

# 8. előadás

Dinamikus memóriakezelés. Mutatók.  
Láncolt adatszerkezetek.

# 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 szemétgyűjtés
- 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;
```

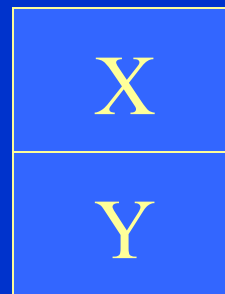
```
X := new Integer;
```

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

```
Y := X;
```

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

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



# Indirekció (3)

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

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

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

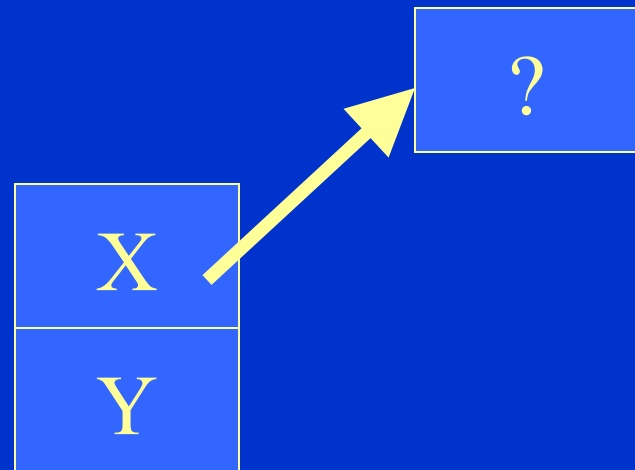
```
X := new Integer;
```

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

```
Y := X;
```

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

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



# Indirekció (4)

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

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

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

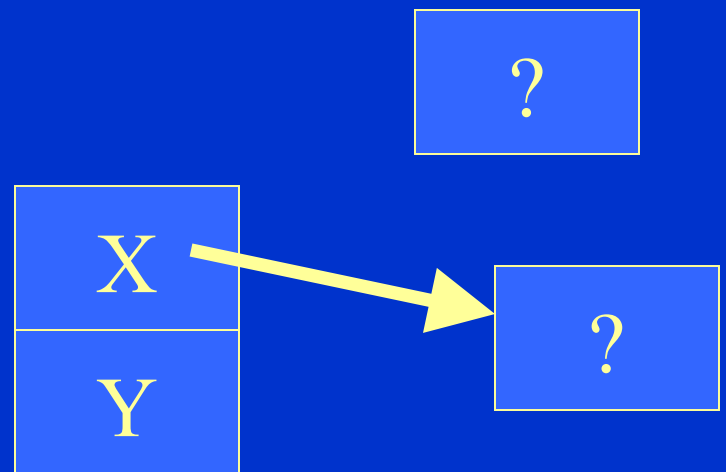
```
X := new Integer;
```

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

```
Y := X;
```

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

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



# Indirekció (5)

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

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

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

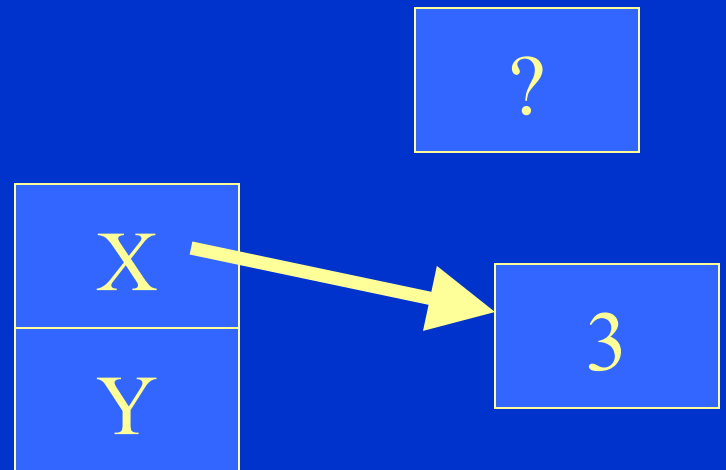
```
X := new Integer;
```

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

```
Y := X;
```

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

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





# Indirekció (6)

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

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

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

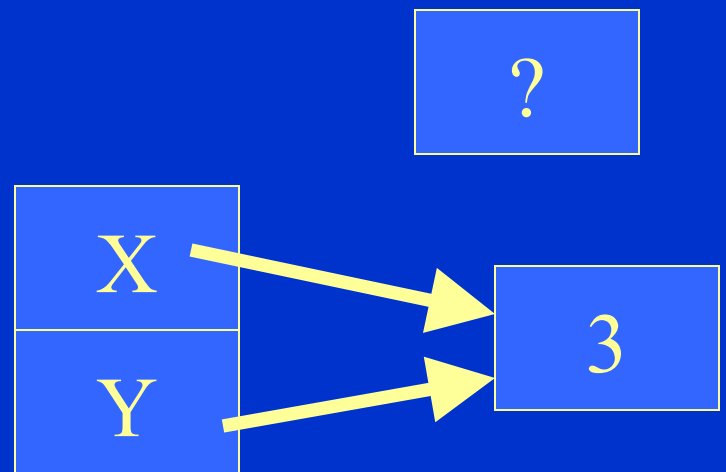
```
X := new Integer;
```

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

```
Y := X;
```

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

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



# Indirekció (7)

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

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

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

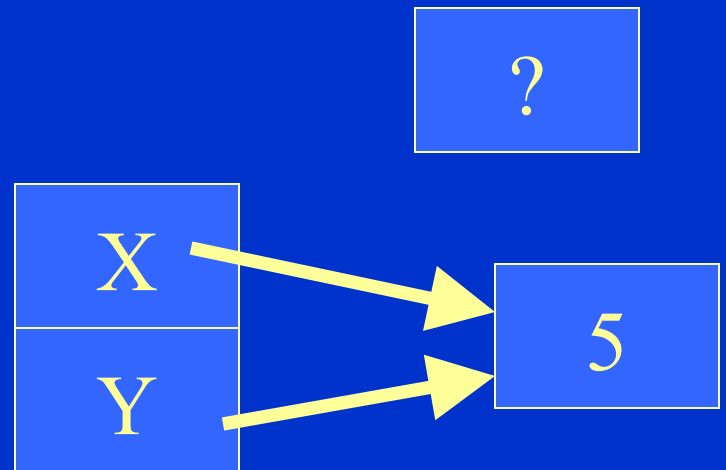
```
X := new Integer;
```

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

```
Y := X;
```

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

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



# Indirekció (8)

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

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

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

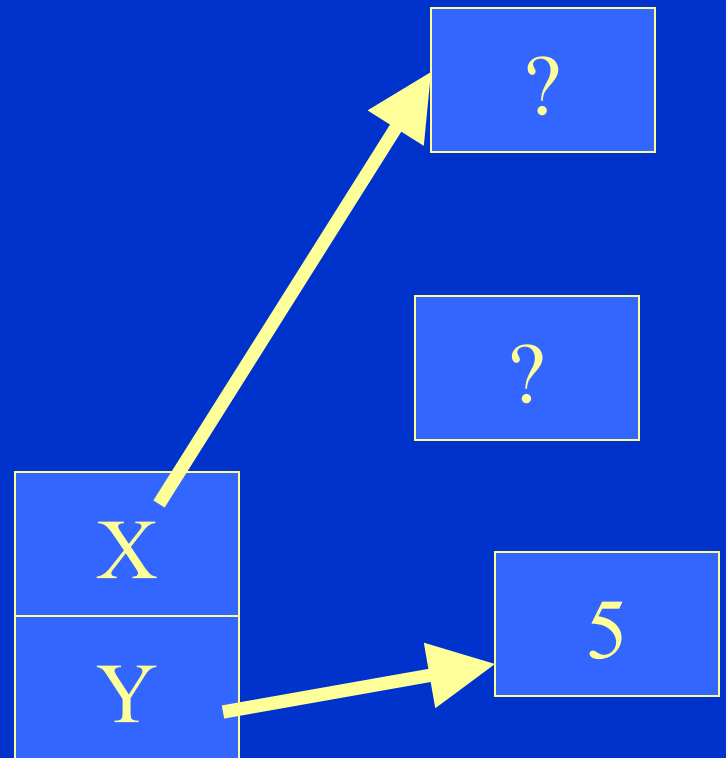
```
X := new Integer;
```

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

```
Y := X;
```

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

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



# Alias kialakulása

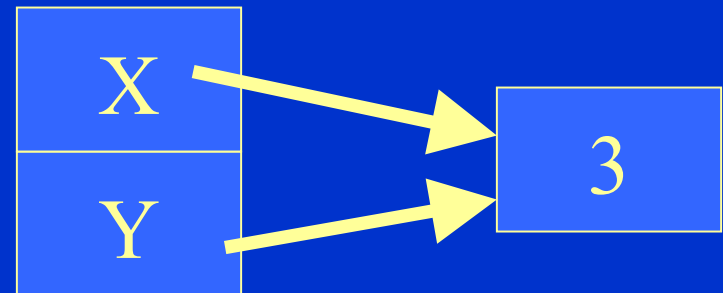
- Ha ugyanazt a változót többféleképpen is elértem
- Például cím szerinti paraméterátadásnál
- Például mutatók használatával

```
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 szemétyűjtés
- ▣ Használjunk automatikus változókat
  - Ott a hatókörhöz kapcsolódik az élettartam
  - C++: automatikus objektum destruktora



# Szemétgyűjtés

- ▣ 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 szemétgyűjtési 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
  - diszkriminánsos 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

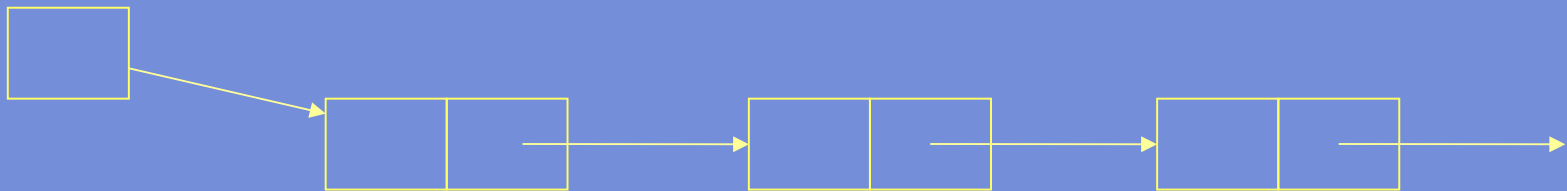
```
procedure A is
  type P_Verem is access Verem;
  function Létrehoz return P_Verem is
    N: Positive;
  begin
    Get(N);
    return new Verem(N);
  end Létrehoz;
  V: P_Verem := Létrehoz;
begin
  ...
end A;
```

ha vissza kell adni  
egy dinamikus  
méretű

objektumot

# 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

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

deklaráljuk a  
típust

# 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 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;
```

allokálás és  
inicializálá  
s 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 allokáltak
- 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);
```

# 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

$X := N' \text{Access};$

- 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



# 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); ...
```

# Alprogrammal való paraméterezés

▣ Sablon

**generic**

**with function F ( A: Float ) return Float;**

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

▣ Alprogramra mutató típus

**type F\_Mutató is access function ( A: Float ) return Float;**


**function Integrál ( F: F\_Mutató; Alsó, Felső, Lépés: Float )  
return Float;**

# Eltérő megvalósítás

- ▣ Az alprogramra mutató típus használatával csak könyvtári szinten megvalósított alprogramok adhatók át
- ▣ Sablon esetén nincs ilyen megkötés
  - Viheti magával a környezetét
  - Ez jobban hasonlít a lezárt (closure) átadására, lásd pl. Modula-3

# Sablon a sablonban

```
generic
  type Elem is private;
package Sorok is
  type Sor is new Limited_Controlled with private;
  ...
  generic
    with procedure Feladat ( E: in Elem );
  procedure Iterál ( S: in Sor );
private
  ...
end Sorok;
```



# Implementáció (Iterál)

```
package body Sorok is
    ...
    procedure Iterál ( S: in Sor ) is
        P: Mutató := S.Eleje;
    begin
        while P /= null loop
            Feladat( P.Adatt );
            P := P.Következő;
        end loop;
    end Iterál;
end Sorok;
```

# Összegzés

Controlled leszámazottja  
csak könyvtári szintű  
lehet

```
with Sorok;  
package Int_Sorok is new Sorok(Integer);
```

```
with Int_Sorok; use Int_Sorok;  
function Összeg ( S: Sor ) return Integer is  
  N: Integer := 0;  
  procedure Hozzáad ( E: in Integer ) is begin N := N + E; end;  
  procedure Összegez is new Iterál(Hozzáad);  
begin  
  Összegez(S);  
  return N;  
end Összeg;
```



# Appendix



```
type Tömb is array(1 .. 10) of Integer;
```

```
X: aliased Tömb;
```

```
X'Access          --OK
```

```
X(1)'Access       -- hibás kifejezés
```

```
type A_Tömb is array(1..10) of aliased Integer;
```

```
Y: A_Tömb;
```

```
Y(1)'Access       --OK
```

```
Y'Access          -- hibás kifejezés
```

```
Z: aliased A_Tömb;
```

```
Z'Access és Z(1)'Access is helyes
```



```
type Rekord is record
```

```
    A: aliased Integer;
```

```
    B: Integer;
```

```
end record;
```

```
R1: aliased Rekord;
```

```
R2: Rekord;
```

```
R1.A'Access    R2.A'Access    -- helyes
```

```
R1.B'Access    R2.B'Access    -- helytelen
```

```
R1'Access      -- helyes
```

```
R2'Access      -- helytelen
```

# Konverzió két mutató típusra

```
type P is access Integer;  
type Q is access all Integer;  
X: P := new Integer'(3);  
Y: Q := Q(X);
```

```
type R is access all Integer;  
Z: R := Y.all'Access;
```

▫ Az 'Access attribútum tetszőleges típushoz jó

```

procedure Program is
  type P is access all Integer;
  X: P;
  procedure Értékadás is
    I: aliased Integer := 0;
  begin
    X := I'Access;    -- hibás értékadás, mert I az eljárás
                     -- után megszűnik, míg az X típusa nem
  end Értékadás;
begin
  Értékadás;
  X.all := X.all + 1;  -- itt lenne a nagy gond
end Program;

```

```

procedure Program is
  type P is access all Integer;
  X: P;
  procedure Értékadás is
    type Q is access Integer;
    Y: Q := new Integer'(0);
  begin
    X := Y.all'Access;      -- hibás értékadás
  end Értékadás;
begin
  Értékadás;
  X.all := X.all + 1;      -- itt lenne a nagy gond
end Program;

```

# 'Unchecked\_Access

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

```

procedure A is
  type P is access all Integer;
  procedure B is
    N: aliased Integer;
    type Q is access all Dátum;
    X: P := N'Access;           -- ez hibás
    Y: Q := N'Access;         -- ez jó
    Z: P := N'Unchecked_Access; -- ez is!
  begin
    ...
  end B;
begin
  ...
end A;

```

```

procedure A is
  type P is access all Integer;
  W: P;
  procedure B is
    N: aliased Integer;
    type Q is access all Dátum;
    X: P := N'Access;           -- ez hibás
    Y: Q := N'Access;         -- ez jó
    Z: P := N'Unchecked_Access; -- ez is!
  begin
    W := Z;
  end B;
begin
  B;                          -- W.all definiálatlan
end A;

```

# constant

- „Nem változhat az értéke”
- Szerepelhet:
  - Mutató deklarációjában
  - Mutató típus definíciójában



# Konstans mutató

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

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

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

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

```
X := null;
```

# Konstansra is mutatható típus

```
type P is access constant Integer;  
X: P;
```

- ▣ Az *X*-en keresztül nem módosítható a mutatott objektum
- ▣ Maga az *X* módosítható (másik objektumra állítható)
  - hacsak nem tesszük konstanssá az *X*-et is...
- ▣ Az *X* mutathat konstans objektumra **is**

# constant az all helyett

- Nem csak dinamikusan létrehozott objektumra mutathat

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

```
N: aliased Integer;
```

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

# constant az all helyett

- ▣ Nem csak dinamikusan létrehozott objektumra mutathat

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

```
N: aliased constant Integer :=3;
```

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

# constant az all helyett

- Nem csak dinamikusan létrehozott objektumra mutathat

DE:

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

```
N: aliased constant Integer :=3;
```

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

## „konstans” hozzáférés

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

```
N: aliased constant Integer := 42;
```

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

```
X.all := 33;      -- hibát okozna, helytelen
```

# „konstans” hozzáférés

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

```
N: aliased Integer := 42;
```

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

```
X.all := 33;      -- helytelen
```

```
type PKonst_int is access constant Integer;
type PInt is access all Integer;
  P: PKonst_int;
  Q: PInt;
  I: aliased Integer;
  K: aliased constant Integer := 20;
-- a következő értékadások helyesek:
  P := K'Access; P := I'Access;
  Q := I'Access; Q.all := 29;
  P := new Integer'(0);
-- a következő értékadások pedig helytelenek:
  P.all := 1;
  Q := K'Access;
  P := new Integer;
```



# A mutató is lehet konstans

```
type PKonst_Int is access constant Integer;
type PInt is access all Integer;
I: aliased Integer;
K: aliased constant Integer := 20;
P: PKonst_Int;
Q: PInt;
KP: constant PKonst_int := K'Access;
KQ: constant PInt := I'Access;
-- a következő értékadás helyes:
KQ.all := 1;
-- a következő értékadások hibásak:
KP := new Integer'(100);
KQ := new Integer'(200);
KP.all := 10;
```

# Példák:

```
type Dátum is record
```

```
    Év, Hónap, Nap: Integer;
```

```
end record;
```

```
type P1 is access Dátum;
```

```
type P is access all Dátum;
```

```
D1, D2: P;
```

```
U: aliased Dátum;
```

```
D1 := U'Access;
```

```
D1.all := (1997,2,2); -- a D1.Nap, illetve az U.Nap ugyanaz
```

```
type P is access constant Dátum; -- read-only elérést ad
```

```
D: P; -- most ez konstans lesz abban az értelemben, hogy:
```

```
D := U'Access; -- megengedett,
```

```
U.Nap := 3; -- ez is,
```

```
D.Nap := 2; -- ez hibás, de a következő nem:
```

```
put(D.Nap);
```

□ Megengedett viszont az is, hogy

```
D := new Dátum'(1999,2,2);
```

□ Ha az U-t akarjuk konstanssá tenni, akkor azt kell írni, hogy:

```
U: aliased constant Dátum := (...);
```

# 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)