Dagens tema

- Vektorer (array-er)
- Tekster (string-er)
- Adresser og pekere
- Dynamisk allokering

Vektorer

Alle programmeringsspråk har mulighet til å definere en såkalte **vektor** (også kalt **matrise** eller «array» på engelsk). Dette er en samling variable av samme type hvor man bruker en **indeks** til å skille dem.

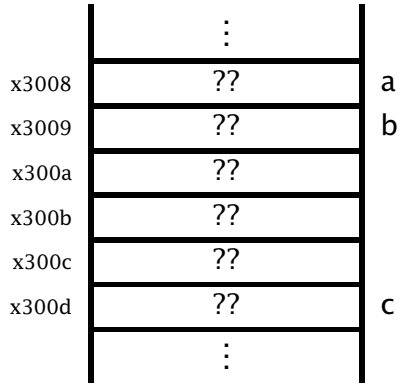
Deklarasjon

I C deklareres vektorer ved å sette antallet elementer i hakeparenteser etter variabelnavnet:

```
char a, b[4], c;
```

Antallet elementer må være en *konstant*.

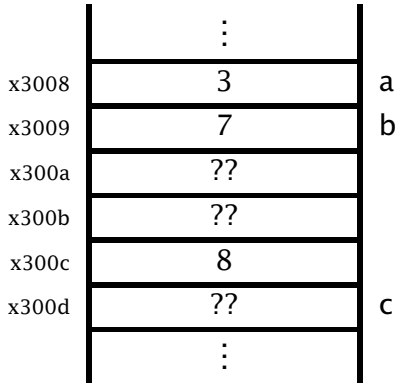
char a, b[4], c;



Bruk

a = 3;
b[0] = 7; b[a] = 8;

Etter dette er situasjonen:



Beregning av adresse

Adressen til vanlige variable er kjent[1] men adressen til vektorelementer må beregnes. Formelen er

$$\textit{Startadresse} + \textit{Indeks} \times \textit{Størrelse}$$

Hva skjer med ulovlig indeks?

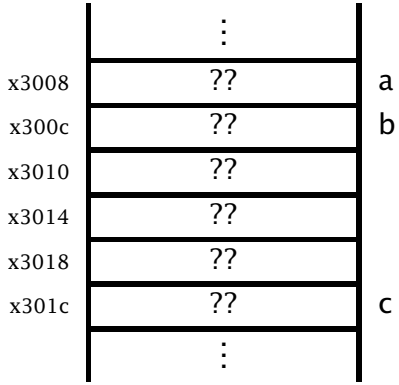
I C sjekkes ikke indeksen. Dette gjør det mulig å ødelegge andre variable, kode eller i noen tilfelle hele systemet.

[1] Dette er ikke helt sant, men vi kan tro det er slik en stund.

Størrelse over 1 byte

Anta at int er 4 byte.

int a, b[4], c;



Tekster

I C lagres tekster som tegnvektorer med en spesiell konvensjon: Etter siste tegn står en byte med verdien 0.[2]

Variable

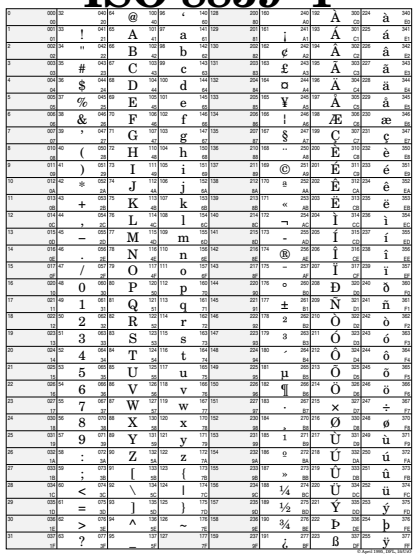
Når man deklarerer en tekstvariabel, må man angi hvor mange tegn det er plass til (samt plass til 0-byten).

```
char str[6];
```

Tekstvariabel str har plass til 5 tegn.

[2] En byte med verdien 0 er ikke det samme som sifferet «0»; sifferet «0» er representert av verdien 48, som vist på neste lysark.

ISO 8859–1

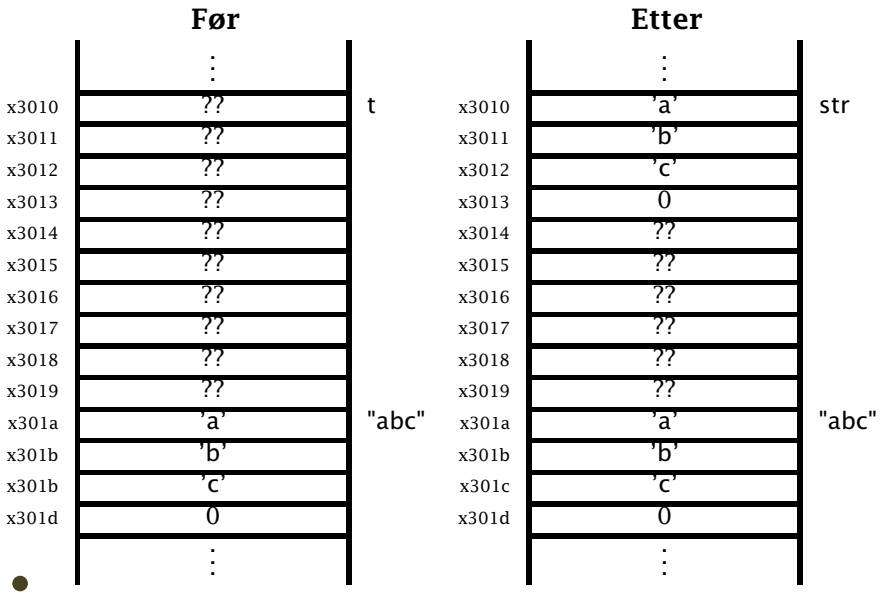


Kopiering av tekst

Flytting av tekst skjer med standardfunksjonen strcpy:

```
char *strcpy (char til[], char fra[])
{
  int i = 0;

  while (1) {
    til[i] = fra[i];
    if (fra[i] == 0) return til;
    ++i;
  }
}
```

Et eksempel

```
#include <stdio.h>

int main (void)
{
  char t[10];
  int i;

  strcpy(t, "abc");
  for (i = 0;  i < 10;  ++i) {
    printf("t[%2d] = %4d = '%c'\n", i, t[i], t[i]);
  }
  return 0;
}
```


Her er utskriften fra kjøringen:

```
t[ 0] =   97 = 'a'
t[ 1] =   98 = 'b'
t[ 2] =   99 = 'c'
t[ 3] =    0 = ''
t[ 4] =  -12 = 'ô'
t[ 5] =  -17 = 'ï'
t[ 6] = -114 = ''
t[ 7] =    0 = ''
t[ 8] =   68 = 'D'
t[ 9] = -107 = ''
```

Andre tekstoperasjoner

strlen(str) beregner den nåværende lengden av teksten i str. (Dette gjør den ved å lete seg frem til 0-byten.)

strcat(str1,str2) utvider teksten i str1 med den i str2.

strcmp(str1,str2) sammenligner de to tekstene.
Returverdien er

< 0	om str1 < str2
0	om str1 = str2
> 0	om str1 > str2

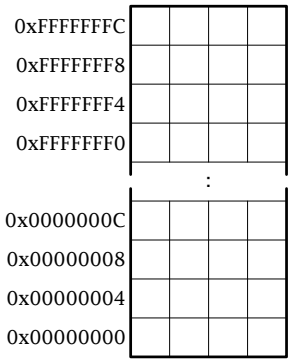
sprintf(str, "...", v1, v2, ...) fungerer som printf men resultatet legges i str i stedet for å skrives ut.

Hva om teksten er for lang?

Siden tekstvariable er vektorer, er det ingen sjekk på plassen. Det er derfor fullt mulig å ødelegge for seg selv (og noen ganger for andre).

Variable, adresser og pekere

Variable ligger lagret i *hurtiglageret* (ofte kalt *RAM*) i en eller annen adresse.



Operatoren &

I C kan man få vite i hvilken adresse en variabel ligger ved å bruke operatoren &.

```
#include <stdio.h>

int a, b, c;

int main(void)
{
  printf("Skriv to tall: ");
  scanf("%d", &a);  scanf("%d", &b);
  c = a + b;
  printf("Summen er %d.\n", c);

  printf("I adresse %08x ligger a med verdien %d.\n", &a, a);
  printf("I adresse %08x ligger b med verdien %d.\n", &b, b);
  printf("I adresse %08x ligger c med verdien %d.\n", &c, c);
}
```

La oss kjøre dette programmet:

```
Skriv to tall: 47 9
Summen er 56.
I adresse 00020e00 ligger a med verdien 47.
I adresse 00020e04 ligger b med verdien 9.
I adresse 00020e08 ligger c med verdien 56.
```

NB Det kan variere fra gang til gang hvilke adresser man får.
Her ser vi at variablene ligger pent etter hverandre og at hver av dem opptar 4 byte.

Adressevariable

I C kan vi legge adresser i variable; disse deklareres med en stjerne:

```
int v, *p;
```

Her er v en vanlig variabel mens p er en adresse som kan peke på int-variable. (Vi må alltid oppgi hva slags variable adresser skal peke på.)

Bruk av adressevariable

Vi kan sette adressen til variable inn i pekervariabelen; vi sier at vi får adressen til å «peke på» variabelen.

```
p = &v;
```


Vi kan «følge en adresse» ved å bruke operatoren *; da får vi variabelen som adressen peker på.

```
v = 7;
printf("v = %d, *p = %d.\n", v, *p);
v = -17;
printf("v = %d, *p = %d.\n", v, *p);
```

Denne koden skriver ut

```
v = 7, *p = 7.
v = -17, *p = -17.
```


Både v og *p angir altså samme variabel:

```
*p = 123;
printf("v = %d, *p = %d.\n", v, *p);
```

Utskriften av denne koden er

```
v = 123, *p = 123.
```

Et eksempel

La oss lage en funksjon som bytter om de to parametrene sine.
Til selve ombyttingen trengs en hjelpevariabel:

```
temp = v1;
v1 = v2;
v2 = temp;
```

Når vi kjører programmet, får vi en overraskelse:

```
Før:   a = 3 og b = 4
Etter: a = 3 og b = 4
```

Grunnen er: Parametre overføres som *verdier* i C (som i Java). Følgelig er det bare lokale kopier som endres. Når funksjonen er ferdig, er alt glemt.

Løsning

Løsningen er å overføre *adressene* til de to variablene i stedet for verdiene. Adressene overføres som kopier, men vi kan allikevel endre det de peker på.


```
#include <stdio.h>

void swap (int v1, int v2)
{
  int temp;

  temp = v1;
  v1 = v2;
  v2 = temp;
}

int main (void)
{
  int a = 3;
  int b = 4;

  printf("Før:   a = %d og b = %d\n", a, b);
  swap(a, b);
  printf("Etter: a = %d og b = %d\n", a, b);
}
```


```
#include <stdio.h>

void swap (int *v1, int *v2)
{
  int temp;

  temp = *v1;
  *v1 = *v2;
  *v2 = temp;
}

int main (void)
{
  int a = 3, b = 4;

  printf("Før:   a = %d og b = %d\n", a, b);
  swap(&a, &b);
  printf("Etter: a = %d og b = %d\n", a, b);
}
```

Legg merke til at både funksjonsdeklarasjonen og kallet er endret!

Når dette programmet kjører, skjer alt som vi forventer:

```
Før:   a = 3 og b = 4
Etter: a = 4 og b = 3
```

- Det er ulike måter å overføre parametre på.
- I C og i Java brukes *verdioverføring*.
- Man kan allikevel oppdatere variable ved å sende over *adressene* til dem. Dette gjøres for eksempel i

  scanf("%d", &v);

Frigivelse av objekter

Når objekter ikke trengs mer, må de gis tilbake til systemet med funksjonen free:

```
free(p);
```

Dynamisk allokering

Ofte trenger man å opprette objekter under kjøringen i tillegg til variablene. Standardfunksjonen malloc («memory allocate») benyttes til dette. Parameter er antall byte den skal opprette; operatoren sizeof kan gi oss dette.

Vi må ha med stdlib.h for at malloc skal fungere skikkelig.

```
#include <stdlib.h>
      ⋮
int *p;
      ⋮
p = malloc(sizeof(int));
```

Et eksempel

Anta at vi skal lese et navn (dvs en tekst) og skrive det ut. For at navnet ikke skal oppta plass når vi ikke trenger det, bruker vi dynamisk allokering.

```
char *navn;
   ⋮
printf("Hva heter du? ");
navn = malloc(200);
scanf("%s", navn);
printf("Hei, %s.\n", navn);
free(navn);
```

Hva hvis noe går galt?

Følgende Java-program inneholder en feil:

```
1  class Feil {
2      void m() {
3      }
4
5      public static void main (String args[]) {
6          Feil fp = null;
7
8          fp.m();
9      }
10 }
```


Når vi kjører det, får vi beskjed om hva som gikk galt:

```
> javac Feil.java
> java Feil
Exception in thread "main" java.lang.NullPointerException
        at Feil.main(Feil.java:8)
```


Her er et C-program med tilsvarende feil:

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5    char *s;
6
7    strcpy(s, "Abc");
8    return 0;
9  }
```


Når vi kompilerer og kjører det, skjer følgende:

```
> gcc feil.c -o feil
> ./feil
Segmentation fault
```

Konklusjon Vær nøye med å få programmet riktig.

(Vi kommer ellers tilbake med verktøy for feilfinning siden.)