Programmeringsspråket C
Del 2

Michael Welzl
E-mail: michawe@ifi.uio.no

Et eksempel

Dette er lite eksempel som ber om et tall, leser det og så teller fra det ned til 0.

```
> cc countdown.c -o countdown
> countdown
===== Countdown Program =====
Enter a positive number: 5
5
4
3
2
1
0
```

```c
/*  Program Name : countdown, our first C program
 *
 *  Description  : This program prompts the user to type in a
 *  positive number and counts down from that number to 0,
 *  displaying each number along the way.                    */

/* The next two lines are preprocessor directives */
#include <stdio.h>
#define STOP 0

/* Function    : main                                        */
/* Description : prompts user for input, then display countdown */
int main(void)
{
  /* Variable declarations */
  int counter;               /* Hold intermediate count values */
  int startPoint;            /* Starting point for count down  */

  /* Prompt the user for input */
  printf("===== Countdown Program =====\n");
  printf("Enter a positive integer: ");
  scanf("%d", &startPoint);

  /* Count down from the input number to 0 */
  for (counter = startPoint; counter >= STOP; counter--)
    printf("%d\n", counter);
}
```

Forklaring:

- Definisjonspakker hentes inn med #include. Den vanligste er for lesing og skriving:

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

- Det er mulig å definere konstanter med:

```
#define STOP 0
```

(Mer om dette siden.)

- Man kan lese tall (og tegn eller ord) med scanf.
  Første parameter angir formatet (som for printf).
  Øvrige parametre gir variablene verdiene skal legges inn i.
  NB! Variablene *må* ha en ”&” foran seg!
  (Forklaring kommer siden.)

Forklaring (forts.):

- for-løkka benyttes til å utføre kode et fast antall ganger. Den litt eiendommelige syntaksen er egentlig en kortform for:

```
Counter = startPoint;
While (counter >= STOP) {
   printf(”%d\n”, counter);
   counter--;
}
```

- Operatoren ”++” brukes til å øke en variabel med 1. Om den står *foran* variabelen, foretas økningen *før* vi henter verdien.

Koden . . .	. . . gir samme resultat som . . .
a = ++x;	x = x+1; a = x;
a = x++;	a = x; x = x+1;

Tilsvarende brukes ”--” til å senke en variabel med 1.

Nok et eksempel:

Dette eksemplet gjør ikke noe fornuftig, men demonstrerer bruk av variable og operatorer i C. En kjøring ser slik ut:

```
> cc simple.c -o simple
> ./simple
The results are : outLocalA = 0, outLocalB = 6
```

```
#include <stdio.h>
int inGlobal;           /* Variable inGlobal is a global variable */
                           /* because it is declared outside of
                              all blocks */
Int main(void)
{
     int inLocal;          /* Variables inLocal, outLocalA,
                              outLocalB */
     int outLocalA;      /* are all local to main */
     int outLocalB;

     /* Initialize */
     inLocal = 2;
     inGlobal = 3;

    /* Perform calculations */
    outLocalA = inLocal++ & ~inGlobal;
    outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

    /* Print out results */
    printf("The results are : outLocalA = %d, outLocalB = %d\n",
               outLocalA, outLocalB);
}
```

Funksjoner

En **funksjon** (også kalt **metode**, **prosedyre** eller **rutine**) er en bit av programkoden som kan *kalles* fra andre steder. Under utførelsen ”hopper” man da til funksjonen, utfører programkoden til funksjonen, og hopper så tilbake.

En metode i Java:

```
class MetodeDemo {
   public static void main (String arg[]) {
      demo(7,3);
      demo(18,24);
   }

   static void demo (int n, int m) {
      System.out.println("GCD(" + n + "," + m +
                         ") = " + gcd(n,m));
   }

   static int gcd (int a, int b) {
      while (a != b) {
         if (a > b) a -= b;
         else       b -= a;
      }
      return a;
   }
}
```

Her er det samme programmet som en funksjon i C:

```
#include <stdio.h>

int gcd (int a, int b)
{
  while (a != b) {
    if (a > b) a -= b;
    else       b -= a;
  }
  return a;
}

void demo (int n, int m)
{
  printf("GCD(%d,%d) = %d\n", n, m, gcd(n,m));
}

int main (void)
{
  demo(7,3);
  demo(18,24);
}
```

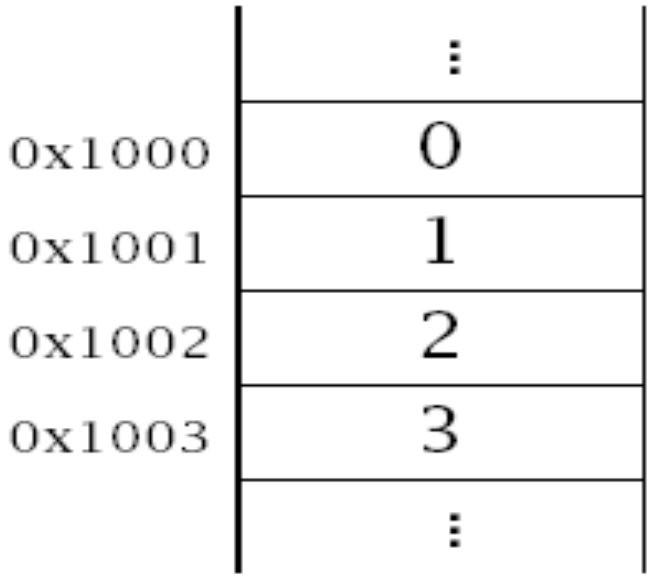
Funksjoner i C (forts.)

Merk at i C kan man bare kalle funksjoner definert ***tidligere***.†

† Dette er ikke helt riktig, men en god regel for nybegynnere...

Størrelser og adresser

De fleste datamaskiner idag er *byte-maskiner der man adresserer hver enkelt byte;*



char lagres i én byte.

short lagres i to bytes.

long lagres i fire bytes.

Prosessorene har instruksjoner som kan hente et gitt antall bytes av gangen. Dette kalles **bussbredden** og er oftest 2, 4 eller 8 idag. En **64-bits prosessor** kan hente 8 bytes av gangen.

Byte-rekkefølgen

Anta at vi har en 32-bits prosessor. Hvis et ord som inneholder 32-bit verdien 0x01020304 og lagres som;

01	02	03	04

kaller vi maskinen **big-endian**. Hvis det lagres som;

04	03	02	01

kalles den **little-endian.**

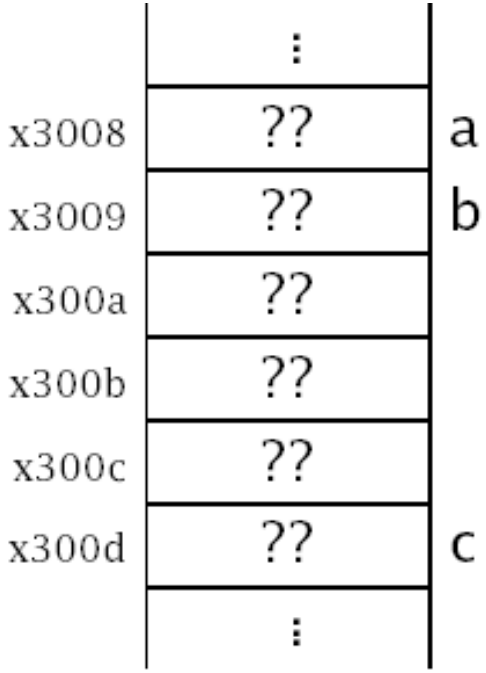
Vektorer

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 hakeparentes etter variabelnavnet

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

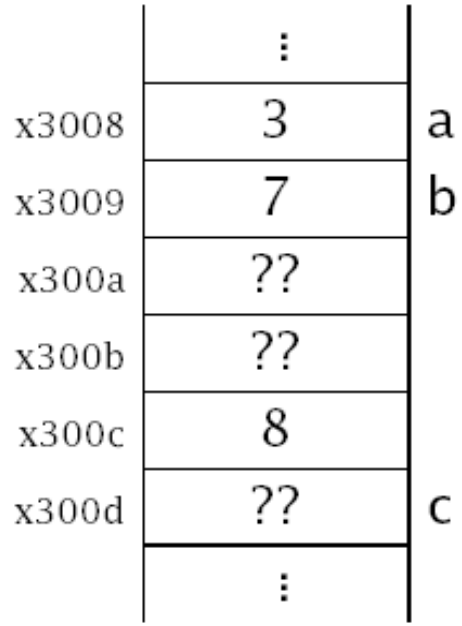


Bruk:

Ved bruk angir indeksen hvilket element vi ønsker. I C er alltid første element nr. 0, neste nr. 1, osv.

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

Etter dette er situasjonen:



Beregning av adresser

Adressen til vanlige variable er kjent[†], men adressen til vektorelementer må beregnes.

Formelen er

Startadresse + *Indeks x 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.

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

Tekster

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

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.

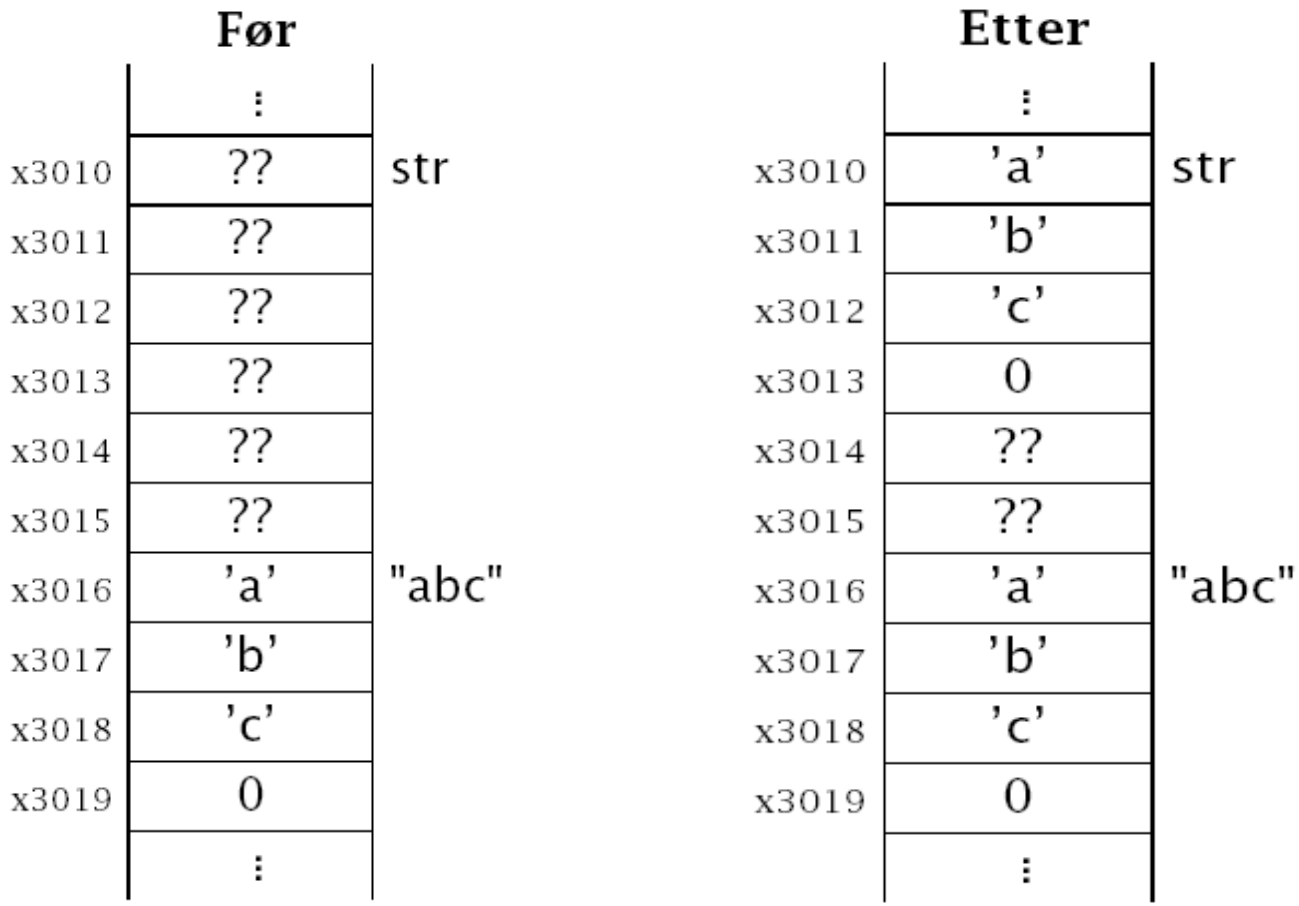
Tekstfunksjoner: **#include <string.h>**

† En byte med verdien 0 er ikke det samme som sifferet “0”; det er representert av verdien 48.

Kopiering av tekst

Flytting av tekst skjer med standardfunksjonen strcpy:

```
strcpy(str, "abc");
```



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).

C´s preprosessor

Før selve kompileringen går C-kompilatoren gjennom koden med en preprosessor (som er programmet /usr/lib/cpp). Dette er en programmerbar tekstbehandler som gjør følgende:

- Henter inn filer

```
#include "incl.h"
#include <stdio.h>
```

Hvis filen er angitt med spisse klammer (som for eksempel <stdio.h>), hentes filen fra området /usr/include. Ellers benyttes vanlig notasjon for filer.

C´s preprosessor (forts.)

- Leser makro-definisjoner og ekspanderer disse i teksten:

```
#define LINUX
#define N 100
#define MIN(x,y) ((x)<(y) ? (x) : (y))
```

Av gammel tradisjon gis makroer navn med store bokstaver.

(En **makro** er en navngitt programtekst. Når navnet brukes, blir det **ekspandert**, dvs erstattet av definisjonen. Dette er ren tekstbehandling uten noen forbindelse med programmeringsspråkets regler.)

Benytter man makroer med parametre, bør disse settes i parenteser. Likeledes, hvis definisjonen er et uttrykk med flere symboler, bør det stå parenteser rundt hele uttrykket.

C´s preprosessor (forts.)

Betinget kompilering. Her angis hvilke linjer som skal tas med i kompileringen og hvilke som skal utelates. Følgende direktiver finnes for betinget kompilering:

#if Hvis uttrykket etterpå er noe annet enn 0, skal etterfølgende linjer tas med. Uttrykket kan ikke inneholde variable eller funksjoner.

#ifdef Hvis symbolet er definert (med en #define), skal etterfølgende linjer tas med.

#ifndef Motsatt av #ifdef.

#else Skille mellom det som skal tas med og det som ikke skal tas med.

#endif Slutt med betinget kompilering.

Betinget kompilering

Eksempel:

```
#define LINUX

#ifdef LINUX
  int x;
#else
  long x;
#endif
```

Betinget kompilering (forts.)

Det er også mulig å styre betinget kompilering gjennom cc-kommandoen:

```
> cc -c -DLINUX
```

gir samme effekt som om det sto

```
#define LINUX
```

i program-koden.

På denne måten er det mulig å ha flere versjoner av koden (for eksempel for flere maskin-typer) og så kontrollere dette utelukkende gjennom kompileringen.

Fare med betinget kompilering
Man kan risikere å ha kode som aldri har vært kompilert, og som kan inneholde de merkeligste feil.

Separat kompilering

I utgangspunktet er det ingen problem med separat-kompilering i C; hver fil utgjør en enhet som kan kompileres for seg selv,uavhengig av alle andre filer i programmet.

```
> cc -c del.c
```

vil kompilere filen del.c og lage del.o som inneholder den kompilerte koden.

Separat kompilering; Eksempel

Anta at vi har to filer:

Filen sum.c:

```
int sum (int n)
{ /* Beregner 1+2+...+n */
  return n*(n+1)/2;
}
```

Filen vissum.c

```
#include <stdio.h>

extern int sum (int n);

int main (void)
{
  int i;
  for (i = 1;  i <= 10;  ++i)
    printf("%2d:%4d\n", i, sum(i));
}
```

Separat kompilering; Eksempel (forts)

Kompilering

Disse kan kompileres hver for seg:

```
> cc -c sum.c
> cc -c vissum.c
```

Linking

De kompilerte filene kan siden **linkes** sammen:

```
> cc vissum.o sum.o -o vissum
```

Kjøring

Da får vi et ferdig program som kan kjøres:

```
> vissum
 1:    1
 2:    3
 3:    6
 4:   10
 5:   15
 6:   21
 7:   28
 8:   36
 9:   45
10:   55
```

Definisjonsfiler

Det er en fare for at funksjonssignaturer, strukturer, makroer, typer og andre elementer ikke blir skrevet likt i hver fil. Dette løses ved hjelp av definisjonsfiler ("header files"), hvis navn gjerne slutter med '.h'.

Filen incl.h:

```
#define N 100
```

Definisjonsfiler (forts.)

Filen prog.c:

```
#include "incl.h"

int main(void)
{
  char *s[N];
     ⋮
}
```

Definisjonsfiler inneholder gjerne følgende:

- Makrodefinisjoner (#define)
- Typedefinisjoner (typedef, union, struct)
- Eksterne spesifikasjoner (extern)
- Funksjoner som

  extern int f(int, char);

  (Legg merke til semikolonet sist! Det betyr at funksjonen ikke defineres her, men et annet sted.)

Utskrift

Som nevnt brukes printf til utskrift. Første parameter er *formatet*, og det tolkes slik:

- Alle tegn unntatt ”%” kopieres.
- Tegnkombinasjonen ”%...” angir noe spesielt:
  %% setter inn et %-tegn
  %d setter inn et heltall hentet fra parameterlisten.
  %u setter inn et heltall uten fortegns-bit fra parameterlisten.
  %x setter inn et heltall (uten fortegns-bit), men skriver det på heksadesimal form.
  %o setter inn et heltall (uten fortegns-bit), men skriver det på oktal form.
  %f setter inn et flyt-tall.
  %c skriver ut et heltall tolket som et tegn.
  %s skriver ut en tekst fra en tekstvariabel.

Et eksempel:

```
#include <stdio.h>

int main (void)
{
  int   a = 102;
  int   b =  65;
  char  c = 'z';
  char  banner[10] = "Hola!";
  float pi = 3.14159;

  printf("The variable 'a' decimal: %d\n", a);
  printf("The variable 'a' hex: %x\n", a);
  printf("The variable 'a' octal: %o\n", a);
  printf("'a' plus 'b' as character: %c\n", a+b);
  printf("A char %c.\tA string \"%s\"\nA float %f\n",
         c, banner, pi);
  return 0;
}
```

Utskriften blir

```
The variable 'a' decimal: 102
The variable 'a' hex: 66
The variable 'a' octal: 146
'a' plus 'b' as character: §
A char z.       A string "Hola!"
A float 3.141590
```

Lesing fra fil:

Man må gjøre følgende når man skal lese fra fil:

- Deklarere fil-variabelen som FILE *.
- Åpne filen med funksjonen fopen med filnavn og ”r” (for ”read”) som parametre. Hvis filen ikke finnes, returneres NULL.
- Lese fra filen med fscanf (som virker som scanf) eller fgetc (som leser ett og ett tegn).
- Lukke filen med fclose.

Et eksempel:

```
#include <stdio.h>

int main(void)
{
  FILE *f;
  int  a, b;

  f = fopen("2tall","r");
  if (f == NULL) {
    printf("Umulig å lese filen 2tall.\n");  return 1;
  }

  fscanf(f, "%d%d", &a, &b);
  fclose(f);

  printf("%d + %d = %d\n", a, b, a+b);
  return 0;
}
```

Skriving til fil:

Skriving til fil er tilsvarende lesing fra fil.
Man må gjøre følgende:

- Deklarere fil-variabelen som FILE*.
- Åpne filen med funksjonen fopen med filnavn og ”w” (for ”write”) som parameter. **Lag en ny file hvis den ikke finnes allerede!** Hvis filen ikke lar seg åpne, returneres NULL.
- Skrive til filen med fprintf (som virker som printf).
- Lukke filen med fclose.

Et eksempel:

```
#include <stdio.h>

int main(void)
{
  FILE *f;
  int i;

  f = fopen("potens.tab","w");
  if (f == NULL) {
    printf("Umulig å skrive til potens.tab\n");  return 1;
  }

  fprintf(f, " n   n2   n3\n");
  for (i = 0;  i <= 10;  ++i) {
    fprintf(f, "%2d %3d %4d\n", i, i*i, i*i*i);
  }
  fclose(f);
  return 0;
}
```

Et eksempel (forts.)

Dette gir følgende resultat:

```
> more potens.tab
 n   n2   n3
 0    0    0
 1    1    1
 2    4    8
 3    9   27
 4   16   64
 5   25  125
 6   36  216
 7   49  343
 8   64  512
 9   81  729
10  100 1000
```

Utskrift i faste kolonner:
Ved å angi et tall i formatet (som i %2d) får vi printf til å sette av *minst* så mange posisjoner til tallet.

Nok et eksempel:

Dette programmet leser en fil min.fil tegn for tegn. Så skriver det ut hvor mange forekomster det er av hvert tegn.

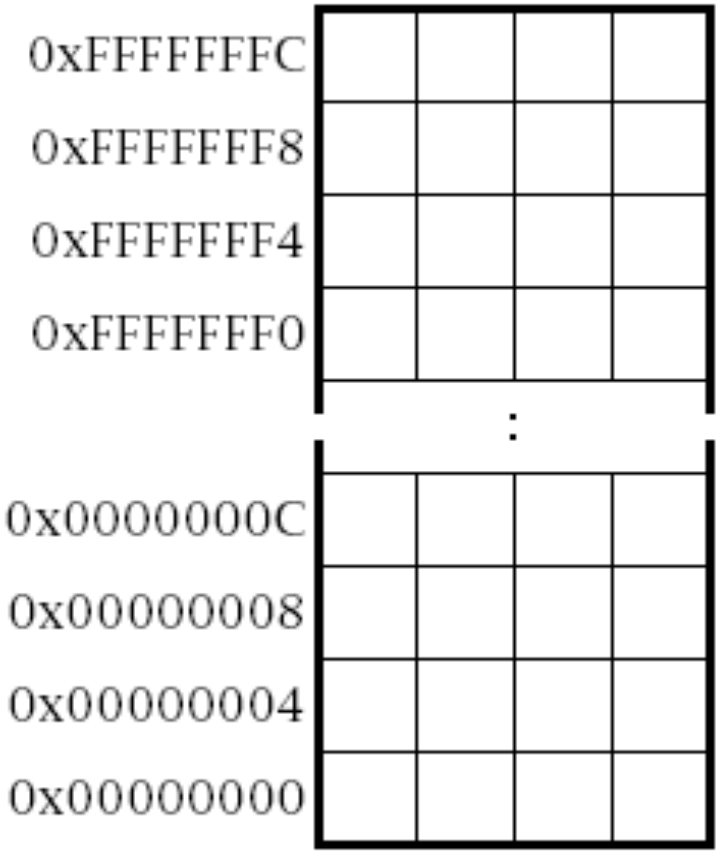
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *f;
  int  nc[256], i, c;

  for (i = 0;  i <= 255;  ++i)
    nc[i] = 0;

  f = fopen("min.fil","r");
  if (f == NULL) {
    printf("Kan ikke lese 'min.fil'!\n");
    exit(1);
  }

  c = fgetc(f);
  while (c != EOF) {
    ++nc[c];  c = fgetc(f);
  }
  fclose(f);

  for (i = 0;  i <= 255;  ++i) {
    if (nc[i] > 0) {
      if ((32<=i && i<=126) || 161<=i) printf("'%c'", i);
      else printf("%3d", i);
      printf(":%4d\n", nc[i]);
    }
  }
  return 0;
}
```

Bruker vi programmet på seg selv, får vi følgende resultat:

```
 10:  32  ' ': 131  '!':   2  '"':  12  '#':   2  '%':   3
'&':   2  ''':   3  '(':  17  ')':  17  '*':   1  '+':   6
',':   6  '.':   4  '0':   5  '1':   4  '2':   5  '3':   2
'4':   1  '5':   5  '6':   3  ':':   1  ';':  18  '<':   7
'=':  14  '>':   3  'E':   2  'F':   2  'I':   1  'K':   1
'L':   3  'N':   1  'O':   1  'U':   1  '[':   5  '\':   2
']':   5  '`':   1  'a':   2  'b':   1  'c':  16  'd':   7
'e':  14  'f':  21  'g':   2  'h':   3  'i':  37  'k':   2
'l':   9  'm':   3  'n':  21  'o':   6  'p':   5  'r':   9
's':   5  't':  12  'u':   3  'v':   1  'w':   1  'x':   1
'{':   5  '|':   2  '}':   5
```

Adresser

Som nevnt tidligere, ligger data og programkode lagret i minnet (gjerne kalt RAM for ”Random Access Memory”.



Et slikt minne er typisk for dagens maskiner, og er gjerne byte-adressert med f.eks. 32 bits adresser.

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 bytes.

Pekervariable

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 peker som kan peke på int-variable. (Vi må alltid oppgi hva slags variable pekere skal peke på.)

Bruk av pekervariable:
Vi kan sette adressen til variable inn i pekervariabelen;
Vi sier at vi får pekeren til å ”peke på” variabelen.

```
p = &v;
```

Bruk av pekervariable (forts.)

Vi kan ”følge en peker” ved å bruke operatoren *; da får vi variabelen som pekeren 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.
```

Eksempel:

La oss lage en funksjon som bytter om de to parametrene sine.

Til selve ombyttingen trengs en hjelpevariabel:

```
tempVal = firstVal;
firstVal = secondVal;
secondVal = tempVal;
```

Vi skriver følgende program:

```
#include <stdio.h>

void Swap(int firstVal, int secondVal);

main()
{
  int valueA = 3;
  int valueB = 4;

  printf("Before Swap: valueA = %d and valueB = %d\n", valueA, valueB);
  Swap(valueA, valueB);
  printf("After Swap : valueA = %d and valueB = %d\n", valueA, valueB);
}

void Swap(int firstVal, int secondVal)
{
  int tempVal;            /* Needed to hold firstVal when swapping */

  tempVal = firstVal;
  firstVal = secondVal;
  secondVal = tempVal;
}
```

Når vi kjører programmet, får vi en overraskelse:

```
Before Swap: valueA = 3 and valueB = 4
After Swap : valueA = 3 and valueB = 4
```

Grunnen er: Parametre overføres som verdier i C (som i Java).

Systemet tar altså en kopi av parameterverdien og legger denne i et register (eller et annet sted for parametre).

Følgelig er det bare lokale kopier som endres. Når funksjonen er ferdig, er alt glemt!

Løsning:

Løsningen er å overføre *pekere* til de to variablene i steden for verdiene.

Pekerne overføres som kopier, men vi kan allikevel endre det de peker på.

Et program som fungerer vil være som følger:

```
#include <stdio.h>

void NewSwap(int *firstVal, int *secondVal);

main()
{
  int valueA = 3;
  int valueB = 4;

  printf("Before NewSwap: valueA = %d and valueB = %d\n", valueA, valueB);
  NewSwap(&valueA, &valueB);
  printf("After NewSwap : valueA = %d and valueB = %d\n", valueA, valueB);
}

void NewSwap(int *firstVal, int *secondVal)
{
  int tempVal;              /* Needed to hold firstVal when swapping */

  tempVal = *firstVal;
  *firstVal = *secondVal;
  *secondVal = tempVal;
}
```

Eksempel (forts.)

Legg merke til at både funksjonsdeklarasjonen og kallet er endret!

Når dette programmet kjører, skjer alt som vi forventer:

```
Before NewSwap: valueA = 3 and valueB = 4
After NewSwap : valueA = 4 and valueB = 3
```

Konklusjon om parametre:

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

  scanf(”%d”, &v);