43. Introduktion til filer

Dette kapitel starter den sidste lektion i programmeringskurset. Vi diskuterer først et antal overordnede filbegreber. Dernæst glider vi over i filbegrebet i C, herunder en mere uddybende behandling af formateret output og input end vi var igennem i kapitel 4. Lektionen sluttes af med en diskussion af input og output af structures.

43.1. Oversigt over filbegreber

Lektion 10 - slide 2

En fil er en samling af data på et eksternt datamedie. Herunder sondrer vi mellem to forskellige klassificeringer af filer, dels efter måden hvorpå indholdet tilgås, og dels efter repræsentationen af indholdet.

- **Datatilgang:**
  - Sekventiel
  - Random access
- **Dataindhold:**
  - Tegn
    - Filer der indeholder bytes der tolkes som tegn fra ASCII alfabetet
  - Andre binære data
    - Filer der indeholder bitmønstre fra andre datatyper end tegn

Filer er ressourcer som administreres af operativsystemet

Fra et programmeringssprog er det muligt at knytte en forbindelse til en fil i operativsystemet

43.2. Sekventielle filer

Lektion 10 - slide 3

Selv om sekventielle filer ret beset er et levn fra fortiden, er størstedelen af vores filbehandling den dag i dag af sekventiel natur. Det ytrer sig ved at vi som regel læser filen fra start til slut, at vi hele tiden vedligeholder 'den nuværende possition', og at vi kun sjældent eksplict vælger at hoppe fra et sted til et andet i filen.

Sekventielle filer er modelleret efter egenskaberne af sekventielle medier, såsom magnetbånd

En *sekventiel fil* læses altid i den rækkefølge den er skrevet. Rækkefølgen af enheder i en sekventiel fil afspejler direkte rækkefølgen af de udførte skriveoperationer på filen.

- Mode of operation
  - En sekventiel fil tilgås enten i *læse mode* eller *skrive mode*
  - I det simple tilfælde kan man ikke både læse fra og skrive til en fil
- Ydre enheder
  - Tastaturet modelleres typisk som en sekventiel fil i læse mode
    - Standard input
  - Skærmen modellers typisk som en sekventiel fil i skrive mode
    - Standard output

Vi har lige fra starten af dette materiale benyttet os af funktioner som tilgår tastatur og skærm - altså standard input og standard output filerne. Vores første møde med disse emner var i kapitel 4.

43.3. Random access filer

Lektion 10 - slide 4

På det begrebslige plan har vi ikke så meget at sige om random access filer.

Filer på en harddisk er ikke af natur sekventielle filer

Det er muligt og naturligt at skrive og læse i en mere vilkårlig rækkefølge

Langt de fleste filer vi omgås i vores dagligdag er lagret på harddiske, og sådanne filer kan uden problemer tilgås pr. random access. Andre filer er dog datastrømme af forskellige slags, såsom filer der modtages over et netværk. Sådanne filer er som oftest sekventielle.

44. Filer i C

Vi vender nu blikket mod filer i C. Vi ser på åbning og lukning af filer i C, strukturen FILE som repræsenterer filer i C, tegnvis skrivning og læsning af filer, standard input, output og error, opening modes, og endelig et lille repertoire af funktioner fra standard biblioteket til håndtering af både sekventielle og random access filer i C.

44.1. Sekventielle filer i C

Lektion 10 - slide 6

Filer i C håndteres via biblioteket **`stdio.h`**

Sekventielle filer i C kan ses som abstraktioner over filerne i operativsystemet

Sådanne abstraktioner omtales ofte som en *stream*

- Åbning af en fil
  - Knytter forbindelsen til en fil i operativsystemets filhierarki
  - Angivelse af *opening mode*
  - Etablerer en pointer til en **`FILE`** struct
- Processering af filen
  - **`stdio.h`** tilbyder et stort antal forskellige funktioner til sekventiel processering af en fil
- Lukning af filen
  - Tømmer buffere og frigiver interne ressourcer
  - Bryder forbindelsen til filen i operativsystemet

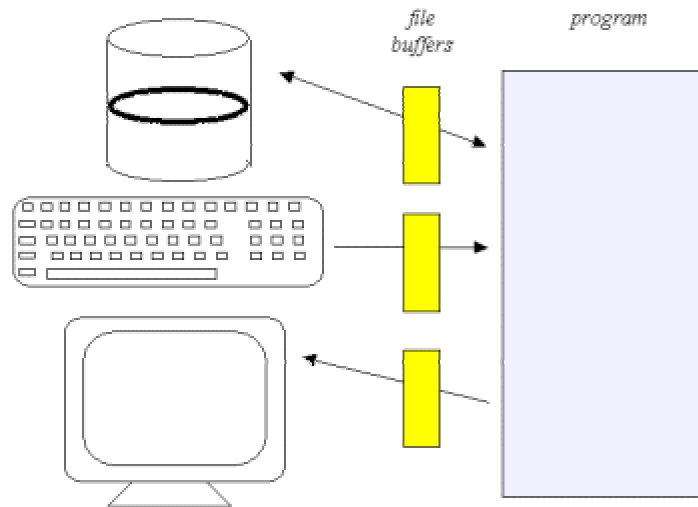
44.2. Filbuffering

Lektion 10 - slide 7

En filbuffer er et lagerområde mellem anvendelsesprogrammet og det fysiske lager, hvor filen er gemt. Buffer ideen er illustreret i figur 44.1.

Hvis man fra en harddisk beder om ganske få bytes af data i en sekventiel læsning af en fil vil dette mest sandsynligt gå meget langsomt, med mindre man indfører en buffer. Årsagen er at programmet gentagne gange skal vente på at disken kommer forbi læsehovedet. Med buffering læses en forholdsvis stor portion af disken ind i bufferen når lejlighed gives, og efterfølgende tilgang til disse data fra programmet kræver derfor ikke fysisk læsning på selve disken.

Når vi har indført buffering skal vi være opmærksomme på få disken fuldt opdateret inden programmet lukkes ned. Dette sker når vi lukker filen, eller når vi eksplicit beder om det ved at kalde f.eks. C funktionen `fflush` fra `stdio.h`.



Figur 44.1 *En illustration af filbuffere mellem den ydre enhed og programmet*

44.3. Strukturen FILE

Lektion 10 - slide 8

En fil i et C program er i praksis en pointer til en FILE structure. Structures blev behandlet i kapitel 39. Vi vil ikke her beskrive de enkelte felter i FILE strukturen. Vi gør en dyd ud af ikke at gøre os afhængige af denne viden. I den forstand opfatter vi filer som objekter i en abstrakt datatype, jf. kapitel 42.

Strukturen FILE beskriver egenskaberne af en fil, herunder det afsatte bufferområde

Helt overordnet er følgende blandt egenskaberne af FILE:

- Den nuværende filposition
- Detaljer om filbufferen
- Andre interne aspekter

Når talen er om filens øjeblikkelige tilstand er den *nuværende filposition* af stor betydning. Det er vigtigt at forstå, at i takt med at vi læser en sekventiel fil vil filpositionen gradvis forøges.

Programmøren udnytter normalt ikke kendskab til detaljerne i FILE

FILE og de tilknyttede operationer er et godt eksempel på en abstrakt datatype

44.4. Simpel skrivning og læsning af en fil

Lektion 10 - slide 9

Vi vender os nu mod små, illustrative eksempler. Vi ser først på tegnvis skrivning og læsning. Eksemplerne i program 44.1, program 44.2 og senere program 44.4 arbejder alle på en fast fil, som vi kalder `"first-file"`.

```
#include <stdio.h>

int main(void) {

  FILE *output_file_pointer;
  char *str = "0123456789abcdefghijklmnopqrstuvw";
  int i;

  output_file_pointer = fopen("first-file", "w");

  while (*str != '\0'){
    fputc(*str, output_file_pointer);
    str++;
  }

  fclose(output_file_pointer);
  return 0;
}
```

Program 44.1   *Skrivning af tegn fra en tekststreng på en tekstfil.*

Vi diskuterer her program 44.1. Overordnet skriver programmet - tegn for tegn - strengen `str` på filen `first-file`. På det blå sted erklærer vi variablen `ouput_file_pointer` som en pointer til `FILE`. På det første røde sted åbner vi filen i skrive mode, `"w"` for write. While-løkken gennemløber tegnene i `str`, og ved brug af standard io funktionen `fputc` skrives hvert tegn i `str` til output filen. Vi erindrer `*str` dereferencer pointeren - dvs. at vi tilgår hvert enkelt tegn i strengen gennem pointeren. (Dereferencing er beskrevet i afsnit 22.4). `str++` tæller pointeren op pr. pointer aritmetik, jf. afsnit 24.5. Afsluttende, på det sidste røde sted, lukker vi filen.

Når vi kører program 44.1 får vi lavet `first-file`. I program 44.2 læser vi denne fil og putter tegnene tilbage i en tekststreng.

```c
#include <stdio.h>
#define MAX_STR_LEN 100

int main(void) {

  FILE *input_file_pointer;
  char str[MAX_STR_LEN];
  int ch;
  int i = 0;

  input_file_pointer = fopen("first-file", "r");

  while ((ch = fgetc(input_file_pointer)) != EOF){
    str[i] = ch;
    i++;
  }
  str[i] = '\0';

  printf("Read from file: %s\n", str);

  fclose(input_file_pointer);
  return 0;
}
```

Program 44.2 *Læsning af tegn fra en tekstfil til en tekststreng.*

Vi forklarer nu program 44.2. Ligesom i program 44.1 erklærer vi en `FILE*` variabel, og vi åbner filen, dog her i læse mode. I en while-løkke læser vi tegnene i filen, indtil vi møder EOF. Det boolske udtryk i while-løkken er et typisk C mønster som kalder `fgetc`, assigner til variablen `ch`, og tester om det læste tegn er forskelligt fra EOF. Bemærk at `ch` er erklæret som en `int`, ikke en `char`. Årsagen er at EOF ikke er en gyldig char værdi, men derimod typisk -1. Et kald af `fgetc` læser netop ét tegn. De læste tegn lægges i arrayet `str`. Vi afslutter med lukning af filen.

Det er altid en god ide at placere filåbning og fillukning i samme funktion, idet det sikrer (langt hen ad vejen i det mindste) at en åben fil altid bliver lukket. Åbning og lukning optræder således altid i par.

I program 44.3 viser vi et program fra lærebogen, som laver dobbelt linieafstand i en tekstfil.

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

void   double_space(FILE *ifp, FILE *ofp);
void   prn_info(char *pgm_name);

int main(int argc, char **argv)
{
   FILE   *ifp, *ofp;

   if (argc != 3) {
      prn_info(argv[0]);
      exit(1);
```

```
   }
   ifp = fopen(argv[1], "r");      /* open for reading */
   ofp = fopen(argv[2], "w");      /* open for writing */
   double_space(ifp, ofp);
   fclose(ifp);
   fclose(ofp);
   return 0;
}

void  double_space(FILE *ifp, FILE *ofp)
{
   int  c;

   while ((c = fgetc(ifp)) != EOF) {
      fputc(c, ofp);
      if (c == '\n')
       fputc('\n', ofp); /* found newline - duplicate it */
   }
}

void prn_info(char *pgm_name)
{
   printf("\n%s%s%s\n\n%s%s\n\n",
      "Usage:  ", pgm_name, "  infile  outfile",
      "The contents of infile will be double-spaced ",
      "and written to outfile.");
}
```

Program 44.3 *Et program der laver dobbelt linieafstand i en tekstfil.*

Programmet accepterer to programparametre, jf. afsnit 31.3. Et typisk kald af programmet er således

```
dbl_space infile outfile
```

forudsat at det oversatte program placeres i filen `dbl_space`. Det kan gøres med en `-o` option til gcc compileren. Filåbning og lukning er vist med blåt. Med brunt vises kaldet af den centrale funktion `double_space`, der laver arbejdet. I denne funktion læses et tegn med `fgetc`, og det skrives med `fputc`. Med rødt ser vi en test for at variablen `c` er newline tegnet. Når dette tegn mødes udskrives det endnu engang. Dermed opnås effekten af dobbelt linieafstand.

Når en fil er læst til ende vil **`fgetc`** og andre tilsvarende funktioner returnere **`EOF`** værdien

44.5. Standard input, output og error

Lektion 10 - slide 10

I dette afsnit skærper vi vores forstand af begreberne standard input og standard output.

De tre filer **stdin**, **stdout**, og **stderr** åbnes automatisk ved program start

Hver af **stdin**, **stdout**, og **stderr** er pointere til FILE

- Standard input - **stdin**
    - Default knyttet til tastaturet
    - Kan omdirigeres til en fil med *file redirection*
- Standard output - **stdout**
    - Default knyttet til skærmen
    - Kan omdirigeres til en fil med *file redirection*
- Standard error - **stderr**
    - Default knyttet til skærmen
    - Benyttes hvis **stdout** ikke tåler output af fejlmeddelelser

File redirection er illustreret i program 16.11.

Der findes en række behændige filfunktioner, som implicit arbejder på **stdin** og **stdout** i stedet for at få overført en pointer til en FILE structure

Bemærkning i rammen ovenfor er f.eks. rettet mod funktionerne `printf`, `scanf`, `putchar`, og `getchar`.

44.6. Fil opening modes

Lektion 10 - slide 11

Vi skal have helt styr på et antal forskellige opening modes af filer i C.

Foruden læse- og skrivemodes tilbydes yderligere et antal andre opening modes

r	Åbne eksisterende fil for input
w	Skabe ny fil for output
a	Skabe ny fil eller tilføje til eksisterende fil for output
r+	Åbne eksisterende fil for både læsning og skrivning. Start ved begyndelse af fil.
w+	Skabe en ny fil for både læsning og skrivning
a+	Skabe en ny fil eller tilføje til eksisterende fil for læsning og skrivning

Tabel 44.1 *En tabel der beskriver betydningen af de forskellige opening modes af filer.*

Opening modes "r" og "w" er enkle at forstå. Bemærk at med opening mode "w" skabes en ny fil, hvis filen ikke findes i forvejen. Hvis filen findes i forvejen slettes den! Opening mode "a" er bekvem for tilføjelse til en eksisterende fil. Som vi vil se i program 44.4 sker tilføjelsen i bagende af filen. Heraf naturligvis navnet 'append mode'.

Opening mode "r+" er ligesom "r", blot med mulighed for skrivning side om side med læsningen. Dette virker i Unix, men ikke i Windows og Dos. Der er nærmere regler for kald af (eksempelvis) `fflush` mellem læsninger og skrivninger. Vi beskriver ikke disse nærmere her. Dette illustreres i program 44.5.

Opening mode "w+" er ligesom "w" med mulighed for læsning side om side med skrivning.

I program 44.4 viser vi tilføjning af "`Another string`" til indholdet af "`first-file`". Bemærk brug af "a" opening mode.

```
#include <stdio.h>

int main(void) {

  FILE *output_file_pointer;
  char *str = "Another string";
  int i;

  output_file_pointer = fopen("first-file", "a");

  while (*str != '\0'){
    fputc(*str, output_file_pointer);
    str++;
  }

  fclose(output_file_pointer);
  return 0;
}
```

Program 44.4 *Tilføjelse af tegn til en tekstfil - simple-append.*

Programmet i program 44.5 benytter opening mode `"r+"`. For hver læsning skrives et `'X'` tegn. Det betyder at hvert andet tegn i filen bliver `'X'` tegnet. Så hvis `first-file` indledningsvis indeholder strengen "`abcdefg`", og vi dernæst kører programmet, bliver filindholdet af `first-file` ændret til at være "`aXcXeXgX`". Variablen `str` ender med at have værdien "`aceg`", som udskrives på standard output.

Som allerede omtalt virker dette program kun på Unix, idet der er begrænsninger for skrivning i en fil i "r" modes i Windows og Dos.

```
#include <stdio.h>
#define MAX_STR_LEN 100

int main(void) {

  FILE *input_file_pointer;
  char str[MAX_STR_LEN], ch;
```

```
  int i = 0;

  input_file_pointer = fopen("first-file", "r+");

  while ((ch = fgetc(input_file_pointer)) != EOF){
    str[i] = ch;

    fflush(input_file_pointer);
    fputc('X', input_file_pointer);
    fflush(input_file_pointer);

    i++;
  }
  str[i] = '\0';

  printf("Read from file: %s\n", str);

  fclose(input_file_pointer);
  return 0;
}
```

Program 44.5 *Læsning og skrivning af fil med r+ - simple-read-update.*

C muliggør åbning af binære filer ved at tilføje et `b` til opening mode

I C på Unix er der ikke forskel på binære filer og tekstfiler

44.7. Funktioner på sekventielle filer

Lektion 10 - slide 12

Vi giver i dette afsnit en oversigt over de mest basale tegn-orienterede funktioner, som virker på sekventielle filer i C.

- `int fgetc(FILE *stream)`
  - Læser og returnerer næste tegn fra stream. Returnerer EOF hvis der er end of file.
- `int fputc(int c, FILE *stream)`
  - Skriver tegnet c til stream.
- `int ungetc(int c, FILE *stream)`
  - Omgør læsningen af c. Tegnet c puttes altså tilbage i stream så det kan læses endnu en gang. Kun én pushback understøttes generelt.
- `char *fgets(char *line, int size, FILE *stream)`
  - Læser en linie ind i line, dog højst size-1 tegn. Stopper ved både newline og EOF. Afslutter line med '\0' tegnet.
- `int fputs(const char *s, FILE *stream)`
  - Skriver strengen s til stream, uden det afsluttende '\0' tegn.

Læsere af C by Dissection kan slå disse og en del af andre funktioner op i appendix A.

44.8. Funktioner på random access filer

Lektion 10 - slide 13

Ligesom for sekventielle filer i afsnit 44.7 viser i i dette afsnit de væsenligste funktioner på random access filer i C.

Begrebsligt kan man tilgå en random access fil på samme måde som et array

Følgende funktioner er centrale for random access filer:

- **`int fseek(FILE *fp, long offset, int place)`**
  - Sætter filpositionen for næste læse- eller skriveoperation
  - **`offset`** er en relativ angivelse i forhold til **`place`**
  - **`place`** er **`SEEK_SET`**, **`SEEK_CUR`**, **`SEEK_END`** svarende til filstart, nuværende position, og filafslutning
- **`long ftell(FILE *fp)`**
  - Returnerer den nuværende værdi af filpositionen relativ til filstart
- **`void rewind(FILE *fp)`**
  - Ækvivalent til **`fseek(fp, 0L, SEEK_SET)`**

Vi illustrerer random access filer ved et af lærebogens eksempler, nemlig læsning af en fil baglæns. Altså læsning stik modsat af sekventiel læsning.

I program 44.6 prompter vi brugeren for filnavnet. Efter filåbning placerer vi os ved sidste tegn i filen. Dette sker med det første røde `fseek`. Vi går straks ét tegn tilbage med det andet røde `fseek`. I while-løkken læser vi tegn indtil vi når første position i filen. For hvert læst tegn går vi to trin tilbage. Husk at læsning med `getc` bringer os ét tegn frem. Kaldet af `getc(ifp)` er ækvivalent med `fgetc(ifp)`. Det læste tegn udskrives på standard output med `putchar`. Det første tegn skal behandles specielt af det tredje røde `fseek` kald. Årsagen er naturligvis, at vi ikke skal rykke to pladser tilbage på dette tidspunkt.

```
#include <stdio.h>

#define   MAXSTRING   100

int main(void){
   char    filename[MAXSTRING];
   int     c;
   FILE    *ifp;

   fprintf(stderr, "\nInput a filename:  ");
   scanf("%s", filename);
   ifp = fopen(filename, "r");
   fseek(ifp, 0, SEEK_END);
   fseek(ifp, -1, SEEK_CUR);
```

```
    while (ftell(ifp) > 0) {
       c = getc(ifp);
       putchar(c);
       fseek(ifp, -2, SEEK_CUR);
    }
    /*  The only character that has not been printed
        is the very first character in the file. */

    fseek(ifp, 0, SEEK_SET);
    c = getc(ifp);
    putchar(c);
    return 0;
}
```

Program 44.6 *Et program der læser en fil baglæns.*

45. Formateret output og input

Formateret output er kendetegnet af at værdier af forskellig typer kan udskrives med en stor grad af kontrol over deres tekstuelle præsentation. Vi så første gang på `printf` i afsnit 4.1.

Formateret input involverer læsning og omformning af tekst til værdier i forskellige scalare typer. Scalartyper blev diskuteret i afsnit 18.1.

45.1. Formateret output - printf familien (1)

Lektion 10 - slide 15

Vi taler om en familie af `printf` funktioner, idet der findes næsten identiske `printf` funktioner som virker på standard output, på en fil, og på tekststrenge. Disse adskilles af forskellige prefixes (det der indleder navnet): **`f`**, **`s`** og intet prefix.

Suffixet **`f`** (det der afslutter navnet) er en forkortelse for "formated".

> Familien af `printf` funktioner tilbyder tekstuel output af værdier i forskellige typer, og god kontrol af output formatet
>
> Familien omfatter funktioner der skriver på en bestemt fil, på standard output, og i en streng

Det overordnede formål med kald af `printf` kan formuleres således:

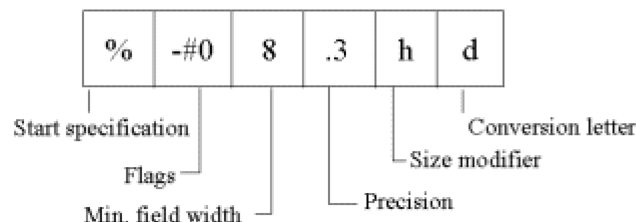
- Pretty printing
- Konvertering af værdier fra forskellige typer til tekst
- Kontrol af det tekstuelle format

I dette materiale vil vi på ingen måde forsøge at dække alle detaljer om `printf`. C by Dissection har en del detaljer. I de næste afsnit peger vi på nogle af de væsentligste detaljer.

45.2. Formateret output - printf familien (2)

Lektion 10 - slide 16

I dette afsnit vil vi anskueliggøre bestanddelene af kontrolstrengen i `printf`. I figur 45.1 viser vi en nedbrydning af kontrolstrengen. For at føre diskussionen rimelig konkret anvender vi et konkret eksempel: **`%-#08.3hd`**.



Figur 45.1 *Nedbrydning af kontrolstrengen for %-#08.3hd. Denne figur er inspireret fra 'A C Reference Manual' af Harbison & Steele*

Forklaringen af **`%-#08.3hd`**, som vist ovenfor, følger her:

- Conversion letter: **`d`** for heltalsformatering
- Flags: **`-`** left justify, **`#`** use variant (ikke anvendelig her), **`0`** zero padding
- Min. field width: **`8`**. Mindste plads som afsættes
- Precision: **`3`** Mindste antal cifre der udskrives - bruger 0 paddings om nødvendig
- Size modifier: **`h`** short

Her følger et antal udvalgte, væsentlige observationer om funktioner i `printf` familien:

- Returnerer antallet af udskrevne tegn
- Man kan kontrollere højre og venstre justering i felt
  - Et **`-`** flag betyder venstrejustering
- Field width og precision kan indlæses som input
  - Angivelse med **`*`**

45.3. Formateret input - scanf familien (1)

Lektion 10 - slide 18

I samme stil som diskussionen af `printf` i afsnit 45.1 foretager vi her en overordnet gennemgang af funktioner i `scanf` familien.

De familiemæssige forhold for `scanf` modsvarer familieforholdene for `printf`, som omtalt i starten af afsnit 45.1.

> Familien af `scanf` funktioner tilbyder tekstuel input (parsing, scanning) af værdier i forskellige typer
>
> Familien omfatter funktioner der læser fra en bestemt fil, fra standard input, og fra en streng

Det overordnede formål med `scanf` er følgende:

- Fortolkning og parsing af tekst
- Konvertering af tekstudsnit til værdier i forskellige typer
- Assignment af værdier til parametre overført pr. reference - *call by reference parametre*

> Parsning er vanskeligere end pretty printing
>
> Ligesom `printf`, er der mange detaljer som knytter sig til de mulige typer, hvortil `scanf` kan konvertere

45.4. Formateret input - scanf familien (2)

Lektion 10 - slide 19

Vi opregner i dette afsnit en række nyttige begreber, som gør det lettere at forstå anvendelser af `scanf`.

- *Kontrolstreng* med et antal *direktiver*
- Tre slags *direktiver*
  - *Almindelige tegn*: Matcher de angivne tegn
  - *White space*: Matcher så mange mellemrum, tabs, og linieskift som mulig
  - *Konverteringsspecifikation*: `%...c`:
    Matcher input efter givne regler og konverteres til en værdi som angivet af `c`.
- *Scan field*:
  - Et sammenhængende område i inputet som forsøges konverteret med en given konverteringsspecifikation.
  - *Scan width*: En angivelse i konverteringsspecifikationen af den maksimale længde af et scan field.
  - Består af et område uden white space (på nær for `%c`).

Bid mærke i følgende observationer om `scanf`:

- Returværdi
  - EOF hvis input 'løber tør' - *input failure*
  - Antallet af gennemførte konverteringer (indtil evt. *matching failure*)
- Assignment suppression `*`
  - Matching - men ingen assignment til pointervariable
- Scan set af formen `[cde]`
  - Indlæsning af en streng fra alfabetet bestående af 'c', 'd' og 'e'.
- Scan set af formen `[^cde]`
  - Indlæsning af en streng afsluttet af et tegn fra alfabetet bestående af 'c', 'd' og 'e'.

Anvendelse af scansets i stedet for et bestemt kontroltegn, ala `c` eller `d`, er nyttig i mange situationer. Eksemplerne i afsnit 45.5 er centreret om scansets.

45.5. Formateret input - scanf familien (4)

Lektion 10 - slide 21

Det første eksempel, vist i program 45.1, er et eksempel fra C by Dissection. Vi læser en fil, hvis navn er overført som programparameter (se afsnit 31.3). Alle tomme liner, og alle blanke indrykninger elimineres. Det afkortede indhold udskrives på standard output.

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  FILE *ifp = fopen(argv[1],"r");

  char line[300];
  int i = 0;

  while (fscanf(ifp," %[^\n]", line) == 1){
    printf("%s\n", line);
    i++;
  }

  printf("\n\ni = %i\n",i);

  fclose(ifp);

  return 0;
}
```

Program 45.1 *Et program der læser ikke-blanke liner fra en fil og udskriver disse på standard output.*

Den afgørende detalje i program 45.2 er fremhævet med rødt. Vi ser et scanset, og ikke mindst et white space directive (et mellemrum før `%[^\n]`). Scansettet `%[^\n]` vil få `fscanf` til at læse indtil et newline mødes. Næste tur i while-løkken, nærmere betegnet næste kald af `fscanf` i løkken, vil pga. white space directivet overspringe alt white space. Dette er det newline der stoppede det

forrige kald af fscanf, tomme linier, og slutteligt blanke tegn i indrykningen på den næste ikke tomme line. Alt dette er en lille smule tricket!

Vi ser nu på et (antal versioner af et) program der separerer alfabetiske tegn og numeriske tegn i input, der læses fra standard input med scanf. Det er stadig en pointe for os at illustrere scan sets.

```
#include <stdio.h>

int main(void) {

  char alph[25], numb[25];
  int numres;

  printf("Enter input, such as \"aaa111bbb222\"\n");

  while (scanf("%[abcdefghijklmnopqrstuvwxyz]", alph) == 1){
    numres = scanf("%[0123456789]", numb);

    if (numres != 1)
       printf("MISMATCH");
    else
       printf("Alphabetic: %s\n"
              "Numeric: %s\n\n",
              alph, numb);
  }

  return 0;
}
```

Program 45.2 *Et andet program der adskiller alfabetiske og numeriske afsnit i en tekst - læser fra stdin.*

Lad os forklare hvad vi ser i program 45.2. Det røde kald af scanf læser igennem et alfabetisk afsnit af inputtet. Læsningen stoppes ved første tegn, som ikke er i det røde scanset. Det blå scanf læser tilsvarende igennem numeriske tegn, og stopper ved først tegn som ikke er et ciffer. Læg mærke til hvordan vi bruger returværdierne af scanf. Hvis der er en streng vekselvirkning mellem alfabetiske afsnit og numeriske afsnit af inputtet, afsluttet af et numerisk afsnit kører programmet til ende. Ellers får vi et "MISMATCH".

Herunder ser vi på en variant af program 45.3 som læser fra en streng med sscanf i stedet for at læse fra standard input (en fil) med scanf eller fscanf.

```
#include <stdio.h>

int main(void) {

  char *str = "abc135def24681ghi3579";
  char alph[25], numb[25];
  int numres;

  while (sscanf(str,"%[abcdefghijklmnopqrstuvwxyz]", alph) == 1){
    numres = sscanf(str,"%[0123456789]", numb);

    if (numres != 1)
```

```
      printf("MISMATCH");
    else
      printf("Alphabetic: %s\n"
             "Numeric: %s\n\n",
             alph, numb);
  }

  return 0;
}
```

Program 45.3 *Et andet program der adskiller alfabetiske og numeriske afsnit i en tekst - læser fra en streng - virker ikke.*

Hvis vi prøvekører program 45.3 må vi konstatere, at det ikke virker. Hvorfor ikke? Vi har input i strengen `str`. De to røde kald af sscanf læser fra `str`.

Pointen i program 45.3 er at `sscanf` læser fra starten af strengen i hvert kald. Der er altså ingen nuværende position som bringes fremad for hvert kald af `sscanf`. Når vi læser fra filer med `scanf` eller `fscanf` læser vi altid relativ til den nuværende position i filen. Denne position tælles op i takt med vi læser. Men sådan er det altså ikke, når vi læser fra en streng!

Herunder, i program 45.4, har vi repareret på problemet. Vi "bringer fremdrift i strengen" ved at tælle pointeren op på de røde steder i program 45.4. Med denne ændring virker programmet tilfredsstillende.

```
#include <stdio.h>

int main(void) {

  char *str = "abc135def24681ghi3579";
  char alph[25], numb[25];
  int numres;

  while (sscanf(str,"%[abcdefghijklmnopqrstuvwxyz]", alph) == 1){
    str += strlen(alph);

    numres = sscanf(str,"%[0123456789]", numb);
    str += strlen(numb);

    if (numres != 1)
      printf("MISMATCH");
    else
      printf("Alphabetic: %s\n"
             "Numeric: %s\n\n",
             alph, numb);
  }

  return 0;
}
```

Program 45.4 *Et andet program der adskiller alfabetiske og numeriske afsnit i en tekst - læser fra en streng - virker!.*

Med disse eksempler afsluttes vores diskussion af formateret output og input.

46. Input og output af structures

Vi har indtil dette sted kun beskæftiget os med input og output af de fundamentale, skalare typer i C, samt af tekststrenge. I dette afsnit diskuterer vi input og output af structures. Structures blev introduceret i kapitel 39.

I mange praktiske sammenhænge er det nyttigt at kunne udskrive en struct på en fil og at kunne indlæse den igen på et senere tidspunkt, måske fra et helt andet program.

46.1. Input/Output af structures (1)

Lektion 10 - slide 23

Det er ofte nyttigt at udskrive en eller flere structs på en fil, og tilsvarende at kunne indlæse en eller flere structs fra en fil Det er nyttigt at have en standard løsning - et mønster - for dette problem

I dette afsnit vil vi udvikle et mønster for hvordan man kan udskrive og genindlæse structures på/fra filer. Løsningen kræver desværre en del arbejde, som knytter sig de specifikke detaljer i structure typen. I visse situationer kan man bruge en mere maskinnær løsning baseret på udskrivning af et bestemt bitmønster. Sidstnævnte er emnet i afsnit 46.3.

Vi ser på en række sammenhængende C programmer, og et tilhørende programbibliotek som udskriver og indlæser `struct book`, som vi introducerede i afsnit 39.6.

Det første program, program 46.1, laver to bøger med `make_book`, som vi mødte i program 39.6. Disse to bøger skrives ud med funktionen `print_book` på det røde sted. Udskrivningen sker i filen `books.dat`. Som vi vil se lidt senere er `books.dat` en tekstfil, dvs. en fil som er inddelt i tegn. Læg mærke til, at vi inkluderer biblioteket `book-read-write.h`. I denne header fil findes `struct book` og prototyper af de nødvendig tværgående funktioner for bog output og bog input. Biblioteket `book-read-write.h` kan ses i program 46.3. Implementationen af funktionerne i biblioteket ses i program 46.4.

```
#include <stdio.h>
#include "book-read-write.h"

int main(void) {

  book *b1, *b2;
  FILE *output_file;

  b1 = make_book("C by Dissection", "Kelly and Pohl",
                 "Addison Wesley", 2002, 1);
  b2 = make_book("The C Programming Language",
                 "Kernighhan and Ritchie",
                 "Prentice Hall", 1988, 1);
```

```
  output_file = fopen("books.dat", "w");

  print_book(b1, output_file);
  print_book(b2, output_file);

  fclose(output_file);

  return 0;

}
```

Program 46.1 *Programmet der udskriver bøger på en output fil.*

Når vi nu har udskrevet bøgerne på filen `books.dat` skal vi naturligvis også være i stand til at indlæse disse igen. Det kan evt. ske i et helt andet program. I program 46.2 inkluderer vi igen `book-read-write.h`, og vi åbner en fil i læse mode. Dernæst læser vi to bøger fra input filen på de røde steder. Vi kalder også `prnt_book` med det formål at skriver bogdata ud på skærmen. På denne måde kan vi overbevise os selv om, at det læste svarer til det skrevne.

```
#include <stdio.h>
#include "book-read-write.h"

int main(void) {

  book *b1, *b2;

  FILE *input_file;
  input_file = fopen("books.dat", "r");

  b1 = read_book(input_file);
  b2 = read_book(input_file);

  prnt_book(b1);   prnt_book(b2);

  fclose(input_file);

  return 0;

}
```

Program 46.2 *Programmet der indlæses bøger fra en input fil.*

Næste punkt på dagsordenen, i program 46.3 er header filen `book-read-write.h`. Vi ser først nogle symbolske konstanter, hvoraf `PROTECTED_SPACE` og `PROTECTED_NEWLINE` anvendes i indkodningen af bogdata som en tekststreng. Dernæst følger `struct book` og den tilhørende typedef. De sidste linier i `book-read-write.h` er prototyper (funktionshoveder) af de funktioner, vi har anvendt til bogkonstruktion, bogskrivning og boglæsning i program 46.1 og program 46.2.

```
#define PROTECTED_SPACE '@'
#define PROTECTED_NEWLINE '$'
#define BUFFER_MAX 1000

struct book {
  char *title, *author, *publisher;
  int publishing_year;
  int university_text_book;
};

typedef struct book book;

book *make_book(const char *title, const char *author,
                const char *publisher,
               int year, int text_book);

void prnt_book(book *b);

void print_book(book *b, FILE *ofp);

book *read_book(FILE *ifp);
```

Program 46.3 *Header filen book-read-write.h.*

Endelig følger implementationen af funktionerne i book-read-write biblioteket. Den fulde implementation kan ses vi den tilhørende slide. I udgaven i program 46.4 har vi udeladt funktionerne `make_book` og `prnt_book`. Disse funktioner er vist tidligere i program 39.6.

Funktionen `print_book` indkoder først bogens felter i et tegn array, kaldet `buffer`. Indkodningen foregår med funktionen `encode_book`. I `encode_book` skriver vi hvert felt i bogen ind i en tekststreng med hjælp af `sprintf`. Hver bog indkodes på én linie i en tekst fil. Felterne er adskilt med mellemrum (space). For at dette kan virke skal vi sikre os at mellemrum og evt. linieskift i tekstfelter transformeres til andre tegn. Dette sker i `white_space_protect`, som kaldes af `encode_book` på alle tekstfelter.

Tilsvarende indlæser funktionen `read_book` en bog fra en åben fil i læsemode. Da hver bog er repræsenteret som én linie i filen er det let at læse denne med `fgets`. Funktionen `decode_book` afkoder denne tekststreng. Essentielt kan arbejdet udføres med `sscanf`, som jo læser/parser tekst fra en tekststreng. I funktionen `white_space_deprotect` udfører vi det omvendte arbejde af `white_space_protect`. Det betyder at de normale mellemrum og linieskift vil vende tilbage i bøgernes tekstfelter.

```
char *white_space_protect(char *str){
  int str_lgt = strlen(str), i, j;
  for(i = 0; i < str_lgt; i++){
    if (str[i] == ' ')
      str[i] = PROTECTED_SPACE;
    else if (str[i] == '\n')
      str[i] = PROTECTED_NEWLINE;
  }
  return str;
```

```
}

char *white_space_deprotect(char *str){
  int str_lgt = strlen(str), i;
  for(i = 0; i < str_lgt; i++){
    if (str[i] == PROTECTED_SPACE)
      str[i] =  ' ';
    else if (str[i] == PROTECTED_NEWLINE)
      str[i] = '\n';
  }
  return str;
}

/* Encode the book pointed to by p in the string str */
void encode_book(book *b, char *str){
  sprintf(str, "%s %s %s %i %i\n",
          white_space_protect(b->title),
          white_space_protect(b->author),
          white_space_protect(b->publisher),
          b->publishing_year, b->university_text_book);
}

book *decode_book(char *str){
  char book_title[100], book_author[100], book_publisher[100];
  int book_year, book_uni;

  sscanf(str, "%s %s %s %i %i",
         book_title, book_author, book_publisher,
         &book_year, &book_uni);

  return make_book(white_space_deprotect(book_title),
                   white_space_deprotect(book_author),
                   white_space_deprotect(book_publisher),
                   book_year, book_uni);
}


void print_book(book *b, FILE *ofp){
  char buffer[BUFFER_MAX];
  encode_book(b, buffer);
  fprintf(ofp, "%s", buffer);
}

book *read_book(FILE *ifp){
  char buffer[BUFFER_MAX];
  fgets(buffer, BUFFER_MAX, ifp);
  return decode_book(buffer);
}
```

Program 46.4 *Implementationen af biblioteket - book-read-write.c.*

I program 46.5 viser vi hvordan det samlede program, bestående af biblioteket, skriveprogrammet og læseprogrammet, oversættes med gcc.

```
gcc -c book-read-write.c

gcc book-write-prog.c book-read-write.o -o book-write-prog

gcc book-read-prog.c book-read-write.o -o book-read-prog
```

Program 46.5 *Compilering af programmerne.*

Opgave 10.1. *Input og Output af structs*

We will assume that we have a struct that describes data about a person, such like

```
struct person {
  char *name;
  int age;
  char sex;
}
```

where sex is either 'm' or 'f'. In this exercise, you are asked to program a library that can write a number of persons to a file, and restore the data as structs.

Concretely, write two functions

```
print_person(person *p, FILE *ofp)
person *read_person(FILE *ifp)
```

We suggest a line-oriented approach where data about a single person is represented as a single line of text on the file. Special attention should be payed to strings with spaces and newlines.

46.2. Input/Output af structures (2)

Lektion 10 - slide 24

I afsnit 46.1 gennemgik vi alle detaljer om structure input/output via et eksempel. Vi vil nu hæve abstraktionsniveauet en del, og diskutere den generelle problemstilling ved structure IO.

Følgende observationer er relevante:

- Kun felter af taltyper, char, og string håndteres
- Pointers til andre typer end strenge kan ikke udskrives og indlæses
- Evt. nestede structs og arrays kræver specialbehandling
- Vi vælger en løsning med én struct per linie i en tekst fil
  - Newline tegn må derfor ikke indgå i strenge
  - Det er behændigt hvis en streng altid kan læses med scanf og %s

En mere generelt anvendelig løsning ville være ønskeligt

Nogle sprog understøtter en meget simpel, binær *file of records* for dette behov

I C er funktionerne `fread` og `fwrite` nyttige for læsning og skrivning af binære filer

I afsnit 46.3 viser vi et simpelt eksempel på brug af binær input/output med `fread` og `fwrite`.

46.3. Binær input/output med fread og fwrite

Lektion 10 - slide 25

Som allerede omtalt er det forholdsvis let at bruge `fwrite` og `fread` til direkte udskrivning og indlæsning af bitmønstre i C. Vi skal dog være opmærksomme på, at vi kun kan udskrive værdier af de simple aritmetiske typer. Pointere kan under ingen omstændigheder udskrives og genindlæses meningsfyldt. Sidstnævnte betyder, at de fleste strenge ikke kan håndteres med `fwrite` og `fread`, idet strenge jo er pointere til det første tegn, og idet mange strenge er dynamisk allokerede.

I program 46.6 ser vi et program der udskriver en struct bestående af en int, en double og en char. Anden og tredje parameter af `fwrite` er størrelsen af strukturen (i bytes) og antallet af disse.

```
#include <stdio.h>

struct str {
  int f1;
  double f2;
  char f3;
};

int main(void){
  FILE *ofp;
  struct str rec= {5, 13.71, 'c'};
  ofp = fopen("data", "w");

  fwrite(&rec, sizeof(struct str), 1, ofp);
  fclose(ofp);
  return 0;
}
```

Program 46.6 *Skrivning af en struct på en binær fil.*

I program 46.7 ser vi det tilsvarende læseprogram, som anvender `fread`.

```c
#include <stdio.h>

struct str {
  int f1;
  double f2;
  char f3;
};

int main(void){
  FILE *ifp;
  struct str rec;
  ifp = fopen("data", "r");

  fread(&rec, sizeof(struct str), 1, ifp);

  printf("f1=%d, f2= %f, f3 = %c\n", rec.f1, rec.f2, rec.f3);
  fclose(ifp);
  return 0;
}
```

Program 46.7 *Tilsvarende læsning af en struct fra en binær fil.*