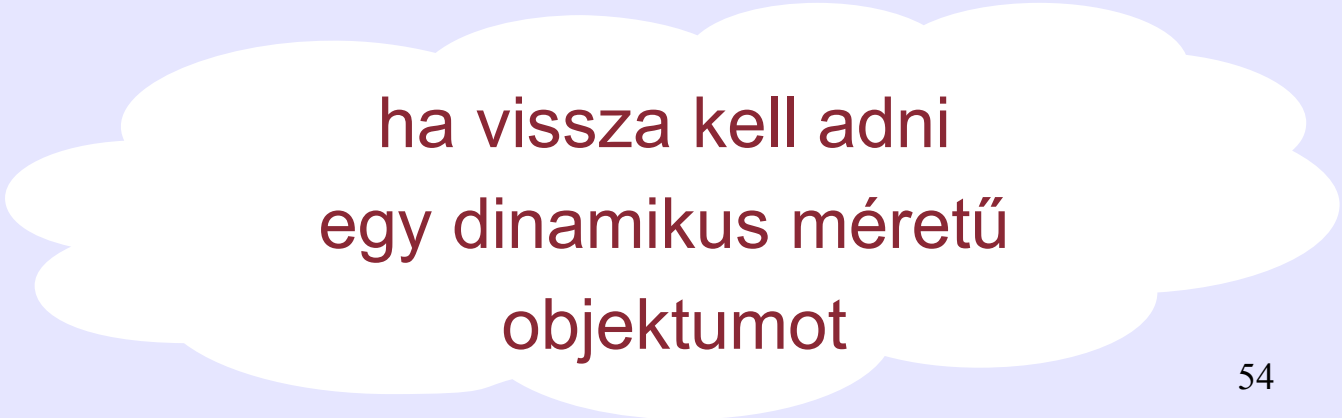
Blokk utasítás vagy mutató

```
procedure A is
  N: Natural;
begin
  Get(N);
  declare
    V: Verem(N);
  begin
    ...
  end;
end A;
```

```
procedure A is
  N: Natural;
  type P_Verem is access Verem;
  V: P_Verem;
begin
  Get(N);
  V := new Verem(N);
  ...
end A;
```

Amikor mutató kell

```
procedure A is
  type P_Verem is access Verem;
  function Létrehoz return P_Verem is
    N: Positive;
  begin
    Get(N);
    return new Verem(N);
  end Létrehoz;
  V: P_Verem := Létrehoz;
begin
  ...
end A;
```



ha vissza kell adni
egy dinamikus méretű
objektumot

Taszkok létrehozása allokátorral

- Taszk típusra mutató típus:

```
task type Üdvözlő ( Szöveg: PString );  
type Üdvözlő_Access is access Üdvözlő;  
P: Üdvözlő_Access;
```

- A mutató típus gyűjtőtípusa egy taszk típus
- Az utasítások között:

```
P := new Üdvözlő( new String'("Szia!") );
```

Taszk elindulása és megállása

procedure Fő is

task type Üdvözlő (Szöveg: PString);

type Üdvözlő_Access is access Üdvözlő;

P: Üdvözlő_Access;

task body Üdvözlő is ... begin ... end;

begin

...

P := new Üdvözlő(new String("Szia!"));

...

end Fő;



itt indul



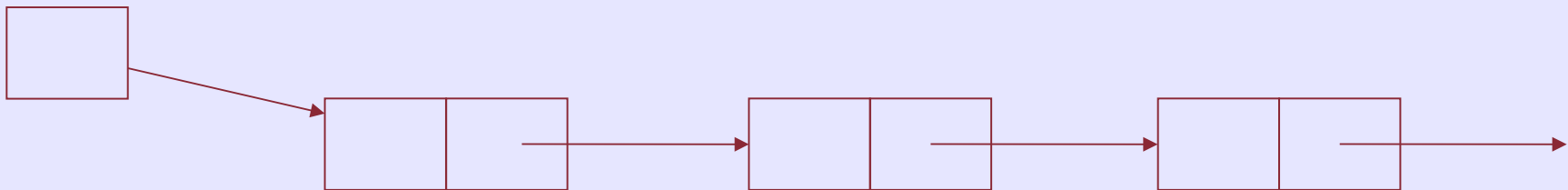
Fő itt bevéárja

Szülő egység

- Az a taszk / alprogram / könyvtári csomag / blokk,
 - amelyben deklaráltuk
 - amely a mutató típust deklarálta
- Elindulás:
 - a szülő deklarációs részének kiértékelése után, a szülő első utasítása előtt
 - az allokátor kiértékelésekor
- A szülő nem ér véget, amíg a gyerek véget nem ér
- Függőségi kapcsolatok, befejeződés

Láncolt adatszerkezet: rekurzív típus

- Pl. Lista adatszerkezet megvalósításához
- Mutató: egy csúcsra mutat
- Csúcs: tartalmaz adatot, valamint mutatót a következő elemre
- Melyiket definiáljuk előbb?



Rekurzív típusok definiálása

deklaráljuk a típust

```
type Csúcs;  
type Mutató is access Csúcs;  
type Csúcs is record  
    Adat: Elem;  
    Következő: Mutató;  
end record;
```

Átlátszatlan rekurzív típusok

```
package Listák is
  type Csúcs is private;
  type Mutató is private;
  ...
private
  type Mutató is access Csúcs;
  type Csúcs is record
    Adat: Elem;
    Következő: Mutató;
  end record;
end Listák;
```

Láncolt adatszerkezet használata (1)

```
-- előfeltétel: M /= null
procedure Mögé_Beszúr ( M: in out Mutató; E: in Elem ) is
  Új: Mutató;
begin
  Új := new Csúcs;
  Új.all.Adat := E;
  Új.all.Következő := M.all.Következő;
  M.all.Következő := Új;
end Mögé_Beszúr;
```

Láncolt adatszerkezet használata (2)

```
-- előfeltétel: M /= null
procedure Mögé_Beszúr ( M: in out Mutató; E: in Elem ) is
  Új: Mutató;
begin
  Új := new Csúcs;
  Új.all.Adat := E;
  Új.all.Következő := M.all.Következő;
  M.all.Következő := Új;
end Mögé_Beszúr;
```

Láncolt adatszerkezet használata (3)

```
-- előfeltétel: M /= null
procedure Mögé_Beszúr ( M: in out Mutató; E: in Elem ) is
    Új: Mutató;
begin
    Új := new Csúcs;
    Új.Adat := E;
    Új.Következő := M.Következő;
    M.Következő := Új;
end Mögé_Beszúr;
```

Láncolt adatszerkezet használata (4)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

 Új: Mutató;

begin

 Új := new Csúcs;

Új.Adat := E;

Új.Következő := M.Következő;

 M.Következő := Új;

end Mögé_Beszúr;



aggregátum?

Láncolt adatszerkezet használata (5)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

 Új: Mutató;

begin

 Új := new Csúcs;

Új.all := (E, M.Következő);

 M.Következő := Új;

end Mögé_Beszúr;

Láncolt adatszerkezet használata (6)

-- előfeltétel: M /= null

procedure Mögė_Beszűr (M: in out Mutató; E: in Elem) is

Új: Mutató;

begin

Új := new Csűcs;

Új.all := (E, M.Következő);

M.Következő := Új;

end Mögė_Beszűr;

allokálás és
inicializálás
egyben?

Láncolt adatszerkezet használata (7)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

 Új: Mutató;

begin

Új := new Csúcs ' (E, M.Következő);

 M.Következő := Új;

end Mögé_Beszúr;

Láncolt adatszerkezet használata (8)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

Új: Mutató;

begin

Új := new Csúcs ' (E, M.Következő);

M.Következő := Új;

end Mögé_Beszúr;

Láncolt adatszerkezet használata (9)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

Új: Mutató := new Csúcs ' (E, M.Következő);

begin

M.Következő := Új;

end Mögé_Beszúr;

Láncolt adatszerkezet használata (10)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

Új: Mutató := new Csúcs ' (E, M.Következő);

begin

M.Következő := **Új**;

end Mögé_Beszúr;

Láncolt adatszerkezet használata (11)

-- előfeltétel: M /= null

procedure Mögé_Beszúr (M: in out Mutató; E: in Elem) is

begin

 M.Következő := new Csúcs ' (E, M.Következő);

end Mögé_Beszúr;

Láncolt adatszerkezet használata (12)

```
-- előfeltétel: M /= null
procedure Mögé_Beszúr ( M: in out Mutató; E: in Elem ) is
  Új: Mutató;
begin
  Új := new Csúcs;
  Új.all.Adat := E;
  Új.all.Következő := M.all.Következő;
  M.all.Következő := Új;
end Mögé_Beszúr;
```


Bizonyos esetekben az all opcionális

- A mutatott objektum egy komponensére történő hivatkozáskor elhagyható

function F (A, B: Integer) return Mutató;

P: Mutató := new Csúcs; S: PString := new String(1..5);

P.all.Adat + 1

F(X,Y).all.Adat + 1

S.all(1) := 'a';

P.Adat + 1

F(X,Y).Adat + 1

S(1) := 'a';

- Akkor szokás használni, ha az egész hivatkozott objektummal csinálunk valamit

if F(X,Y).all /= P.all then Put(S.all); end if;

Sor típus láncolt ábrázolással

generic

type Elem is private;

package Sorok is

type Sor is limited private;

procedure Betesz (S: in out Sor; E: in Elem);

procedure Kivesz (S: in out Sor; E: out Elem);

...

private

...

end Sorok;

fejelem nélküli
egyszeresen láncolt
listával

Reprezentáció

private

type Csúcs;

type Mutató is access Csúcs;

type Csúcs is record

 Adat: Elem;

 Következő: Mutató;

end record;

type Sor is record

 Eleje, Vége: Mutató := null;

end record;

Implementáció (Betesz)

```
package body Sorok is
  procedure Betesz ( S: in out Sor; E: in Elem ) is
    Új: Mutató := new Csúcs ' (E,null);
  begin
    if S.Vége = null then
      S := (Új, Új);
    else
      S.Vége.Következő := Új;
      S.Vége := Új;
    end if;
  end Betesz;
  ...
end Sorok;
```

Implementáció (Kivesz)

```
procedure Kivesz ( S: in out Sor; E: out Elem ) is
begin
    if S.Eleje = null then raise Üres_Sor;
    else
        E := S.Eleje.Adat;
        if S.Eleje = S.Vége then
            S := (null, null);
        else
            S.Eleje := S.Eleje.Következő;
        end if;
    end if;
end Kivesz;
```

Memóriaszivárgás: felszabadítás kell!

```
with Ada.Unchecked_Deallocation;  
package body Sorok is  
    ...  
    procedure Felszabadít is  
        new Ada.Unchecked_Deallocation(Csúcs, Mutató);  
    procedure Kivesz ( S: in out Sor; E: out Elem ) is ...  
    ...  
end Sorok;
```

Implementáció (Kivesz) javítva

procedure Kivesz (S: in out Sor; E: out Elem) is

Régi: Mutató := S.Eleje;

begin

if Régi = null then raise Üres_Sor;

else E := Régi.Adat;

if S.Eleje = S.Vége then S := (null, null);

else S.Eleje := S.Eleje.Következő;

end if;

Felszabadít(Régi);

end if;

end Kivesz;

Használjuk a Sor típust

```
with Sorok;  
procedure A is  
    package Int_Sorok is new Sorok(Integer);  
    procedure B is  
        S: Int_Sorok.Sor;  
    begin  
        Int_Sorok.Betesz(S,1);  
    end B;  
begin  
    B;  
    ...  
end A;
```

Nem szabadul fel a
sort alkotó lista

Memóriaszivárgás

```
with Sorok;  
procedure A is  
    package Int_Sorok is new Sorok(Integer);  
    procedure B is  
        S: Int_Sorok.Sor;  
    begin  
        Int_Sorok.Betesz(S,1);  
    end B;  
begin  
    B; B; B; B; B; B; B; B; B; B; B; B; B; B; B;  
    ...  
end A;
```

Mi történik?

- A sor objektumok a stack-en jönnek létre
- A sor elemei a heap-en allokkáltak
- Amikor a sor objektum megszűnik (automatikusan), az elemek nem szabadulnak fel
- Megoldás
 - Felszámoló eljárást írni, és azt ilyenkor meghívni
 - C++: a sor destruktorában felszabadítani
 - Ada 95: Controlled típust használni

Destruktor az Adában

```
with Ada.Finalization; use Ada.Finalization;  
generic  
    type Elem is private;  
package Sorok is  
    type Sor is new Limited_Controlled with private;  
    procedure Finalize ( S: in out Sor );  
    procedure Betesz ( S: in out Sor; E: in Elem );  
    procedure Kivesz ( S: in out Sor; E: out Elem );  
    ...  
private  
    ...  
end Sorok;
```

Reprezentáció

private

type Csúcs;

type Mutató is access Csúcs;

type Csúcs is record

 Adat: Elem;

 Következő: Mutató;

end record;

type Sor is **new Limited_Controlled with** record

 Eleje, Vége: Mutató := null;

end record;

A Limited_Controlled típus

- Jelölt (tagged) típus
 - Újabb komponensekkel bővíthető rekord típus
 - Az OOP támogatásához (pl. dinamikus kötés)
- A Sor típus ennek egy leszármazottja (new)
- Definiál konstruktort és destruktort, amelyeket a leszármazott felüldefiniálhat
 - A konstruktor és a destruktorkor automatikusan lefut létrehozáskor, illetve felszámolásakor

```
type Limited_Controlled is abstract tagged limited private;  
procedure Initialize (Object : in out Limited_Controlled);  
procedure Finalize (Object : in out Limited_Controlled);
```

Implementáció (Finalize)

```
procedure Finalize ( S: in out Sor ) is
    P: Mutató;
begin
    while S.Eleje /= null loop
        P := S.Eleje;
        S.Eleje := S.Eleje.Következő;
        Felszabadít(P);
    end loop;
end Finalize;
```

Implementáció (visszatérés)

```
procedure Betesz ( S: in out Sor; E: in Elem ) is
```

```
    Új: Mutató := new Csúcs ' (E,null);
```

```
begin
```

```
    if S.Vége = null then S.Eleje := Új; else S.Vége.Következő := Új; end if;
```

```
    S.Vége := Új;
```

```
end Betesz;
```

```
procedure Kivesz ( S: in out Sor; E: out Elem ) is
```

```
    Régi: Mutató := S.Eleje;
```

```
begin
```

```
    if Régi = null then raise Üres_Sor;
```

```
-- ha üres a sor
```

```
    else E := Régi.Adat;
```

```
        if S.Eleje = S.Vége then S.Vége := null; end if;
```

```
-- ha egyelemű volt
```

```
        S.Eleje := S.Eleje.Következő;
```

```
-- itt csatolom ki
```

```
        Felszabadít(Régi);
```

```
    end if;
```

```
end Kivesz;
```

A Controlled típus

- Olyan, mint a Limited_Controlled
- De nem korlátozott típus
- A konstruktor és a destruktor mellett definiál értékadáskor automatikusan lefutó műveletet
 - Ez a primitív művelet is felüldefiniálható
 - Olyasmi, mint a C++ értékadó operátor felüldefiniálása
 - Saját értékadási stratégia (shallow/deep copy)
 - Szokás az = operátort is felüldefiniálni vele együtt

procedure Adjust (Object : in out Controlled);

Másolás és egyenlőségvizsgálat

- Sekély:
 - a struktúrában a mutatók értékadása (alising kialakítása)
 - a struktúrában a mutatók egyenlősége (azonosságvizsgálat)
- Custom:
 - bizonyos mutatókon lehet sekély, másokon mély
 - ahogy az adott típus logikája kívánja
- Mély:
 - a struktúrában a mutatók által hivatkozott objektumon tranzitívan
 - a struktúrában a mutatók által hivatkozott objektumon tranzitívan

Adjust és “=”

```
with Ada.Finalization; use Ada.Finalization;
```

```
generic
```

```
    type Elem is private;
```

```
package Sorok is
```

```
    type Sor is new Controlled with private;
```

```
    ...
```

```
    procedure Adjust( S: in out Sor );
```

```
    function "="( S, Z: Sor ) return Boolean;
```

```
private
```

```
    ...
```

```
    type Sor is new Controlled with record Eleje, Vege: Mutato := null; end record;
```

```
end Sorok;
```

Mélyebb (custom) másolás

```
procedure Adjust ( S: in out Sor ) is
  Eddig_Kesz: Mutato;
begin
  if S.Eleje /= null then                -- ha nem üres
    S.Eleje := new Csucs'(S.Eleje.all);  -- első csúcsról másolat
    Eddig_Kesz := S.Eleje;              -- az első csúcs kész
    while Eddig_Kesz.Kovetkezo /= null loop -- amíg van következő csúcs
      Eddig_Kesz.Kovetkezo := new Csucs'(Eddig_Kesz.Kovetkezo.all); -- lemásoljuk
      Eddig_Kesz := Eddig_Kesz.Kovetkezo;
    end loop;                            -- Eddig_Kesz után nincs további csúcs
    S.Vege := Eddig_Kesz;
  end if;
end Adjust;
```

Mélyebb (custom) egyenlőségvizsgálat

```
function "="( S, Z: Sor ) return Boolean is
    P, Q: Mutato;
begin
    P := S.Eleje;   Q := Z.Eleje;
    if P = Q then return true;    -- shortcut reflexivitásra
    else   while P /= null loop
            if Q = null or else P.Adat /= Q.Adat then return false;
            else P := P.Kovetkezo;   Q := Q.Kovetkezo;
            end if;
        end loop;
        return Q = null;
    end if;
end "=";
```

Alias mutatók

- Olyan objektumra mutat, amelyet nem a heap-en hoztunk létre (Ada 95)
- Az **aliased** és az **all** kulcsszavak kellenek
- Az **Access** attribútum „cím” lekérdezésére való, minden típushoz használható

```
type P is access all Integer;
```

```
N: aliased Integer := 1;
```

```
X: P := N ' Access;
```

- Az X.all egy alias lesz az N-hez

Ilyesmi a C++ nyelvben

```
void f()  
{  
    int n = 1;  
    int& r = n;  
    int* p = &n;           // ez hasonlít a leginkább  
}
```

Borzasztó veszélyes

```
int *f ( int p ) {  
    int n = p;  
    return &n;  
}
```

```
int main() {  
    int i = *f(3); // illegális memóriahivatkozás  
}
```

Az Ada szigorúbb, biztonságosabb

```
procedure A is
  type P is access all Integer;
  procedure B ( X: out P ) is
    N: aliased Integer := 1;
  begin
    X := N'Access;
  end B;
  X: P;
begin
  B(X);
end;
```



Fordítási hiba

Élettartam ellenőrzése

`X := N'Access;`

- Csak akkor helyes, ha az N objektum legalább ugyanannyi ideig fennmarad, mint az X típusa.
- Az 'Access elérhetőségi ellenőrzést is végez, hogy a mutatott objektum élettartama legalább akkora-e, mint a mutató típusának hatásköre.
- Megkerülni az ellenőrzést: 'Unchecked_Access

'Unchecked_Access

- Vannak szituációk, ahol kényelmetlen az ellenőrzés
- Kiskapu: az élettartam ellenőrzése ellen
- Nagyon veszélyes, mert ilyenkor a programozó felelőssége az, hogy ne legyen baj
- A fordító elfogadja az előző programokat, ha az Unchecked_Access attribútumot használjuk
- A program működése azonban definiálatlan

Alprogramra mutató típus

```
procedure A ( N: in Integer ) is ... begin ... end A;  
type P is access procedure ( N: in Integer );  
X: P := A'Access;
```

```
... X.all(3); ... X(3); ...
```



Ada 95

```
void a ( int n ) { ... }  
void (*x) (int) = a;  
... (*x)(3); ... x(3); ...
```

Paraméterek

- Paraméterezhető dolgok
 - Alprogramok
 - Típusok (diszkrimináns, indexhatár)
 - Sablonok
- Formális paraméter - aktuális paraméter
- Paraméterek megfeleltetése
- Alprogram: paraméterátadás

Alprogram-paraméterek

- Értékek és objektumok
- Bemenő és kimenő paraméter
 - Az információáramlás iránya
 - Bemenő paraméter: jobbérték
 - Kimenő paraméter: balérték
- Paraméterátadási módok
 - Technikai, nyelvimplementációs kérdés

Paraméterátadási módok

- Érték szerinti (C, Pascal)
- Cím szerinti (Pascal, C++)
- Eredmény szerinti (Ada)
- Érték/eredmény szerinti (Algol W)
- Megosztás szerinti (Java, Eiffel, CLU)
- Igény szerinti (Haskell)
- Név szerinti (Scala, Algol 60, Simula 67)
- Szövegszerű helyettesítés (makróknál)

Érték szerinti paraméterátadás

- Call-by-value
- Nagyon elterjedt (C, C++, Pascal, Ada)
- Bemenő szemantikájú
- A formális paraméter az alprogram lokális változója
- Híváskor az aktuális értéke bemásolódik a formálisba
- A vermen készül egy másolat az aktuálisról
- Kilépéskor a formális megszűnik

Érték szerint a C++ nyelvben

```
int Inko ( int a, int b )  
{  
    while (a != b)  
        if (a>b) a-=b; else b-=a;  
}  
int f ()  
{  
    int x = 10, y = 45;  
    return Inko(x,y) + Inko(4,8);  
}
```

x és y nem
változik

hívható bal- és
jobbértékkal


Cím szerinti paraméterátadás

- Call-by-reference
- A Fortran óta széles körben használt
- Kimenő szemantikájú
 - pontosabban be- és kimenő
 - csak balértékkel hívható
- A formális paraméter egy címet jelent
- A híváskor az aktuális paraméter címe adódik át
- A formális és az aktuális paraméter ugyanazt az objektumot jelentik (alias)

Cím szerint a C++ nyelvben

```
void swap ( int& a, int& b )  
{  
    int c = a;  
    a = b;  b = c;  
}
```

```
void f ()  
{  
    int x = 10, y = 45;  
    swap ( x, y );  
    // értelmetlen: swap ( 3, 4 );  
}
```



x és y
megváltozik

Pascal-szerű nyelvek

`int Inko (int a, int b) ...`

`void swap (int& a, int& b) ...`

`function Inko (a, b: integer) : integer ...`

`procedure swap (var a, b: integer) ...`

Érték/eredmény szerinti paraméterátadás

- Call-by-value/result
- Algol-W, Ada (kis különbséggel)
- Kimenő szemantika
 - pontosabban be- és kimenő
 - csak balértékkel hívható
- A formális paraméter az alprogram lokális változója
- Híváskor az aktuális értéke bemásolódik a formálisba
- A vermen készül egy másolat az aktuálisról
- Kilépéskor a formális értéke visszamásolódik az aktuálisba

Érték/eredmény szerint az Adában

```
procedure Swap ( A, B: in out Integer ) is
    C: Integer := A;
begin
    A := B; B := C;
end Swap;
procedure P is
    X: Integer := 10;
    Y: Integer := 45;
begin
    Swap ( X, Y );
end P;
```



X és Y
megváltozik

-- a Swap(3,4) értelmetlen

Eredmény szerinti paraméterátadás

- Call-by-result
- Ada
- Kimenő szemantika
 - csak balértékkal hívható
- A formális paraméter az alprogram lokális változója
- Kilépéskor a formális értéke bemásolódik az aktuálisba
 - Híváskor az aktuális értéke nem másolódik be a formálisba

Eredmény szerint az Adában

```
procedure Betűt_Olvas ( C: out Character ) is
begin
    Ada.Text_IO.Get ( C );
    if C < 'A' or else C > 'Z' then
        raise Constraint_Error;
    end if;
end;
C: Character;
...
    Betűt_Olvas(C);
```

- C nem volt inicializálva
- C értéket kapott

Adatmozgatással járó paraméterátadás

- Data transfer
- Ilyen az érték, az eredmény és az érték/eredmény szerinti
 - Nem ilyen a cím szerinti
- Az aktuális paraméterről másolat készül
 - Független az aktuális a formálistól
 - Ha valamelyik változik, a másik nem
 - Könnyebben követhető

Cím versus érték/eredmény szerinti

- Mindkettő (be- és) kimenő szemantikájú
- Az utóbbi adatmozgatásos
 - nagy adat esetén a cím szerinti hatékonyabb
 - kis adat esetén az érték/eredmény szerinti hatékonyabb lehet (ha sok a hivatkozás)
- „Ugyanaz” a program másként működhet
- Az érték/eredmény szerinti általában jobban érthető viselkedést produkál
 - Mindkettőnél adható csúful viselkedő példa

Példa: egy Ada kódrészlet

```
N: Integer := 4;  
procedure P ( X: in out Integer ) is  
begin  
    X := X + 1;  
    X := X + N;  
end P;  
...  
P ( N );
```

Cím szerinti: 10

Érték/eredmény: 9

Paraméterátadási módok az Adában

- Bemenő (in) módú paraméterek:
érték szerint vagy cím szerint
- Kimenő (out), illetve be- és kimenő (in out) módú paraméterek:
 - Bizonyos típusoknál (érték/)eredmény szerint
(pl. skalár típusok, rekordok, mutatók)
pass by copy
 - Más típusoknál cím szerint
(pl. jelölt típusok, taszkok, védett egységek)
 - Egyes típusoknál implementációfüggő
(pl. tömbök)

Megosztás szerinti (pass-by-sharing)

- objektumelvű nyelvek, pl. Java

```
void method( int primitive, StringBuilder reference ){  
    ++primitive;    reference.append(primitive);    reference=null;  
}
```

```
int n = 0;    StringBuilder sb = new StringBuilder();  
method( n, sb );  
System.out.println( n );    System.out.println( sb );  
//            0                "1"
```

Igény szerinti (pass-by-need)

- lusta kiértékelésű funkcionális nyelvek, pl. Haskell

`null [] = true`

`f x = x + x`

`null _ = false`

`null [1..]`

`f (1+1)`

`let x = 1+1 in x + x`

Név szerinti (pass-by-name)

- Scala, régebbi nyelvek (Algol, Simula)
- Lustaság kifejezésére

```
scala> def f ( x: => Int ) = { println("inside"); x + x }
```

```
f: (x: => Int)Int
```

```
scala> f { println("parameter"); 1+1 }
```

```
inside
```

```
parameter
```

```
parameter
```

```
res0: Int = 4
```

access alprogram-paraméter

procedure A (X: access Integer) is ...

- in módnak felel meg
- Az aktuális paraméter valamilyen Integer-re mutató típusba kell tartozzon
- A formális paraméter sosem null
 - A futtató rendszer ellenőrzi, hogy az aktuális ne legyen az
 - A programozónak nem kell ezzel foglalkozni
- Korlátozott a paraméter
(nincs rá értékadás és egyenlőségvizsgálat)

Alprogrammal való paraméterezés

- Sablon

generic

with function F (A: Float) return Float;

function Integrál (Alsó, Felső, Lépés: Float) return Float;

- Alprogramra mutató típus

type Fun is access function (A: Float) return Float;

function Integrál (F: Fun;

Alsó, Felső, Lépés: Float) return Float;

- Alprogramra mutató paraméter

function Integrál (F: access function (A: Float) return Float;

Alsó, Felső, Lépés: Float) return Float;

Closure

```
procedure Closures is
  procedure Repeat( N: in Positive; Proc: access procedure( T: in Integer ) ) is
  begin
    for I in 1..N loop Proc.all( I ); end loop;
  end Repeat;
begin
  declare
    V: Integer := 0;
    procedure Increase( N: in Integer ) is begin V := V + N; end Increase;
  begin
    Repeat( 10, Increase'Access );
  end;
end;
```

Szövegszerű helyettesítés

- A legelső programozási nyelvek assembly-k voltak, abban makrókat lehetett írni
- A makrók még mindig fontosak: pl. C, C++
- A makró törzsében a formális paraméter helyére beíródik az aktuális paraméter szövege
- Egyszerű működési elv, de szemantikusan bonyolult: könnyű hibát véteni
- A makró törzse beíródik a hívás helyére

C makró

```
#define square(x) x*x  
int x = square(3+7);
```

3+7*3+7

```
#define max(a,b) ((a)>(b) ? (a) : (b))  
int x = 5, y = max(++x, 3);
```

x ← 7

- Egyéb furcsaságokat okoz az, hogy a makró törzse a hívás helyére behelyettesítődik
- Nem lehet swap makrót írni

Inline alprogram

- A makró „hatékonyabb”, mint az alprogram
 - kisebb futási idő
 - nagyobb programkód
- Ugyanezt tudja a kifejtett (inline) alprogram
 - rendes paraéterátadás
 - szemantikus biztonság

```
function Max ( A, B: Integer ) return Integer...  
pragma Inline(Max);
```

Javaslat a fordítónak

```
inline int max( int a, int b ) { return a > b ? a : b; }
```

Optimalizálás

- Egy jó fordító esetleg jobban optimalizál, mint a programozó
- Inline kifejtés
- Tár vagy végrehajtási idő?
- Ada: Optimize fordítóvezérlő direktíva

```
pragma Optimize(Time);    -- Space, Off
```