C-programmering

Pekere, arrays, datastrukturer, minneallokering

TTK4235

Læringsmål for i dag

Kunnskap

- Grunnleggende (og litt praktisk) kunnskap om
  - Pekere i c (prinsipper, funksjonalitet og aritmetikk)
  - Nyttige datastrukturer (struct og union)
  - Minnemodellen i c-programmer

Ferdigheter

- Bruke kunnskapen i utvikling av tilpassede datasystemer
  - Når/hvordan bør pekere brukes/ikke brukes?
  - Når kan structs og union være nyttig?
  - Fornuftig minnehåndtering i tilpassede datasystemer

Agenda

Pekere
- Arrays
- Strings

Structs og Union
- Nyttige datatyper
- Typedef

Minnestruktur i et c-program
- Malloc og free
- Minnelekkasje

Pensum:
K & R, kap. 5 + 6
Kompendium av Hendseth, kap.3 (ikke 3.10!)

Vil holde fokus på de tema som er viktige for tilpassede datasystemer!

Grunnleggende om pekere

Pekere: hva/hvorfor?

Pekere er variable som inneholder minneadressen til andre variabler/funksjoner

Når bruker vi pekere?

Hvis vi vil bruke «pass by reference» ved funksjonskall – peker som argument

Som referanse til minne allokert dynamisk med *malloc*

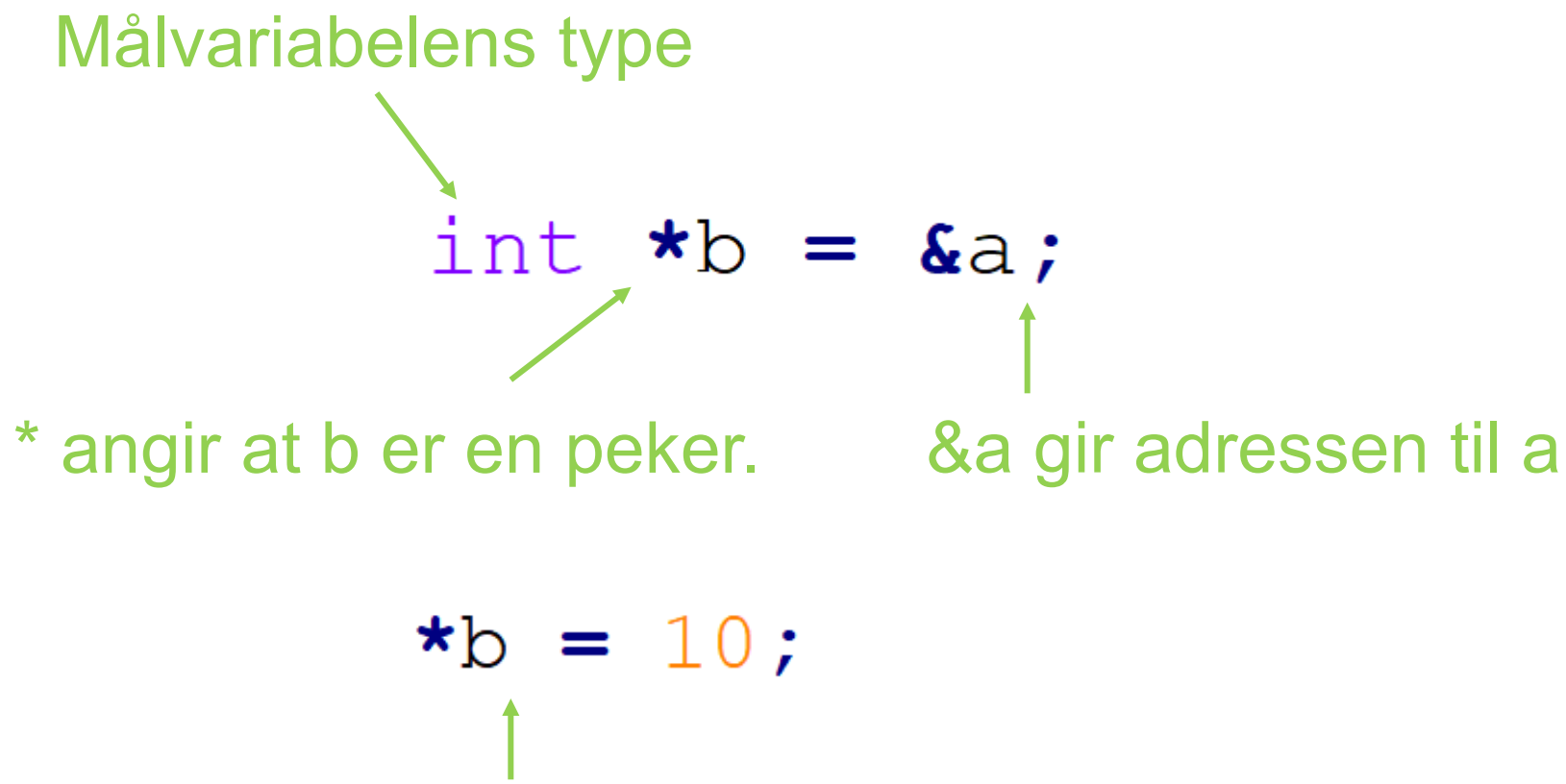
Pekere brukes ofte ved aksessering av arrays

Funksjonspekere

Tilpassede datasystemer: For operasjoner mot IO-registre (memory mapped IO)

OBS: feil bruk kan føre til kaos!

Pekere: notasjon



Pekere: enkelt eksempel

Eksempel:

```
int a=25;
int *b = &a;
printf("a = %d\n", a);
printf("*b = %d\n", *b);
```

Output:

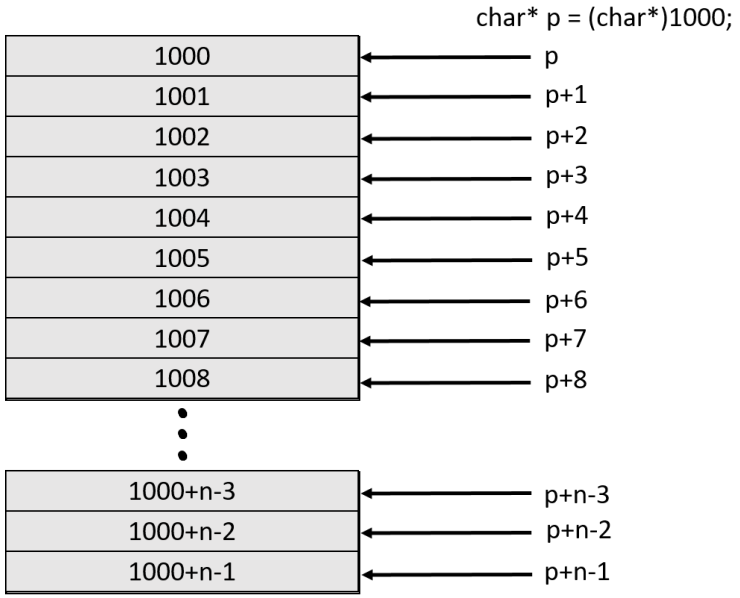
```
$ ./pekere.exe
a = 25
*b = 25
```

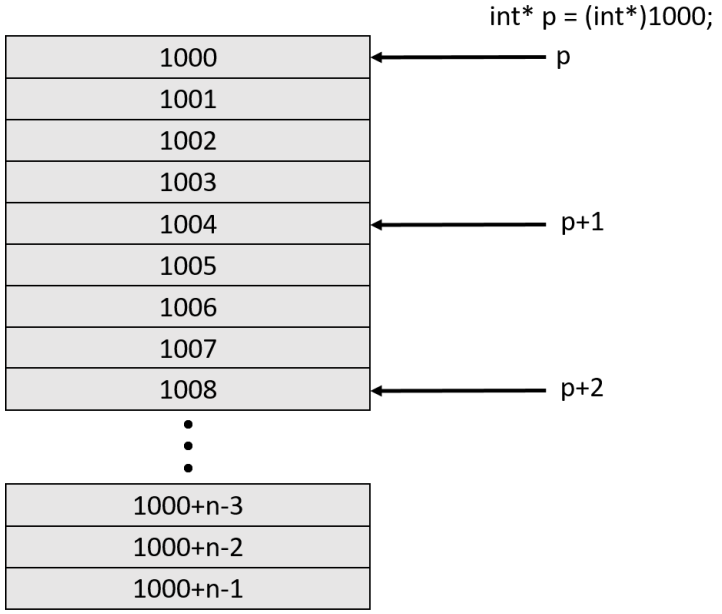
Pekere: flere notasjonseksempler

```
int x = 1, y = 2, z[10];
int *ip;      /*Peker til en int*/
ip   = &x;    /*ip peker nå til integerverdien x (=1)*/
y  = *ip;     /*integerverdien y settes til 1*/
*ip = 0;      /*x (som ip peker til) settes til 0*/
ip   = &z[0];  /*ip peker nå til z[0] som er første el. i z*/
```

Pekere: pekertyper

- Meningen av typen
  - `int*` betyr ”adresse til et heltall” / ”minneplasseringen av et heltall”
  - Kan derfor også bety ”adresse til det første av flere heltall”
  - Bestemmer forhold mellom pekerverdi og **bytes** i minnet
  - Mulig å bruke pekeren på samme måte som typen
    - «*p» kan brukes som en int når «int *p»
  - Men hva med voidpekere, `void*`?





Pekere: verdien av en peker

- Meningen til verdien til pekeren
  - Adressen kan være til **minne vi eier**
  - Adressen kan være **uinitialisert** (har tilfeldig verdi)
  - Adressen kan være **eksplisitt** ugyldig (NULL)
  - Adressen kan ha **blitt ugyldig**, f.eks. adressen til en lokal variabel i en funksjon som har returnert

```
int a = 25;
int *b = &a;

int *b;

int *b = NULL;
```

Peker som argument

C passerer egentlig alle argumenter som value

Enkelt case: bruke funksjon for å bytte to verdier

Effektivt med pekere!

Kan gjøres manuelt men blir mye mer bokholderi

```
void swap (int x, int y){
     int temp;

     temp = x;
     x = y;
     y = temp;

}

void main(void){
     int a = 1;
     int b = 2;

     swap(a,b);

}
```

```
void swap (int *px, int *py){
     int temp;

     temp = *px;
     *px = *py;
     *py = temp;

}

void main(void){
     int a = 1;
     int b = 2;

     swap(&a,&b);

}
```

Eksempel: peker som argument

Denne funksjonen

- Leser et tegn
- Setter verdien til variabelen *c peker på.
- Returnerer en statusverdi

```
/** Read a character from standard input.
    Return 0 if the character is an x.
    */
int readChar(char *c) {
    int value = getchar();
    *c = (char)value;
    return value != 'x';
}
```

Gitt en *char c* kan funksjonen kalles med:

- *&c*
- En eksplisitt peker mot *c*

```
char c;
while (readChar(&c)) {
    printf("-> %c\n", c);
}

char *d = &c;
while (readChar(d)) {
    printf("-> %c\n", *d);
}
```

Arrays og tekststrenger

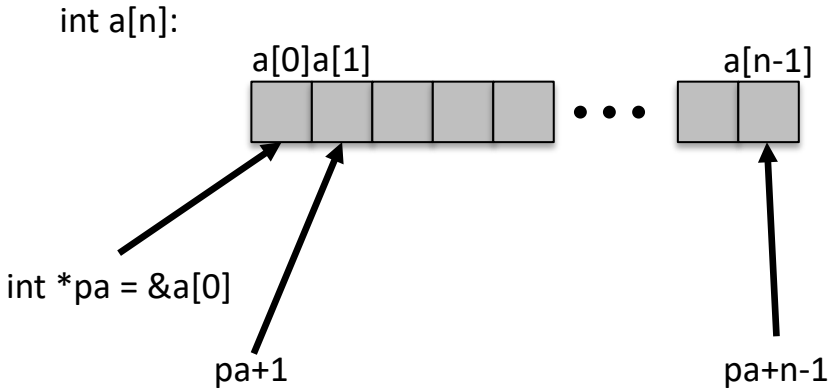
Arrays

Sekvensielt i minne

Alle elementer har samme datatype
- Og samme størrelse!

Oppslag er **0-indeksert!**
- Fordi minne er sekvensielt, slår vi opp med en ”offset”
  - En offset på 0 er da starten på arrayet
- Offset er antall *elementer* unna starten, ikke antall *bytes*
  - (Som for pekere)
- Hva skjer med index = -1?

Initialisering {} eller []




```
void main(void){
    int arr1[3];

    for (int i = 0;i<3;i++){
        arr1[i] = i;
    }
    int arr2[3] = {0,1,2};

}
```

Arrays

```c
#include <stdio.h>

void main(void){
    int array[10];
    for (int i =0; i<9;i++){
        printf("array[%d]: %p sizeof(array[%d]): %d\n",i,&array[i],i,sizeof(array[i]));

    }
}
```

```
$ ./a
array[0]: 0xffffcbf0 sizeof(array[0]): 4
array[1]: 0xffffcbf4 sizeof(array[1]): 4
array[2]: 0xffffcbf8 sizeof(array[2]): 4
array[3]: 0xffffcbfc sizeof(array[3]): 4
array[4]: 0xffffcc00 sizeof(array[4]): 4
array[5]: 0xffffcc04 sizeof(array[5]): 4
array[6]: 0xffffcc08 sizeof(array[6]): 4
array[7]: 0xffffcc0c sizeof(array[7]): 4
array[8]: 0xffffcc10 sizeof(array[8]): 4
```

Tekststrenger

- Tekststrenger er arrays av bokstaver
  - Terminert med en 0-byte
  - Hvordan finne lengden?
  - Kan defineres som «read only» eller i form av char-array:

```
char* s1 = ”hello”;     // minneadressen s1 er read-only
s1[1] = 'a';           // ugyldig

char s2[] = ”hello”;    // tar en kopi
s2[1] = 'a';           // gyldig
```

Pekere til pekere, multidimensjonale arrays, funksjonspekere

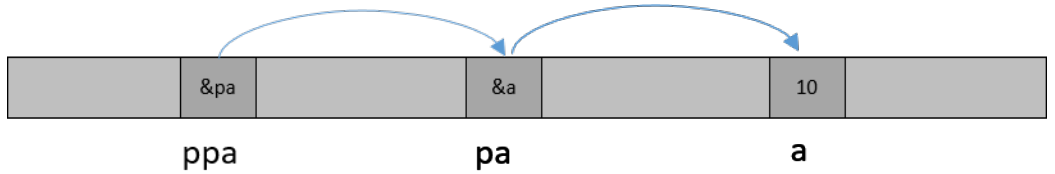
Pekere til pekere

En peker kan peke til en annen peker

Kan brukes til å holde oversikt over pekere til alle typer (f.eks.: int **p, char **p,…)

Praktisk triks for å f.eks. lage lister av tekststrenger

```
void main(void){
    int a      = 10;
    int *pa    = &a;
    int **ppa = &pa;
}
```




```
#include <stdio.h>

void main(void){
    char *names[3]=
    {
        "one\n",
        "two\n",
        "three\n"
    };

    for (int i = 0;i<3; i++){
        printf(names[i]);
    }
}
```

Multidimensjonale arrays

Angis som antall rader x antall kolonner

Kan i c ses på på lik linje med i andre språk

- Indekseres med rad og kolonne

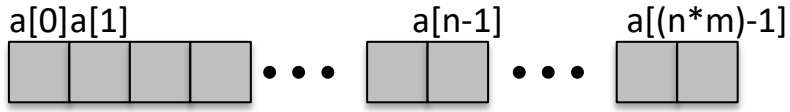
Men hva er forskjellen på dette og peker til peker?

- Bestemt antall elementer
- Fast minnestørrelse

```
int two_dim_array[10][3];

for (int i = 0;i<10;i++){
    for (int j = 0;j<3;j++){
        two_dim_array[i][j] = i*3+j;
    }
}
```

int a[n][m]:



Funksjonspekere

Peker på en funksjon

Nyttig når en har behov for å bruke forskjellige funksjoner under forskjellige betingelser

Kan være nyttig i tilpassede datasystemer

F.eks. tilstandsmaskin der funksjon som beskriver transisjon kan variere

```c
#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

void main(void)
{
     int a = 2;
     int b = 3;
     void (*fun_ptr)(int,int);

     for (int i = 0;i<3;i++){
          switch(i){
               case 0:
                    fun_ptr = &add;
                    break;
               case 1:
                    fun_ptr = &subtract;
                    break;
               case 2:
                    fun_ptr = &multiply;
                    break;
          }
          (*fun_ptr)(a,b);
     }
}
```

Funksjonspekere

Også i bruk ved interrupts

Koble interruptvektor med adresse til Interrup Service Routine (ISR)

ISR == funksjon

Interrupttabell = liste med funksjonspekere

```
49      /* Hvis programmet stopper her, har du funnet en bug
50      avr_break();
51  }
52
53  #pragma vector=TWI_vect
54  __interrupt void twi_interrupt( void )
55  {
56      unsigned char status = TWSR & 0xf8;
57      static unsigned char i = 0;
58
59      /* Dersom current == NULL skal ikke TWI-avbrudd være
60         og vi bør dermed aldri komme hit */
61      assert( current );
62
63      /* Tilstandsmaskin som sender all informasjon i curr
64      switch( status ) {
```

```
14  void init_adc(unsigned char inngang)
15  {
16    ADMUX = (1 << REFS0) |
17            (1 << ADLAR) |
18            (inngang & 0x07);
19
20    ADCSR = (1 << ADEN) |
21            (1 << ADSC) |
22            (1 << ADFR) |
23            (1 << ADIE) ;
24  }
25  #pragma vector=ADC_vect
26  __interrupt void speilegg(void)
27  {
28      dac_write(0, ADCH);
29  }
30
31  void main( void )
32  {
33      //TCCR0 = 4;
34
```

Pekeraritmetikk

Pekeraritmetikk

Datatypen angir typen til variablene pekeren skal peke på.
Merk: Antall bytes i datatypen tas hensyn til ved pekeraritmetikk

Pekere kan behandles med visse operatorer:

- **++** / **--** flytter pekeren til neste/forrige element i minnet
- **+** / **-** tillater å adressere et antall elementer etter eller foran i minnet
- *Dersom pekerne peker på elementer innenfor samme array:*
  - **-** tillater subtraksjon av to pekere for å finne antall minneelementer mellom dem
  - **==**, **!=**, **<**, **>**, **>=** og **<=** kan brukes til å sammenlikne pekere
- Addisjon av to pekere er **ikke** tillatt

Pekeraritmetikk: aksess til elementer

```c
#include <stdio.h>
#include <stdlib.h>

void main(void){
    int* p1 = malloc(sizeof(int)*3);
    int* p2;

    p1[0] = 1;
    p1[1] = 5;
    p1[2] = 10;

    p2 = p1;

    for (int i = 0;i<3;i++){
        printf("------%d-------\n",i);
        printf("p1 elem %d = %d\n",i,p1[i]);
        printf("p1 elem %d = %d\n",i,p1+i);
        printf("p1 elem %d = %d\n",i,*p1+i);
        printf("p1 elem %d = %d\n",i,*(p1+i));
        printf("p2 elem %d = %d\n",i,*p2);
        p2++;
    }
}
```

```
martinfo@NTNU19160 ~/src/pointer
$ gcc p_arithm_1.c

martinfo@NTNU19160 ~/src/pointer
$ ./a.exe
------0-------
p1 elem 0 = 1
p1 elem 0 = 928
p1 elem 0 = 1
p1 elem 0 = 1
p2 elem 0 = 1
------1-------
p1 elem 1 = 5
p1 elem 1 = 932
p1 elem 1 = 2
p1 elem 1 = 5
p2 elem 1 = 5
------2-------
p1 elem 2 = 10
p1 elem 2 = 936
p1 elem 2 = 3
p1 elem 2 = 10
p2 elem 2 = 10
```

Structs og union

Structs

Samling av variabler, vilkårlig antall og typer

Deklarasjoner, tar ingen lagringsplass, f.eks.:

```
struct point {
    int x;
    int y;
    double value;
};
```

Men instansiering av en struct gjør det

Kan initialiseres direkte:

```
struct point mypoint = {100, 150, 9.8};
```

Variablene («feltene») aksesseres med: *struct.navn*:

```
printf("mypoint.x = %d\n", mypoint.x);
printf("mypoint.y = %d\n", mypoint.y);
printf("mypoint.value = %f\n", mypoint.value);
```

Structs

Variablene i en struct kan ha samme navn som variable utenfor kontekst:

Kan også ha en struct i en struct

Aritmetikk:

kopiering, finne adresse(&), manipulere elementer

Direkte sammenlikning ikke mulig!

```c
struct point{
     int x;
     int y;
};
struct rectangle {
     char *color;
     struct point p1;
     struct point p2;
};
void main(void){
     int x,y,z;

     struct point p1;

     p1.x = 0;
     p1.y = 0;
     struct point p2 = {100,50};

     struct rectangle r1 = {"red",p1,p2};

     struct rectangle r2 = r1;

     struct rectangle *ptr = &r1;

     if(r1 == r2){  /* dette går ikke */
          printf("They are the same\n");
     }

}
```

Pekere til structs

Pekere for structs defineres som for andre datatyper:

```
struct point *pp = &mypoint;
```

For å aksessere feltene kan pekeren derefereres som vanlig:

```
printf("(%d, %d)\n", (*pp).x, (*pp).y);
```

Men struct-pekere brukes så mye at C også har en egen notasjon:

```
printf("(%d, %d)\n", pp->x, pp->y);
```

Pekere foretrekkes ofte fremfor kopier av structer som input til funksjoner, hvorfor?

Structer kan ta _*mye*_ plass i minnet

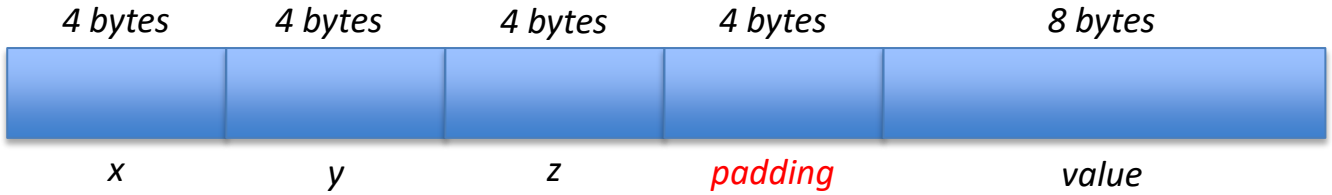
Structs i minnet

```
struct point {
    int x;
    int y;
    int z;
    double value;
};
```

En struct legges ut i minnet etter visse regler:

- Variablene lagres i rekkefølgen de deklareres
- Hver variabel må starte på en offset som er delelig på sin egen størrelse (***sizeof()***) – hvis nødvendig «paddes» det med bytes imellom

Dersom *sizeof(int)=4* og *sizeof(double)=8* vil denne structen bli slik:



Structs og lenkede lister

Er mulig å lage lenkede lister med structer

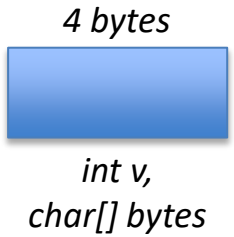
- En struct peker til andre structer

```
struct menuelem {    /* menyelement-struct */
    char *name;/* tekst i meny */
    struct menuelem *next;/* neste el */
    struct menuelem *previous;/* forrige */
};
```

- Når kan dette være relevant for tilpassede datasystemer?
  - F.eks. lage navigasjonsmeny
    - Eksempel i boka
    - Godt men litt komplisert

Union

Samling av flere måter å aksessere det samme minneområdet på



F.eks:

```
typedef union {
    int v;
    char bytes[sizeof(int)];
} Tall;|
```

Kan initialiseres ved å sette verdiene på enten den ene eller den andre måten:

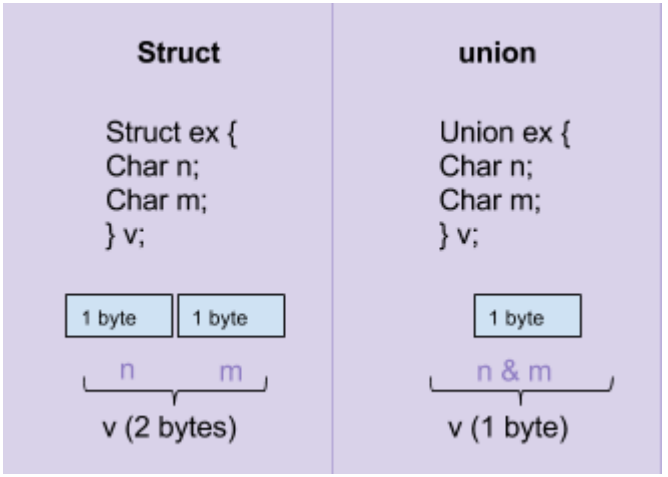
```
Tall t;
Tall t2;

t.v = 258;
t2.bytes[0] = 2;
t2.bytes[1] = 1;
```

Union

Nyttig verktøy når en skal anvende et minneområde til flere formål…

….men hvorfor er dette nyttig for tilpassede datasystemer?



Redusert minnebruk for midlertidige variable:

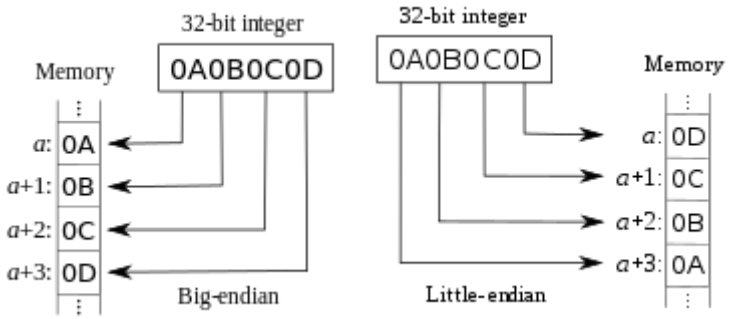
```
union temp{
    int som_int;
    long som_long;
    char som_char;
    short som_short;
} skisseblokk;
```

Sentralt verktøy i kommunikasjon, f.eks. canbus

Men veldig viktig å ikke bruke for permanent lagring av viktige data!

Union

Endianness er helt sentralt!



Eksempel:

```
martinfo@NTNU19160 ~/src/union
$ gcc union.c

martinfo@NTNU19160 ~/src/union
$ ./a.exe
Byte 0 i t er 0
Byte 1 i t er 0
Byte 2 i t er 1
Byte 3 i t er 2
Verdien av t2 er 33619968
```

```c
#include <stdio.h>

typedef union {
      int v;
      char bytes[sizeof(int)];
} Tall;

void main(void){
      Tall t;
      Tall t2;

      t.v = 258;
      t2.bytes[0] = 2;
      t2.bytes[1] = 1;
      for (int i=0;i<sizeof(int);i++){
            printf("Byte %d i t er %d\n",i,t.bytes[i]);
      }
      printf("Verdien av t2 er %d\n",t2.v);
}
```

Typedef

Nyttig verktøy for å gjenbruke definisjoner av typer

Spesielt nyttig for structs, unioner…

…men også helt sentralt for å spesifisere variable for tilpassede datasystemer

- f.eks. for å sikre riktige verdier til registre

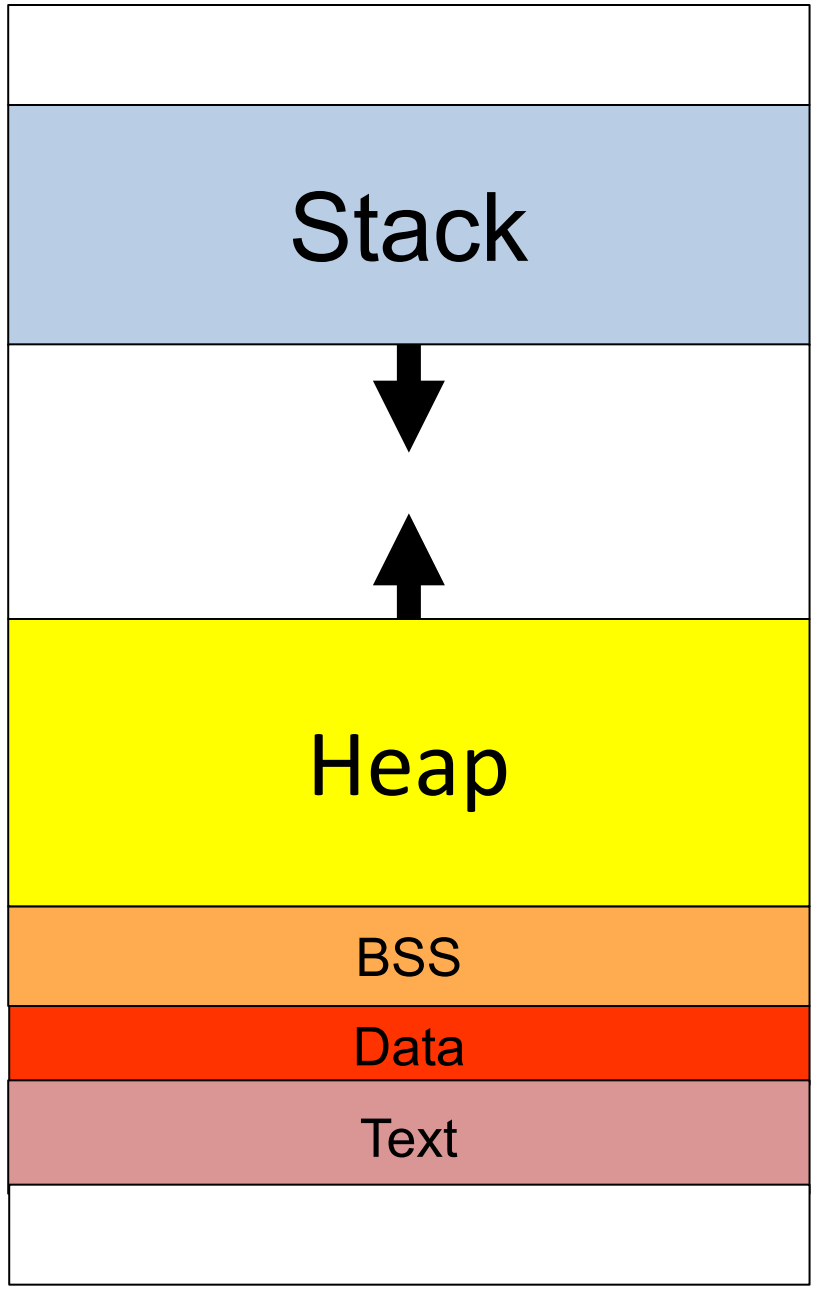
```
struct point{
    int x;
    int y;
};
typedef struct {
    char *color;
    struct point p1;
    struct point p2;
} r_struct;
typedef unsigned char reg_8bit;

void main(void){
    r_struct r1;
    reg_8bit reg1;
}
```

Minnehåndtering, -allokering og –frigjøring i c

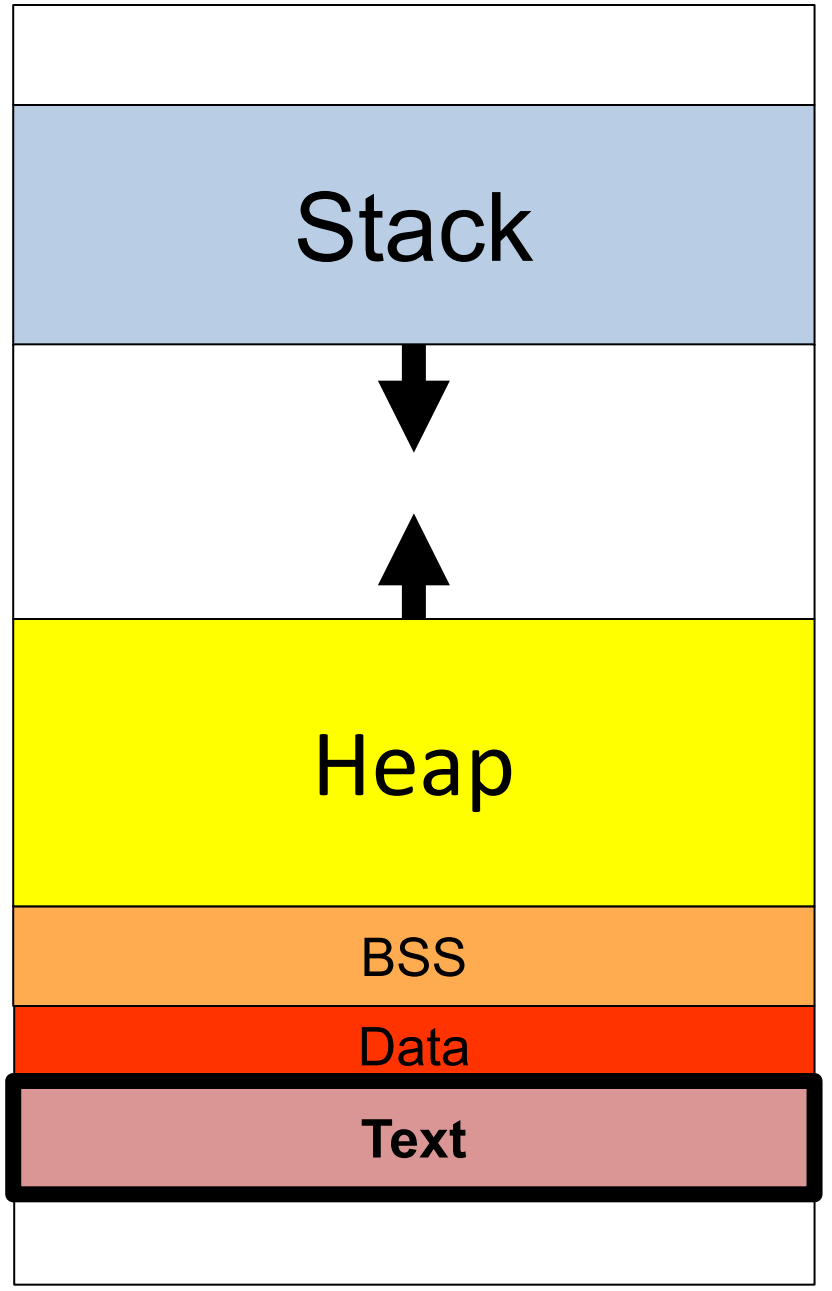
Recap om minne-strukturen i c

- Når du kjører et C-program, legges *bildet* i RAM
- Kalles ”Memory layout”
- Typisk *organisering*



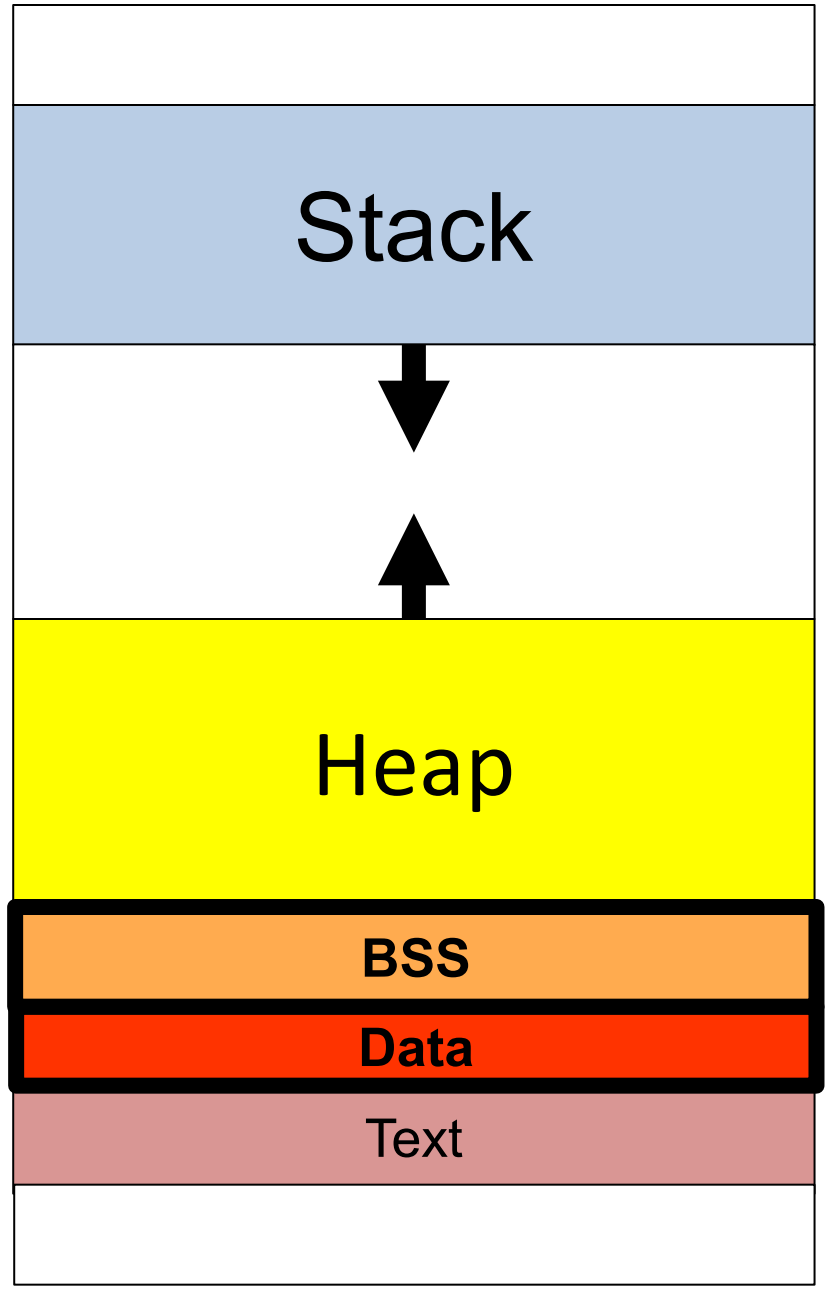
Minnehåndtering *Text*

- Innholder
  - Instruksjoner
- Delbart mellom prosesser
- Vanligvis ”*read-only*”
  - Hvorfor?



Minnehåndtering *Data og BSS*

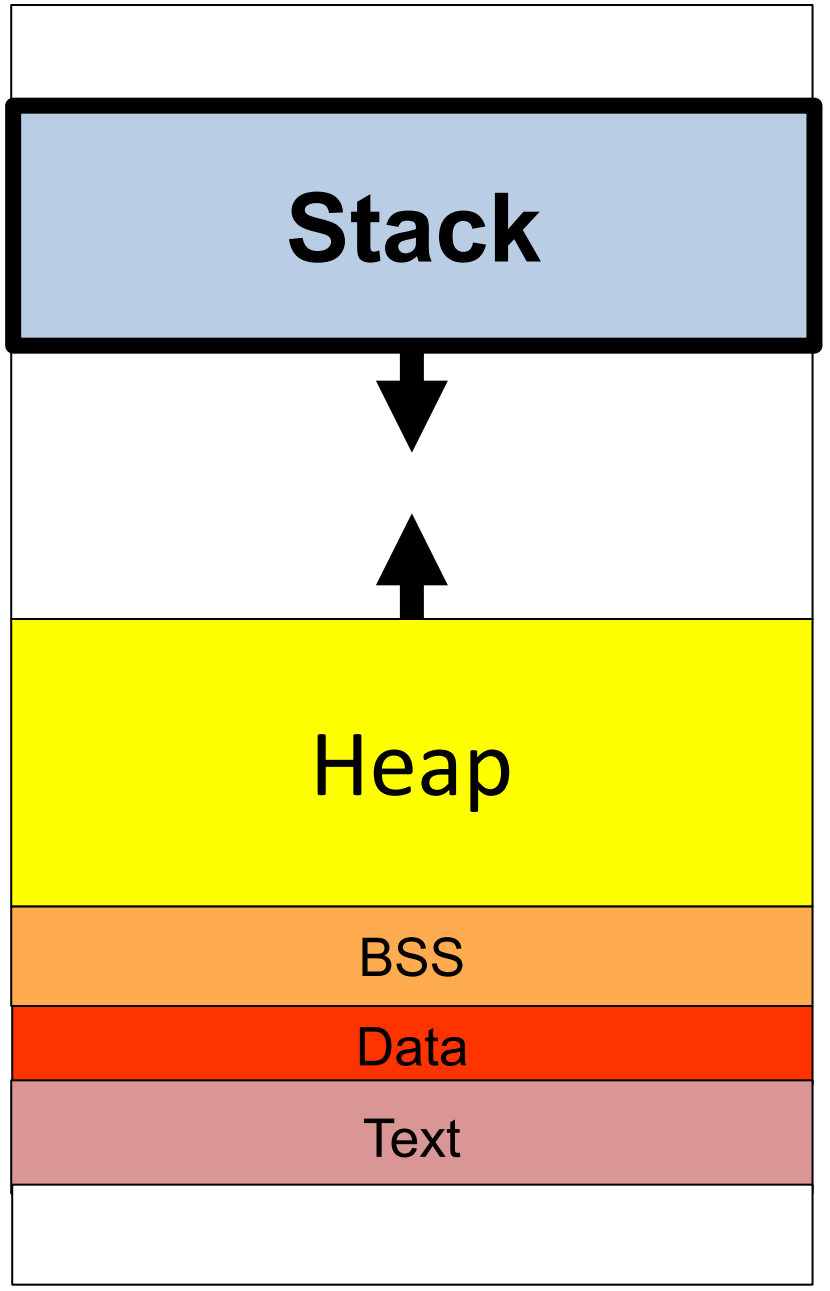
- Data inneholder:
  - Initialisert data
    - Globale og statiske variabler eksplisitt initialisert
- BSS inneholder:
  - Uinitialiserte variabler
    - Typisk fylt med ”0”
- Bruker ikke ”faktisk” plass i .o-filer



Minnehåndtering *Stack*

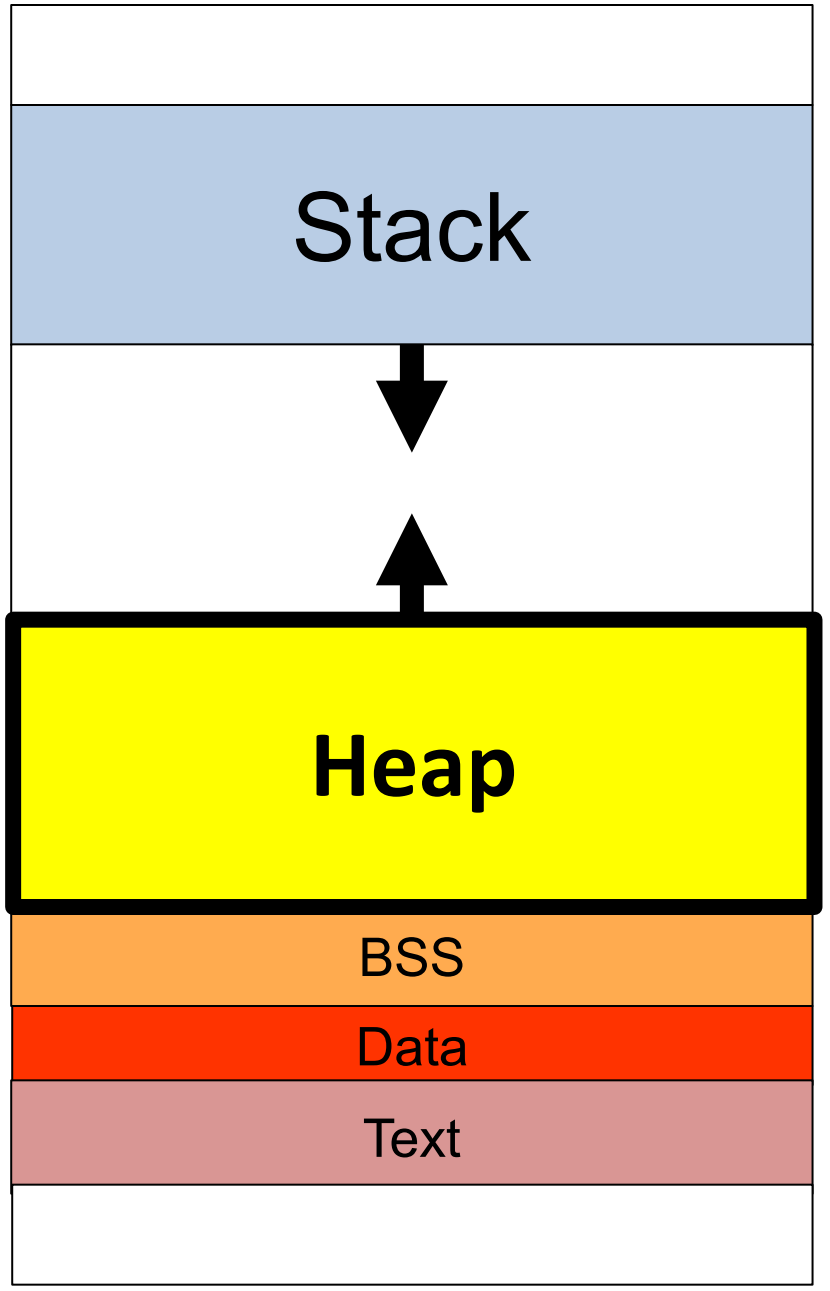
- Inneholder:
  - Lokale variabler (også funksjonspekere osv.)

- ”Stack-frames” lages når funksjoner kalles
- Disse slettes igjen når funksjoner returnerer



Minnehåndtering *Heap*

- Inneholder:
  - Alt ”dynamisk” allokert (malloc(), calloc(), realloc()))
- Unngå at heap vokser inn i stack!
- Minnelekkasjer...
- ”Heap fragmentation”



Malloc/free

Viktige funksjoner for å styre minnebruk:

malloc(): setter av et minnesegment til variabel (**realloc**() kan brukes til å endre på segmentet)

free(): frigjør minnet for en variabel

Tilsvarer «**new**» og «**delete**» i c++

I andre språk (f.eks. java) gjøres «delete» automatisk

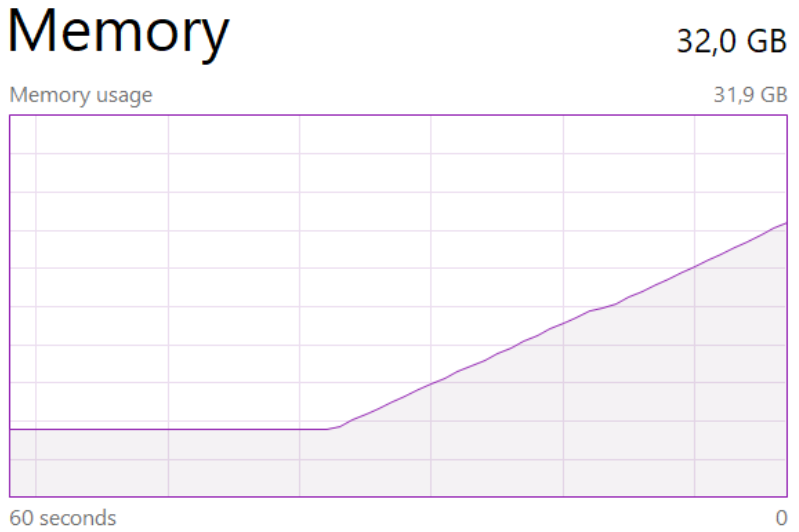
Malloc/free

Om malloc() brukes uten free() kan en få minnelekkasje

Enkelt eksempel:

```
#include <stdlib.h>

void main(void){
    int* p;
    for (int i=0;i<1000000000;i++){
        p = malloc(sizeof(int)*2);
        p[0] = 1;
        p[1] = 2;
        // Do something useful with
value here
    }
}
```



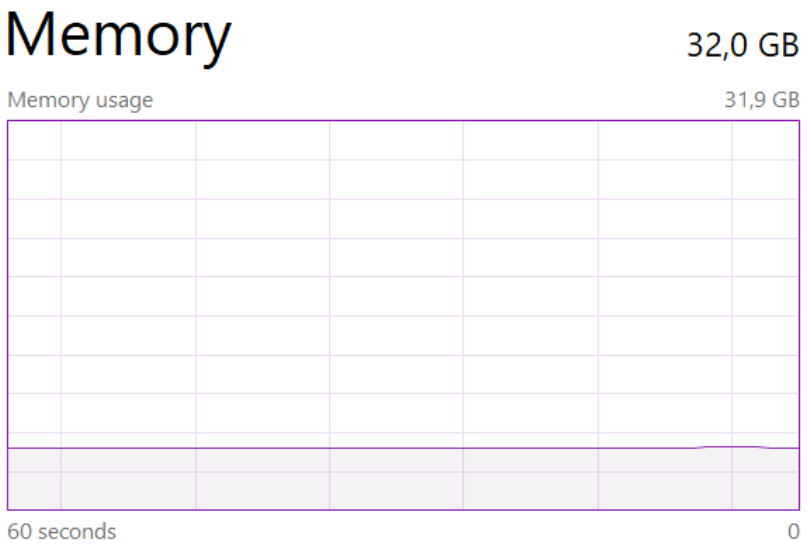
Malloc/free

Om malloc() brukes uten free() kan en få minnelekkasje

Enkelt eksempel, med free():

```
#include <stdlib.h>

void main(void){
    int* p;
    for (int i=0;i<1000000000;i++){
        p = malloc(sizeof(int)*2);
        p[0] = 1;
        p[1] = 2;
        // Do something useful with
value here
        free(p);
    }
}
```



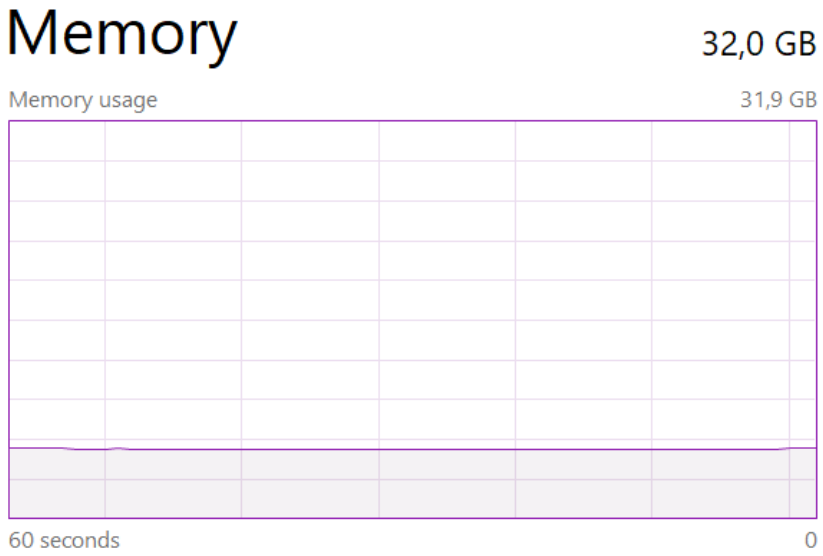
Malloc/free

Om malloc() brukes uten free() kan en få minnelekkasje

Enkelt eksempel, med preallokering:

```
#include <stdlib.h>

void main(void){
    int* p = malloc(sizeof(int)*2);
    for (int i=0;i<1000000000;i++){
        p[0] = 1;
        p[1] = 2;
        // Do something useful with
value here
    }
}
```



Minnelekkasje er et enda større problem i tilpassede datasystemer.