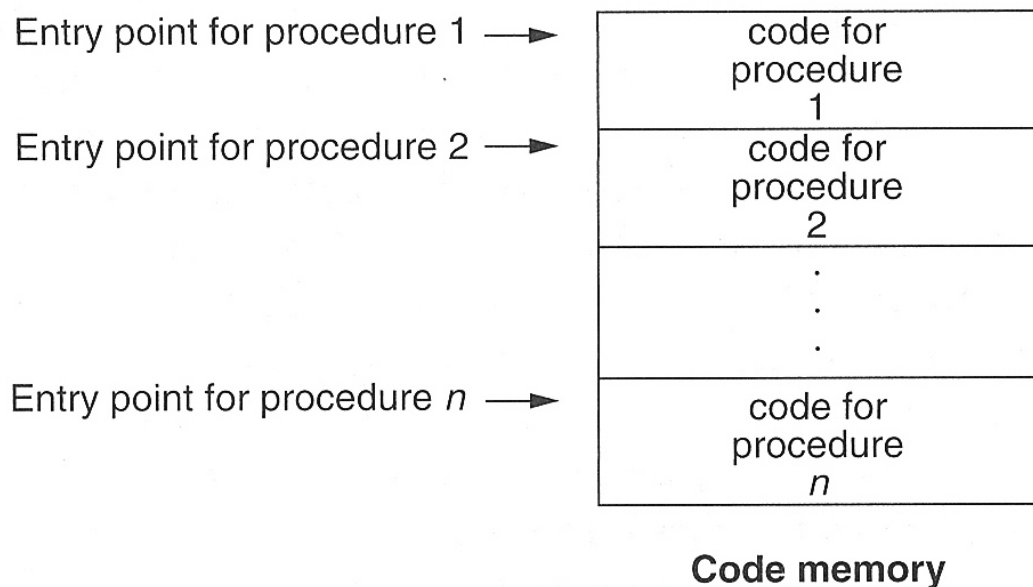


# Runtime-omgivelser Kap 7 - I

- Generelt
- Språk som bare trenger statiske omgivelser
- Språk som trenger stakk-orienterte omgivelser
  
- Språk som trenger mer generelle omgivelser
  - Vel så riktig å si at forskjellige slags begreper i et gitt språk krever at omgivelsen organiserer forskjellige deler av en programutførelse forskjelligt
- Parameteroverføring

# Den oversatte programkoden

- kan nesten alltid betraktes som statisk allokert
  - skal hverken flyttes eller forandres under utførelse
- Kompilatoren kjenner alle adresser til kodebiter



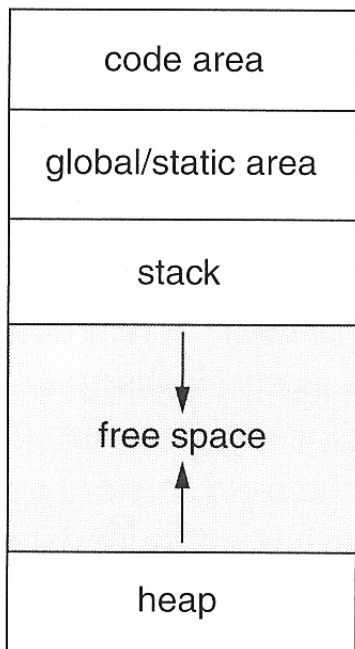
## Men husk

Koden blir ofte produsert som relokerbare kode, som får sin endelige plassering av linker/loader

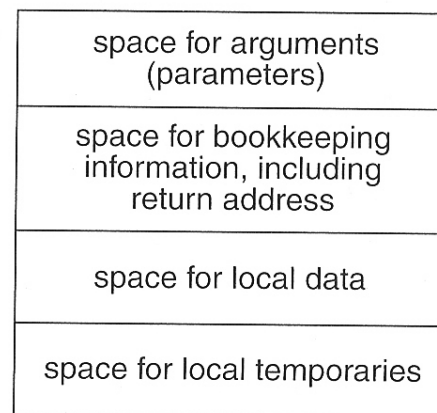
OS kan flytte rundt på kode, men det forstyrrer ikke kodens eget adresserum

# Lagerorganisering

- Typisk organisering under utførelse dersom et programmeringsspråk har alle slags data (statisk, stakk, dynamisk)



- Typisk organisering av data for et prosedyrekall (aktiveringsblokk)

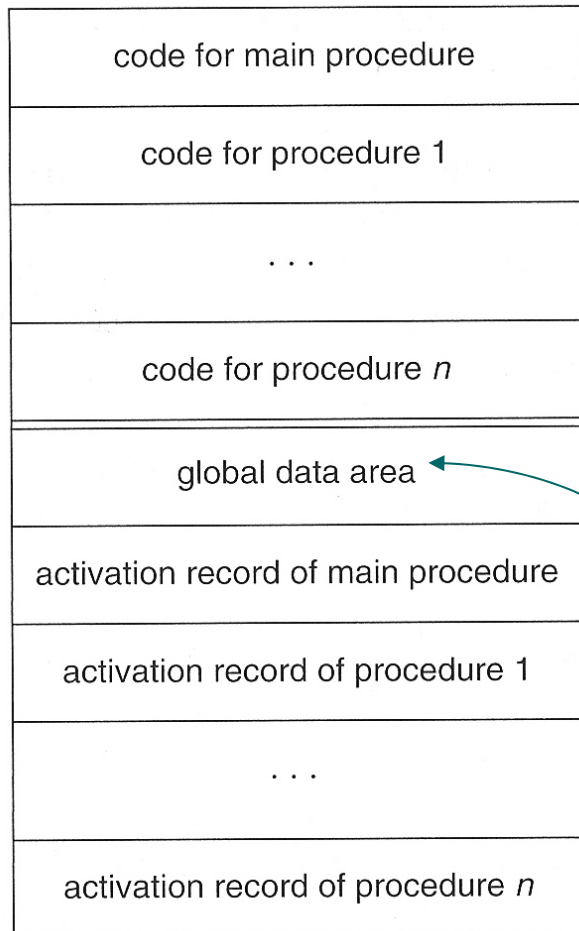


Det er gjerne ut fra plasseringen her man karakteriserer språk til være

- statisk organisert
- stakk-organisert
- heap/dynamisk organisert

# Full statisk organisering (eks. Fortran)

- Kompilatoren kan beregne hvor alt ligger
  - Utførbar kode
  - Variable
  - Alle slags hjelpedata



bl.a. alle slags større konstanter i programmet

# Et eksempel i Fortran

```

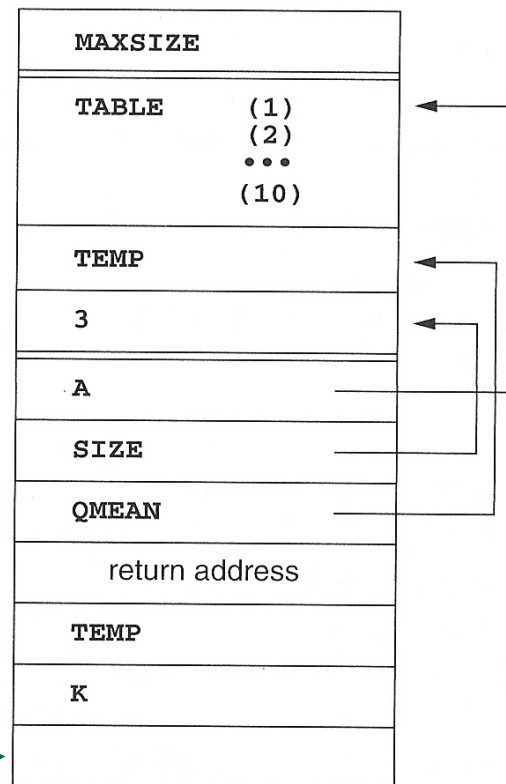
PROGRAM TEST
COMMON MAXSIZE
INTEGER MAXSIZE
REAL TABLE(10),TEMP
MAXSIZE = 10
READ *, TABLE(1),TABLE(2),TABLE(3)
CALL QUADMEAN(TABLE,3,TEMP)
PRINT *, TEMP
END

SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
COMMON MAXSIZE
INTEGER MAXSIZE,SIZE
REAL A(SIZE),QMEAN, TEMP
INTEGER K
TEMP = 0.0
IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
DO 10 K = 1,SIZE
    TEMP = TEMP + A(K)*A(K)
10 CONTINUE
99 QMEAN = SQRT(TEMP/SIZE)
RETURN
END
    
```

Global area

Activation record  
of main procedure

Activation record  
of procedure  
QUADMEAN



Plass til mellomresultater o.l.  
Kompilatoren kan beregne hvor mye som trengs

I Fortran overføres parametere som pekere til de aktuelle verdier/variable

# Et eksempel i C

```
#include <stdio.h>

int x,y;

int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf ("%d%d",&x,&y);
  printf ("%d\n",gcd(x,y));
  return 0;
}
```

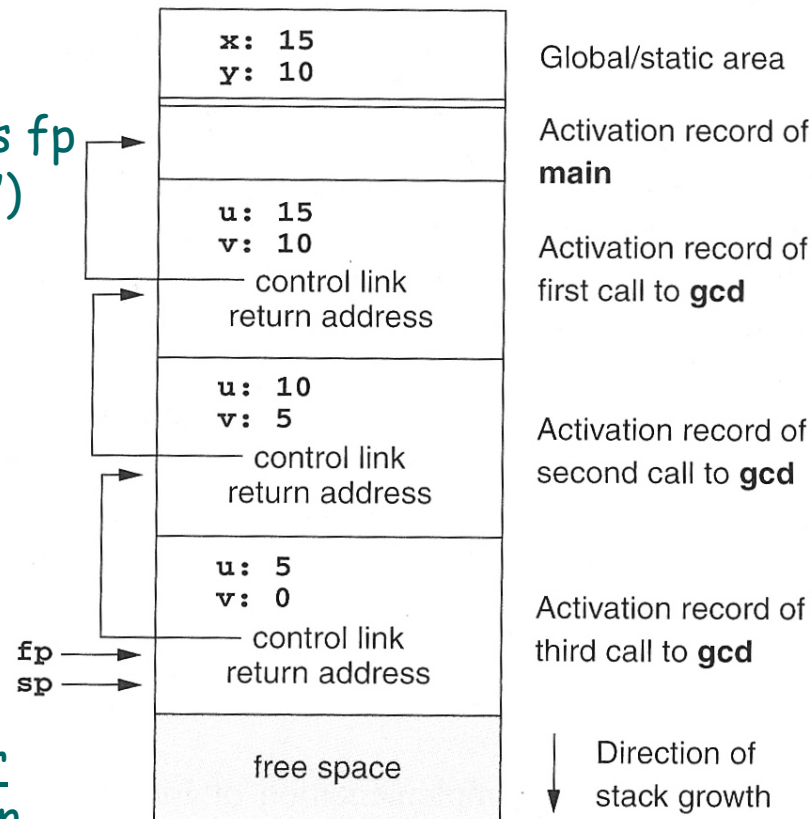
- Aktiverings-blokkene kan organiseres som en stakk. Krevs om man tillater rekursive kall

Return address  
Program-adressen man er kalt fra

control link  
Angir kallerens fp ('dynamisk link')

frame pointer  
Peker på fast sted i den aktuelle aktiveringsblokken

stack pointer  
Angir grensen mellom brukt og ledig lagerareal



# aktiveringstrær

```

int x = 2;

void g(int); /* prototype */

void f(int n)
{ static int x = 1;
  g(n);
  x--;
}

void g(int m)
{ int y = m-1;
  if (y > 0)
  { f(y);
    x--;
    g(y);
  }
}

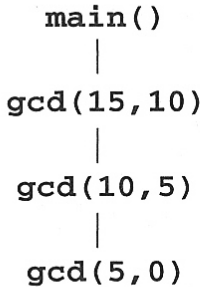
main()
{ g(x);
  return 0;
}

```

Blir én global variabel bare synlig fra f

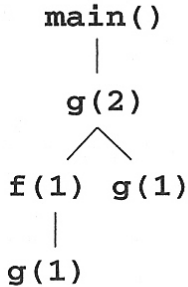
Relativ-adresser:  
 mOffset = +4  
 yOffset = -6  
 Aksess av y: -6(fp)

forrige program



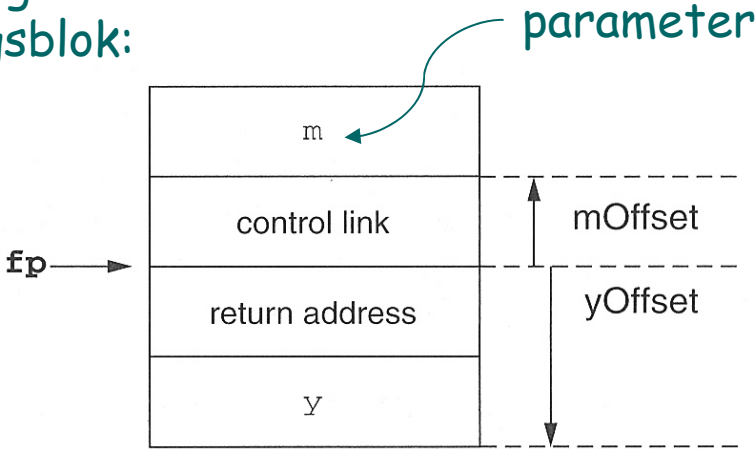
(a)

dette program

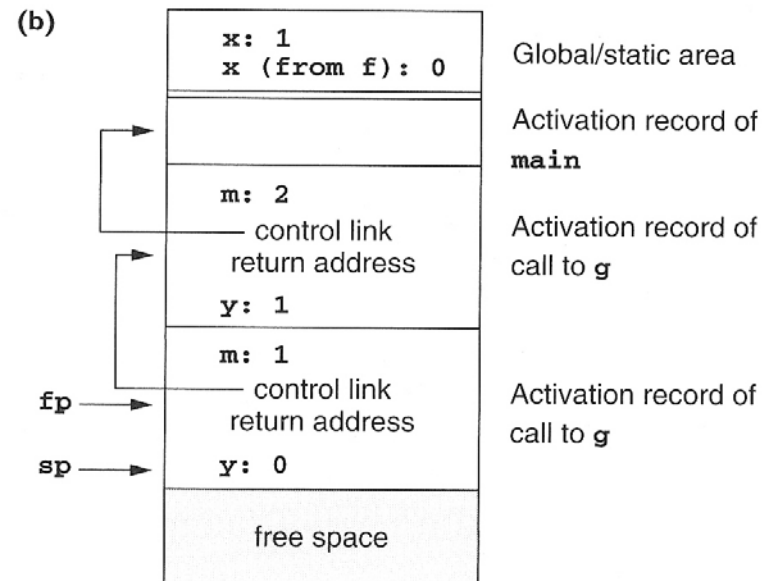
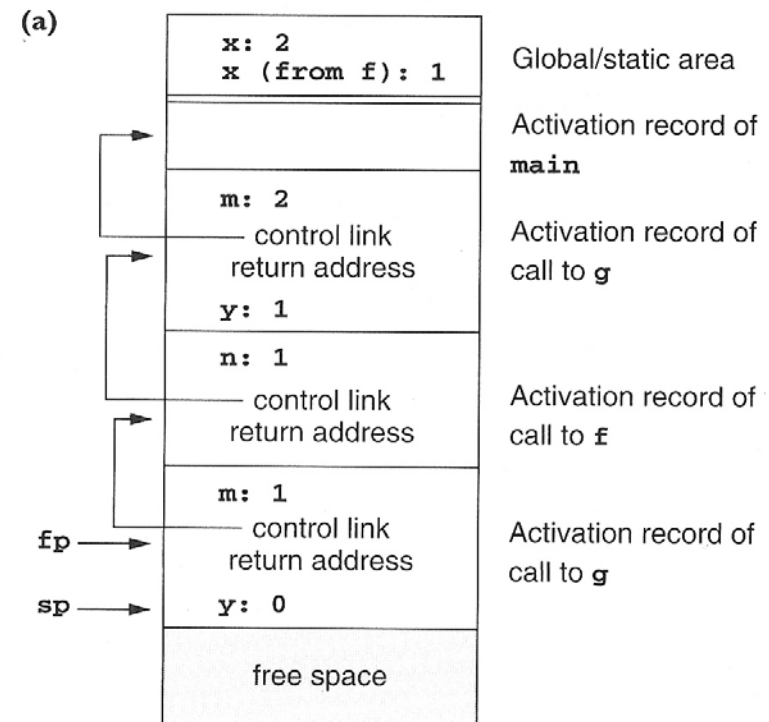


(b)

Layout av g sin aktiveringsblokk:



# Stadier i utførelse av programmet

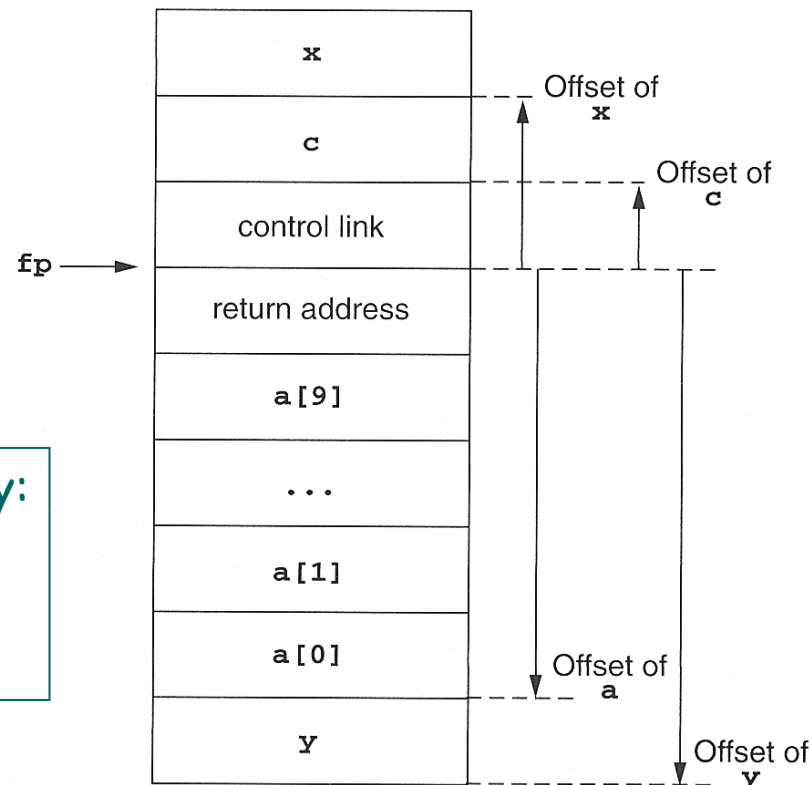




# Arrayer av kjent (statisk) lengde

```
void f(int x, char c)
{ int a[10];
  double y;
  ...
}
```

Layout av aktiveringsblokk:



## Relativ-adresser

Name	Offset
<code>x</code>	+5
<code>c</code>	+4
<code>a</code>	-24
<code>y</code>	-32

Aksess av `c` og `y`:

`c`: 4(`fp`)  
`y`: -32(`fp`)

`A[i]` beregnes som adressen

$(-24 + 2*i)(fp)$

kan ofte gjøres i én instruksjon


# Hvordan utføre et kall

## ▪ Ved prosedyrekall (entry)

1. Compute the arguments and store them in their correct positions in the new activation record of the procedure (pushing them in order onto the runtime stack will achieve this).
2. Store (push) the fp as the control link in the new activation record.
3. Change the fp so that it points to the beginning of the new activation record (if there is an sp, copying the sp into the fp at this point will achieve this).
4. Store the return address in the new activation record (if necessary).
5. Perform a jump to the code of the procedure to be called.

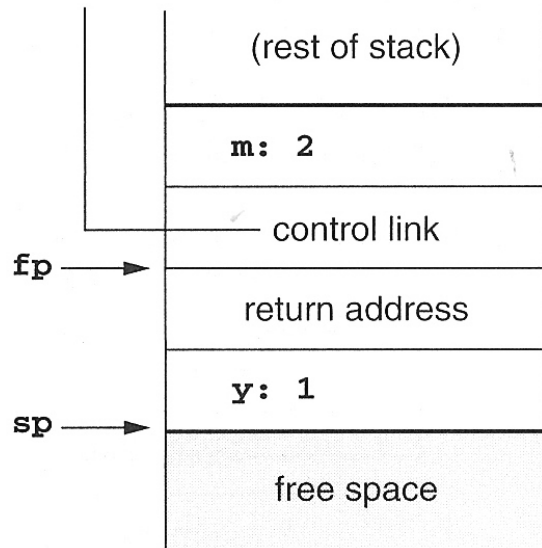
## ▪ Ved prosedyre-exit

1. Copy the fp to the sp.
2. Load the control link into the fp.
3. Perform a jump to the return address.
4. Change the sp to pop the arguments.

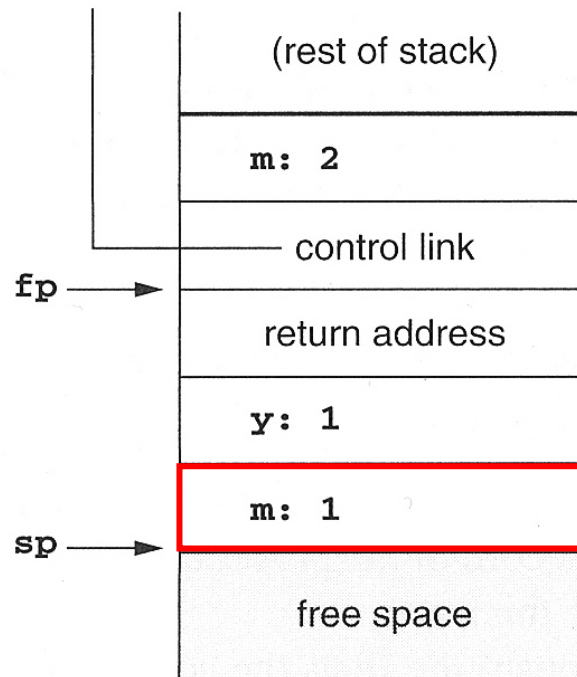


Denne setter av plass til lokale variable, ved å flytte sp  
Kan evt. også initialisere disse.  
(Kan tenkes på som push)

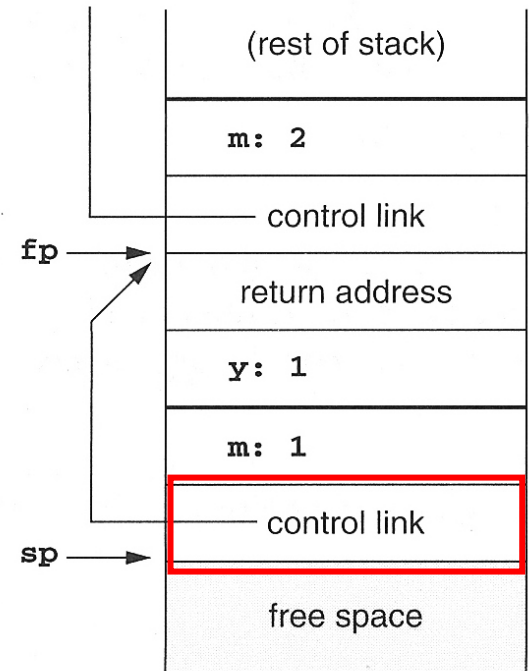
# Gjennomføring av et kall - I



før kall på *g*

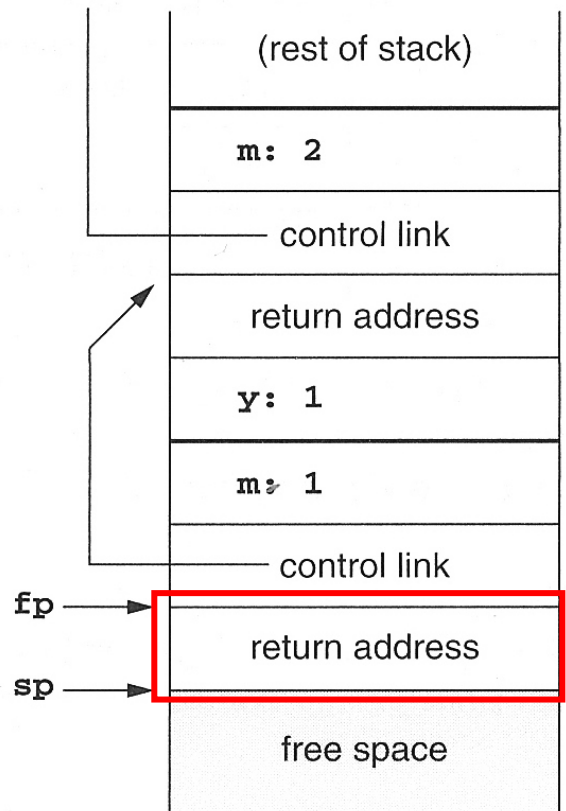


push parameter

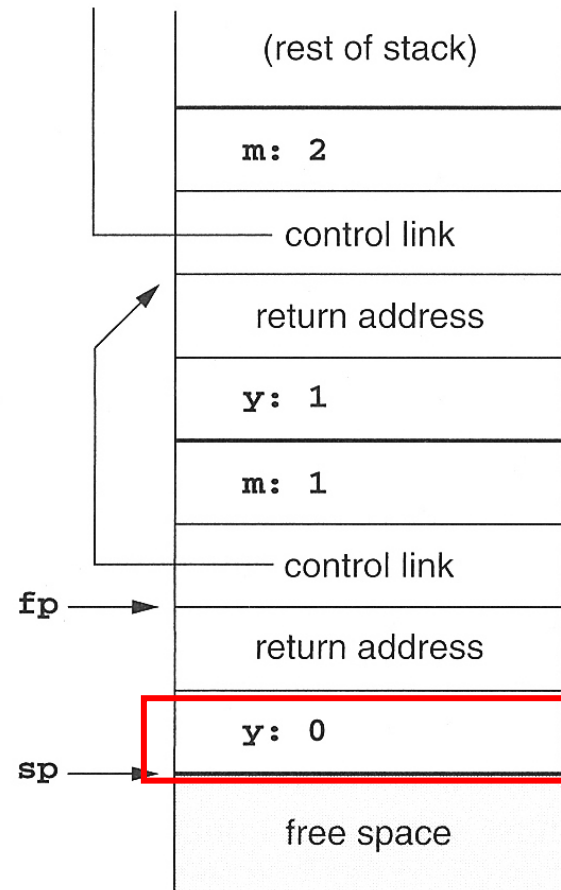


push *fp*

# Gjennomføring av et kall – II



1.  $fp = sp$
2. Push returadresse



alloker lokal var (y)

# Data av variabel lengde

```

type Int_Vector is
    array(INTEGER range <>) of INTEGER;

procedure Sum (low,high: INTEGER;
               A: Int_Vector) return INTEGER
is
    i: integer
begin
    ...
end Sum;
    
```

antar at A overføres ved full kopiering

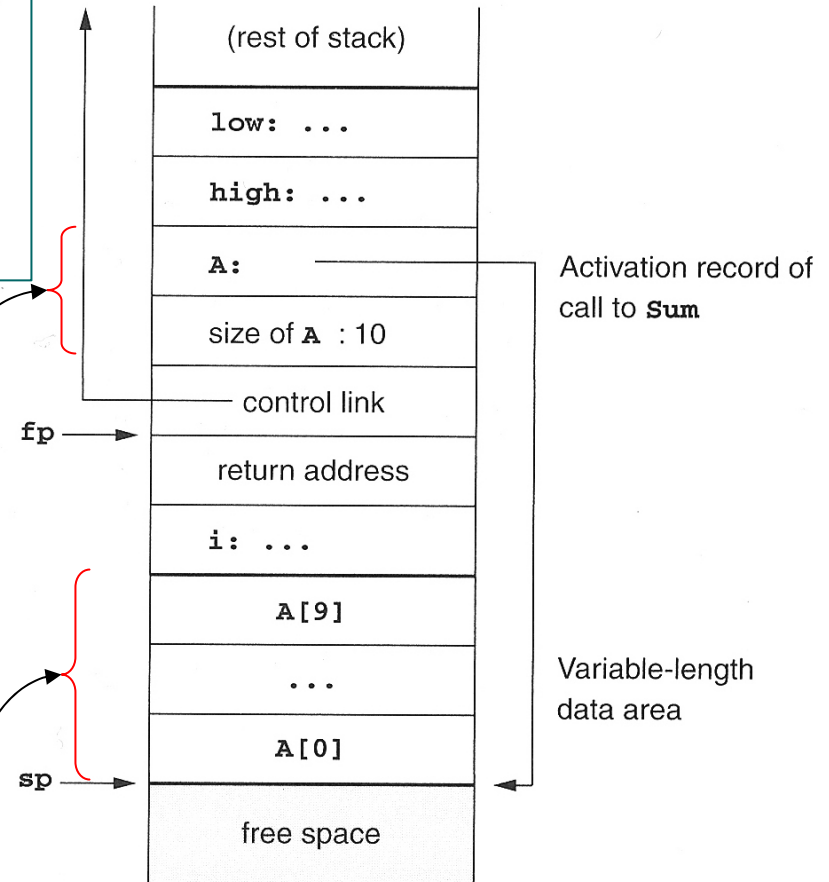
Fast lengde

A[i] beregnes som

$@6(fp) + 2*i$

I Java legges disse på heapen

- 'variabel' betyr at data ikke har samme størrelse ved hvert kall



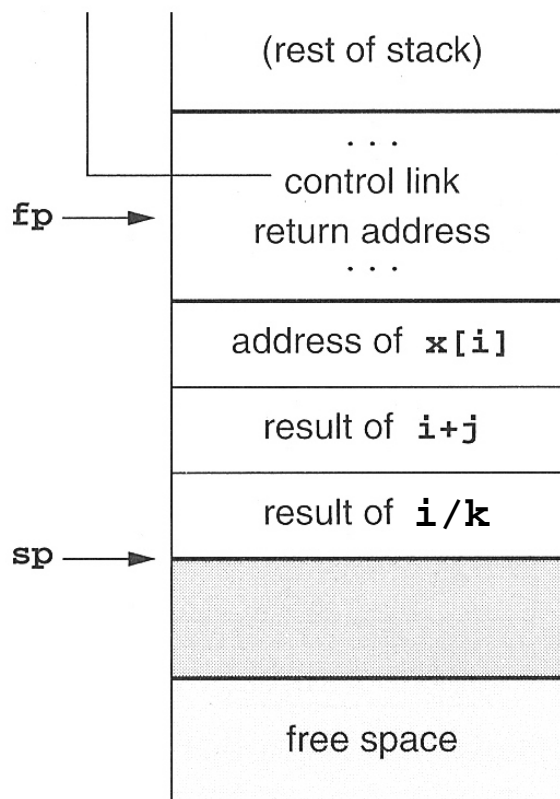
# Behandling av mellomresultater

$$x[i] = (i + j) * (i/k + f(j))$$

$\underbrace{\hspace{2em}}$   $\underbrace{\hspace{2em}}$   $\underbrace{\hspace{2em}}$   
adresse      verdi      verdi

Antar strikt beregning fra venstre mot høyre. Kallet  $f(j)$  kan forandre verdier.

Trenger ikke sette av fast maksimal plass til slike mellomresultater for hele blokkens levetid. I modsetning til hva man naturlig gjør i Fortran.



Activation record of procedure containing the expression

Stack of temporaries

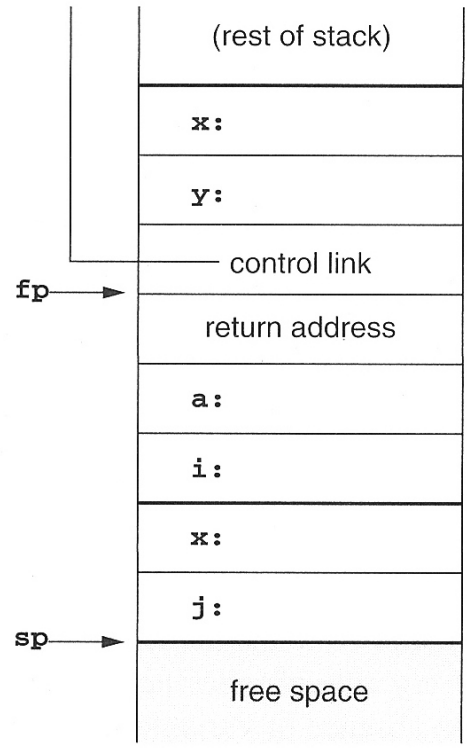
New activation record of call to  $f$  (about to be created)

# Mulig plass- allokering ved 'indre blokker'

```

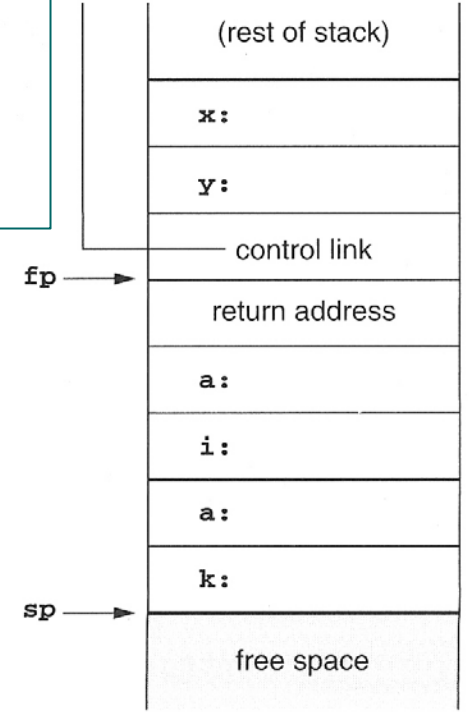
void p( int x, double y)
{ char a;
  int i;
  ...
A: { double x;
    int j;
    ...
  }
  ...
B: { char * a;
    int k;
    ...
  }
}

```



Activation record of call to P

Allocated area for block A



Activation record of call to P

Allocated area for block B

# Prosedyrer inne i prosedyrer

- Nestede prosedyrer
- Nested klasser (inner classes) kan behandles på samme måte.

```
program nonLocalRef;

procedure p;
var n: integer;

    procedure q;
    begin
        (* a reference to n is now
           non-local non-global *)
    end; (* q *)

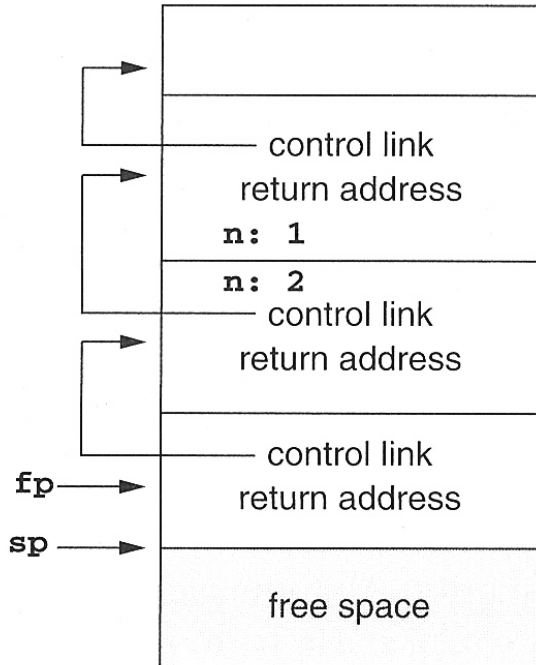
    procedure r(n: integer);
    begin
        q;
    end; (* r *)

begin (* p *)
    n := 1;
    r(2);
end; (* p *)

begin (* main *)
    p;
end.
```



# Et første forsøk



Hvordan kan vi aksessere 'n' i 'p' ?

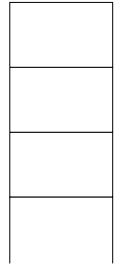
Activation record of main program

Activation record of call to **p**

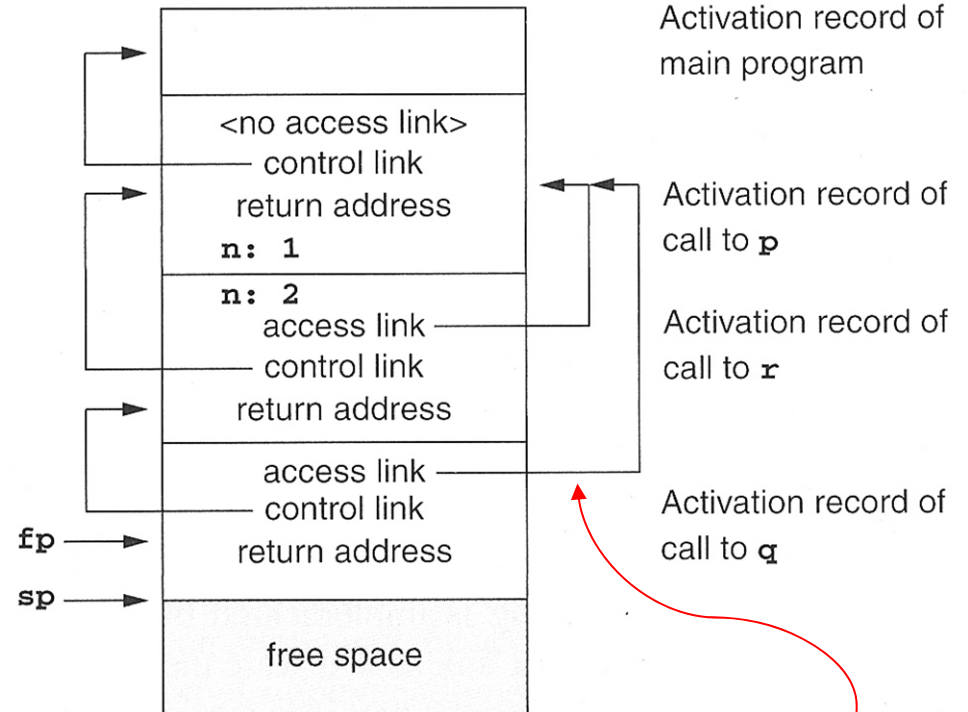
Activation record of call to **r**

Activation record of call to **q**

Kontekstvektor ('display')



Vi trenger noe ekstra (aksess-link/statisk link)



Går alltid til aktuell utgave av tekstlig omgivelse

# Eksempel med flere nivåer

```

program chain;

procedure p;
var x: integer;

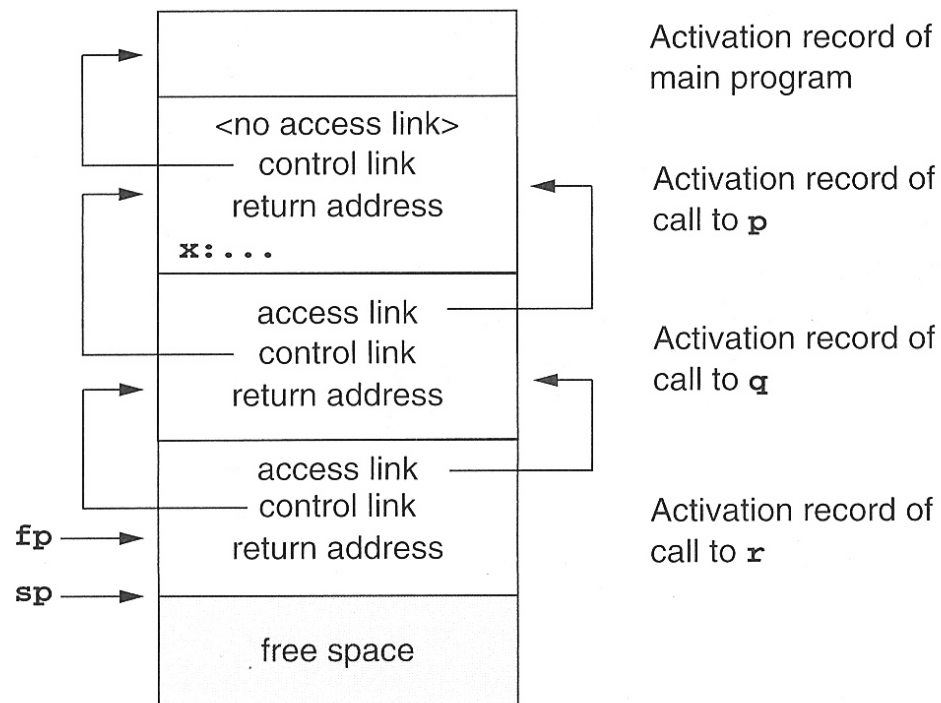
    procedure q;
        procedure r;
        begin
            x := 2;
            ...
            if ... then p;
        end; (* r *)
    begin
        r;
    end; (* q *)
end;

begin
    q;
end; (* p *)

begin (* main *)
    p;
end.

```

Program-  
blokkene  
får da et  
blokk-nivå



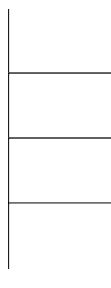
```

begin
    q;
end; (* p *)

begin (* main *)
    p;
end.

```

display



fp.al.al.x  
}  
 diff i blokknivå

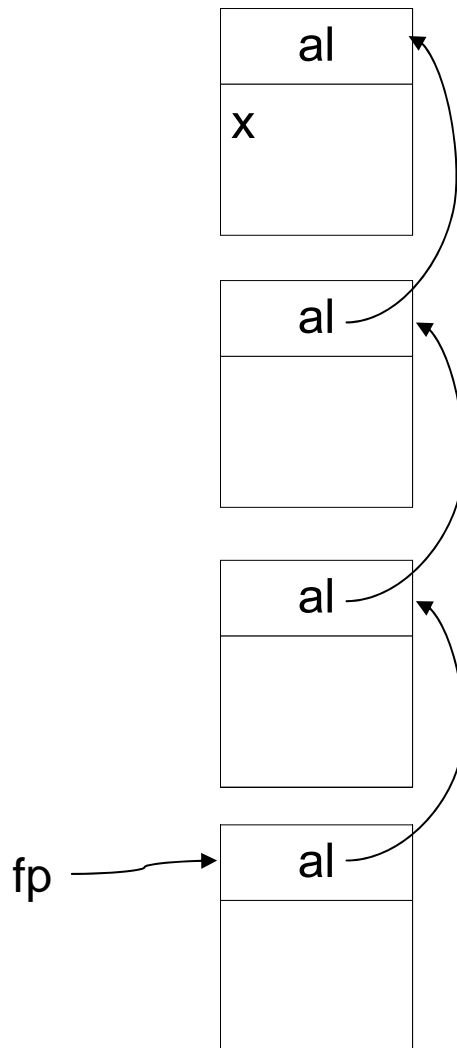
# Implementasjon av fp.al.al.al. ... al.x

Antar at fp ligger fast i et register

4(fp) -> reg  
4(reg) -> reg  
...  
4(reg) -> reg

} diff i blokknivå

X kan nå aksesseres som 6(reg)



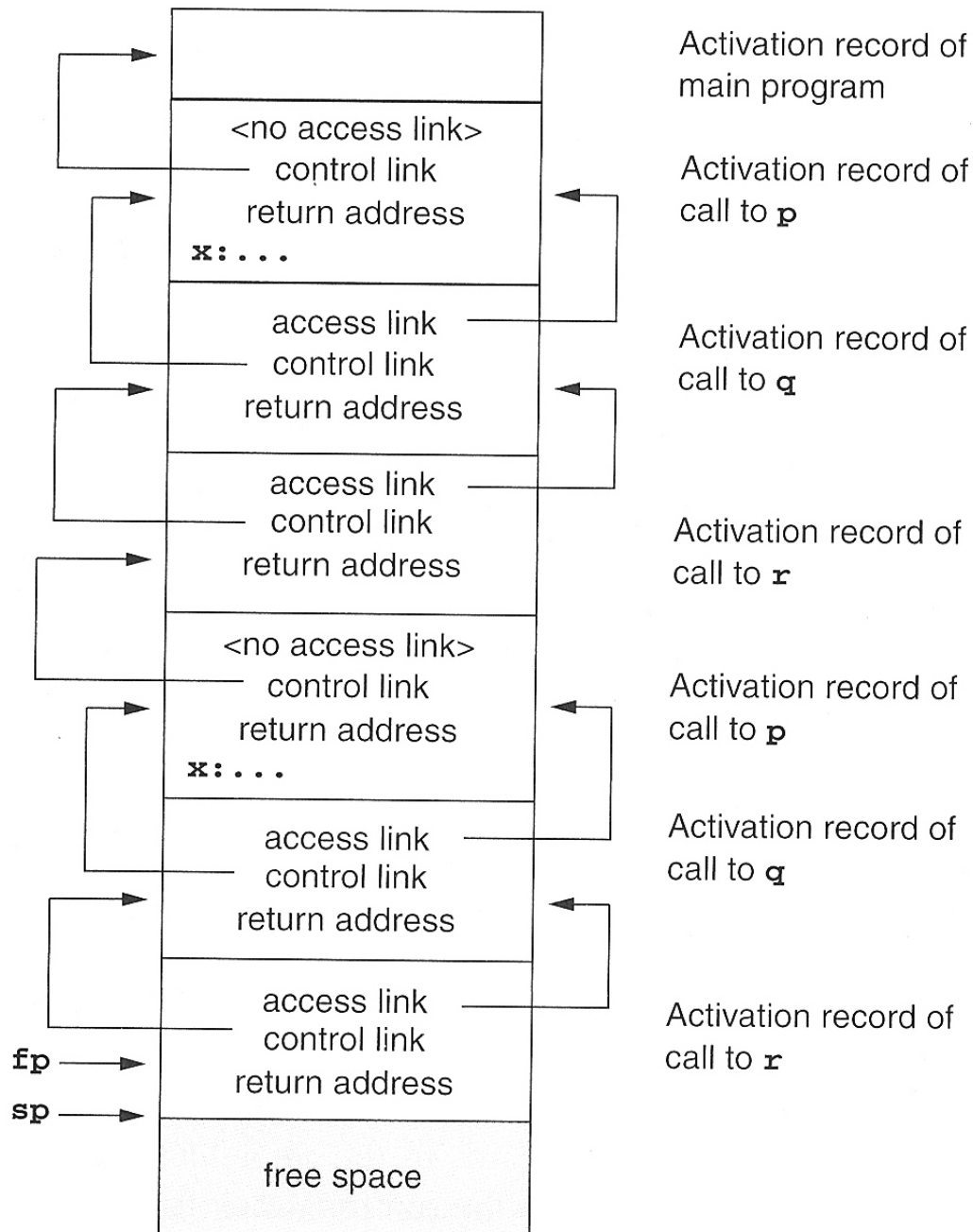
Ofte ikke så mange blokknivåer

# Videre utførelse

Hvordan skaffe  
access-link ved kall?  
Kalleren vet hvor den  
er, og utfører

ny aksess-link =  
fp.al.al....

(så mange som  
nivåforskjellen er)



# Prosedyrer som parametere

```
program closureEx(output);
```

```
procedure p(procedure a);
```

```
begin
```

```
  a;
```

```
end;
```

```
procedure q;
```

```
var x:integer;
```

```
  procedure r;
```

```
  begin
```

```
    writeln(x);
```

```
  end;
```

```
begin
```

```
  x := 2;
```

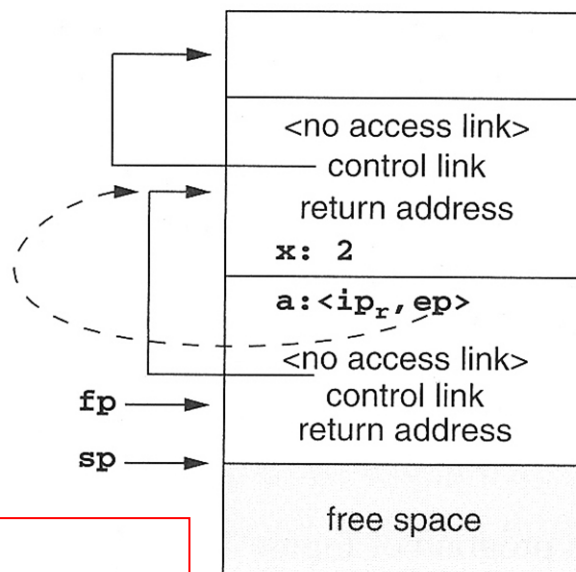
```
  p(r);
```

```
end; (* q *)
```

```
begin (* main *)
```

```
  q;
```

```
end.
```



Activation record of  
main program

Activation record of  
call to q

Activation record of  
call to p

Dette må da  
oversettes helt  
spesielt:

1. aksess-peker = ep
2. hopp til ip

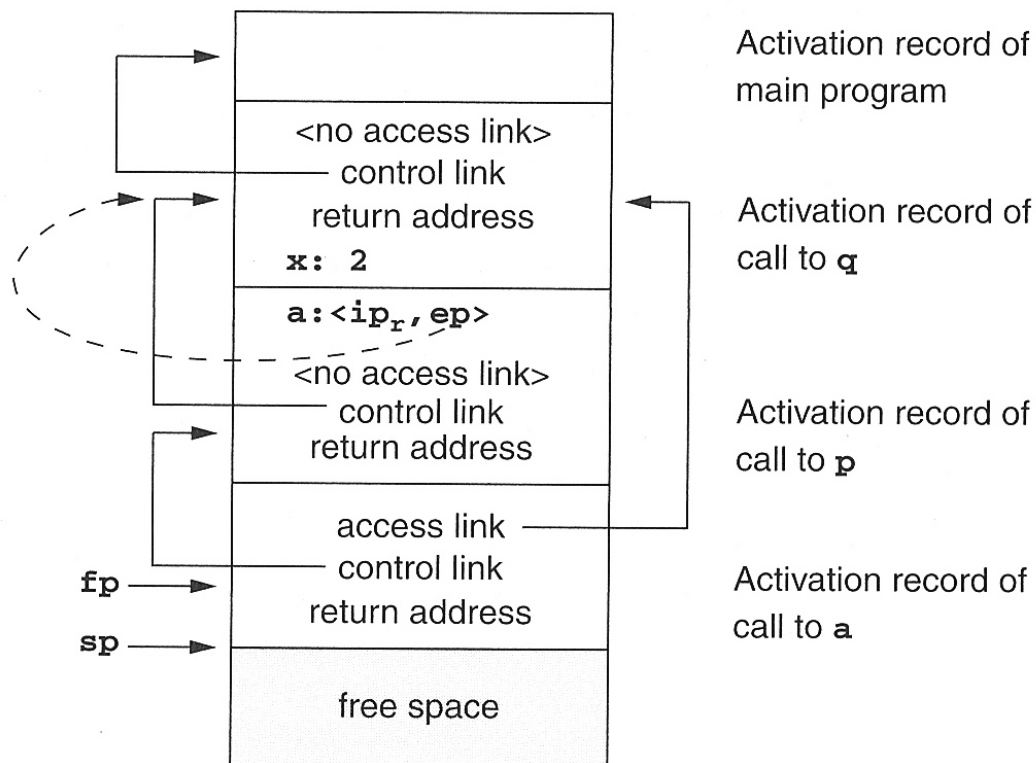
$ip_r$   
 $ep_r = fp$

Den aktuelle parameteren må være:

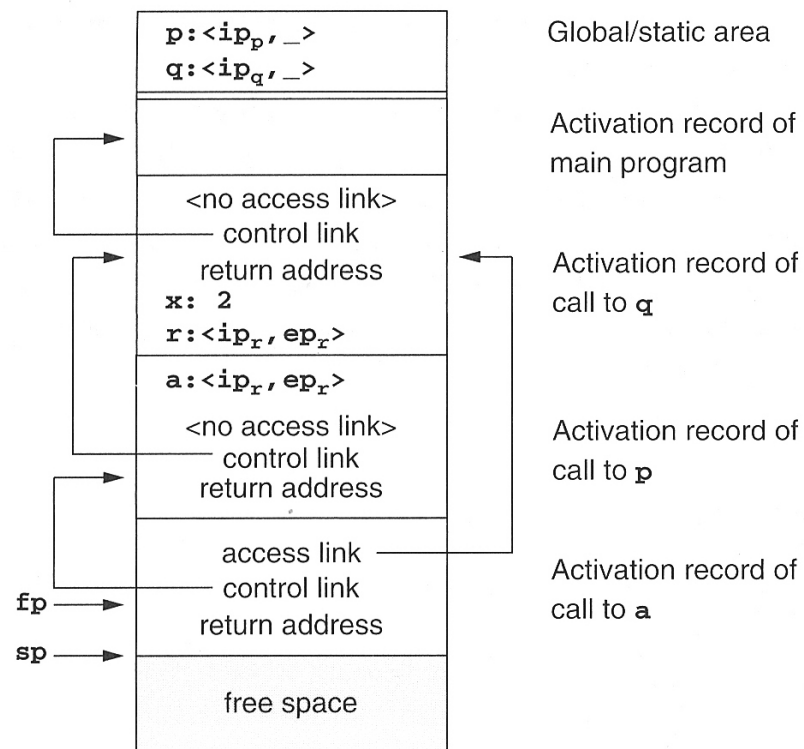
- Kode-adressen til prosedyren (ip)
- Prosedyrens aksess-link (ep)

# Kall av prosedyre levert som parameter

- Etter kallet på den formelle parameteren 'a' som aktuelt er 'r' i Q:



- Denne ser vi ikke på:



# Hva om vi skal ha 'access-link'?

## ▪ Ved prosedyrekall (entry)

1. Compute the arguments and store them in their correct positions in the new activation record of the procedure (pushing them in order onto the runtime stack will achieve this).
2. Store (push) the fp as the control link in the new activation record.
3. Change the fp so that it points to the beginning of the new activation record (if there is an sp, copying the sp into the fp at this point will achieve this).
4. Store the return address in the new activation record (if necessary).
5. Perform a jump to the code of the procedure to be called.

## ▪ Ved prosedyre-exit

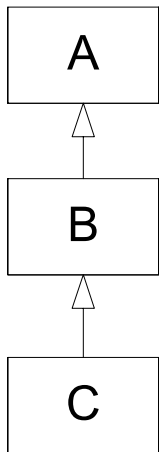
1. Copy the fp to the sp.
2. Load the control link into the fp.
3. Perform a jump to the return address.
4. Change the sp to pop the arguments.

+ aksess-link

1. Beregn ny aksess-link som  $nyal = fp.al.al \dots$  (tilsvarende diff. i blokknivå mellom kalleren og kallet - er 0 om den kalle er lokal i kalleren)
2. Push nyal på stakken

# Objekt-orientering

- Klasser og subklasser
- Typedede pekere
- Virtuelle og ikke-virtuelle metoder

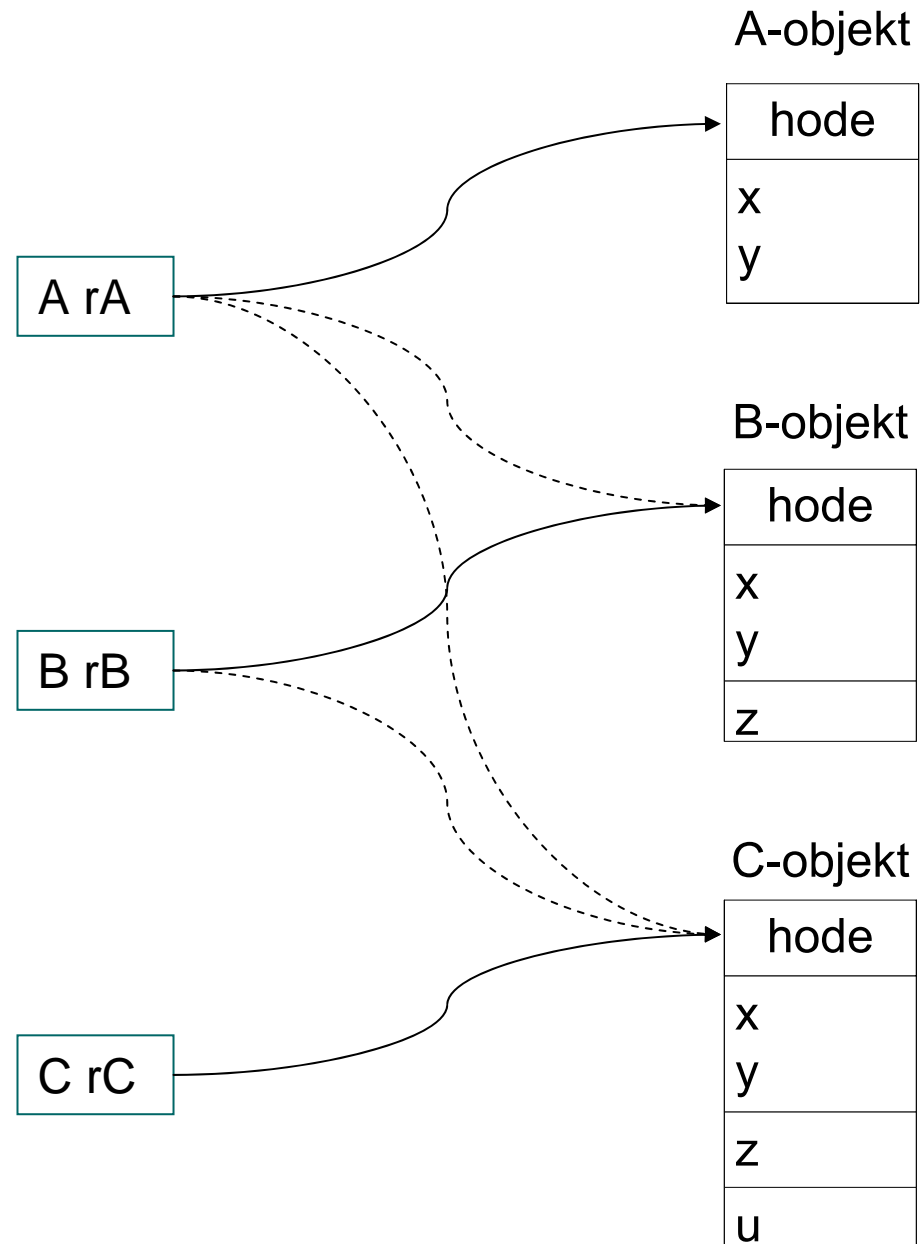


```

class A {
  int x,y;
  void f(s,t) {...K...};
  virtual void g(p,q) {...L...}
}

class B extends A{
  int z;
  void f(s,t) {...Q...};
  redef void g(p,q) {...M...};
  virtual void h(r) {...N...}
}

class C extends B{
  int u;
  redef void h(r) {...P...}
}
  
```





# Kall på metoder

- Kall på ikke-virtuelle metoder (bruk pekerens type)
  - rA.f(1,2) gir alltid f i A (K)
  - rB.f(1,2) gir alltid f i B (Q)
  - rC.f(1,2) gir alltid f i B (Q)
- Kall på virtuelle metoder (bruk objektets type)
  - rA.g(3,4) } (L eller M)
  - rB.g(3,4) } gir den dybeste versjonen (M)
  - rC.g(3,4) } i det aktuelle objektet (M)
  
  - rA.h(5) ulovlig (statisk, kompileringstid)
  
  - rB.h(5) } (N eller P)
  - rC.h(5) } i det aktuelle objektet (P)

# Implementasjon (typede pekere)

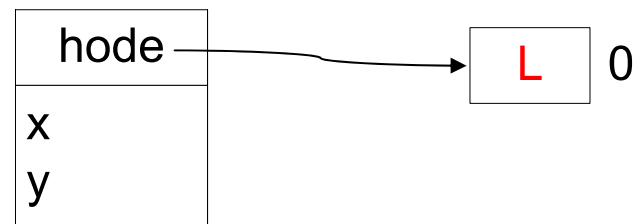
- Kompilatorsjekk av `rX.f(...)`, både virtuelle og ikke-virtuelle): `f` må være definert i `X` eller i superklassen til `X`
- De ikke-virtuelle bindes ferdig i kompilatoren
- De virtuelle nummereres (med 'offset') fra ytterste klasse og innover – redefinisjoner får samme nummer
- La obekthodene inneholde en peker til klassen felles virtuell-tabell

`rA.g(...)` implementeres slik:

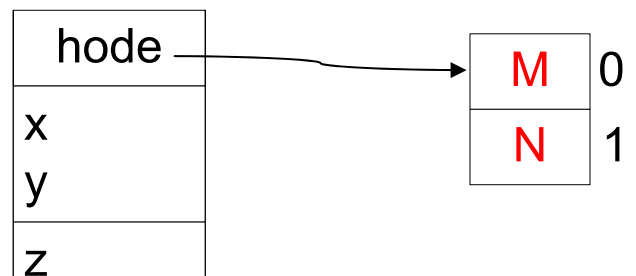
`call(rA.virttab[g_offset])`

Kompilatoren vet:  $g\_offset = 0$   
 $h\_offset = 1$

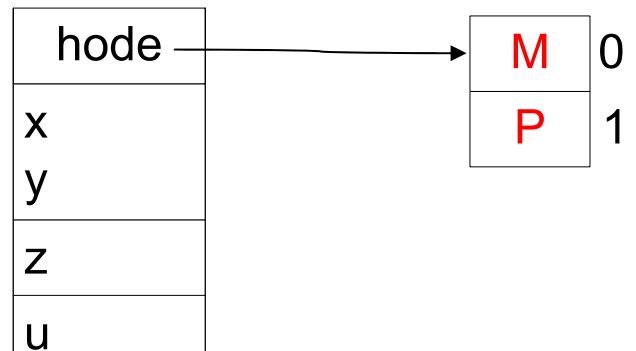
A-objekt



B-objekt



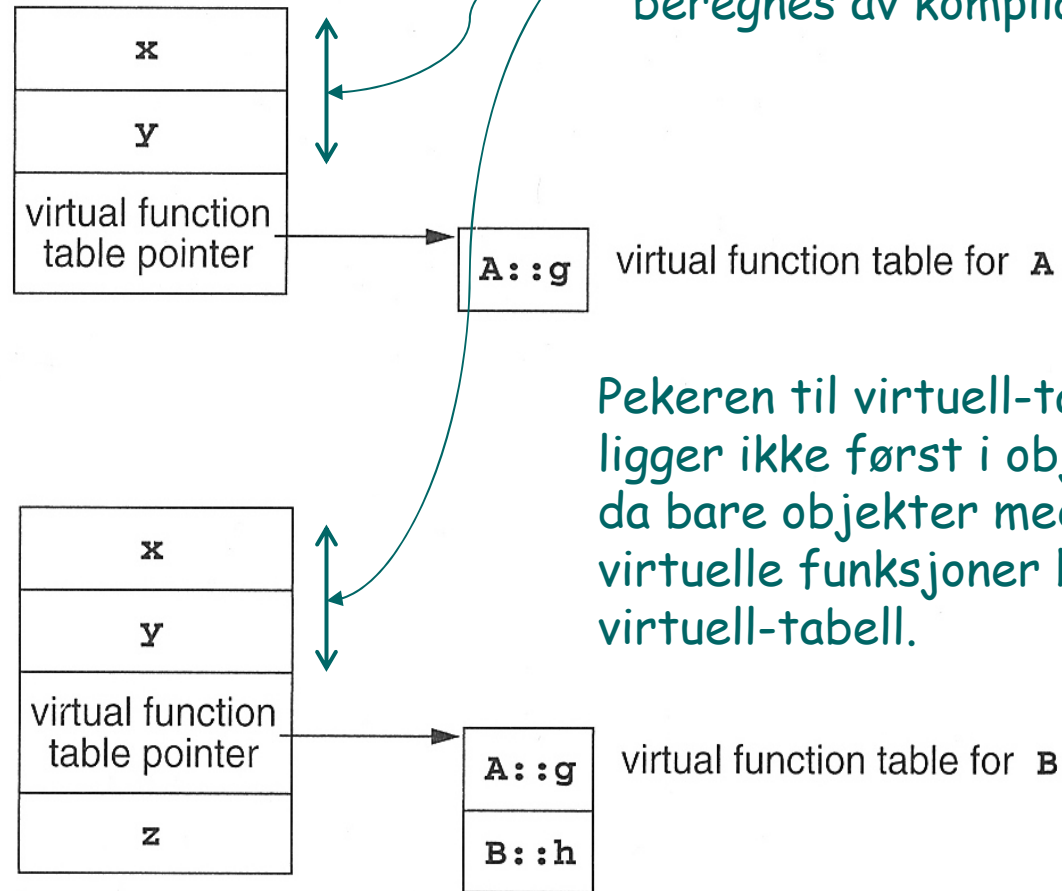
C-objekt



# Impl. av virtuelle metoder som i boken (C++)

```
class A
{ public:
  double x,y;
  void f();
  virtual void g();
};
```

```
class B: public A
{ public:
  double z;
  void f();
  virtual void h();
};
```



Virtuell-tabell pekeren har relativ-adresse som kan beregnes av kompilatoren

Pekeren til virtuell-tabellen ligger ikke først i objektet, da bare objekter med virtuelle funksjoner har en virtuell-tabell.

# Utypede pekere (f.eks. Smalltalk)

- Ikke-virtuelle metoder finnes ikke
- Problem med virtuell-tabeller: Alle virtuell-tabeller måtte innholde alle metoder i alle klasser, altså for stor.
- I tillegg: I Smalltalk kan man legge til metoder underveis
- Derfor (antar at f er fjernet):

r.g(...) implementeres slik:

1. Gå til objektets klasse
2. Let etter 'h' ut gjennom superklassene

