# Föreläsning 8

- Dynamiska datastrukturer

  Länkade strukturer utan "pekare" (forts)

  Minnets struktur i en von Neumann-dator

  Pekare och dynamisk minnesallokering

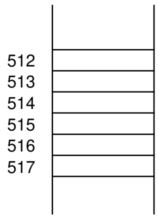
  Länkade strukturer med pekare

  Destruktiva operationer på datastrukturer



# Datastrukturer

## Minnet i en von Neumann-dator

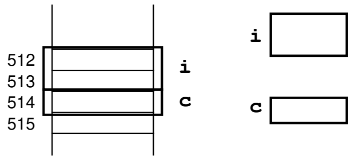



- Minnet linkar en ARRAY!

  **VAR** *minnet* **: ARRAY [0..Max] OF BYTE**

- Varje byte har en egen *adress*.
- Dataobjekt i Modula-2 lagras i en sammanhängande följd av bytes med början på en bestämd adress.

```
VAR i:INTEGER;
    c:CHAR;
```




- Kompilatorn väljer adresser för alla variabler i programmet.



# Dynamiska datastrukturer

## Pekare

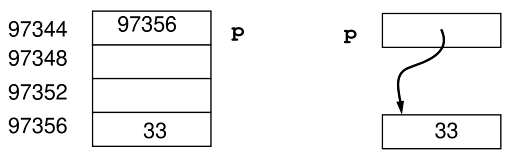
### Pekartyper

```
Type = ... | PointerType | ...
PointerType = POINTER TO Type
```

### Exempel

```
VAR p:POINTER TO INTEGER;
```

- Pekare är variabler som innehåller minnesadresser.




**p** pekar på en plats där heltalet 33 är lagrat.

- Hur kommer man åt det en pekare pekar på?

```
WriteInt(p^,1)  (* skriver ut 33 *)
p^:=42;         (* byter ut 33 mot 42 *)
```

Dereferering av pekare, motsvarar indexering i arrayer.

**p^ =** *minnet* **[p]**



# Dynamiska datastrukturer

- Var i minnet kan pekare peka?

```
p:=97344; (* ??? fungerar inte *)
p^:=42;
```

## Dynamisk minnesallokering

- Proceduren NEW ger adressen till en oanvänd plats i minnet.

  **NEW(p);**

- Platsen innehåller ett skäpvärde tills man lägger dit något annat

  **p^:=42;**

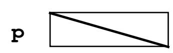
- Platsen i minnet är reserverad (=allokerad) tills man explicit lämnar tillbaka det med proceduren DISPOSE.

  **DISPOSE(p);**

## Ingenstans

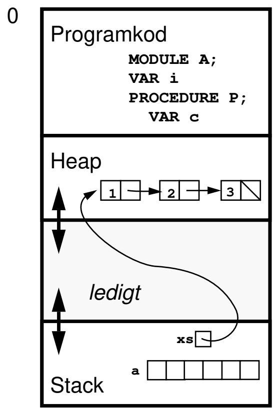
- Det finns ett speciellt pekarevärde som betecknar "ingen plats i minnet"

  **p:=NIL**



## Dynamiska datastrukturer

### Hur datorns minne används.




- Programkoden ändras inte under körningen
- Vanliga "statiska" variabler lagras i stacken. Stacken växer/krymper automatiskt vid anrop/återhopp från procedurer.
- Heapen används för dynamiska datastrukturer. Den växer/krymper vid anrop av NEW/DISPOSE. Det kan också finnas lediga hål i heapen.



## Dynamiska datastrukturer

### Dynamisk minnesallokering i Modula-2

#### Modulen Storage

```
DEFINITION MODULE Storage;

FROM SYSTEM IMPORT ADDRESS;

PROCEDURE ALLOCATE (VAR a : ADDRESS; size : CARDINAL);
(* Allocates an area of the given size 'size' and returns it's
 * address in 'a'. If no space is available, 'a' becomes 'NIL'.
*)

PROCEDURE DEALLOCATE (VAR a : ADDRESS; size : CARDINAL);
(* Frees the area of size 'size' starting at address 'a'. *)
(* Upon return 'a' is set 'NIL' *)

END Storage.
```

#### NEW & DISPOSE

**VAR p : POINTER TO** *typ*

- **NEW(p)** = **ALLOCATE(p,SIZE(***typ***))**
- **DISPOSE(p)** = **DEALLOCATE(p,SIZE(***typ***))**

- Genom att använda **NEW** & **DISPOSE** eliminerar man risken att skriva fel storlek.
- Kompilatorn översätter automatiskt **NEW** och **DISPOSE** till lämpliga anrop av **ALLOCATE** och **DEALLOCATE**. Dessa måste man själv importera!



## Dynamiska datastrukturer

### Faror med pekare

- Dessa fel upptäcks inte av kompilatorn!

#### "Dangling references"

- Användning av pekare som pekar på återlämnat minne.

```
NEW(p);
...
q:=p;
...
DISPOSE(p);
...
... q^ ...
```

- Leder till "Segmentation fault" eller slumpmässiga fel under körningen av programmet.

#### Skräp, minnesläckor

- Reserverat minne som inte kan användas till något

```
NEW(p);
...
NEW(q);
...
q:=p;
```



## Dynamiska datastrukturer

### Listor

```
data 'a list = nil | cons of 'a * 'a list
```

### Länkad lagringsstruktur för listor med pekare

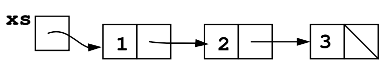
- Pekare kan vara **NIL**, som kan representera tomma listan (**nil**), så det räcker att ta hand om **cons**-fallet i typen:

```
TYPE List = POINTER TO ListNod;
     ListNod = RECORD
                 head: INTEGER;
                 tail: List
               END;
```

- Typer måste deklareras innan de kan användas, utom typer som står efter **POINTER TO**. Därför behövs både **List** och **ListNod**.

**VAR xs:List**

**xs = [1,2,3]**



# Dynamiska datastrukturer

## Modulen lists2

```
DEFINITION MODULE lists2;

(* Länkade lagringsstrukturer:
   listor, med pekare & dynamisk minnesallokering.
 *)

TYPE List; (* Typen är abstrakt (hemligt). *)

TYPE  Info = INTEGER; (* Typen på elementen i listan *)

(* === Konstrurerare === *)

PROCEDURE nil():List;  (* tomma listan *)

PROCEDURE cons(x:Info; xs:List):List;
(* cons lägger till x först i listan xs *)

(* === Predikat === *)

PROCEDURE null(l:List):BOOLEAN;
(* null testar om en lista är tom *)

(* Selektorer *)

PROCEDURE hd(l:List):Info;
(* hd ger första elementet i listan l *)

PROCEDURE tl(l:List):List;
(* tl ger resten av listan l utan första elementet *)

END lists2.
```



# Dynamiska datastrukturer

## Modulen lists2

```
IMPLEMENTATION MODULE lists2;
(* Länkade listor med pekare & dynamisk minnesallokering *)

FROM Storage IMPORT ALLOCATE,DEALLOCATE;

TYPE NodIndex = POINTER TO ListNod;
     List = NodIndex;

     ListNod = RECORD
                 head:Info; tail:List
               END;

PROCEDURE null(l:List):BOOLEAN;
(* null testar om en lista är tom *)
BEGIN
  RETURN l=NIL
END null;

PROCEDURE nil():List;
BEGIN
  RETURN NIL
END nil;

PROCEDURE cons(x:Info; xs:List):List;
(* cons lägger till x först i listan xs *)
VAR i:NodIndex;
BEGIN
  (* tag en ny nod *)
  NEW(i);
  (* fyll i den *)
  i^.head:=x;
  i^.tail:=xs;
  RETURN i
END cons;
```



# Dynamiska datastrukturer

## Modulen lists2 (forts)

```
PROCEDURE hd(l:List):Info;
(* hd ger första elementet i listan l *)
BEGIN
  RETURN l^.head
END hd;

PROCEDURE tl(l:List):List;
(* tl ger resten av listan l utan första elementet *)
BEGIN
  RETURN l^.tail
END tl;

PROCEDURE disposeList(l:List);
(* återlämnar allt minne som används av en lista *)
VAR i:List;
BEGIN
  WHILE NOT null(l) DO
    i:=l;
    l:=tl(l);
    DISPOSE(i)
  END
END disposeList;

END lists2.
```



# Dynamiska datastrukturer

## Destruktiva operationer på datastrukturer

### Statiska datatyper

- Exempel: vanlig tilldelning: **`x:=5`**, **`a[3]:=7`**
- Exempel: byt alla små bokstäver mot stora i en sträng.

  Icke-destruktiv version

  ...

  Destruktiv version

  ...
- Man kan kopiera statiska objekt med tilldelning.

### Dynamiska typer

- Exempel: insättning i en sorterad lista

  Destruktiv version

  ...
- Den gamla listan förstörs.
- Det behöver inte bli problem med skräp.
- Kopiering med tilldelning? Alias?

  ...