Runtimesystemer Kap 7 - I

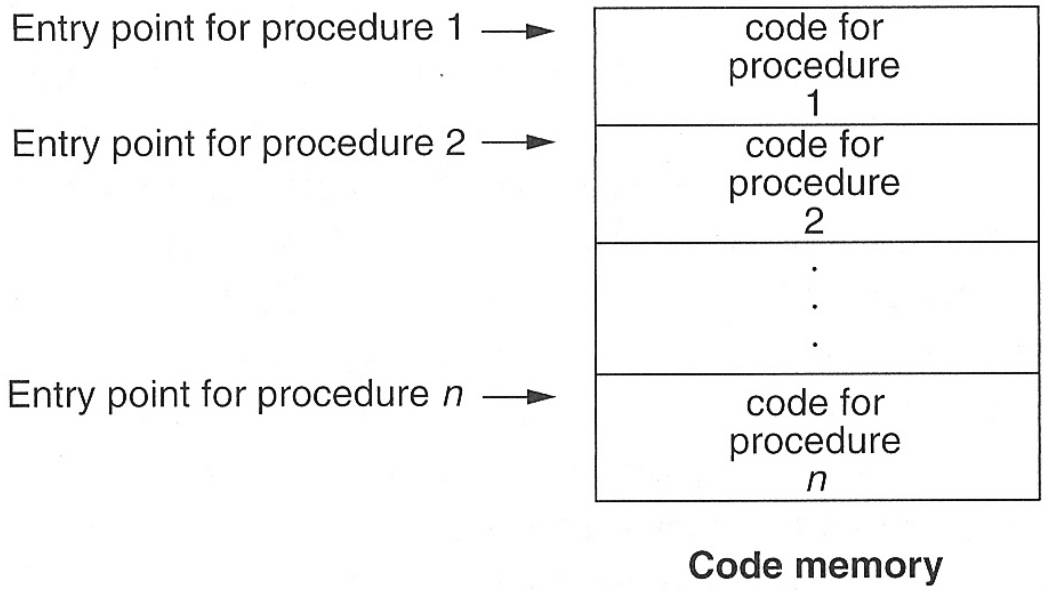
- Språk som bare trenger statisk allokering
- Språk som trenger stakk-orientert allokering
- Språk som trenger mer generell allokering
  - Forskjellige slags begreper i et gitt språk krever at runtimesystemet organiserer forskellige deler av en programutførelse forskjelligt
- Parameteroverføring

INF 5110 - 2012

Den oversatte programkoden

- kan nesten altid betraktes som statisk allokert
- skal hverken flyttes eller forandres under utførelse

- Kompilatoren kjenner alle adresser til kodebiter



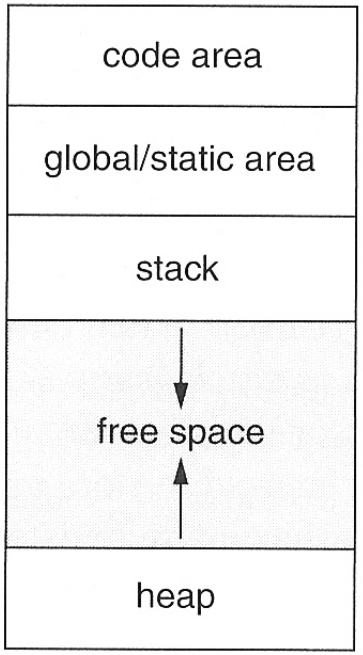
Men husk

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

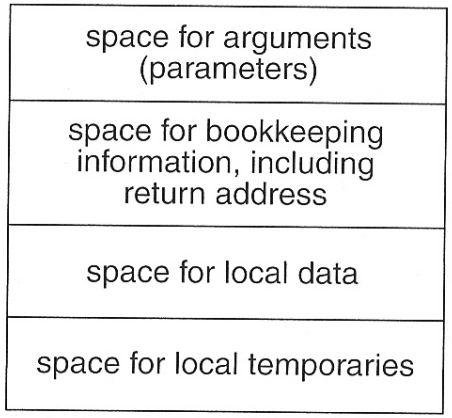
OS kan flytte rundt på kode, men det forstyrrer ikke kodens eget adresserum

Lagerorganisering

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

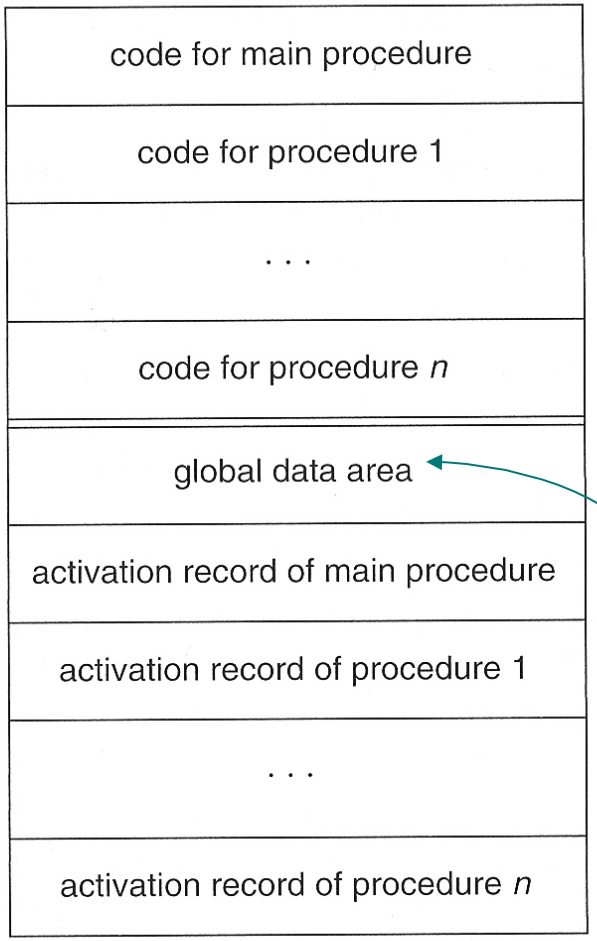


- Typisk organisering av data for et prosedyrekall (aktiveringsblokk)



Det er gjerne ut fra plasseringen her man karakteriserer språk til være
- statisk organisert
- stakk-organisert
- heap/dynamisk organisert

Full statisk organisering (eks. Fortran)

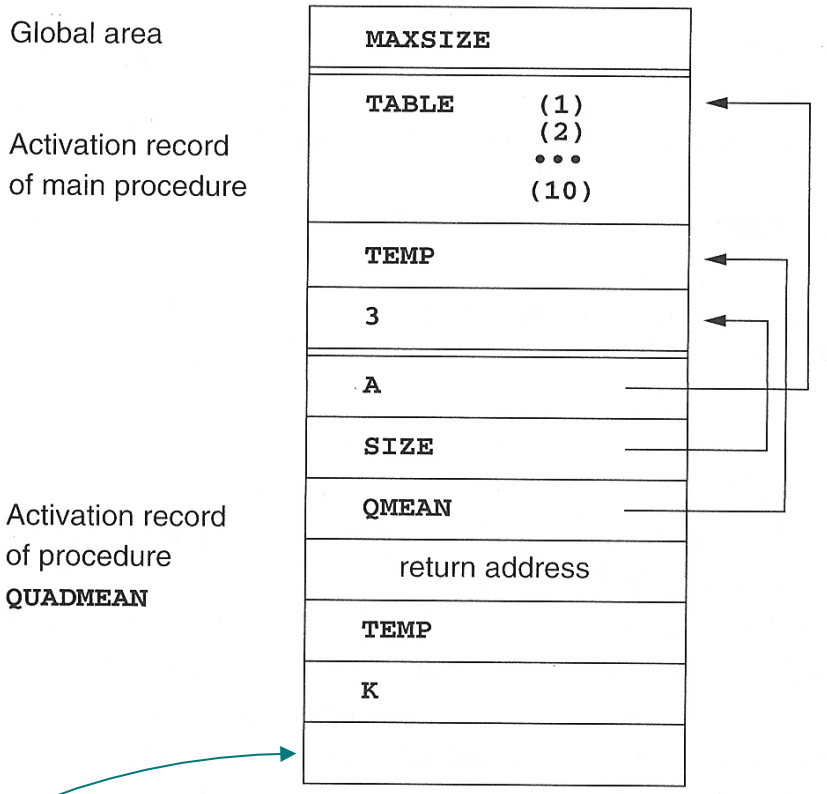


- Kompilatoen kan beregne hvor alt ligger
  - Utførbar kode
  - Variable
  - Alle slags hjelpedata

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
```



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

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

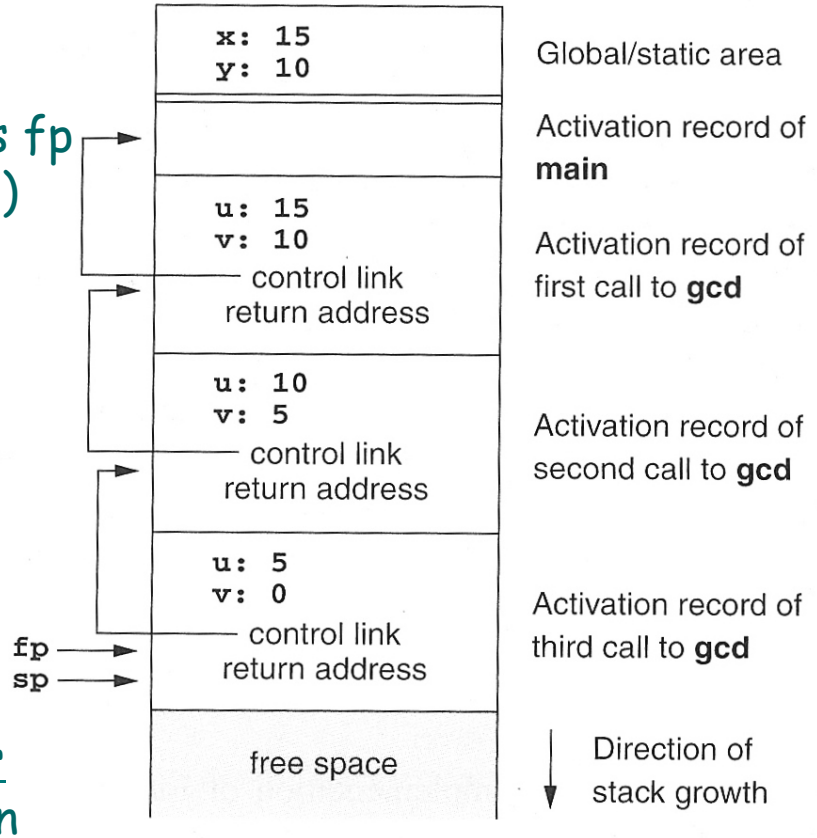
Et eksempel i C

```
#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. Kreves 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 aktiveringsblokkken

stack pointer
Angir grensen mellom brukt og ledig lagerareal

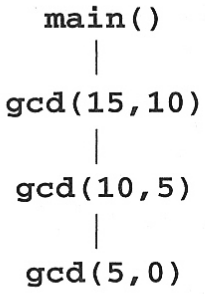
INF 5110 - 2012

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;
}
```

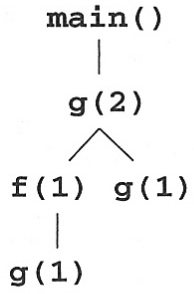
Blir én global variabel bare synlig fra f

forrige program



(a)

dette program

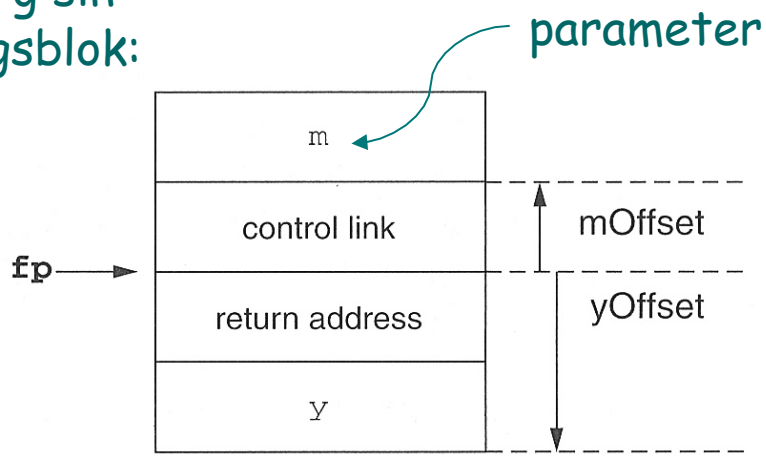


(b)

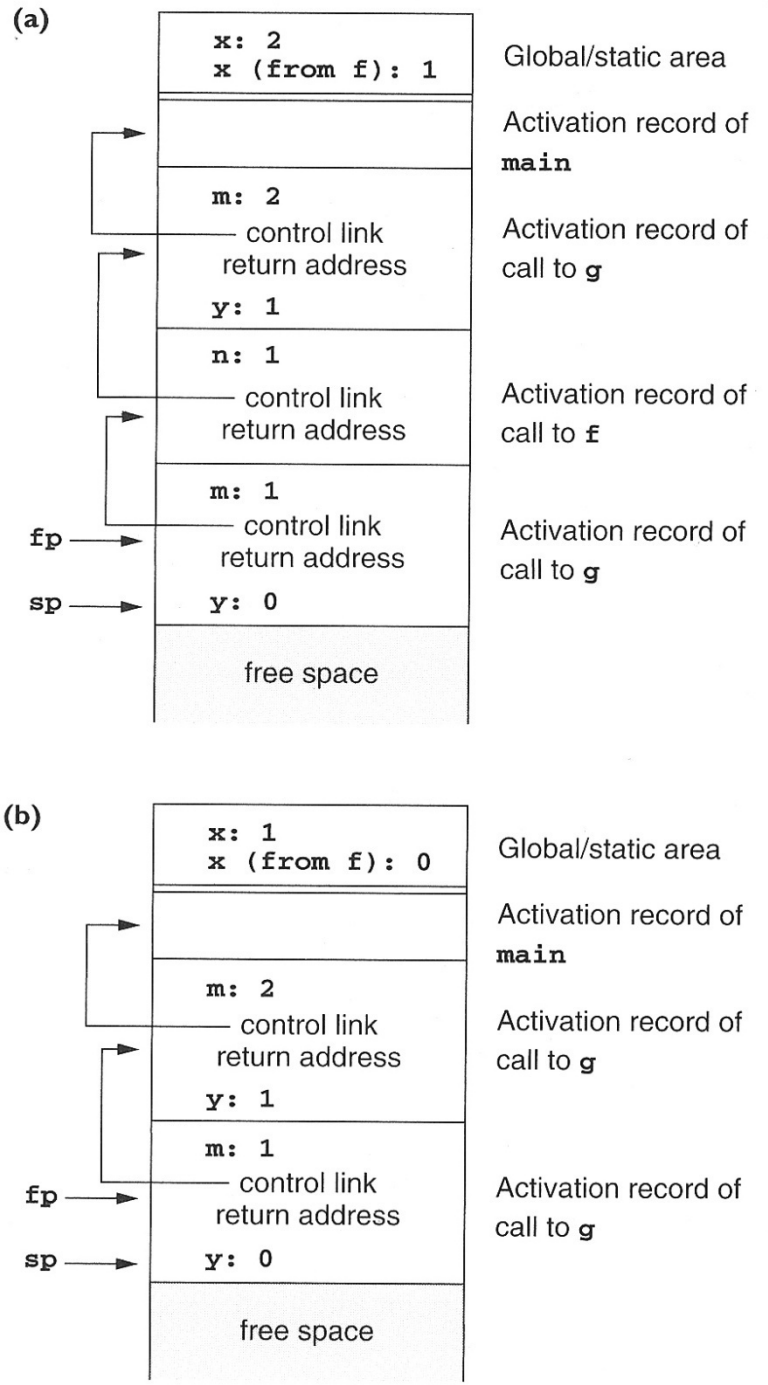
Relativ-adresser:
mOffset = +4
yOffset = -6

Aksess av y: -6(fp)

Layout av g sin aktiveringsblok:



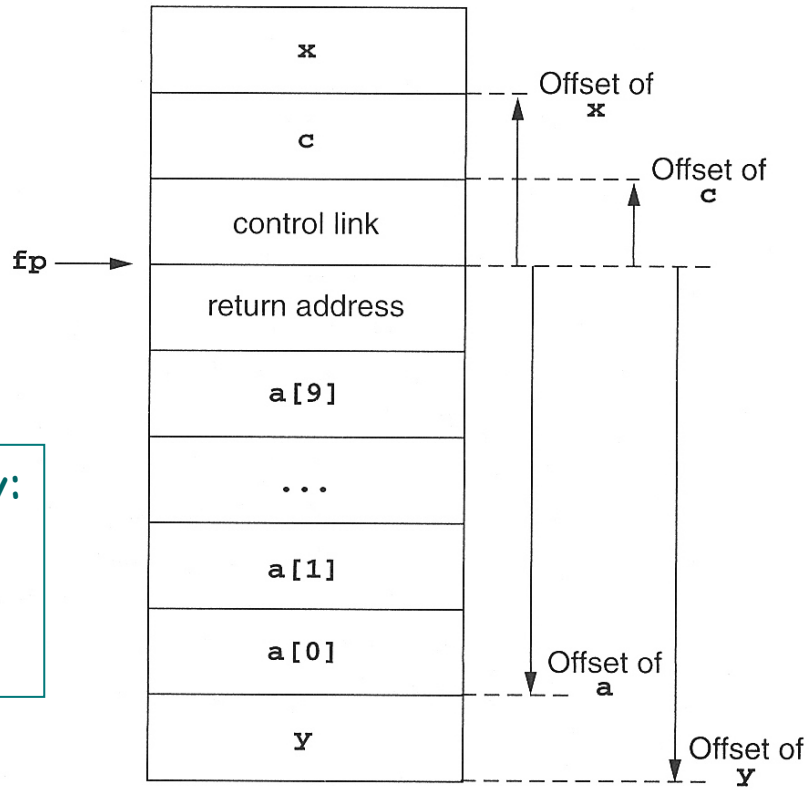
```
int x = 2;

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;
}
```



Arrayer av kjent (statisk) lengde

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

Layout av aktiveringsblokk:



Relativ-adresser

Name	Offset
x	+5
c	+4
a	−24
y	−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

INF 5110 - 2012

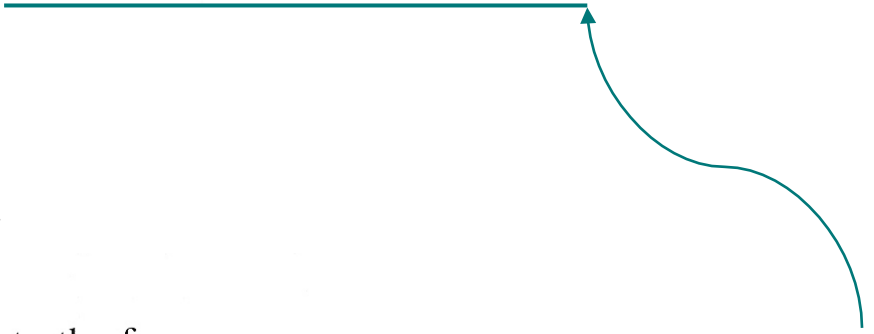
Hvordan utføre et kall

- 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.

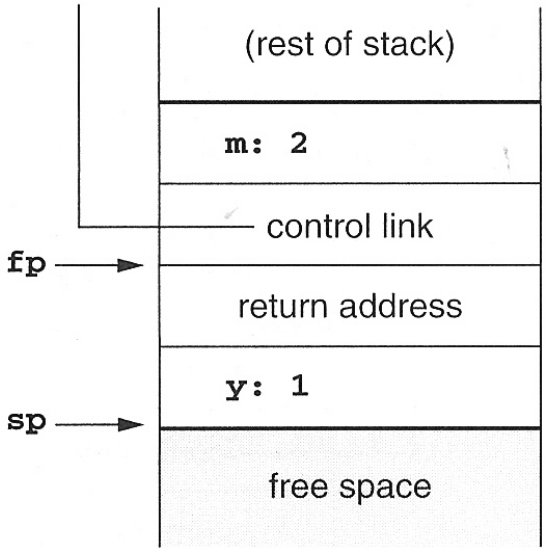
- Ved prosedyre-exit

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.

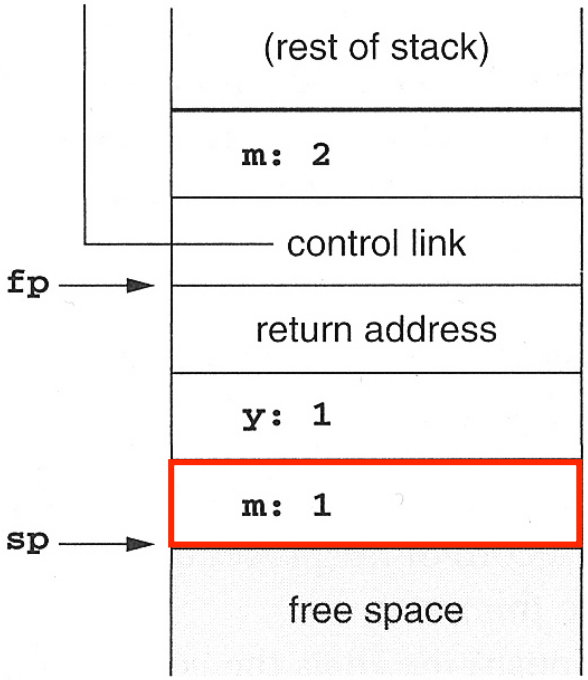


Denne setter av plass til lokale variable, ved å flytte sp
Kan evt. også initialisere disse.
(Kan tenkes på som push)

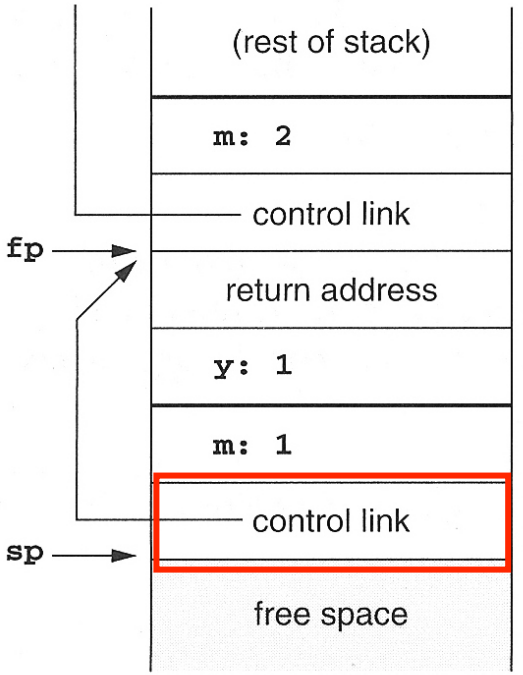
Gjennomføring av et kall - I



før kall på g



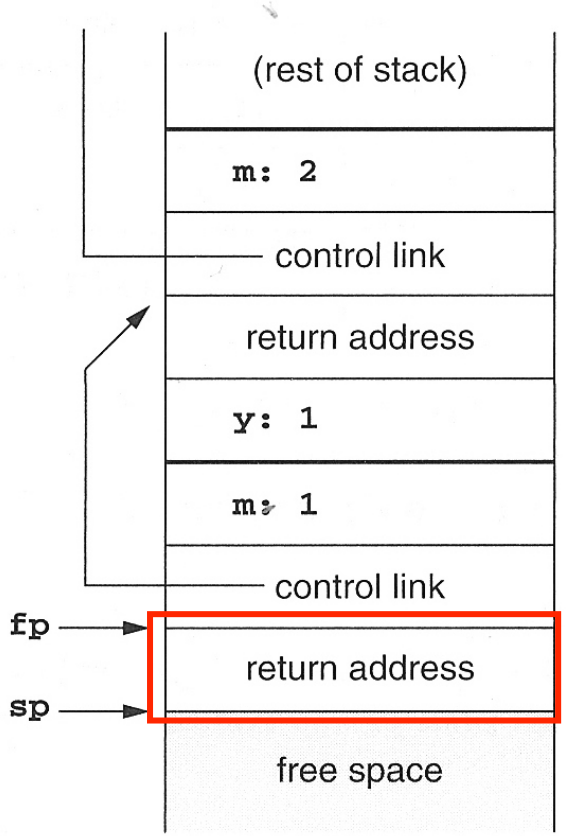
push parameter



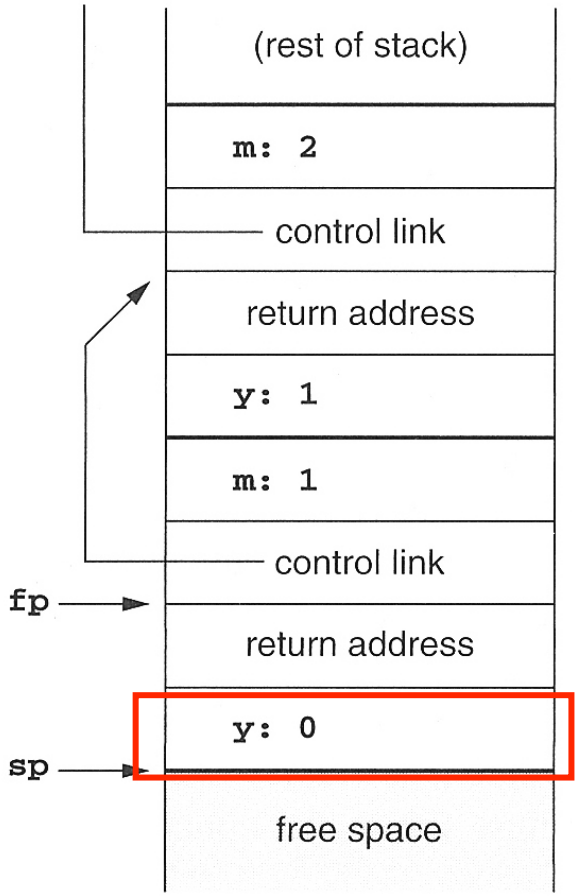
push fp

INF 5110 - 2012

Gjennomføring av et kall – II



1. fp = sp
2. Push returadressse



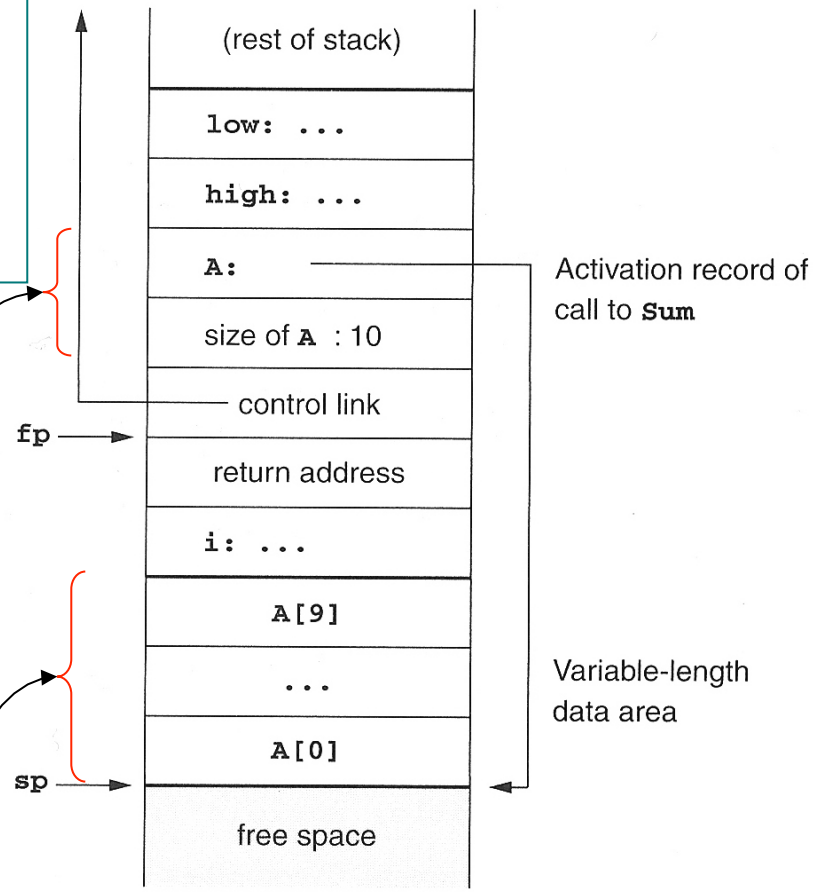
alloker lokal var (y)

Data av variabel lengde

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

```
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

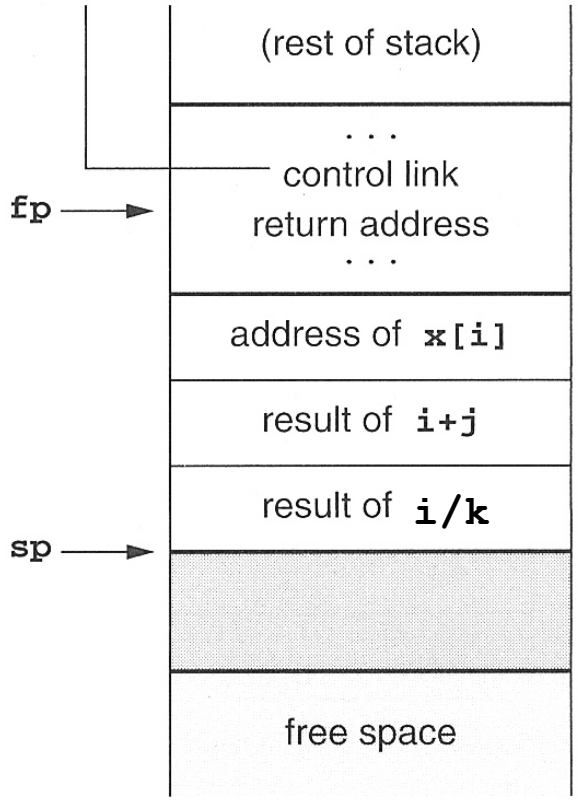
I Java legges disse på heapen

Behandling av mellomresultater

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

adresse verdi verdi

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



Activation record of procedure containing the expression

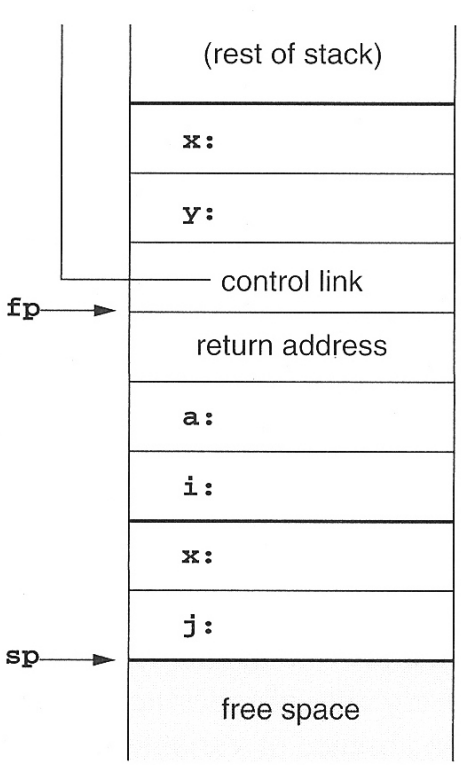
Stack of temporaries

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

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

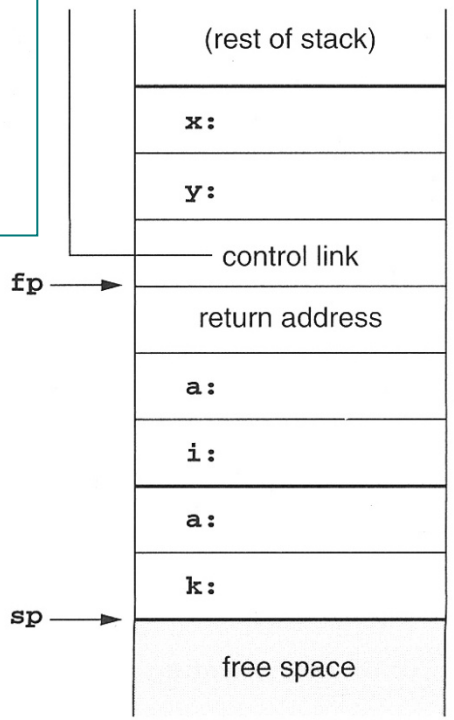
Mulig plass-allokering ved 'indre blokker'

```
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`

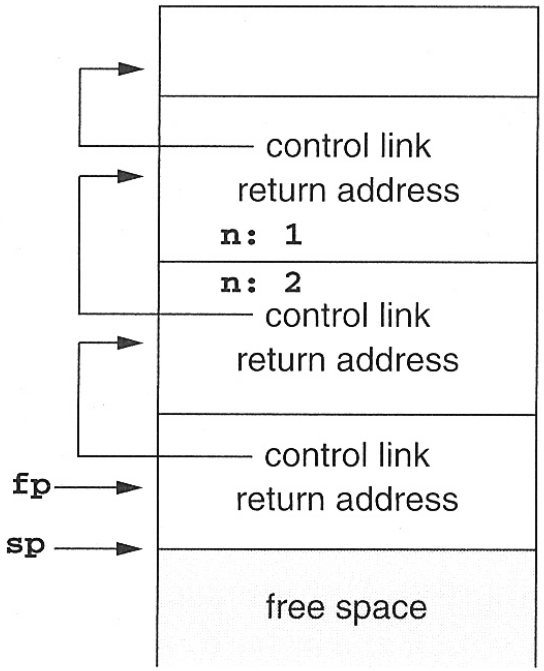
Allocated area for block `B`

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



Activation record of main program

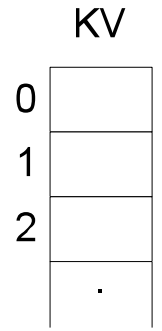
Activation record of call to `p`

Activation record of call to `r`

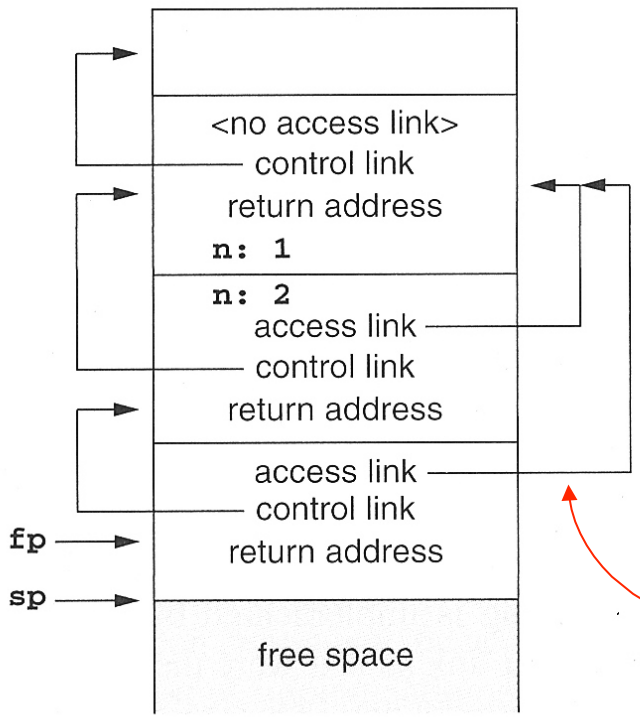
Activation record of call to `q`

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

Kontekstvektor ('display')



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



Activation record of main program

Activation record of call to `p`

Activation record of call to `r`

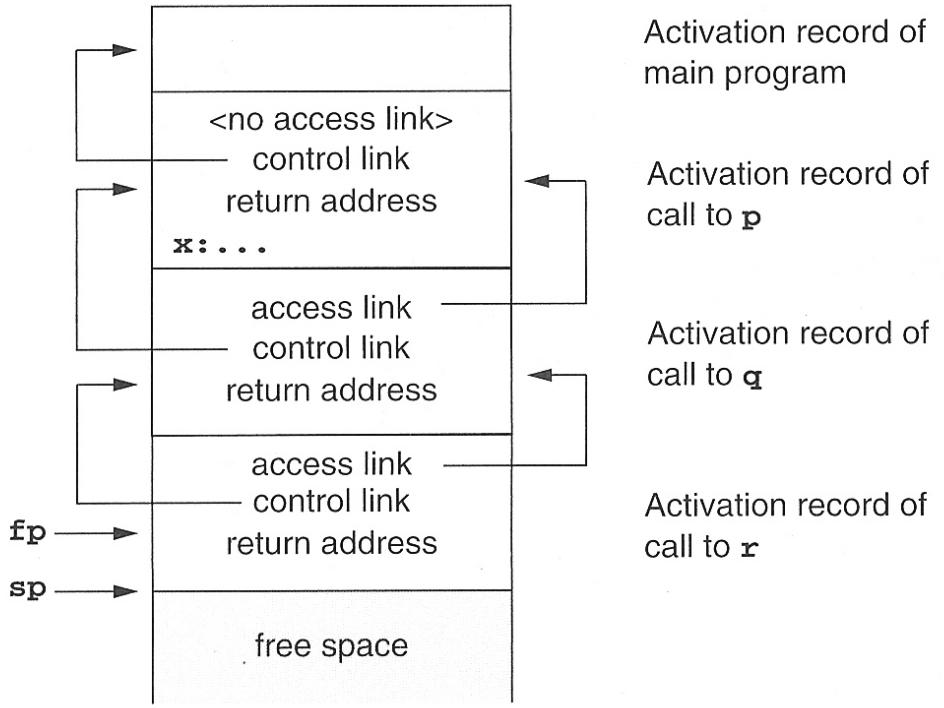
Activation record of call to `q`

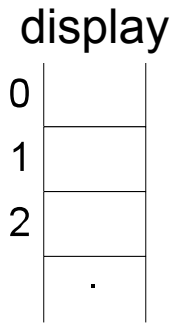
Går altid til aktuell utgave av tekstlig omgivelse

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 *)

begin
  q;
end; (* p *)

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

Program-blokkene får da et blokk-nivå





fp.al.al.x

diff i blokknivå

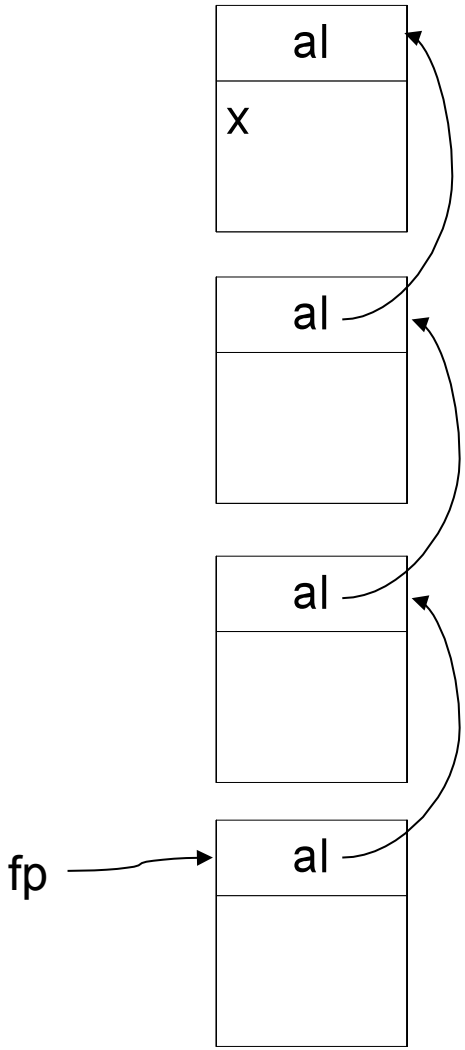
Implementasjon av fp.al.al.al. ... al.x

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

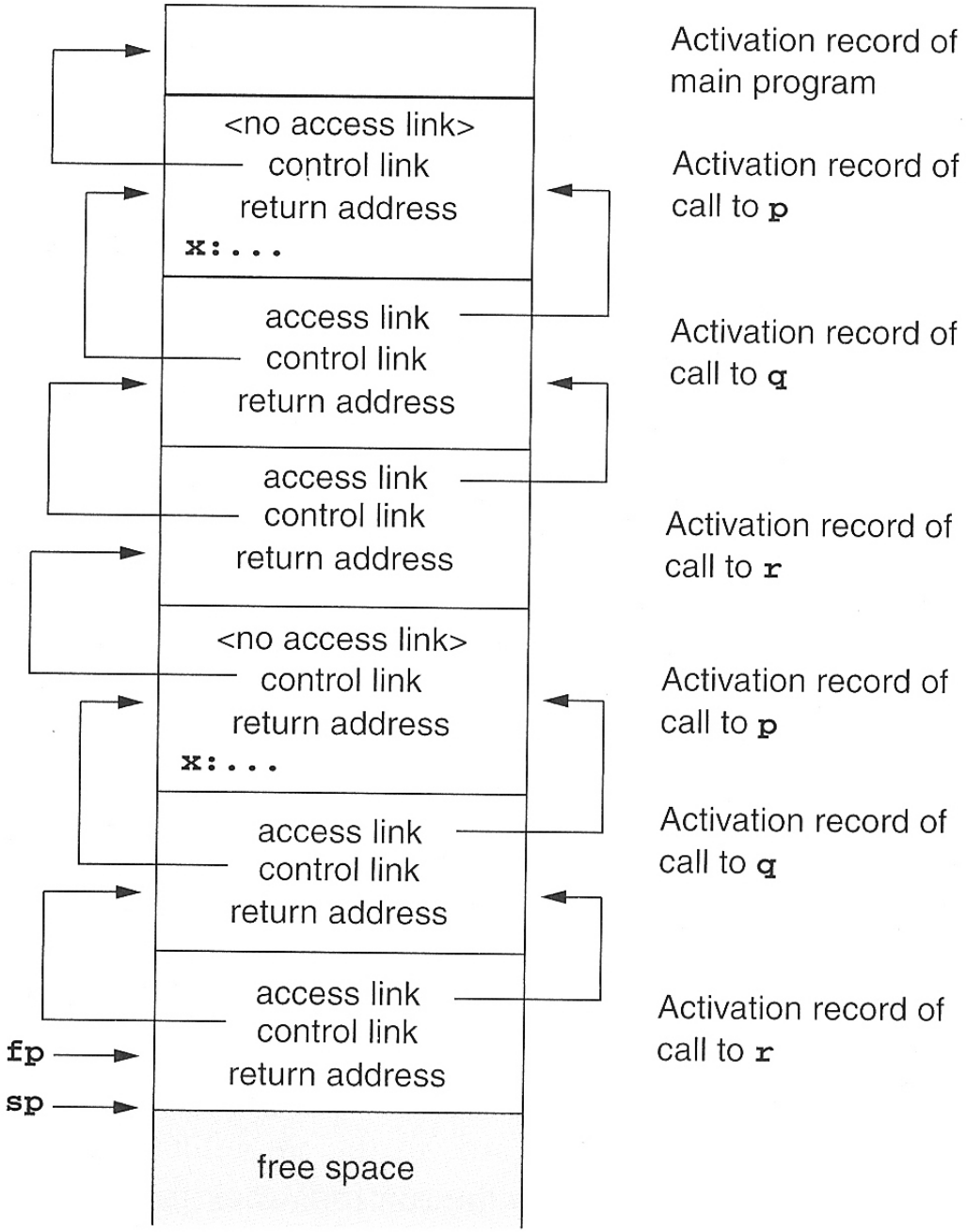
INF 5110 - 2012

Videre utførelse

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

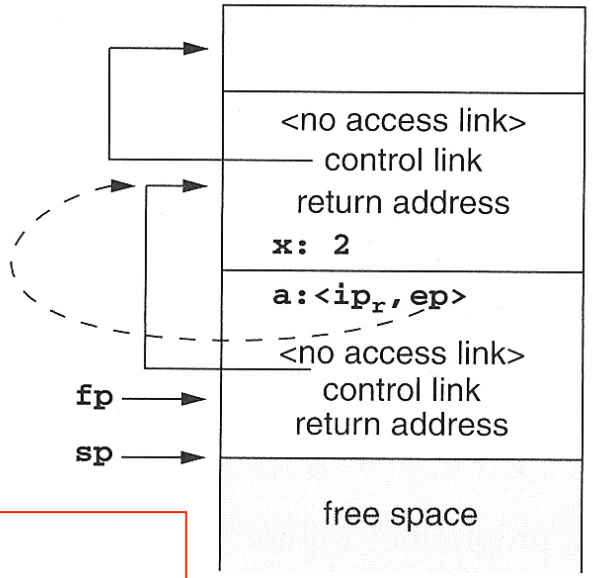
ny aksess-link = fp.al.al....

(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:

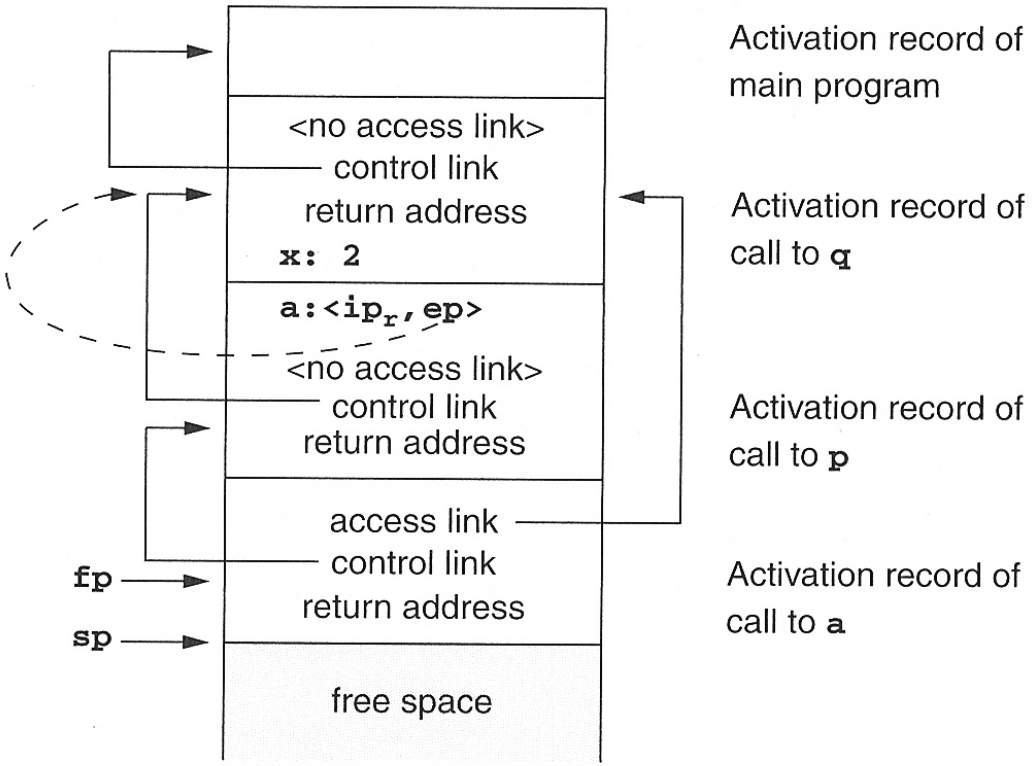
1. aksess-peker = ep
2. hopp til ip

ip_r
ep_r

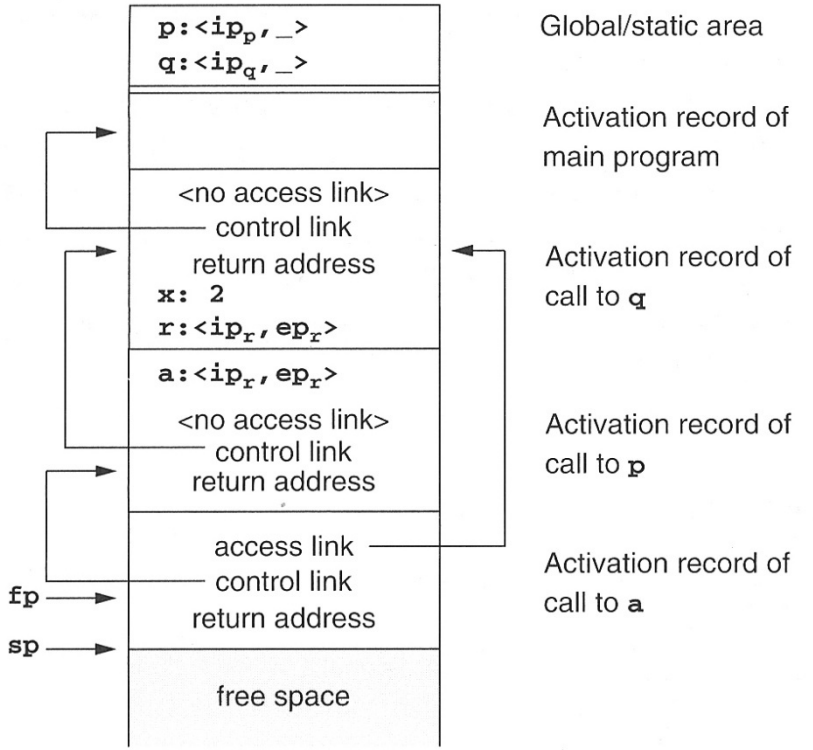
Den aktuelle parameteren må være:
- Kode-adressen til prosedyren (ip)
- Prosedyrens aksess-link (ep)

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.

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

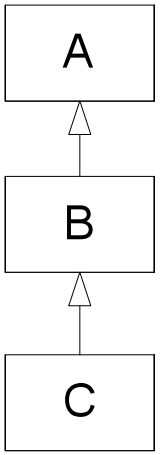
- Ved prosedyre-exit

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.

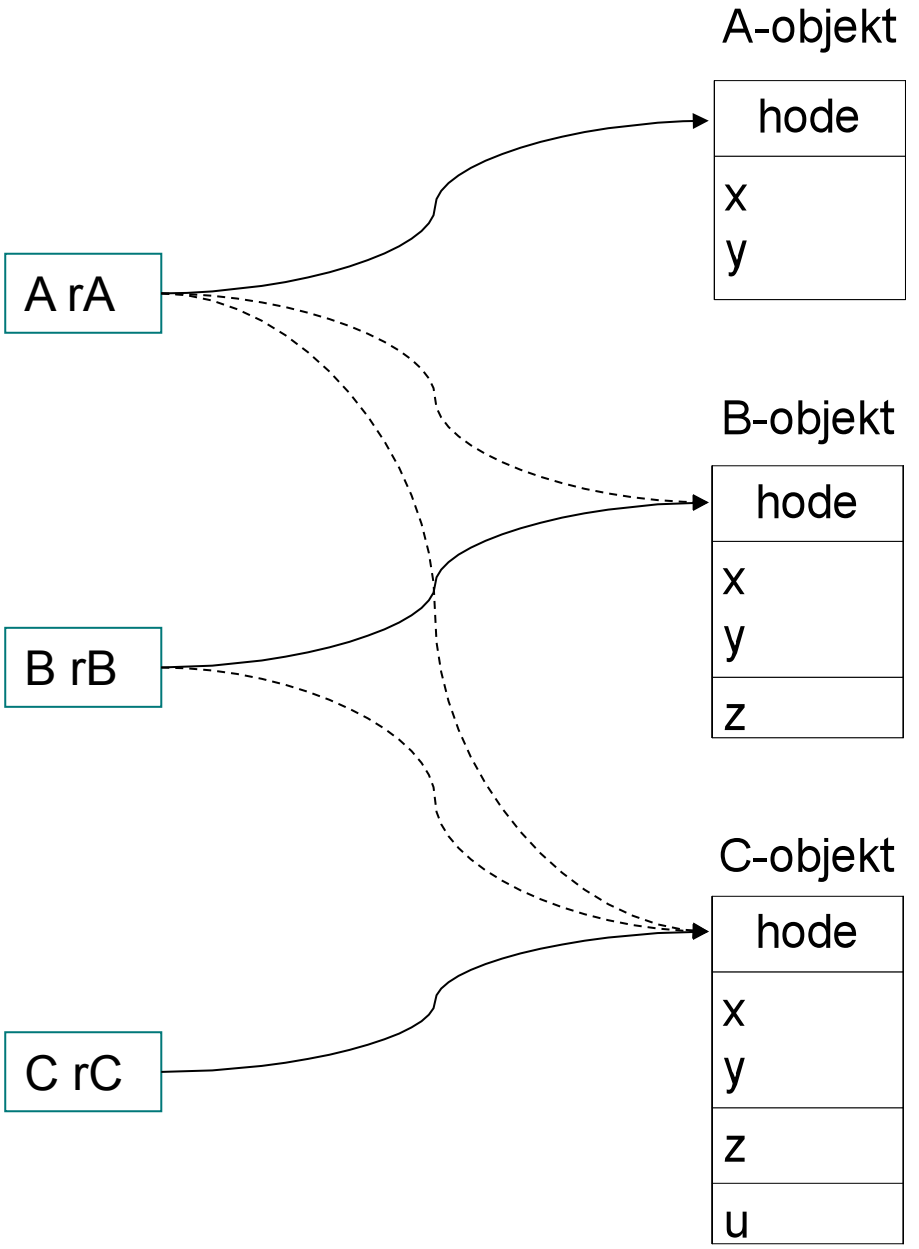
+ aksess-link

Objekt-orientering

- Klasser og subklasser
- Typede 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...}
}
```

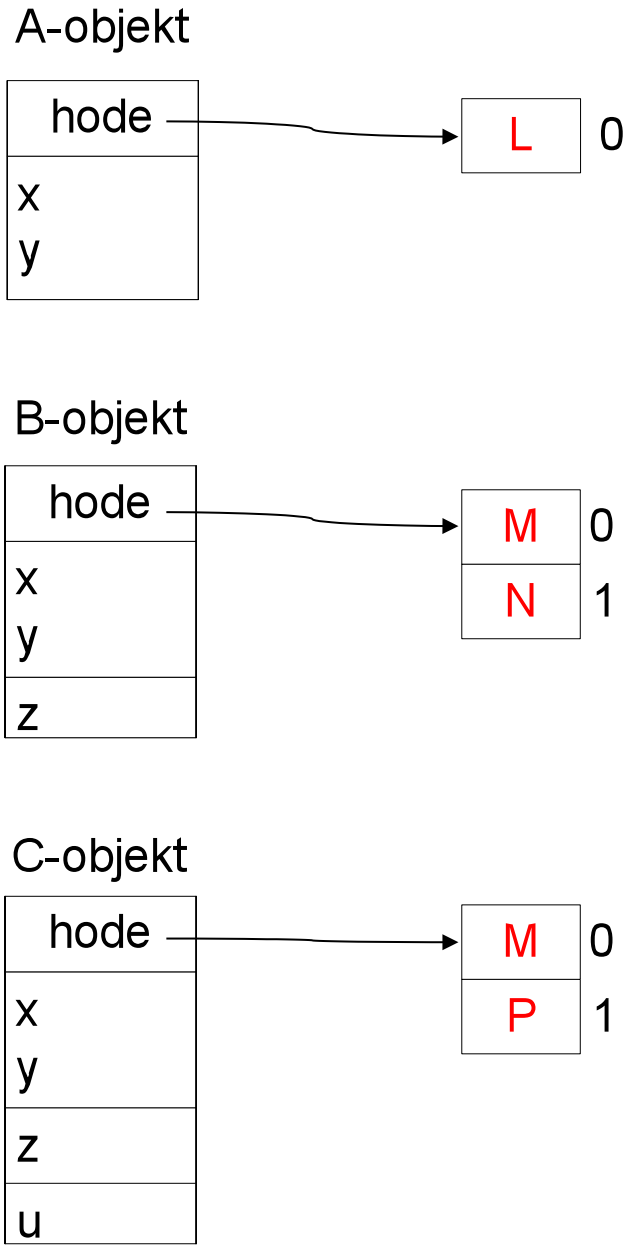


Kall på metoder

- Kall på ikke-virtuelle metoder (bruk pekerens type)
  - rA.f(1,2) gir altid f i A (K)
  - rB.f(1,2) gir altid f i B (Q)
  - rC.f(1,2) gir altid 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) gir den dybeste versjonen (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 objekthodene inneholde en peker til klassens felles virtuell-tabell



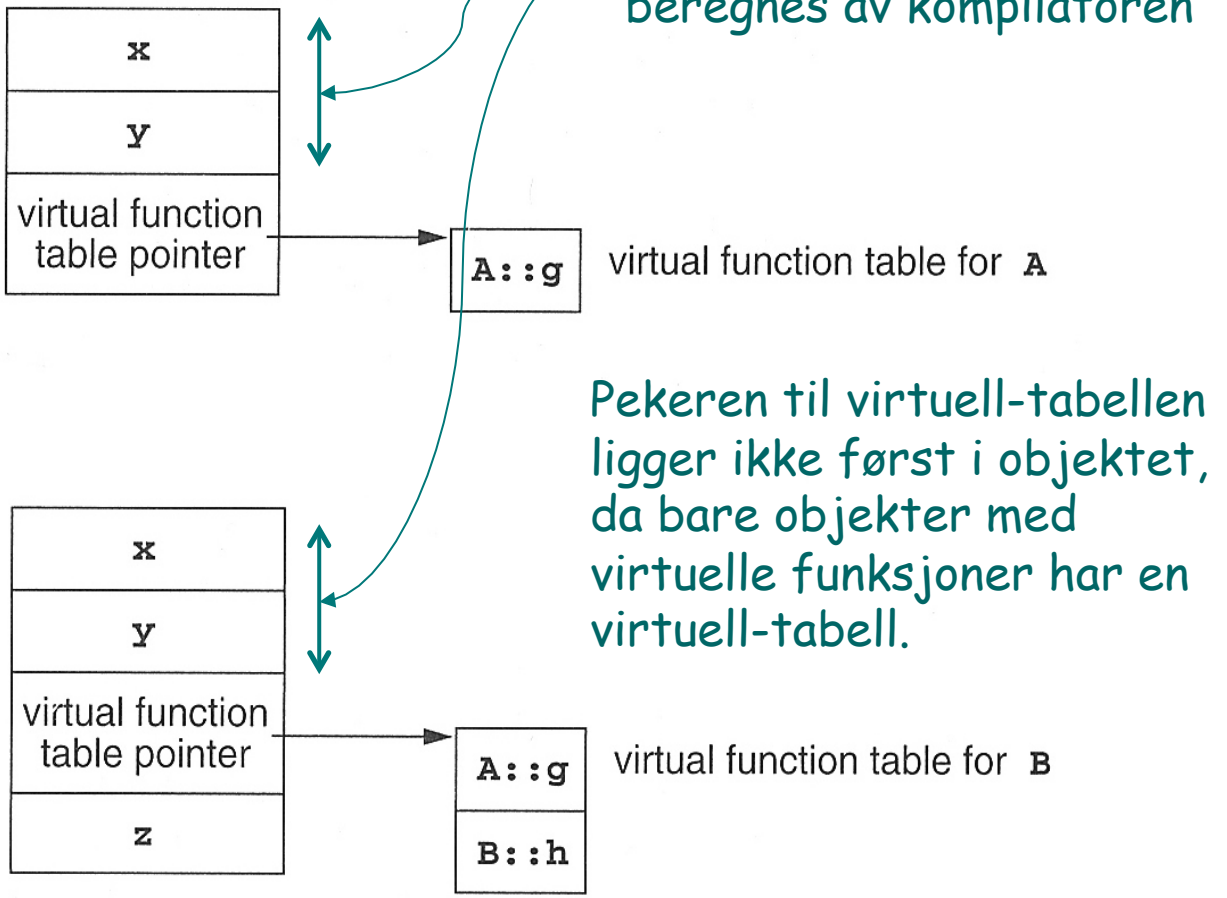
- rA.g(...) implementeres slik:

  call(rA.virttab[g_offsett])

Kompilatoren vet: g_offset = 0
h_offset = 1

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();
};
```



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 'g' ut gjennom superklassene