

Runtime-omgivelser Kap 7 - II

- Dynamisk lager-allokering/når trenger vi en heap
 - For objekter/recorder som allokeres dynamisk (new) og som man kan ha pekere til
 - Gjelder ofte også array-objekter
 - Under visse forhold også aktiveringsblokker for prosedyrer
 - Simula, pga korutiner
 - Hvis man har
 - Prosedyre-variable
 - Nestede prosedyrer

Noen alternativer

- Når blir plass ledig?
 1. Brukeren sier selv fra (alloc/free)
 - Kan lett bli feil og inkonsistenser
 2. Systemet finner automatisk hva som blir ledig
 - Krever ekstra administrasjon/informasjon
- Gjenbruk av frigjort plass
 1. Man flytter aldrig objekter
 - Fører lett til fragmentering
 2. Man flytter sammen de objekter som skal bevares
 - Krever ekstra administrasjon/informasjon
 - Alle pekere til flyttede objekter må forandres
 - All ledig plass samlet i et område

Problemer med frie pekere

```
int * dangle(void)
{ int x;
  return &x;}
```

- Dette og lignende problemer (spesielt i funksjonelle språk, men også for korutiner i Simula) gjør det nødvendig å legge aktiveringsblokker for prosedyrer på heapen
- Altså: dynamisk prosedyre allokering/deallokering- ikke stakk-basert

```
typedef int (* proc)(void);

proc g(int x)
{ int f(void) /* illegal local function */
  { return x; }
  return f; }

main()
{ proc c;
  c = g(2);
  printf("%d\n",c()); /* should print 2 */
  return 0;
}
```

Her defineres navnet 'proc'

Leverer en prosedyre

Prosedyren som leveres er lokal i g, og g() terminerer

Eksempel på gjenvinning

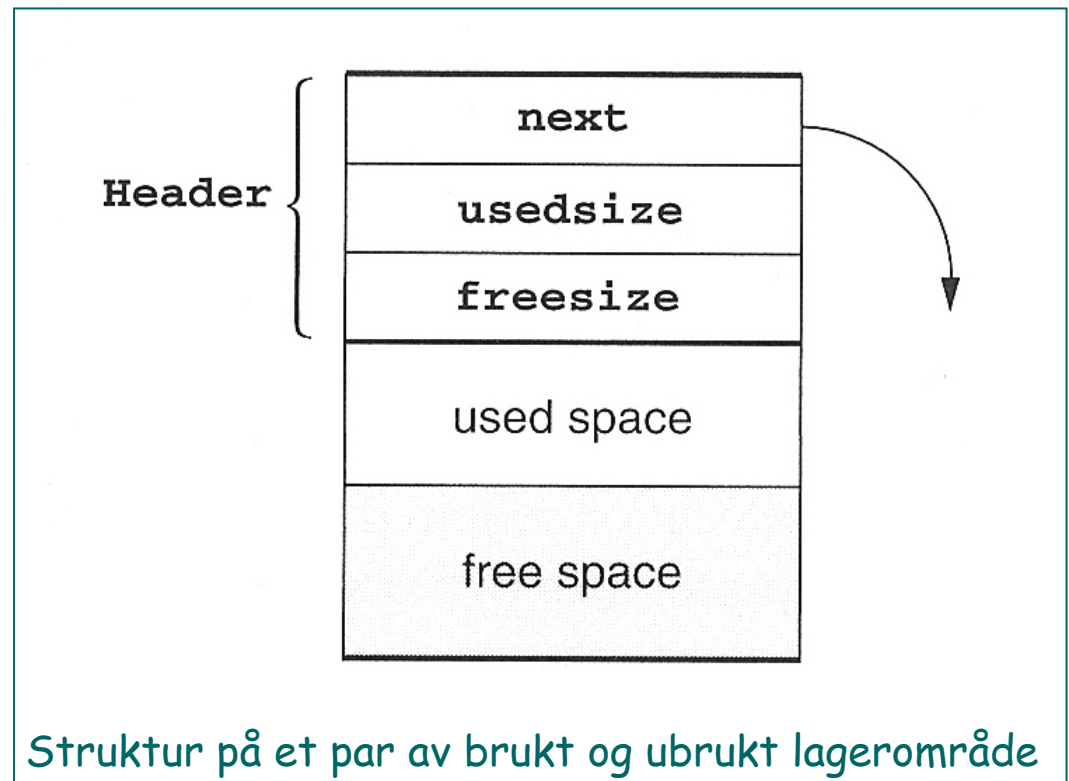
- når objekter ikke kan flyttes

```
#define NULL 0
#define MEMSIZE 8096 /* change for different sizes */

typedef double Align;
typedef union header
{ struct { union header *next;
          unsigned usedsize;
          unsigned freesize;
        } s;
  Align a;
} Header;

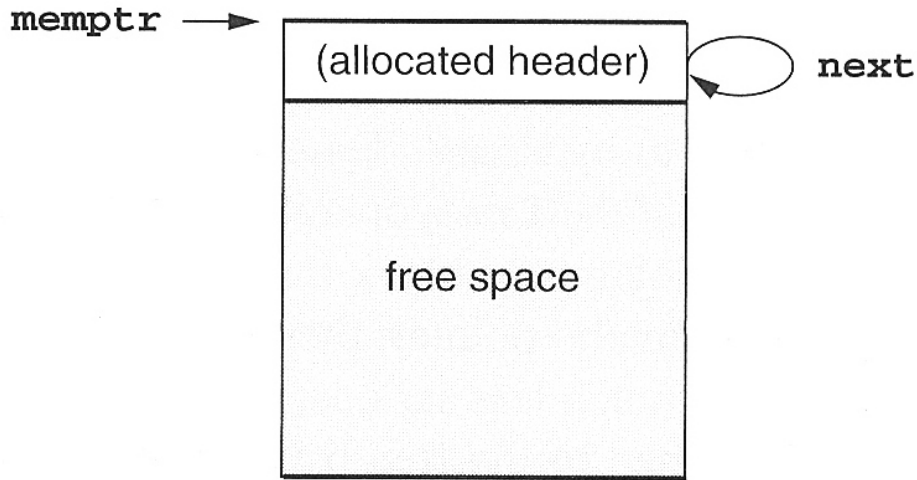
static Header mem[MEMSIZE];
static Header *memptr = NULL;
```

- Vi får noe lineær leting både ved reservering og ved frigivelse
- Finnes metoder der denne letingen bare blir logaritmisk (buddy-algoritmen)

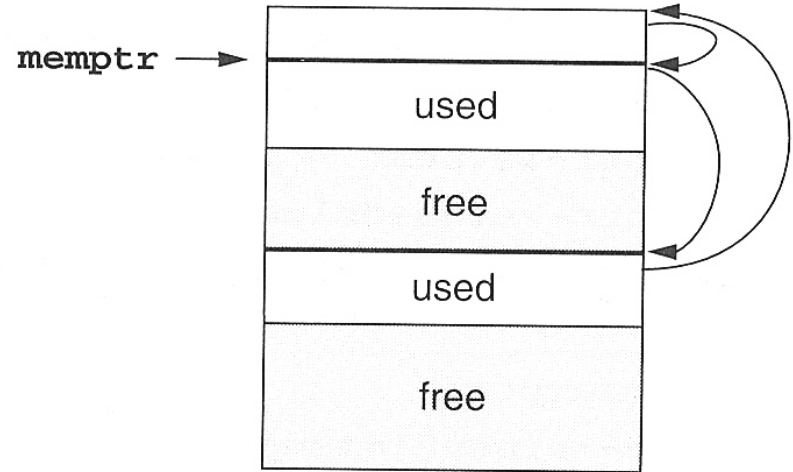


Struktur på et par av brukt og ubrukt lagerområde

1)

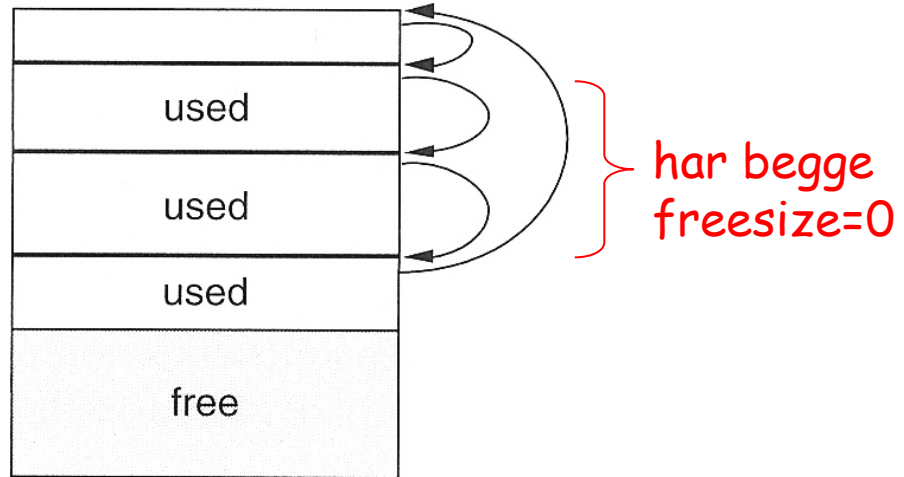


3)



2)

Peker til en eller annen blokk som har ledig plass'



```
void *malloc(unsigned nbytes)
```

```
{ Header *p, *newp;  
  unsigned nunits;  
  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;  
  if (memptr == NULL)  
  { memptr->s.next = memptr = mem;  
    memptr->s.usedsize = 1;  
    memptr->s.freesize = MEMSIZE-1;  
  }  
  for(p=memptr;  
      (p->s.next!=memptr) && (p->s.freesize<nunits);  
      p=p->s.next);  
  if (p->s.freesize < nunits) return NULL;  
  /* no block big enough */  
  newp = p+p->s.usedsize;  
  newp->s.usedsize = nunits;  
  newp->s.freesize = p->s.freesize - nunits;  
  newp->s.next = p->s.next;  
  p->s.freesize = 0;  
  p->s.next = newp;  
  memptr = newp;  
  return (void *) (newp+1);  
}
```

Leverer en utypet peker

Regn om til antal hele "hode-lengder" + 1

Initialisering

Skil ut et nytt område i bruk, med eget hode

Peker til hodet av den som skal frigjøres

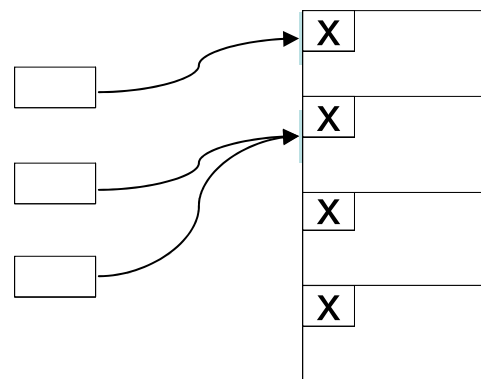
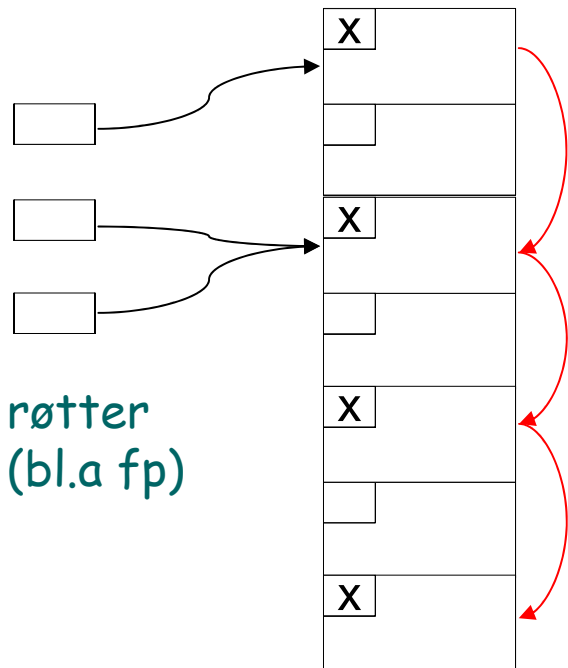
Let etter den som skal returneres, og husk forgjengeren (prev)

```
void free(void *ap)  
{ Header *bp, *p, *prev;  
  bp = (Header *) ap - 1;  
  for (prev=memptr, p=memptr->s.next;  
      (p!=bp) && (p!=memptr); prev=p, p=p->s.next);  
  if (p!=bp) return;  
  /* corrupted list, do nothing */  
  prev->s.freesize += p->s.usedsize + p->s.freesize;  
  prev->s.next = p->s.next;  
  memptr = prev;  
}
```

La den frigjorte inngå i fri-området til prev, og husk forgjengeren (prev)

Garbage Collection I

- Deler ut plass ukritisk så lenge det er plass. Når det ikke er plass mer tar vi en større opprydning
 - Starter alltid med et fullt rekursivt gjennomløp, der vi finner alle objekter som kan nåes fra variable vi kan nå direkte (røtter)
 - Alle objekter som kan nåes merkes. Krever eget bit i hvert objekt.



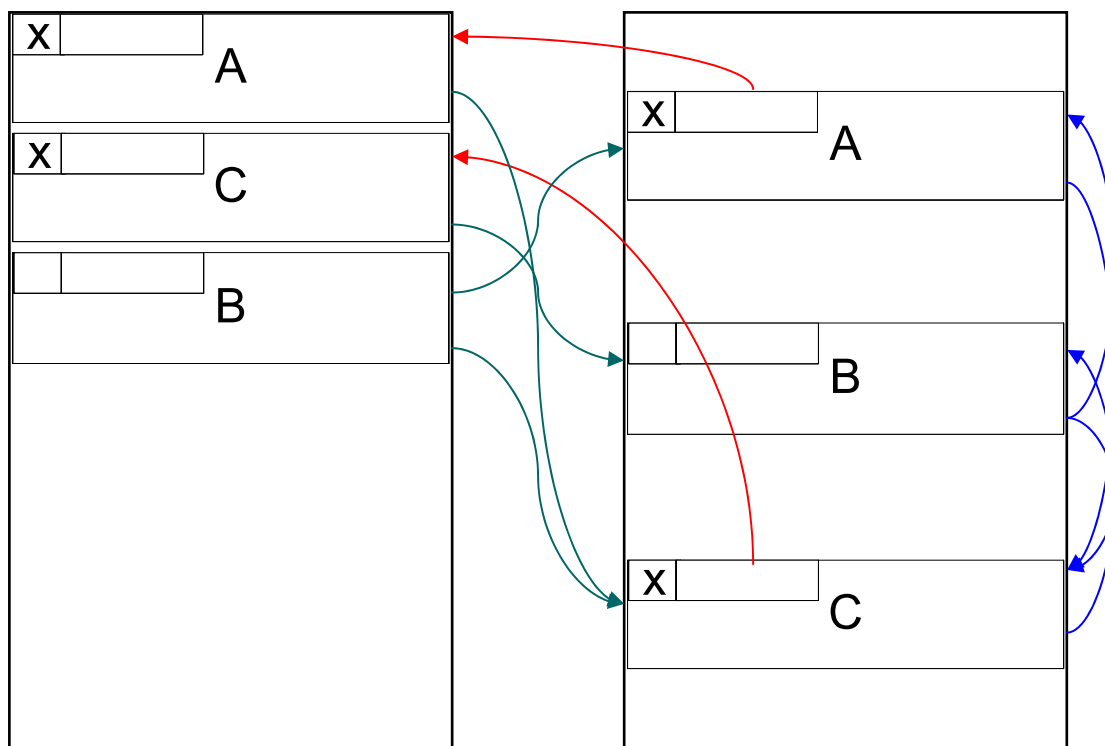
- NB (ikke nevnt i boken): Krever også at man kan finne ut hva som er pekere i et gitt objekt

Garbage Collection II

- Etter merkingen ("mark") gjøres et sekvensielt gjennomløp av lageret ("sweep"), der de umerkede objekter leveres tilbake
 - Må slå sammen ledig naboplass
 - Ledig plass holdes f.eks. i en eller fler frilister
- I stedet for "sweep" kan man gjøre "compaction": flytte alle objektene tett sammen.
 - NB: Da må man også forandre alle pekere til det stedet objektet flyttes til

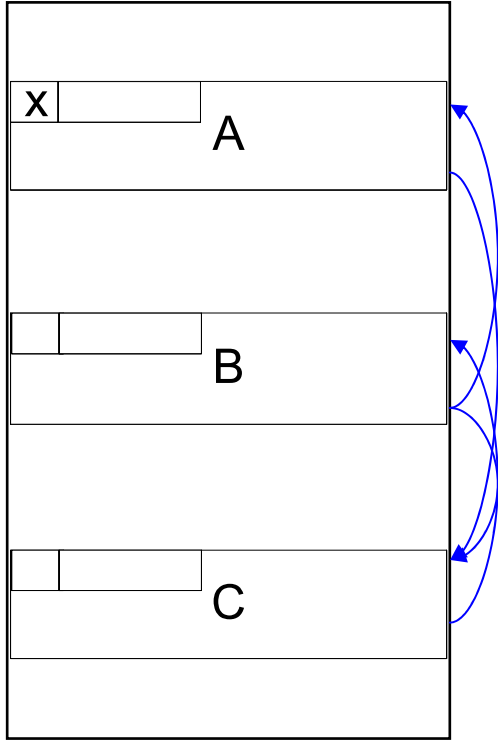
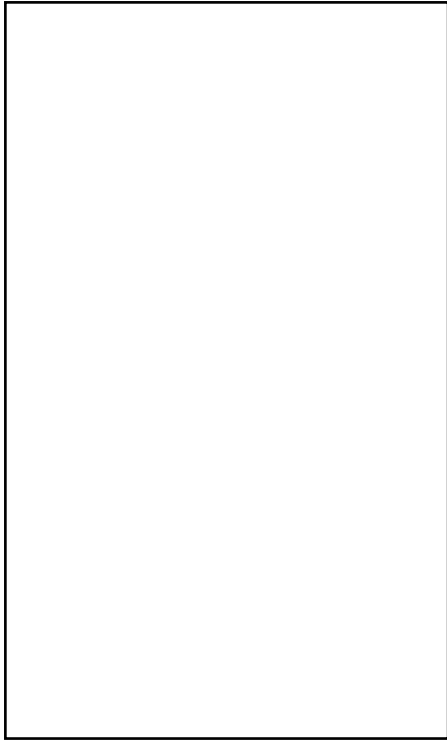
Garbage Collection III

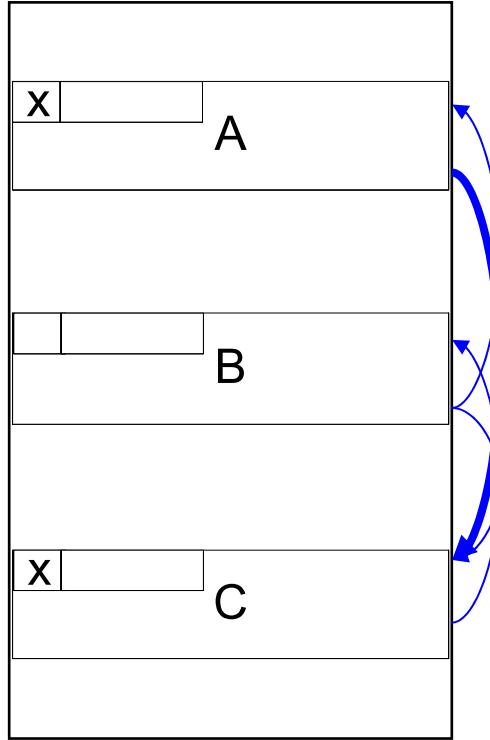
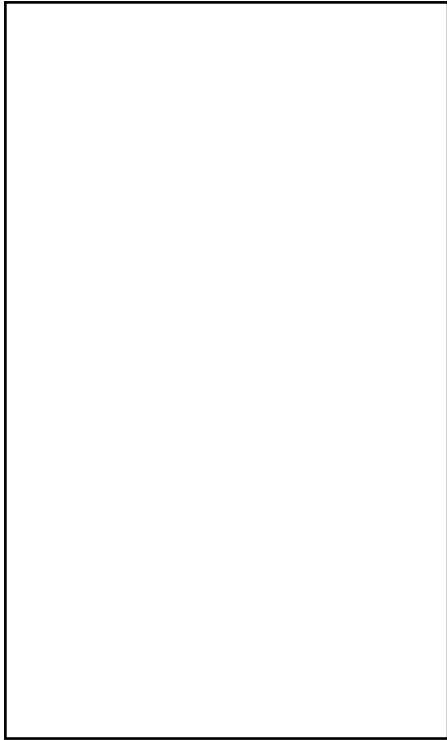
- To-delt lager
 - Deler plassen i to og bruker bare halvparten av gangen
 - "mark" og "compaction" kan da gjøres i ett rekursivt gjennomløp

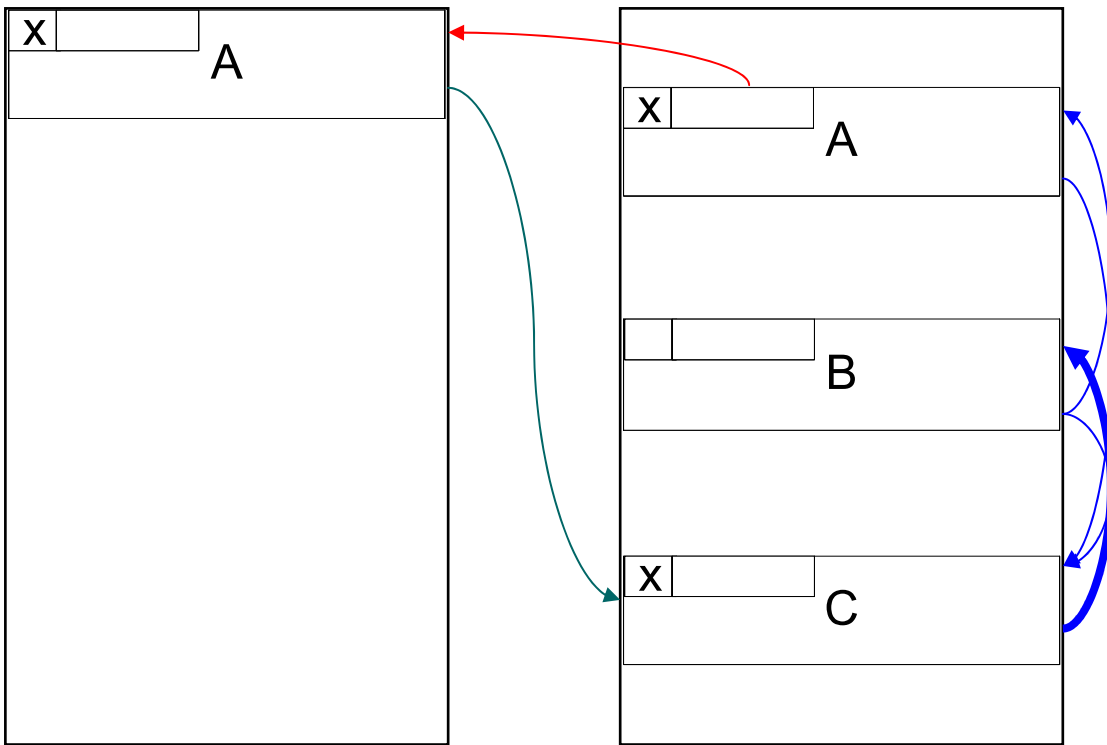


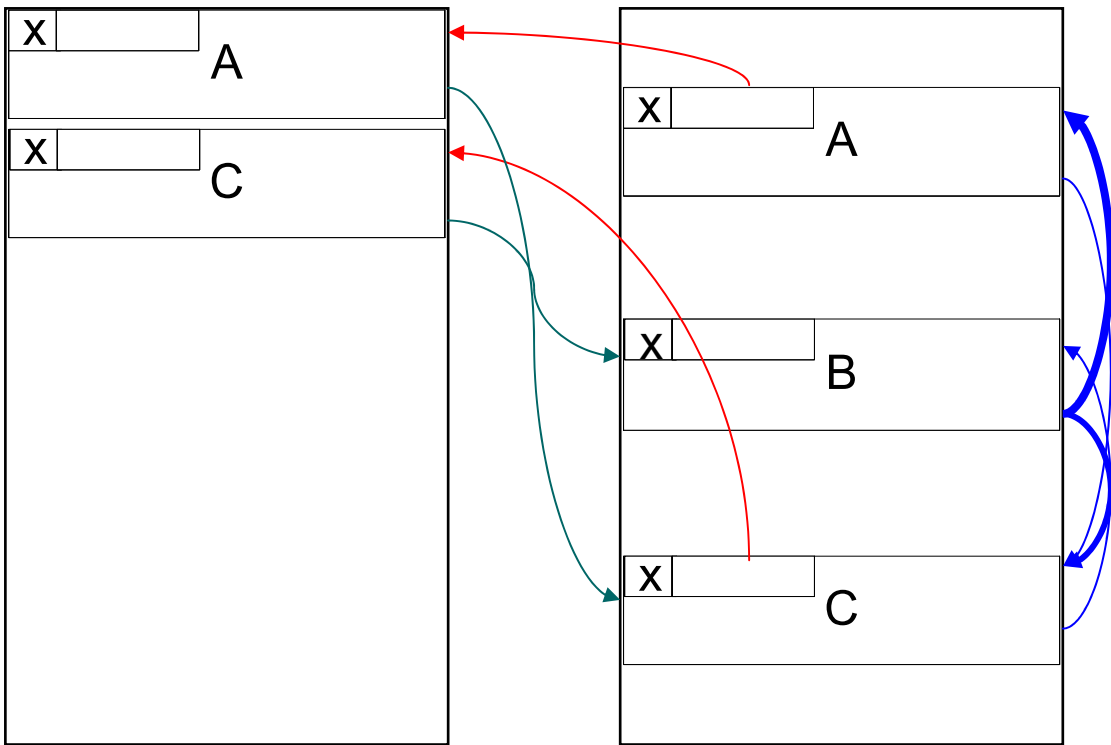
Hvert objekt må ha et ledig bit ("er flyttet")

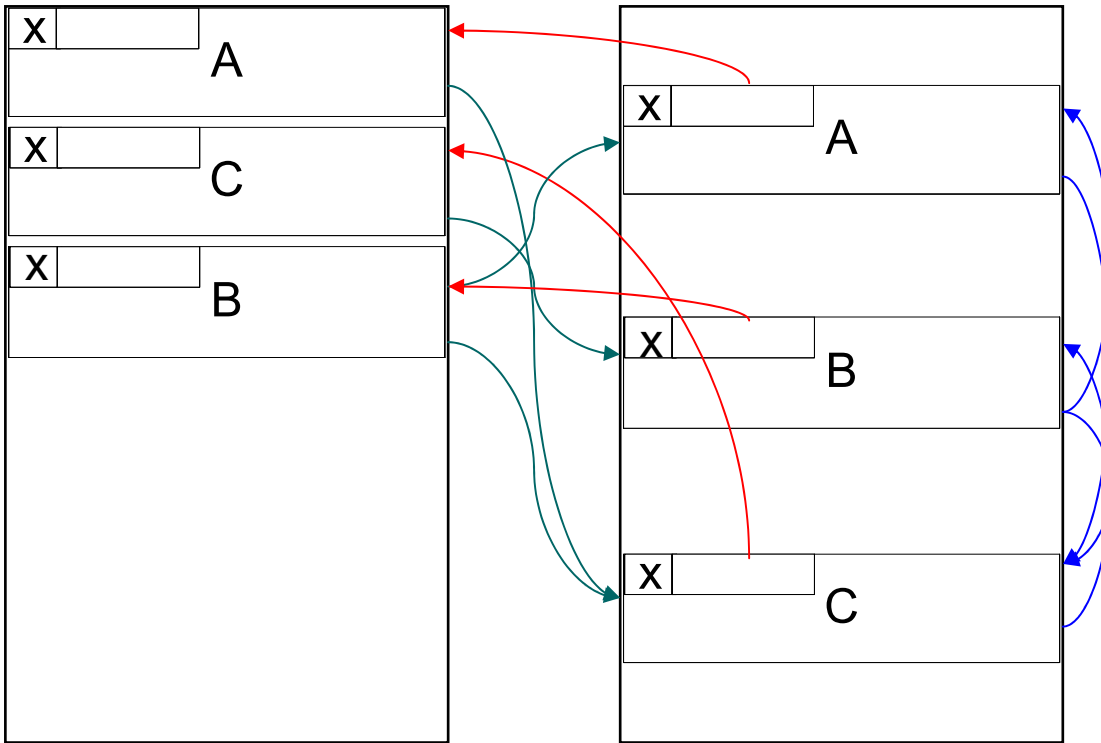
Da angir "neste ordet" adressen det er flyttet til











'by value' parameteroverføring (verdioverføring)

```
void inc2( int x)
/* incorrect! */
{ ++x; ++x; }
```

```
void inc2( int* x)
/* now ok */
{ ++(*x); ++(*x); }
```

Kall: `inc2(&y)`

- Hver formell parameter blir implementert som en lokal variabel i prosedyren
- Ved kall blir disse variable initialisert slik:
formell_var = aktuelt uttrykk
- I noen språk kan den formelle variabelen ikke forandres
- I C er dette eneste overføringsmåte. Man kan dog overføre pekere 'by value'

```
void init(int x[],int size)
/* this works fine when called
   as init(a), where a is an array */
{ int i;
  for(i=0;i<size;++i) x[i]=0;
}
```

Kall: `init(a)`

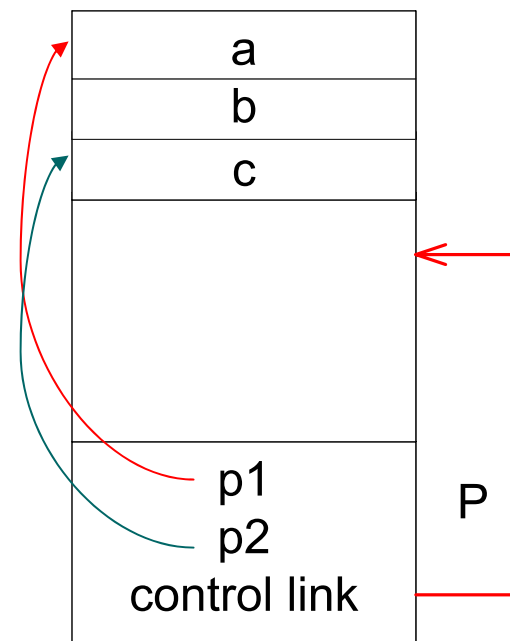
'by reference' parameteroverføring

- Overfører en peker/adresse til den aktuelle variabel
- Bare lov med variable som aktuelle parametere
- Fortran tillater imidlertid
 - P(5,b)
 - P(a+b,c)
- Passer til store datastrukturer

```
void inc2( int & x)
/* C++ reference parameter */
{ ++x; ++x; }
```

Kall: inc2(y)

```
void P(p1,p2) {
  ...
  p1 = 3
}
var a,b,c;
P(a,c)
```



'by value-result' parameteroverføring

- Bare variable kan være aktuelle parametere
- Det allokeres en lokal variabel, som ved 'by value'
- Ved kallet gjøres **formell_var = aktuell_var**
- Ved retur utføres **aktuell_var = formell_var**
- Kan gi effekt forskjellig fra 'by reference' (Men f.eks. Ada godkjenner også dette som en implementasjon)

```
void p(int x, int y)
{ ++x;
  ++y;
}

main()
{ int a = 1;
  p(a, a);
  return 0;
}
```

Eksempel på at det kan være forskjell på 'by reference' og 'by value-result'

'by name' parameteroverføring

- Den aktuelle parameteren blir substituert inn for den formelle ('nesten' rent tekstlig: den aktuelle parameteren beholder sitt skop, så altså ikke makro-ekspansjon)
- Om den aktuelle parameteren er et uttrykk blir det det ikke beregnet før man bruker parameteren inne i prosedyren (lazy evaluering)
- Men: Uttrykket blir beregnet om igjen hver gang
- Implementasjon
 - Se den aktuelle parameteren (f.eks. et uttrykk) som en lien prosedyre ('thunk')
 - Må optimaliseres for det tilfellet hvor parameteren er en enkel variabel (da er effekten som ved 'by reference')

```
void p(int x)           p(a[i])   ++a[i]
{ ++x; }
```

```
int i;
int a[10];

void p(int x)
{ ++i;
  ++x;
}

main()
( i = 1;
  a[1] = 1;
  a[2] = 2;
  p(a[i]);
  return 0;
}
```

'by name' eksempel

```
procedure P(par); name par; int par;  
begin  
  int x,y;  
  ...  
  par := x + y;      a)  
  ...  
  x := par + y;     b)  
  ...  
end;  
...  
P(v);  
P(r.v);  
P(5);  
P(u+v);
```

	v	r.v	5	u + v
a)	OK	OK	feil	feil
b)	OK	OK	OK	OK