16. Tegn og alfabet

I dette kapitel studerer vi tegn. Tegn udgør grundbestanddelen i enhver form for tekstbehandling. I senere kapitler, nærmere betegnet kapitel 27 - kapitel 31, ser vi på sammensætningen af tegn til tekststrenge.

Tegn organiseres i alfabeter, som er mængder af tegn. Dette er emnet for det første afsnit efter denne introduktion. I C er der en tæt sammenhæng mellem tegn og små heltal. Ofte er det bekvemt at notere sådanne tal enten oktalt eller hexadecimalt. I dette kapitel vil vi se hvorfor disse talsystemer er attraktive, og vi vil benytte lejligheden til at studerende opbygningen af talsystemer lidt nærmere.

Vigtigst af alt gennemgår vi også i dette kapitel et antal programeksempler. Her skal fremhæves to, nemlig programmer som kan konvertere mellem forskellige talsystemer, og programmer som kan tælle tegn, ord og linier i en tekst.

16.1. Alfabet

Lektion 4 - slide 2

I dette afsnit vil vi først fastslå betydningen af et alfabet, og dernæst vil vi på punktform diskutere forskellige egenskaber af alfabeter, og primært af det såkaldte ASCII alfabet.

Et *alfabet* er en ordnet mængde af bogstaver og andre tegn

- ASCII alfabetet
  - **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
  - Angiver tegnets placering i alfabetet ved at tildele tegnet en heltalsværdi
  - Det oprindelige ASCII tegnsæt indeholder 128 tegn
  - Det udvidede ASCII tegnsæt indeholder 256 tegn
    - De danske tegn 'æ', 'ø', 'å', 'Æ', 'Ø' og 'Å' er placeret i det udvidede ASCII tegnsæt
  - Appendix E side 609 i *C by Dissection*

I afsnit 16.2 viser vi hvordan vi kan udskrive en tabel over tegnene i ASCII alfabetet.

I det seneste årti er der defineret mere omfattende alfabeter, som indeholder tegn fra mange forskellige kulturer.

Unicode er et sådant alfabet.

16.2. Datatypen char

Lektion 4 - slide 3

Datatypen `char` er en type i C, ligesom f.eks. `int`, `float`, og `double`.

Datatypen **`char`** repræsenterer mængden af mulige tegn i et standard alfabet

- Datatypen char
  - Et tegn repræsenteres i én byte, som svarer til otte bits
  - Otte bit tegn giver netop mulighed for at repræsentere tegnene det udvidede ASCII alfabet
  - Typen **`char`** er en heltalstype i C

En bit er den mindste informationsenhed i en computer. En bit kan skelne mellem to forskellige tilstande, som vi ofte benævner 0 og 1. Otte bit kaldes en byte, og da de alle individuelt og uafhængig af hinanden kan være enten 0 eller 1 kan der repræsenteres $2^8 = 256$ mulige værdier i en byte. Det kan f.eks. være 256 mulige tegn. Videre endnu sammensættes andre informationsenheder af et helt antal byte. Eksempelvis er en `int` værdi i C typisk sammensat af fire bytes, således at vi kan arbejde med $256^4 = 2^{32} = 4294967296$ forskellige `int` værdier i C.

Herunder viser vi et C program, som udskriver en simpel version af de 128 ASCII tegn. I for-løkken gennemløbes et interval fra 0 til 127. Den røde del afgør om tegnet er printbart eller ej. Abstraktionen `isgraph` diskuteres i afsnit 16.8. Den første if i forløkken sørger for linieskift for hvert tiende tegn. Det fremgår også af dette program, at i C skelner vi ikke skarpt mellem tal og tegn.

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

int main(void) {
  int i;

  printf("\n");
  printf("%3s", "");           /* print top line above table */
  for (i = 0; i < 10; i++)
    printf("%6d",i);
  printf("\n");

  for (i = 0; i < 128; i++){   /* print the table */
    if (i % 10 == 0){          /* print leftmost column */
      printf("\n");
      printf("%3d", (i/10) * 10);
    }
    if (isgraph(i))
      printf("%6c", i);
    else printf("%6s", "NP");
  }
  printf("\n\n");

  return 0;
}
```

Program 16.1 *Et C program som udskriver en ASCII tegntabel.*

Outputtet fra program 16.1 vises i program 16.2 herunder. Som det fremgår af programmet ovenfor er en del tegn udskrevet som NP, hvilket betyder 'Non Printable'. NP tegn er netop de tegn, hvorpå `isgraph(i)` i program 16.1 returner false (altså 0).

```
      0   1   2   3   4   5   6   7   8   9

  0  NP  NP  NP  NP  NP  NP  NP  NP  NP  NP
 10  NP  NP  NP  NP  NP  NP  NP  NP  NP  NP
 20  NP  NP  NP  NP  NP  NP  NP  NP  NP  NP
 30  NP  NP  NP   !   "   #   $   %   &   '
 40   (   )   *   +   ,   -   .   /   0   1
 50   2   3   4   5   6   7   8   9   :   ;
 60   <   =   >   ?   @   A   B   C   D   E
 70   F   G   H   I   J   K   L   M   N   O
 80   P   Q   R   S   T   U   V   W   X   Y
 90   Z   [   \   ]   ^   _   `   a   b   c
100   d   e   f   g   h   i   j   k   l   m
110   n   o   p   q   r   s   t   u   v   w
120   x   y   z   {   |   }   ~  NP
```

Program 16.2 *Udskriften fra programmet.*

16.3. Tegnkonstanter (1)

Lektion 4 - slide 4

Vi vil i dette afsnit se på forskellige notationer for konkrete tegn. Den mest basale notation er `'a'`, her for tegnet a.

> Printbare værdier i typen **`char`** noteres i enkelt quotes, eksempelvis **`'a'`**.
>
> Det er vigtigt at kunne skelne mellem tegn, strenge og tal.
>
> Eksempelvis '7', "7" og 7.

- Andre mulige notationer for tegn:
  - Escape notation af ikke printbare tegn
    - *Eksempler:* **`'\n'`** **`'\t'`** **`'\''`** **`'\\'`**
  - Oktal escape notation
    - *Eksempler:* **`'\12'`** **`'\141'`** **`'\156'`**
  - Hexadecimal escape notation
    - *Eksempler:* **`'\xa'`** **`'\x61'`** **`'\x6e'`**

I første eksempel linie (ikke printbare tegn) viser vi escape notation for linieskift, tabulatortegnet, enkelt-anførselstegn, og bagvendt skråstreg.

De tre tegn, som er noteret som oktal og hexadecimal notation er parvis ens.

Tegnene `'\12'` og `'\xa'` er begge tegn 10 i decimal notation, hvilket ifølge ASCII tabellen er newline tegnet, altså `'\n'`.

Tegnene `'\141'` og `'\x61'` er begge tegn 97, som ifølge ASCII tegntabellen er 'a', altså et lille a.

Tegnene `'\156'` og `'\x6e'` er begge tegn 110, som ifølge ASCII tegntabellen er 'n', altså et lille n.

Vi vil senere i denne lektion udvikle et nyttigt program - program 16.8 - som kan omregne ovenstående oktale og hexadecimale tal til 'normale' decimale tal.

16.4. Tegnkonstanter (2)

Lektion 4 - slide 5

Vi viser i dette afsnit et C program, som indeholder forskellige tegnkonstanter.

I program 16.3 indeholder variablene `c`, `d`, `e` og `f` alle en værdi svarende til tegnet `'n'`. Mere konkret indeholder disse variable heltalsværdien 110 (decimalt). Variablen `x` indeholder newline tegnet, svarende til heltalsværdien 10.

Vi udskriver variablene `c`, `d`, `e`, `f`, og `x` som tegn (`%c`), som decimaltal (`%d`), som oktale tal (`%o`), og som hexadecimale tal (`%x`) i de fire `printf` kommandoer i program 16.3.

```
#include <stdio.h>

int main(void) {

  char c = 'n', d = '\156', e = '\x6e';
  int  f = 110;
  char x = '\n';

  printf("c = %c (%d, %o, %x) \n",   c,c,c,c);
  printf("d = %c (%d, %o, %x) \12",  d,d,d,d);
  printf("e = %c (%d, %o, %x) \xa",  e,e,e,e);
  printf("f = %c (%d, %o, %x) \n",   f,f,f,f);

  printf("x = %c (%d, %o, %x) \n",   x,x,x,x);

  return 0;
}
```

Program 16.3 *Et C program med eksempler på forskellige tegn konstanter. Programmet fokuserer på tegnet 'n', som noteres i normal tegnnotation, oktalt, og hexadecimalt.*

Udskriften fra program 16.3 ses herunder. Læg mærke til, at når `'\n'` skrives ud som et tegn, bliver der skiftet linie.

```
c = n (110, 156, 6e)
d = n (110, 156, 6e)
e = n (110, 156, 6e)
f = n (110, 156, 6e)
x =
 (10, 12, a)
```

Program 16.4 *Output fra programmet.*

16.5. Oktale og hexadecimal notation (1)

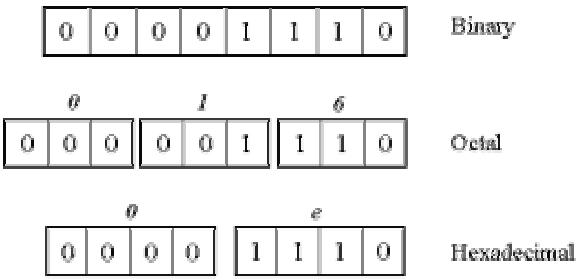
Lektion 4 - slide 6

Vi har talt om binære, oktale og decimale tal i de forrige afsnit. Men det er ikke sikkert vi alle har en god forståelse af systematikken og ideen bag disse talsystemer.

Ligeså vigtigt er det at forstå, hvorfor vi belemrer hinanden med disse fremmede notationer. Mange vil nok mene, at det ikke just er en hjælp til forståelsen. I dette og næste afsnit vil vi indse, at der rimelighed i galskaben...

Herunder viser vi tallet 14 (decimalt) som binære tal. Vi grupperer de indgående bits i grupper af tre (anden linie) og i grupper af 4 (tredie linie).

Nogle vil nok spørge, hvor og hvordan vi får øje på tallet 14. Forklaringen er at $14 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Vi ser på dette i detaljer i afsnit 16.6.



Figur 16.1 *En illustration af binære, oktale og hexadecimale tal.*

Binære tal er relevante fordi digital hardware teknologi er i stand til at skelne mellem to tilstande: Høj og lav spænding, strøm/ikke strøm, etc. Abstrakt set kalder vi disse for 0 og 1. Al indsigt om binære tal - som et matematisk begreb - kan dermed direkte overføres til digital hardware.

Oktale tal tillader ciffervis konvertering af det oktale til grupper af 3 bits. Dette er illustreret i figur 16.1. Tilsvarende tillader hexadecimale tal ciffervis konvertering af de hexadecimale tal til grupper af 4 bits, hvilket også fremgår af figuren.

Det er, med andre ord, meget enkelt for os mennesker at omforme et oktalt eller et hexadecimalt tal til binær notation. Dette er bestemt ikke tilfældet for decimale tal. Vi kan dog relativt let skrive et

program som udfører konverteringen. Vi viser hvordan i afsnit 16.10, nærmere betegnet i program 16.9.

Oktal og hexadecimal tal notation tillader en ciffervis konvertering til binær notation

16.6. Oktale og hexedecimal notation (2)

Lektion 4 - slide 7

I dette afsnit anskueliggør vi logikken bag de binære, oktale og hexedecimale talsystemer.

Herunder ser vi hvordan decimaltallet 14 kan opløses efter basis 2, 8 og 16. Det er et matematisk faktum, at disse opløsninger er entydige.

Vi vil starte med at konstatere, at decimaltallet 14 egentlig betyder $1 \cdot 10^1 + 4 \cdot 10^0$. Ganske tilsvarende forholder det sig når vi opløser tallet efter en anden basis end 10:

- Binære tal: Grundtal 2
  - Eksempel: $14 = \mathbf{1} \cdot 2^3 + \mathbf{1} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{0} \cdot 2^0$
- Oktale tal: Grundtal 8
  - Eksempel: $14 = \mathbf{1} \cdot 8^1 + \mathbf{6} \cdot 8^0$
- Hexadecimale tal: Grundtal 16
  - Eksempel: $14 = 14 \cdot 16^0 = \mathbf{e} \cdot 16^0$

Senere i denne lektion vil vi programmere funktioner der kan konvertere mellem talsystemerne

16.7. Tegn aritmetik

Lektion 4 - slide 8

Vi kan bruge tegn i både aritmetiske og logiske udtryk. Det er ikke så underligt, idet vi jo for længst har indset, at tegn i C blot repræsenteres som det tilsvarende tal fra ASCII tegntabellen.

Det er muligt at udføre aritmetiske operationer på tegn - forstået som aritmetik på de underliggende ASCII værdier

Herunder fortolker vi betydningen af en række forskellige udtryk, der har tegn som operander.

- Addition af et tal til et tegn
  - `'0' + 7 == '7'`
  - Tegnet 7 positioner længere fremme i tegntabellen
- Subtraktion af et tal fra et tegn
  - `'k' - 1 == 'j'`
  - Tegnet før k
- Subtraktion af to tegn
  - `'a' - 'A'`
  - Afstanden mellem små og store bogstaver
- Sammenligning af to tegn
  - `'a' < 'b'`
  - Tegnet a kommer før b. Vurdering af alfabetisk rækkefølge

16.8. Klassificering af tegn

Lektion 4 - slide 9

I dette afsnit vil vi diskutere en hierarkisk klassificering af tegn.

Biblioteket `ctype.h` understøtter en række abstraktioner, som klassificerer tegn

- Tegn
  - Kontrol tegn - `iscntrl(c)`
  - Printbare tegn - `isprint(c)`
    - Blanke tegn - `isspace(c)`
    - Grafiske tegn - `isgraph(c)`
      - Punkt tegn - `ispunct(c)`
      - Alfanumeriske tegn - `isalnum(c)`
        - Alfabetiske tegn - `isalpha(c)`
          - Små bogstaver - `islower(c)`
          - Store bogstaver - `isupper(c)`
        - Numeriske tegn - `isdigit(c)`

Vi kan tænke på ovenstående abstraktioner som funktioner fra biblioteket `ctype`. Typisk er disse dog implementeret som makroer, hvilke tekstuelt substitueres ind i stedet for kaldet. Dette er en helt anderledes mekanisme end kald af funktioner, som vi diskuterede generelt i afsnit 12.3 og eksemplificerede konkret i afsnit 12.4. Brug af makroer giver effektive programmer, idet vi undgår et egentlig funktionskald. Der er dog også en række problemer forbundet med brug af makroer. Se appendix B i *C by Dissection* for nogle detaljer om dette.

16.9. Abstraktionerne getchar og putchar

Lektion 4 - slide 10

Til historien om tegn hører også en lille beretning om hvordan enkelttegn kan udskrives og indlæses. Det sker med C funktionerne `getchar` og `putchar`.

> Abstraktionerne **`getchar`** og **`putchar`** giver mulighed for input og output af enkelttegn.
>
> **`getchar`** og **`putchar`** er lavniveau alternativer til **`scanf`** og **`printf`**.
>
> Som et eksempel på **`readchar`** ser vi på tegnvis indlæsning af heltal.

Ved brug af `scanf` er det (nogenlunde) let og bekvemt at indlæse et heltal. Vi har efterhånden set flere gange, at det kan gøres med `scanf("%d", &var)`. Dette kald af `scanf` overfører det indlæste heltal til variablen `var`.

Vi vil nu antage, at vi ikke har `scanf` funktionen til rådighed, og at vi derfor selv skal programmere en funktion (eller rettere en procedure) som indlæser et heltal. I program 16.5 viser vi en basal udgave. I program 16.7 ser vi på en lidt mere udbygget version.

```
int read_int(char *prompt){
  int res = 0;  char c;
  printf(prompt);

  while (isdigit(c = getchar()))
    res = res * 10 + (c - '0');

  return res;
}
```

Program 16.5 *En funktion der læser et heltal som enkelttegn - basal udgave.*

I programmet vist ovenfor indlæser vi tegn i en for-løkke, med anvendelse af `getchar`. Læg mærke til, at vi faktisk læser tegnene i det logiske udtryk (den røde del) der kontrollerer while-løkken. Dette er en typisk udtryksform i C, også kaldet et *mønster* eller et *idiom*.

I det blå assignment oparbejder vi det resulterende heltal. Oparbejdningen sker efter principperne i afsnit 16.6.

Under indlæsningen af cifre med `getchar` (den røde del) møder vi først de mest betydende cifre, og tilslut det mindst betydende ciffer. Hvis vi indtaster 123 vil `res` i første gentagelse i while-løkken få værdien 1. I andet gennemløb får `res` værdien 1*10 + 2, altså 12. I tredje og sidste gennemløb får `res` værdien 12*10 + 3, altså 123.

Vi har altså hermed indset at indlæsning af tegnene 123 leder til beregningen af tallet 123. *Voila!*

Nedenfor viser vi et helt C program, hvori funktionen `readint` fra program 16.5 indgår. Funktionen readint er vist med fed sort skrift i program 16.6.

```
#include <stdio.h>

int read_int(char*);

int main(void) {
  int i, n = 0;

  for (i = 1; i <= 5; i++){
    n = read_int("Enter an integer: ");
    printf("Decimal value: %d\n", n);
  }

  return 0;
}

int read_int(char *prompt){
  int res = 0;  char c;
  printf(prompt);

  while (isdigit(c = getchar()))
    res = res * 10 + (c - '0');

  return res;
}
```

Program 16.6 *Funktionen read_int og en main funktion som kalder read_int 5 gange.*

Vi afslutter dette delafsnit med en lidt mere generel, og lidt mere robust implementation af funktionen `readint`. Udgaven herunder kan håndtere et + eller - fortegn af det indlæste tal. Funktionen er også lidt bedre til at læse 'resten af linien' efter der ikke er flere cifre.

```
#include <stdio.h>

int read_int(char*);

int main(void) {
  int i, n = 0;

  for (i = 1; i <= 5; i++){
    n = read_int("Enter an integer: ");
    printf("Decimal value: %d\n", n);
  }

  return 0;
}

int read_int(char *prompt){
  int res = 0;  char c; int sign = 1;
  printf(prompt);

  /* Handle initial sign, if any */
  c = getchar();
  if (c == '+') {sign = 1; c = getchar();}
  else if (c == '-') {sign = -1; c = getchar();}

  /* Read digits - first char is ready in c*/
```

```
  while (isdigit(c)){
    res = res * 10 + (c - '0');
    c = getchar();
  }

  /* read rest of line */
  while (c != '\n') c = getchar();

  return sign * res;
}
```

Program 16.7 *En mere brugbar udgave med mulighed for fortegn og bedre linieafslutning.*

16.10. Konvertering mellem talsystemer

Lektion 4 - slide 11

I dette afsnit laver vi to nyttige programmer, som kan konvertere mellem talsystemer. Det første konverterer et tal fra et (næsten) vilkårligt talsystem til det sædvanlige 10-talsystem. Det næste omregner et tal fra 10-talsystemet til et andet talsystem.

De første 10 cifre noteres med tegnene '0' - '9'. Vi har ikke flere naturlige cifre i ASCII tabellen, så derfor benytter vi de alfabetiske tegn 'a' - 'z' for de næste 26 cifre. Det betyder vi kan håndtere talsystemer op til basis 36. Den eneste årsag til at vi ikke kan gå højere op er at vi ikke har naturlig notation for mere end 36 forskellige cifre.

Vi vil nu se på et program, der indlæser et tal i et *n* talsystem, og som konverterer det til et decimalt tal.

Vi vil tilsvarende se på et program, der konverterer et decimalt tal til et tal i et andet talsystem.

Programmerne illustrerer tillige **`getchar`** og **`putchar`**.

Som det fremgår, tager programmet herunder et tal i et talsystem op til basis 36, og konverterer det til et decimaltal. Det betyder, at vi skal kunne håndtere input som både kan være normale cifre og bogstaver. Derfor læser vi cifrene enkeltvis med `getchar` (den blå del i program 16.8).

Funktionen `read_in_base` indlæser således en række tegn, og det returnerer det decimaltal, som modsvarer de indlæste tegn i den base, der er overført som parameter til `read_in_base`. Det ville have været mere tilfredsstillende at overføre allerede indlæste cifre som en parameter til `read_in_base`. Generelt ønsker vi at begrænse de steder i vore programmer, som laver input og output. Det vil dog kræve programmering med arrays, som vi endnu ikke har set på. Med arrays ville vi kunne overføre en ordnet samling af cifre til funktionen, read_in_base, som så kunne returnere et tal. Dette ville være en renere abstraktion, end den vi faktisk har programmeret nedenfor. Arrays mødes første gang i kapitel 21 i dette materiale.

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

int read_in_base(int);
void skip_rest_of_input_line (void);

int main(void) {
  int i, n, base;

  printf("THIS PROGRAM CONVERTS NUMBERS IN A"
         "GIVEN BASE TO A DECIMAL NUMBER.\n\n");
  do {
    printf("Enter number base (0 terminates the program, max 36): ");
    scanf("%d", &base); skip_rest_of_input_line();
    if (base > 0){
      printf("Enter a non-negative number to be converted"
             "to decimal notation: ");
      n = read_in_base(base);
      if (n >= 0)
        printf("The decimal number is: %d\n", n);
      else {
        printf("You have typed an illegal ciffer - TRY AGAIN\n");
        skip_rest_of_input_line();
      }
    }
  } while (base > 0);

  return 0;
}

/* Read a number in base and return the corresponding decimal number.
   Return -1 if erroneous input is encountered. */
int read_in_base(int base){
  int ciffer_number, res = 0, done = 0;
  char ciffer;

  do {
    ciffer = getchar();
    if (ciffer >= '0' && ciffer <= '9')
      ciffer_number = ciffer - '0';
    else if (ciffer >= 'a' && ciffer <= 'z')
      ciffer_number = ciffer - 'a' + 10;
    else if (ciffer == '\n')
      done = 1;
    else return -1;

    if (ciffer_number >= 0 && ciffer_number < base){
      if (!done) res = res * base + ciffer_number;
    }
    else return -1;
  }
  while (!done);

  return res;
}

void skip_rest_of_input_line (void){
  char c;
```

```
  c = getchar();
  while (c != '\n') c = getchar();
}
```

Program 16.8 *Et C program der omregner et tal fra et talsystem til titalsystemet og udskriver resultatet.*

Lad os lige diskutere et par detaljer i program 16.8. I funktionen `read_in_char` bruger vi en do løkker, idet vi altid skal læse mindst ét tegn. If-else kæden skelner mellem normale cifre og udvidede cifre, idet `ciffer_number` beregnes forskelligt i de to tilfælde. Der er også et tilfælde som fanger *end of line* tegnet, og som sætter den boolske `done` til true (1). Hvis vi falder hele vejen gennem if-else kæden til den afsluttende else returnerer vi umiddelbart -1, som tegn på en fejl. I dette tilfælde har vi mødt et ulovligt ciffertegn.

Den sidste if i do-løkken tester om det indlæste ciffer er ikke-negativ og mindre end `base`. Hvis dette ikke er tilfældet returnerer vi igen umiddelbart -1.

Da jeg programmerede `read_in_base` måtte jeg gøre mig umage for at styre tilfældene

1. Udelukkende lovlige cifre afsluttet med '\n', som resulterer i normal afslutning af do løkken styret ved hjælp af variablen `done`.
2. Forekomst af et ulovligt ciffertegn, f.eks. '?'
3. Forekomst af et ciffertegn der er større end `base`.

Normalt bestræber vi os på kun at have ét returneringspunkt i en funktion. Jeg fandt det imidlertid mest naturligt at have flere `return` kommandoer i `read_in_base`, idet én returnering til sidst i funktionen gjorde styringen af kontrollen for kompliceret.

I programmet herefter programmerer vi den omvendte konvertering. Her indlæser vi i `main` et normal heltal, som overføres til C funktionen `print_in_base`. Vi indlæser også et basistal for det talsystem, hvortil vi ønsker at konvertere.

Funktionen `print_in_base` finder cifrene i tallet i det andet talsystem. Endvidere udskriver funktionen de beregnede cifre.

```c
#include <stdio.h>

void print_in_base(int, int);

int main(void) {
  int i, n, base;

  for (i = 1; i <= 5; i++){
    printf("Enter positive decimal number "
           "and number base (at least two): ");
    scanf(" %d %d", &n, &base);
    print_in_base(n, base); printf("\n");
  }

  return 0;
}

/* Convert the decimal number n to base and print the result */
void print_in_base(int n, int base){
  int ciffer;

  if (n > 0){
    ciffer = n % base;    /* find least significant ciffer */

    /* RECURSIVELY, find and print most significant ciffers */
    print_in_base(n / base, base);

    /* print least significant ciffer */
    if (ciffer >= 0 && ciffer <= 9)
      putchar('0' + ciffer);
    else if (ciffer >= 10 && ciffer <= 36)
      putchar('a' + ciffer - 10);
    else putchar('?');
  }
}
```

Program 16.9 *Et C program der omregner et tal fra titalsystemet til et andet talsystem og udskriver resultatet.*

Igen et par detaljer om konverteringen. I `print_in_base` finder vi først det mindst betydende ciffer. Ved at studere formlen for opløsningen af et tal ud fra et grundtal i et talsystem (se afsnit 16.6) ser man let at det mindst betydende ciffer fås ved at finde resten af tallet ved division med grundtallet `base`.

Vi husker på det mindst betydende ciffer i variablen `ciffer`. Den resterende del af problemet kan løses rekursivt, ved at kalde `print_in_base` på `n / base`. Dette vil give de mest betydende cifre, som vi udskriver før vi udskriver det mindst betydende ciffer, som stadig findes i variablen `ciffer`.

I stedet for at bruge rekursion kunne vi lave en løkke. Den rekursive løsning er dog 'fiks', idet den gør det let og naturligt at udskrive cifrene i den ønskede rækkefølge (mest betydende før mindre betydende).

16.11. Eksempel: Kopiering af fil

Lektion 4 - slide 12

Vi finder det naturligt, ligesom i bogen, at vise hvordan vi laver et filkopierings program. Som vi ser i program 16.10 er dette meget let i C. Med brug af file redirection, ala program 16.11 kan vi tilmed nærme os en ganske bekvem aktivering af filkopieringen.

Det er meget simpelt at skrive et program som kopierer en fil til en anden fil.

```
#include <stdio.h>

int main(void) {

  int c;

  while ((c = getchar()) != EOF)
    putchar(c);

  return 0;
}
```

Program 16.10 *Et C program der kopierer input til output.*

```
normark$ gcc -o copy-file copy.c
normark$ copy-file < copy.c > copy-of-copy.c
```

Program 16.11 *Oversættelse og udførelse af copy.c.*

16.12. Eksempel: Optælling af ord

Lektion 4 - slide 13

Vi slutter af med at se på lærebogens eksempel på et program, der tæller antal af ord i en fil. Programmet er vist i program 16.12.

Jeg synes ikke at programmet er vellykket. Der er i alt tre løkker involveret. Ét i main, og to indlejrede løkker i found_next_word. Der skal konstant holdes udgik efter *end of file*, hvilket skæmmer alt for meget.

Man kan forestille sig at programmet startede simpelt, ud fra en rimelig idé om abstraktionen `found_next_word`. Men resultatet virker som en lappeløsning, hvor programmøren er kommet i tanker om flere og flere nødvendige specialtilfælde i kroppen af `found_next_word`.

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

int found_next_word(void);

int main()
{
   int   word_cnt = 0;

   while (found_next_word() == 1)
      ++word_cnt;
   printf("\nNumber of words = %d\n\n", word_cnt);
   return 0;
}

int found_next_word(void)
{
   int   c;

   while (isspace(c = getchar()))
      ;                 /* skip white space */
   if (c != EOF) {      /* found a word */
      while ((c = getchar()) != EOF && !isspace(c))
       ;      /* skip all except EOF and white space */
      return 1;
   }
   return 0;
}
```

Program 16.12 *Bogens udgave af ordtællingsprogrammet - et uskønt program.*

Vi viser et tilsvarende eksempel fra kernighan og Ritcies oprindelige bog om C, *The C Programming Language*. Dette program har én løkke, som læser alle tegn indtil end of file. Programmet tæller både tegn, liner og ord.

Ordtællingen er det mest komplicerede element, som jeg derfor forklarer. Variablen `state` bruges til formålet. `state` starter med at være `OUT`, hvilket betyder at vi er uden for et ord. `OUT` er en symbolsk konstant, defineret med `define`.

Når vi i tilstanden `OUT` møder det første tegn i et ord tælles variablen `nw` op. Vi skifter også tilstand til `IN`. Vi tæller altså et ord ved indgangen til ordet. I tilstand `IN` bliver `nw` ikke talt yderligere op. Først når vi forlader ordet, og kommer i tilstand `OUT`, har vi mulighed for at møde det næste ord, som så bliver talt.

```c
#include <stdio.h>

#define IN 1   /* inside a word */
#define OUT 0  /* outside a word */

int main(void) {
  int c, nl, nw, nc, state;

  state = OUT;
  nl = nw = nc = 0;
  while ((c = getchar()) != EOF) {
    ++nc;
    if (c == '\n')
      ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
      state = OUT;
    else if (state == OUT) {   /* entering a word */
      state = IN;
      ++nw;
    }
  }

  printf("%d %d %d\n", nl, nw, nc);

  return 0;
}
```

Program 16.13 *Kernighan og Ritchies's tilsvarende, men mere generelle program.*

Dette afslutter vores behandling af tegn.