38. Datastrukturer

Lektionen, som starter i dette afsnit, fortsætter vores behandling af datatyper og datastrukturer. Vores første møde med datastrukturer var i kapitel 21 hvor vi studerende arrays. Se også afsnit 18.1 hvor vi gav en oversigt over de fleste datatyper i C.

I dette kapitel vil primært se på structures (records) og datastrukturer som er sammenbundet med pointere.

38.1. Datastrukturer (1)

Lektion 9 - slide 2

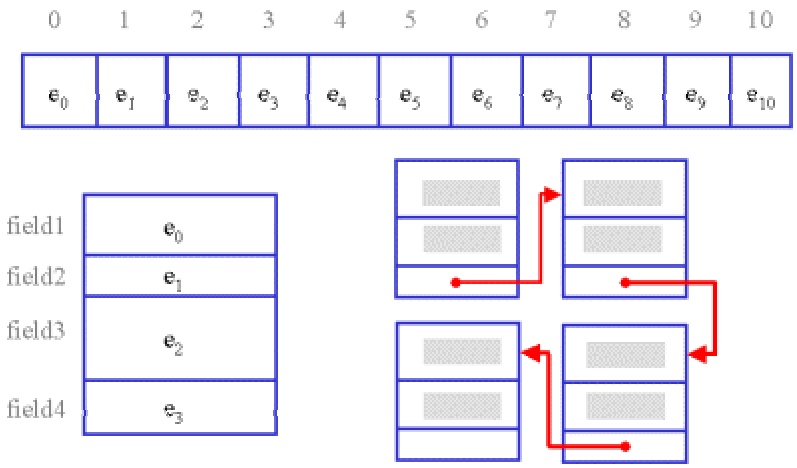
Vi ser først overordnet på datastrukturer.

> Datastrukturer betegner data som er sammensat af forskellige primitive datatyper (skalartyper) ved brug af arrays, records (og andre tilsvarende) samt pointere mellem disse

Sammensatte datastrukturer siges også at *aggregere* et antal data til en helhed, som i mange sammenhænge kan manipuleres samlet. Vi kan skelne mellem nedenstående slags datastrukturering:

- Arrays: Tabeller med elementer af samme type, og tilgang via heltallige indekser
- Records/structures: Tabeller med felter af forskellige typer, og tilgang via feltnavne
- Kædede strukturer: Sammenkædning af records og arrays ved brug af pointere

De tre slags datastrukturer kan illustreres grafisk på følgende måde:



Figur 38.1 *En illustration af arrays, structures og sammenkædede structures.*

39. Records/structures

I dette kapitel foretager vi en grundig gennemgang af records. I C taler vi om structures, eller blot structs, i stedet for records. Recordbetegnelsen anvendes eksempelvis i Pascal. Som altid ser vi på et antal eksempler når vi kommer godt igang med emnet.

39.1. Structures/records (1)

Lektion 9 - slide 5

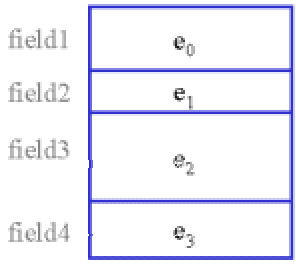
Både arrays og structures kan opfattes som tabeller. Den mest betegnende forskel er dog, at structures tillader 'elementer' af forskellige typer i tabellen. Vi husker fra afsnit 21.1 at alle elementer i et array skal være af samme type.

En record er en tabel med navngivne felter, som kan være af forskellige datatyper

En record gruppérer og sammensætter en mængde variable af forskellige typer i en logisk helhed

I C kaldes records for structures

I figur 39.1 har vi tegnet en tabel, hvor elementerne opfattes som felter. Felter har navne, og ikke indeks numre som vi kender det fra arrays.



Figur 39.1 *En skitse af en structure*

39.2. Structures/records (2)

Lektion 9 - slide 6

Herunder gengiver vi de generelle egenskaber af records.

- Elementerne/felterne i en record kan være af forskellige typer
- Elementerne er individuelt navngivne med *feltnavne*
- Elementerne i en record tilgås med *dot notation*: `record.feltnavn`
- Når en record er skabt og allokeret i lageret kan der ikke tilføjes eller fjernes felter
- Felterne i en record lagres konsekutivt, med afstættelse af passende plads til hvert felt

Der er effektiv tilgang til felterne i en record, men der er typisk meget færre felter i en record end der er elementer i et array

Prisen for 'elementer af forskellige typer' er at vi ikke kan 'beregne os frem til' et felt i tabellen.

39.3. Structures/records (3)

Lektion 9 - slide 7

Lad os et øjeblik antage, at vi ikke har structs i vores programmeringssprog. Ligesom vi gjorde for arrays i afsnit 21.2 vil vi skrive et program, som udstiller genvordighederne ved at undvære structures.

I program 39.1 ser vi tre grupper af logisk sammenhørende variable, som beskriver egenskaber af en far, mor og en baby. Hver person beskrives ved navn, CPR nummer og forældrenes CPR numre. I alt fire egenskaber pr. person.

```
#include <stdio.h>

typedef unsigned long CPR;

int main(void) {

  char *baby_name = "Paul";
  CPR baby_cpr = 2304031577;
  CPR baby_father_cpr = 1605571233;
  CPR baby_mother_cpr = 2412671234;

  char *mother_name = "Anna";
  CPR mother_cpr = 2412671234;
  CPR mother_father_cpr = 1605376789;
  CPR mother_mother_cpr = 1201402468;

  char *father_name = "Frank";
  CPR father_cpr = 1605571233;
  CPR father_father_cpr = 1111331357;
  CPR father_mother_cpr = 1212358642;

  return 0;
}
```

Program 39.1 *Motivation for structures - logisk sammenhørende, men sideordnede variable.*

Vi fornemmer klart, at det vil være hensigtsmæssigt med en mere eksplicit grupperingsmekanisme, som f.eks. sammenknytter variablene `baby_name`, `baby_cpr`, `baby_father_cpr` og `baby_mother_cpt` til en enhed. Structures i C - og records mere generelt - bidrager netop med dette.

I program 39.2 vises et alternativ til program 39.1, hvor vi har introduceret grupperingen af personnavn og de tre relevante CPR numre. Grupperingen hedder blot person. `struct person` skal forstås som en typedefinition.

```
#include <stdio.h>

typedef unsigned long CPR;

struct person {
  char *name;
  CPR cpr;
  CPR father;
  CPR mother;
};

int main(void) {

  struct person
    baby = {"Paul", 2304031577, 1605571233, 2412671234},
    mother = {"Anna", 2412671234, 1605376789, 1201402468},
    father = {"Frank", 1605571233, 1111331357, 1212358642};

  printf("%s's mother and father is %s and %s\n",
         baby.name, mother.name, father.name);


  return 0;
}
```

Program 39.2 *Introduktion af structures - sammenhørende data om en person.*

I `main` i program 39.2 erklærer vi variablene `baby`, `mother` og `father` af typen `struct person`. Vi ser også en direkte record-notation i tuborg klammer, helt analogt til den tilsvarende array-notation introduceret i program 21.3.

Ved indførelse af structures/records får vi en bedre strukturering af data

Alle data om en person kan håndteres som en helhed

39.4. Structures i C (1)

Lektion 9 - slide 8

I dette afsnit viser vi den syntaktiske opbygning, og vi opregner et antal yderligere egenskaber af structures.

```
struct structure-tag{
 type1 field1;
 type2 field2;
 ...
 typen fieldn;
};
```

Syntaks 39.1 *En struct type specifikation*

- Terminologi:
  - Structure tag name: Navnet efter **`struct`**.
  - Member: Variablene, som udgør bestanddelene af strukturen
- Tilgang til et felt via en variabel, som indeholder en **`struct`**
  - **`variabel.felt`**
- Tilgang til felt via en variabel som indeholder en pointer til en **`struct`**
  - **`variabel->felt`**
  - **`(*variabel).felt`**

Det er værd at bemærke tilgangen til et felt via en pointer til en struct. Operatoren `->` befinder sig i den øverste prioritetsgruppe i operatortabellen i tabel 2.3.

Herunder giver vi en oversigt over de lovlige operationer, som kan udføres på structures.

- Lovlige operationer structures:
  - Assignment af structures af samme type: **`struct1 = struct2`**
    - Members assignes parvis til hinanden
  - Uddragning af addressen af en struct med **`&`** operatoren
  - Tilgang til members med dot notation og **`->`** operatoren
- Størrelsen af en struct
  - En struct kan fylde mere en summen af felternes størrelse

Det er bemærkelsesværdigt, at man ikke i C kan sammenligne to structures strukturelt og feltvis, ala strukturel strengsammenligning i figur 28.1.

39.5. Eksempel: Dato structure

Lektion 9 - slide 10

I det første eksempel på en struct viser vi en dato, `date`, som primært er karakteriseret af egenskaberne (felterne) `day`, `month`, og `year`. Sekundært medtager vi også en ugedag, `weekday`. Vi siger undertiden af dette felt er redundant, da den kan afledes entydigt af `day`, `month` og `year`.

```
struct date {
  weekday day_of_week;
  int day;
  int month;
  int year;
};

typedef struct date date;
```

Program 39.3 *En struct for en dato.*

Vi viser `struct date` i en praktisk sammenhæng i program 39.4. Med rødt ser vi en funktion, `date_before`, som afgør om en dato `d1` kommer før en dato `d2`. Bemærk at kroppen af funktionen `date_before` er ét stort logisk udtryk sammensat af en logisk *or* operator, jf. tabel 6.2. Alternativt kunne vi have programmeret kroppen af `date_before` som en udvælgende kontrolstruktur via en if-else kæde. (jf. afsnit 8.5)

```
#include <stdio.h>

enum weekday {sunday, monday, tuesday, wednesday, thursday,
              friday, saturday};
typedef enum weekday weekday;

struct date {
  weekday day_of_week;
  int day;
  int month;
  int year;
};

typedef struct date date;

/* Is date d1 less than date d2 */
int date_before(date d1, date d2){
 return
  (d1.year < d2.year) ||
  (d1.year == d2.year && d1.month < d2.month) ||
  (d1.year == d2.year && d1.month == d2.month && d1.day < d2.day);
}

int main(void) {

  date today = {thursday, 22, 4, 2004};
  date tomorrow = {friday, 23, 4, 2004};

  if (date_before(today, tomorrow))
    printf("OK\n");
  else printf("Problems\n");

  return 0;
}
```

Program 39.4 *Et dato program med en funktion, `date-before`, der vurderer om en dato kommer før en anden dato.*

I `main` funktionen kalder vi `date_before` med to konstruerede datoer. Vi forventer naturligvis at programmet udskriver OK.

39.6. Eksempel: Bog structure

Lektion 9 - slide 11

I program 39.5 ser vi en structure, der beskriver forskellige egenskaber af en bog. Ud over title, forfatter og publiseringsår repræsenterer vi hvorvidt bogen er en universitetslærebog. Vi har også en typedef, som navngiver `struct book` til `book`. Dette er helt analogt til de navngivninger, vi introducerede for enumeration typer i afsnit 19.3.

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

typedef struct book book;
```

Program 39.5 *Et eksempel på en struct for en bog.*

Med blåt i program 39.6 ser vi igen `struct book`. Med brunt under denne viser vi en funktion, `make_book`, som laver og returnerer en bog på baggrund af parametre, der modsvarer de enkelte felter. Funktionen overfører og assigner altså blot informationerne til felterne i `struct book`. Læg mærke til, at den returnerede bog ikke er en pointer. Der er helt reelt tale om returnering af en lokal variabel, som er af typen struct book. Enkelt og lige til.

Med lilla vises en funktion, der via `printf` kan printe en bog på standard output (typisk skærmen). Med sort, og forneden i program 39.6 assignes `b1` og `b2` til resultaterne returneret fra kald af `make_book`. Variablen `b3` assignes til en kopi af `b2`. Bid lige mærke i, at structure assignment involverer en feltvis kopiering af indholdet i en structure.

```
#include <stdio.h>

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

typedef struct book book;

book make_book(char *title, char *author, char *publisher,
               int year, int text_book){
  book result;
  result.title = title; result.author = author;
  result.publisher = publisher; result.publishing_year = year;
  result.university_text_book = text_book;

  return result;
```

```
}

void prnt_book(book b){
  char *yes_or_no;

  yes_or_no = (b.university_text_book ? "yes" : "no");
  printf("Title: %s\n"
         "Author: %s\n"
         "Publisher: %s\n"
         "Year: %4i\n"
         "University text book: %s\n\n",
         b.title, b.author, b.publisher,
         b.publishing_year, yes_or_no);
}

int main(void) {

  book b1, b2, b3;

  b1 = make_book("C by Disssection", "Kelley & Pohl",
                 "Addison Wesley", 2001, 1);
  b2 = make_book("Pelle Erobreren", "Martin Andersen Nexoe",
                 "Gyldendal", 1910, 0);

  b3 = b2;
  b3.title = "Ditte Menneskebarn"; b3.publishing_year = 1917;

  prnt_book(b1);  prnt_book(b2);   prnt_book(b3);

  return 0;
}
```

Program 39.6 *Et program som kan lave og udskrive en bog.*

39.7. Structures ved parameteroverførsel

Lektion 9 - slide 12

Vi har allerede i program 39.6 set hvordan vi helt enkelt kan overføre og tilbageføre structures til/fra funktioner i C. Vi vil her understrege hvordan dette foregår, og vi vil observere hvordan det adskiller sig fra overførsel og tilbageførsel af arrays.

Structures behandles anderledes end arrays ved parameteroverførsel og værdireturnering fra en funktion

- Structures:
  - Kan overføres som en værdiparameter (call by value parameteroverførsel)
  - Kan returneres fra en funktion
  - Det er mere effektivt at overføre og tilbageføre pointere til structures
- Arrays:
  - Overføres som en reference (call by reference parameteroverførsel)
  - Returneres som en reference fra en funktion

C tilbyder udelukkende parameteroverførsel 'by value' (værdiparametre), jf. afsnit 12.7. Når vi overfører et array overfører vi altid en pointer til første element. Når vi overfører en structure kan vi overføre strukturen som helhed. Det indebærer en kopiering felt for felt både i parameteroverførsel, og ved returneringen (som f.eks. foretaget i program 39.6).

Som det illustreres i program 39.7 kan vi også overføre og tilbageføre structures som pointere. I program 39.7 overføres parameteren `b` til `prnt_book` pr. reference (som en pointer). Se afsnit 23.1. Funktionen `make_book` (på det blå sted) forsøger på at tilbageføre den konstruerede bog som en pointer. Versionen i program 39.7 virker dog ikke. Det gør derimod den tilsvarende funktion i program 39.8.

```
/* Incorrect program - see books-ptr.c for a better version */

#include <stdio.h>

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

typedef struct book book;

book *make_book(char *title, char *author, char *publisher,
                int year, int text_book){
  static book result;
  result.title = title; result.author = author;
  result.publisher = publisher; result.publishing_year = year;
  result.university_text_book = text_book;

  return &result;
}

void prnt_book(book *b){
  char *yes_or_no;

  yes_or_no = (b->university_text_book ? "yes" : "no");
  printf("Title: %s\n"
         "Author: %s\n"
         "Publisher: %s\n"
         "Year: %4i\n"
         "University text book: %s\n\n",
         b->title, b->author, b->publisher,
         b->publishing_year, yes_or_no);
}

int main(void) {

  book *b1, *b2;

  b1 = make_book("C by Disssection", "Kelley & Pohl",
                 "Addison Wesley", 2001, 1);
  b2 = make_book("Pelle Eroberen", "Martin Andersen Nexoe",
                 "Gyldendal", 1911, 0);

  prnt_book(b1);
```

```
  prnt_book(b2);

  return 0;
}
```

Program 39.7 *Et eksempel på overførsel og tilbageførsel af bøger per refererence - virker ikke.*

Lad os indse, hvorfor `make_book` (det blå sted) i program 39.7 ikke virker tilfredsstillende. Den lokale `book` variabel er markeret som static (se afsnit 20.3.) Static markeringen bruges for at undgå, at result bogen udraderes når funktionen returnerer. Alle kald af `make_book` deler således `result` variablen, som overlever fra kald til kald. Når `make_book` kaldes gentagne gange vil der blive tilskrevet indhold til denne ene `result` variabel flere gange. Anden gang `make_book` kaldes (med 'Pelle Eroberen') overskrives egenskaberne af den først konstruerede bog ('C by Dissection').

I program 39.8 reparerer vi `make_book` problemet. På det røde sted ser vi forskellen. I stedet for at gøre brug af en lokal variabel allokerer vi dynamisk lager til bogen ved brug af `malloc`, jf. afsnit 26.2. Dermed afsættes lageret på heapen, og ikke på stakken af activation records. Endnu vigtigere i vores sammenhæng, for hvert kald af `make_book` allokeres der plads til bogen. Dette er - i en nødeskal - forskellen mellem denne version af `make_book` og versionen i program 39.7.

```
#include <stdio.h>

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

typedef struct book book;

book *make_book(char *title, char *author, char *publisher,
                int year, int text_book){
  book *result;
  result = (book*)malloc(sizeof(book));
  result->title = title; result->author = author;
  result->publisher = publisher; result->publishing_year = year;
  result->university_text_book = text_book;

  return result;
}

void prnt_book(book *b){
  char *yes_or_no;

  yes_or_no = (b->university_text_book ? "yes" : "no");
  printf("Title: %s\n"
         "Author: %s\n"
         "Publisher: %s\n"
         "Year: %4i\n"
         "University text book: %s\n\n",
         b->title, b->author, b->publisher,
         b->publishing_year, yes_or_no);
}
```

```
int main(void) {

  book *b1, *b2;

  b1 = make_book("C by Disssection", "Kelley & Pohl",
                 "Addison Wesley", 2001, 1);
  b2 = make_book("Pelle Eroberen", "Martin Andersen Nexoe",
                 "Gyldendal", 1911, 0);

  prnt_book(b1);
  prnt_book(b2);

  return 0;
}
```

Program 39.8 *Et eksempel på overførsel og tilbageførsel af bøger per reference - OK.*

39.8. Structures i structures

Lektion 9 - slide 13

Structures kan indlejres i hinanden. Dette betyder at et felt i en structure kan være en anden structure.

I program 39.9 viser vi eksempler på, hvordan structures kan indlejres i hinanden. Med rødt ser vi en point structure, der repræsenterer et punkt i planen, karakteriseret af x og y koordinater.

Med blåt ser vi først en rektangel structure, `struct rect`. Et rektangel repræsenteres i vores udgave ved to modstående hjørnepunkter `p1` og `p2`. Vi viser også en alternativ rektangel structure, `struct another_rect`. I denne rektangel structure har vi indlejret to anonyme punkt structures, i stedet for at gøre brug af `struct point`. På alle måder er struct rect 'smartere' end struct another_rect.

Med sort, forneden i program 39.9 viser vi en `main` funktion, der statisk allokerer to punkter `pt1` og `pt2`, og to rektangler `r` og `ar`. Dernæst viser vi, hvordan egenskaberne af de indlejrede structures kan tilgås via dot notation (se afsnit 39.2). Læg mærke til, at dot operatoren associerer fra venstre mod højre, jf. tabel 2.3. Det betyder at `ar.p1.x` betyder `(ar.p1).x` og ikke `ar.(p1.x)`.

```
#include <stdio.h>

struct point {
  int x;
  int y;
};

struct rect {
  struct point p1;
  struct point p2;
};

struct another_rect {
```

```
  struct {
    int x;
    int y;
  }  p1;
  struct {
    int x;
    int y;
  }  p2;
};

int main(void) {

  struct point pt1, pt2;
  struct rect r;
  struct another_rect ar;

  pt1.x = 5; pt1.y = 6;
  pt2.x = 17; pt2.y = 18;

  r.p1 = pt1; r.p2 = pt2;

  ar.p1.x = 1;   ar.p1.y = 2;
  ar.p2.x = 10;  ar.p2.y = 12;

  ar.p1 = pt1;  /* error: incompatible types */
  ar.p2 = pt2;  /* error: incompatible types */

  return 0;
}
```

Program 39.9 *Eksempler på structures i structures.*

40. Arrays af structures

Vi så i afsnit 39.8 at structures kan indeholde felter som er andre structures. Temaet i dette kapitel er structures som elementer i arrays. Generelt er der fri datastruktur kombinationsmulighed i C.

40.1. Array af bøger

Lektion 9 - slide 15

Array-begrebet blev introduceret i kapitel 21. Et array kan opfattes som en tabel af elementer, hvor alle elementer er af samme type.

Vi vil nu se på situationen, hvor elementerne i et array er stuctures. Vores første eksempel er et array af bøger, som intuitiv svarer til en *bogreol*. Vi erindrer om, at en bog repræsenteres som en structure, jf. vores diskussion i afsnit 39.6.

I program 40.1 ser vi på det røde sted erklæringen af 'bogreolen' kaldet `shelf`. `shelf` er netop et array af `book`, hvor `book` ligesom i program 39.6 er et alias for `struct book`.

På de brune, blå og lilla steder i `main` initialiseres `shelf[1]`, `shelf[2]` og `shelf[3]` på lidt forskellig vis. `shelf[2]` er i udgangspunktet en kopi af `shelf[1]`, som dernæst modificeres fra 'Pelle Erobreren' til 'Ditte Menneskebarn'.

Til sidst i `main` udskrives bøgerne i `shelf` med `prnt_book`, som vi allerede mødte i program 39.6.

```
int main(void) {
  book shelf[MAX_BOOKS];
  int i;

  shelf[0] =
     make_book("C by Disssection", "Kelley & Pohl",
               "Addison Wesley", 2001, 1);

  shelf[1] =
    make_book("Pelle Erobreren", "Martin Andersen Nexoe",
              "Gyldendal", 1910, 0);

  shelf[2] = shelf[1];
  shelf[2].title = "Ditte Menneskebarn";
  shelf[2].publishing_year = 1917;

  for(i = 0; i <=2; i++)
    prnt_book(shelf[i]);

  return 0;
}
```

Program 40.1 *Et array af bøger i funktionen main.*

Via den tilknyttede slide kan man få adgang til et komplet C program, som dækker program 40.1.

40.2. Arrays af datoer: Kalender

Lektion 9 - slide 16

I forlængelse af dato strukturen fra afsnit 39.5 ser vi nu på en tabel af datoer. En sådan tabel kan fortolkes som en form for kalender.

På det røde sted i program 40.2 erklæres `calender` variablen som et array med plads til 1000 datoer. Der er ligesom for `shelf` i program 40.1 tale om *statisk allokering* (jf. afsnit 26.1) hvilket indebærer at lageret til tabellerne allokeres når erklæringen effektueres under programkørslen.

I while-løkken gennemløbes et helt år, med start i `first_date`. Hver dato i året puttes i kalenderen. Bemærk at der er god plads. Fremdriften i kalenderen, som er en lidt tricket detalje, bestyres af funktionen `tomorrow`. Det er en god programmeringsøvelse at lave denne funktion.

```
int main(void) {
  date calendar[1000];

  date first_date = {thursday, 24, 4, 2003},
       last_date  = {thursday, 22, 4, 2004},
       current_date;
  int i = 0, j = 0;

  current_date = first_date;
  while (date_before(current_date, last_date)){
    calendar[i] = current_date;
    current_date = tomorrow(current_date);
    i++;
  }

  for (j = 0; j < i; j++)
    prnt_date(calendar[j]);
}
```

Program 40.2 *Et program der laver en kalender - kun main funktionen.*

Hele programmet (dog uden `tomorrow`) kan tilgås via den tilknyttede slide. Hele programmet inklusive tomorrow er tilgængelig som en løsning på opgave 9.1.

Opgave 9.1. *Funktionen tomorrow*

Implementer funktionen `tomorrow`, som er illustreret på den omkringliggende side.

Funktionen tager en dato som parameter, og den returnerer datoen på den efterfølgende dag.

Hint: Lad i udgangspunktet resultatet være den overførte parameter, og gennemfør (pr. assignment til enkelte felter) de nødvendige ændringer.

41. Sammenkædede datastrukturer

Sammenkædede datastrukturer udgør et fleksibelt alternativ til forskellige statiske (faste) kombinationer af arrays og structures.

41.1. Sammenkædede datastrukturer (1)

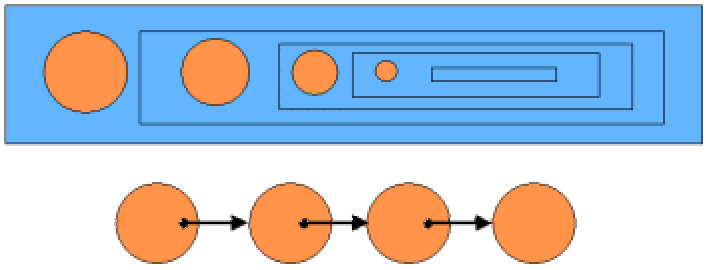
Lektion 9 - slide 18

Sammenkædede datastrukturer opbygges typisk af dynamisk allokerede structs, der sammenbindes af pointere. Pointere var emnet i kapitel 22 og structures har været et af vore emner i denne lektion, startende i afsnit 39.1.

Når én struct via en pointer refererer til en struct af samme type taler vi undertiden om selvrefererende typer, eller på engelsk self-referential structures.

En '*self-referential structure*' er en pointer repræsentation af en rekursiv datatype

Som det fremgår, også af figur 41.1, er selvrefererende strukturer en konkret implementation af rekursive datatyper. Se også afsnit 32.1.



Figur 41.1 *En rekursiv struktur i forhold til en self-referential structure med pointere*

I denne lektion vil vi begrænse os til at studere *kædede lister* (linked lists)

41.2. Sammenkædede datastrukturer (2)

Lektion 9 - slide 19

Vi lader nu op til at studere enkelt-kædede lister. Dette er lineære strukturer, der ofte bruges som et alternativ til arrays. Arrays er mere effektive at slå op i, men langt mindre effektive ved indsættelser og sletninger af elementer. Sidstnævnte illustreres i afsnit 41.6.

Herunder, i program 41.1 ser vi mod blåt den afgørende byggeklods for enkelt-kædede lister. Feltet `next` er en pointer til en `struct`, som svarer til værtstrukturen af `next`. Den røde struct er et forsøg på en rekursiv struktur. Rekursiviteten optræder fordi feltet `next` har samme type som den omkringliggende struct. Vi genfinder altså helheden (strukturen) i detaljen (et felt i strukturen). Rekursive datatyper, ala den røde struktur, er ulovlig i C. Den blå struktur er derimod OK.

```
/* Illegal recursive data structure */
struct first_list {
  int               data;
  struct first_list  next;
};

/* Legal self-referential data structure */
struct second_list {
  int                data;
  struct second_list  *next;
};
```

Program 41.1 *Structures for recursive and self referential structures.*

I de kommende afsnit vil vi udvikle et listebegreb, med udgangspunkt i den blå struktur i program 41.1. Med afsæt i Lisp, som er et LISte Processeringssprog, vil vi kalde denne struktur en *cons celle*.

41.3. Kædede lister ala Lisp og ML

Lektion 9 - slide 20

Som sagt er vil vi nu implementere lister, som de findes i Lisp. Det er for øvrigt det samme listebegreb, som nogle måske kender fra programmeringssproget ML. Både Lisp og ML er funktions-orienterede programmeringssprog.

Den strukturelle byggeklods er cons cellen, som vi etablerer først. Se program 41.2.

```
struct cons_cell {
  void             *data;
  struct cons_cell *next;
};

typedef struct cons_cell cons_cell;
```

Program 41.2 *Typen cons_cell.*

Vi laver nu, i program 41.3, funktionen som hhv. konstruerer og selekterer bestanddele i en cons celle. Funktionen `cons` allokerer med `malloc` plads til en cons celle (i alt to pointere). Funktionen `head` returnerer førstepladsen (en pointer til `data` bestanddelen) af cons cellen. Funktionen `tail` returnerer andenpladsen (en pointer til en næste cons celle). I de fleste Lisp systemer taler vi om `car` and `cdr` i stedet for `head` og `tail`.

```
/* Returns a pointer to a new cons cell,
   which refers data and next via pointers*/
cons_cell *cons(void *data, cons_cell *next){
 cons_cell *result;
 result = malloc(sizeof(cons_cell));
 result->data = data;
 result->next = next;
 return result;
}

/* Return the head reference of the cons cell */
void *head(cons_cell *cell){
  return cell->data;
}

/* Return the tail refererence f the cons cell */
cons_cell *tail(cons_cell *cell){
  return cell->next;
}
```

Program 41.3 *Funktionerne cons, head og tail.*

Givet funktionerne `cons`, `head` og `tail` vil vi nu illustrere hvordan de anvendes til simpel og basal listehåndtering. Dette sker i program 41.4. Vi ser først tre punkter `p1`, `p2` og `p3` af typen `point`. (For god ordens skyld viser vi typen `point` og en `point` print funktion i program 41.5). På det røde sted laver vi en liste af de tre punkter.

Læg mærke til hvordan adresserne på punkterne overføres som førsteparametre til kaldene af `cons`. På de blå steder anvender vi selektor funktionerne `head` og `tail` i et gennemløb af listen af punkter. Læg mærke til at vi har brug for at caste resultatet af `head(points)` inden det kan overføres som en parameter til `prnt_points`. Årsagen er at `head` returnerer en generisk pointer (en pointer til `void`(jf. program 41.3). Compileren har brug for en garanti for, at denne pointer peger på et punkt. Ved brug af cast operationen på det lilla sted udsteder programmøren denne garanti. Det kommer ikke på tale at transformere pointeren på nogen måde.

```
int main(void) {

  cons_cell *points;

  point p1 = {1,2}, p2 = {3,4}, p3 = {5,6};

  points = cons(&p1,
                cons(&p2,
                     cons(&p3,
                          NULL)));

  while (points != NULL) {
    prnt_point(*((point*)(head(points))));
    points = tail(points);
  }

  return 0;
}
```

Program 41.4 *Et eksempel på en liste af punkter håndteret i funktionen main.*

```
struct point {int x; int y;};
typedef struct point point;

void prnt_point(point p){
  printf("Point: %i, %i\n", p.x, p.y);
}
```

Program 41.5 *Typen point og funktionen prnt_point(p).*

I realiteten er en datastruktur, bygget af cons celler, et binært træ

Observationen om det binære træ er enkel. Hver cons celle har to pointere, som faktisk begge kan gå til andre cons celler. Hvis dette er tilfældet kan vi helt naturligt forstå en cons celle struktur som et træ. Bladene i træet vil typisk være pointere til enten `NULL` eller til andre simple værdier, f.eks. tal.

41.4. Mutation af cons celler

Lektion 9 - slide 21

Lad os nu udvide repertoiret af funktioner, som arbejder på cons celler. I afsnit 41.3, nærmere bestemt i program 41.3, så vi hvordan vi lavede funktionerne `cons`, `head` og `tail`. Vi vil nu lave funktioner, der kan ændre head og tail pointerne.

> Foruden funktionerne **`cons`**, **`head`** og **`tail`** laver vi nu også **`set_head`** og **`set_tail`**, som ændrer pladserne i en cons celle

Funktionerne set_head og set_tail er enkle. Vi ser dem i program 41.6.

```
/* Change the data position of cell to new_head */
void set_head(cons_cell *cell, void *new_head){
  cell->data = new_head;
}

/* Change the next position of cell to new_tail */
void set_tail(cons_cell *cell, cons_cell *new_tail){
  cell->next = new_tail;
}
```

Program 41.6 *Funktionerne set_head og set_tail.*

Eneste punkt på dagsordenen i `set_head` og `set_tail` er et assignment af hhv. data- og next pointeren til den pointerværdi, der overføres som anden (og sidste) parameter.

I program 41.7 viser vi et eksempelprogram, som anvender `set_head`. Vi laver først en liste af tre punkter. På det røde sted udskiftes punktet `p1` med `p3`. Ligeledes udskiftes punktet `p2` med `p4`. Når vi udskriver punkterne i den efterfølgende while-løkke får vi altså outputtet `(5,6)`, `(6,7)`, `(5,6)`.

```
int main(void) {

  cons_cell *points;

  point p1 = {1,2}, p2 = {3,4},
        p3 = {5,6}, p4 = {6,7};

  points = cons(&p1,
                cons(&p2,
                     cons(&p3,
                          NULL)));

  set_head(points,&p3);
  set_head(tail(points), &p4);

  while (points != NULL) {
    prnt_point(*((point*)(head(points))));
    points = tail(points);
  }

  return 0;
}
```

Program 41.7 *Et eksempel på mutationer i en liste af punkter.*

41.5. Funktioner på lister

Lektion 9 - slide 22

På basis af listefunktionerne `cons`, `head`, `tail`, `set_head` og `set_tail` fra afsnit 41.3 og afsnit 41.4 vil vi nu programmet udvalgte funktioner på enkeltkædede lister. Både listebegrebet og funktionerne er modelleret efter Lisp.

I program 41.8 starter vi med funktionen `list_length`, der måler længden af en liste. Funktionen tæller altså antallet af elementer i listen.

```
/* Return the number of elements in list */
int list_length(cons_cell *list){
  if (list == NULL)
    return 0;
  else return 1 + list_length(tail(list));
}
```

Program 41.8 *Funktionen list_length.*

Funktionen `list_length` tager som parameter en reference til den første cons celle i listen. Med udgangspunkt i denne gennemløbes listen rekursivt, og cons cellerne tælles. Læg mærke til at funktionen `tail` kaldes for at få fremdrift i rekursionen.

Den næste funktion, i program 41.9 er `append`, som sammensætter to lister `list1` og `list2` til én liste. Funktionen tager de to lister som parametre, i termer af to pointere til cons celler. Der returneres en pointer til den sammensatte listes første cons celle.

```
/* Append list1 and list2. The elements in list2 are shared between
   the appended list and list2 */
cons_cell *append(cons_cell *list1, cons_cell *list2){
  if (list1 == NULL)
    return list2;
  else return cons(head(list1),
                   append(tail(list1),list2));
}
```

Program 41.9 *Funktionen append.*

I `append` funktionen kopieres cons cellerne i `list1`. Den sidste cons celle i den kopierede del refererer blot den første cons celle i `list2`. Læg godt mærke til hvordan rekursion gør det muligt - let og elegant - at gennemføre den beskrevne kopiering og listesammensætning. I program 41.10 programmerer vi en funktion, `member`, der afgør om et element `el` findes i listen `list`. Både `el` og `list` er parametre af `member` funktionen. Sammenligningen af `el` og elementerne i listen fortages som pointersammenligning, ala reference lighed af strenge (se figur 28.1).

```
/* Is el member of list as a data element?
   Comparisons are done by reference equality */
int member(void *el, cons_cell *list){
  if (list == NULL)
    return 0;
  else if (head(list) == el)
    return 1;
  else
    return member(el, tail(list));
}
```

Program 41.10 *Funktionen member.*

I lidt større detalje ser vi at `member` returnerer 0 (false) hvis `list` er `NULL` (den tomme liste). Hvis ikke listen er tom sammenlignes `el` og og `head` af den første cons celle. Hvis disse peger på forskellige objekter sammenlignes `el` rekursivt med halen (`tail`) af `list`.

Den sidste liste funktion, som vi programmerer i dette afsnit, er `reverse`. Se program 41.11. Funktionen `reverse` vender listen om, således at det første element bliver det sidste, og det sidste element bliver det første.

```
/* Returned list in reverse order */
cons_cell *reverse(cons_cell *list){
  if (list == NULL)
    return NULL;
  else
    return append(reverse(tail(list)), cons(head(list), NULL));
}
```

Program 41.11 *Funktionen reverse.*

Funktionen `reverse` virker således: Først, hvis listen er den tomme liste er resultatet også den tomme liste. Hvis listen er ikke tom sammensættes `reverse(tail(list))` med listen, der kun består af `head(list)`. Sammensætningen foretages naturligvis med `append`, som vi programmerede i program 41.9.

Vi viser nu et lidt større program, som bruger alle de funktioner vi har programmeret i program 41.8 - program 41.11. Se program 41.12.

```
int main(void) {

  cons_cell *points, *more_points, *point_list, *pl;

  point p1 = {1,2}, p2 = {3,4}, p3 = {5,6}, p4 = {6,7},
        p5 = {8,9}, p6 = {10,11}, p7 = {12,13}, p8 = {14,15};

  points = cons(&p1,
                cons(&p2,
                     cons(&p3,
                          cons(&p4,NULL))));

  more_points =
           cons(&p5,
                cons(&p6,
                     cons(&p7,
                          cons(&p8,NULL))));

  printf("Number of points in the list points: %i\n",
         list_length(points));

  point_list = append(points, more_points);
  pl = point_list;

  while (pl != NULL) {
    prnt_point(*((point*)(head(pl))));
    pl = tail(pl);
  }

  if (member(&p6, points))
    printf("p6 is member of points\n");
  else  printf("p6 is NOT member of points\n");

  if (member(&p6, more_points))
    printf("p6 is member of more_points\n");
  else  printf("p6 is NOT member of more_points\n");

  if (member(&p6, point_list))
    printf("p6 is member of point_list\n");
  else  printf("p6 is NOT member of point_list\n");


  pl = reverse(points);

  while (pl != NULL) {
    prnt_point(*((point*)(head(pl))));
    pl = tail(pl);
  }
```

```
  return 0;
}
```

Program 41.12 *Eksempler på anvendelse af ovenstående funktioner i en main funktion.*

Vi forklarer kort programmet. Vi starter med at lave to lister af hhv. `p1`, `p2`, `p3` og `p4` samt af `p5`, `p6`, `p7` og `p8`. På det røde sted kalder vi `list_length` på den første liste. Vi forventer naturligvis resultatet 4. På det blå sted sammensætter vi de to lister af punkter. Det forventes at give en liste af de otte punkter `p1` - `p8`. På de tre brune steder tester vi om `p6` tilhører den første punktliste, den anden punktliste og den sammesatte punktliste. Endelig, på det lilla sted, vender vi den sammensatte liste om ved brug af `reverse`.

41.6. Indsættelse og sletning af elementer i en liste

Lektion 9 - slide 23

På dette sted har vi etableret lister (se afsnit 41.5) på grundlag af cons celler (se afsnit 41.3).

Vi vil nu illustrere hvordan man kan indsætte og slette elementer i kædede lister. Det er meget relevant at sammenligne dette med indsættelse og sletning af elementer fra arrays.

Indsættelse og sletning af elementer i et array involverer flytning af mange elementer, og er derfor kostbar

Indsættelse og sletning af elementer i en kædet liste er meget mere effektiv

På den tilhørende slide side kan du konsultere en billedserie, der viser hvilke skridt der er involveret i at indsætte et element i en kædet liste. Herunder, i program 41.13 , viser et program, som implementerer indsættelsen af et element i en kædet liste.

```
/* Insert new_element after the cons-cell ptr_list */
void insert_after(void *new_element, cons_cell *ptr_list){
  cons_cell *new_cons_cell, *ptr_after;

  /* Make a new cons-cell around new_element */
  new_cons_cell= cons(new_element,NULL);

  /* Remember pointer to cons-cell after ptr_list */
  ptr_after = tail(ptr_list);

  /* Chain in the new_cons_cell */
  set_tail(ptr_list, new_cons_cell);

  /* ... and connect to rest of list */
  set_tail(new_cons_cell, ptr_after);
}
```

Program 41.13 *Funktionen insert_after.*

Der er fire trin involveret, som er omhyggelig kommenteret i program 41.13. Først laves en ny cons celle, hvorfra parameteren `new_element` kan refereres. Så fastholdes pointeren til det element der kommer efter det nye element. I de to sidste trin indkædes den nye cons celle i listen. Hvis du ikke allerede har gjort det, så følg billedserien på den tilhørende slide og forstå program 41.13 ud fra denne.

Bemærk at `insert_after` ikke kan bruges til at indsætte et nyt første element i en liste. Dette tilfælde er både specielt og problematisk. 'Specielt' fordi det kræver sin egen indsættelsesteknik. Og 'problematisk' fordi der generelt kan være mere end én reference til det første element. Dette vender vi kort tilbage til i slutbemærkningerne af afsnit 42.4.

Helt symmetrisk til indsættelse af elementer kan vi også programmere sletning af elementer. Konsulter dog først billedserien på den tilhørende slide side Det tilsvarende program er vist i program 41.14.

```
/* Delete the element after ptr_list. Assume as a
   precondition that there exists an element after ptr_list */
void delete_after(cons_cell *ptr_list){
  cons_cell *ptr_after, *ptr_dispose;

  /* cons-cell to delete later */
  ptr_dispose = tail(ptr_list);

  /* The element to follow ptr_list */
  ptr_after = tail(ptr_dispose);

  /* Mutate the tail of ptr_list */
  set_tail(ptr_list, ptr_after);

  /* Free storage - only one cons-cell */
  free(ptr_dispose);
}
```

Program 41.14 *Funktionen delete_after.*

Vores udgangspunkt er en pointer til den cons celle, som er lige før cons cellen for det element, der skal slettes. I programmet laver vi først en reference `ptr_dispose` til den cons celle, der skal slettes. Dernæst etablerer vi `ptr_after` til at pege på cons cellen lige efter elementet, der skal slettes. Nu kan vi i det tredje trin ændre på next pointeren af `ptr_list` ved hjælp af set_tail. Endelig frigør vi lageret af cons cellen, der refereres af `ptr_dispose`. Dette gøres på det blå sted med `free` funktionen, jf. afsnit 26.3.

Sletningen af et element med `delete_after` virker ikke på det første element. Observationen svarer helt til det tilsvarende forhold for `insert_after`.

Vi viser i program 41.15 hvordan `insert_after` og `delete_after` kan bruges i en praktisk sammenhæng.

```
int main(void) {

  cons_cell *points, *pl;

  point p1 = {1,2}, p2 = {3,4}, p3 = {5,6}, p_new = {11,12};

  points = cons(&p1,
                cons(&p2,
                     cons(&p3,
                          NULL)));


  insert_after(&p_new, tail(points));

  pl = points;
  while (pl != NULL) {
    prnt_point(*((point*)(head(pl))));
    pl = tail(pl);
  }

  printf("\n\n");

  delete_after(points);

  pl = points;
  while (pl != NULL) {
    prnt_point(*((point*)(head(pl))));
    pl = tail(pl);
  }

  return 0;
}
```

Program 41.15 *Eksempler på anvendelse af insert_after og delete_after.*

> Tilgang til elementerne i et array er meget effektiv
>
> Tilgang til elementerne i en kædet liste kræver gennemløb af kæden, og er derfor kostbar

42. Abstrakte datatyper

Abstrakte datatyper er helt naturligt en kontrast til det vi kunne kalde "konkrete eller simple datatyper". Som diskuteret i afsnit 17.1 er en type en mængde af værdier med fælles egenskaber. Eksempelvis er heltal, `int` i C, en type. Værdierne er 0, 1, -1, 2, -2, etc. De fælles egenskaber er først og fremmest alle tallenes velkendte aritmetiske egenskaber.

Vi vil nu set på et udvidet typebegreb, hvor de operationelle egenskaber - funktionerne som virker på værdierne - er i fokus.

42.1. Abstrakte datatyper

Lektion 9 - slide 25

Ud over at være en mængde af værdier er en abstrakt datatype også karakteriseret af af de operationer, som kan anvendes på værdierne.

> En *abstrakt datatype* er en mængde af værdier og en tilhørende mængde af operationer på disse værdier

Værdierne i en abstrakt datatype kaldes ofte objekter. Dette er specielt tilfældet i det objekt-orienterede programmeringsparadigme, hvor ideen om abstrakte datatyper er helt central.

Abstrakte datatyper er centrale og velunderstøttede i objekt-orienterede programmeringssprog. I mere enkle sprog, herunder C, håndterer vi udelukkende abstrakte datatype med brug af *programmeringsdisciplin*. Med dette mener vi, at vi skal programmere på en bestemt måde, med brug af bestemte mønstre og bestemte fremgangsmåder. Ofte er structures et centralt element i en sådan disciplin. Vi opsummerer her i punktform programmering med abstrakte datatyper i C.

- Sammenhørende data indkapsles i en `struct`
- Der defineres en samling af funktioner der 'arbejder på' strukturen
  - Disse funktioner udgør grænsefladen
- Funktioner uden for den abstrakte datatype må ikke have kendskab til detaljer om data i strukturen
  - Dette kendskab må kun udnyttes internt i funktionerne, som udgør typens grænseflade
- Der laves en funktion der skaber og initialiserer et objekt af typen
  - Dette er en konstruktør

Vi vil nu se nærmere på abstrakte datatyper i C. Specielt vil vi se hvorledes nogle af liste eksemplerne fra kapitel 41 kan fortolkes som abstrakte datatyper.

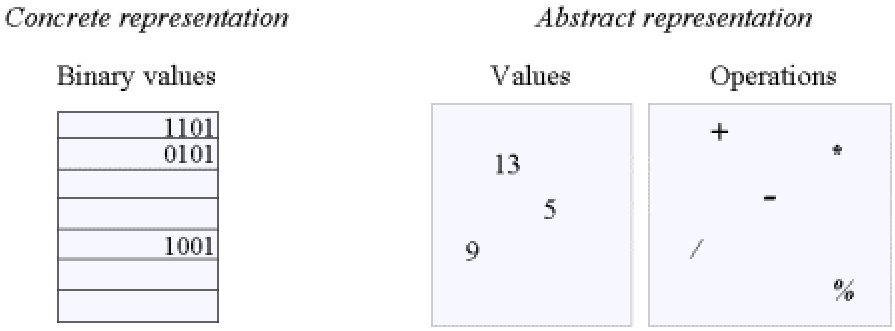
42.2. Tal som en abstrakt datatype

Lektion 9 - slide 26

Vi vil i dette afsnit argumentere for heltal, som vi omgås disse værdier i programmeringssammenhæng, kan opfattes som en abstrakt datatype. I introduktionen til kapitel 42 argumenterede for at heltal er en type. Vores centrale observation i dette afsnit er, at vi næsten udelukkende arbejder med heltal gennem en (abstrakt) notation og en mængde af aritmetiske funktioner. Vi er kun sjældent bevidste om den konkrete binære repræsentation af heltal, som diskuteret i afsnit 16.5.

Vores normale omgang med tal afspejler en abstrakt forståelse

I illustrationen herunder viser vi til venstre den konkrete binære repræsentation af heltal. Til højre skitserer vi heltal i decimal notation og mængden af operationer, noteret som f.eks. +, - og *. Bemærk at den decimale notation i sig selv er forholdsvis abstrakt i forhold til den binære notation.



Figur 42.1 *En konkret og en abstrakt forståelse af heltal*

Når vi programmerer med abstrakte datatyper vil vi tilstræbe en tilsvarende forståelse for alle andre datatyper

Pointen er således at vi ofte ønsker at eftergøre observationerne om heltal i vore egne datatyper. I en nødeskal ønsker vi altså at introducere værdier, som bearbejdes af et antal funktioner, som vi programmerer, og som vi anvender når en type skal nyttiggøres. Kendskabet til detaljerne i værdierne skal dog begrænses så meget som muligt.

I de næste afsnit vil vi give eksempler på den tankegang, som blev udtrykt for heltal i dette afsnit.

42.3. En cons celle som en ADT

Lektion 9 - slide 27

Vi introducerede cons celler som byggeklodser for kædede lister i afsnit 41.3.

En cons celle kan opfattes som en abstrakt datatype

En cons celle er en relativ simpel struktur, der er repræsenteret som en struct i C. En cons celle er repræsenteret som sammensætningen (aggregeringen) af to pointere, *data pointeren* og *next pointeren*. En værdi i typen er altså en sådan to-komponent struktur. Vi ønsker nu at indkapsle denne viden i en abstrakt datatype. Vi konstruerer en cons celle med funktionen `cons`. Vi kan tænke på denne funktion som en *konstruktør*. Vi kan aflæse hver af to to pointere med en funktion. Her kalder vi disse funktioner for hhv. `head` og `tail`. Vi kan endvidere ændre data og next værdierne af en cons cellen med `set_head` og `set_tail`.

Vi opsummerer her operationerne på cons celler.

- `cons`, der spiller rollen som konstruktør
- `head` og `tail` som tilgår bestanddelene
- `set_head` og `set_tail` som muterer bestanddelene

Objekter af typen `cons_cell` benyttes som 'byggeklodser' i nye typer på et højere abstraktionsniveau

I næste afsnit tager vi et trin opad abstraktionsstigen. I stedet for at se på cons celler ser vi nu på typen bestående af listeværdier og de naturlige operationer på lister.

42.4. En liste som en ADT

Lektion 9 - slide 28

Vi opfatter nu hele lister som værdier i en abstrakt datatype. Operationerne på lister kan f.eks. være følgende:

- En funktion/konstruktør der laver en tom liste
- Funktioner der indsætter og sletter et element
  - Udvidede versioner af `insert_after` og `delete_after`, som også håndterer tomme lister fornuftigt
- Funktioner ala `list_length`, `append`, `member` og `reverse`
- *Mange andre...*

En liste er mere end samlingen af elementerne

Listen som så bør repræsenteres af en structure

Det kan måske være vanskeligt helt at forstå essensen af ovenstående observation. Som det fremgår af afsnit 41.6 er det forbundet med specielle udfordringer at indsætte og slette elementer i begyndelsen (forenden) af en liste. Dette skyldes i bund og grund at der kan være flere referencer til det første element i en liste. Hvis vi derimod har sikkerhed for, at der kun er én reference til det første element i en liste, nemlig fra et særligt listeobjekt (som repræsenterer listen som helhed) er problemet løst.

42.5. Fra structures til classes

Lektion 9 - slide 29

Diskussionen om abstrakte datatyper, og eksemplerne vi har studeret i de forrige afsnit, leder frem mod de programmeringssproglige grundbegreber i objekt-orienteret programmering. Mest håndfast glider structure begrebet (C structs) over i klassebegrebet. Vi vil ganske kort gøre nogle få observationer om overgangen fra structures til klasser.

> En klasse i et objekt-orienteret programmeringssprog er en structure ala C, hvor grænsefladens operationer er flyttet ind i strukturen

Følgende punkter karakteriser dataabstraktion på en lidt mere fyldig måde end i afsnit 42.1.

- Logisk sammenhørende værdier ***grupperes*** og ***indkapsles*** på passende vis
  - Sådanne værdier kaldes ofte for ***objekter***
- Operationer på objekterne knyttes tæt til disse
  - Om muligt flytter operationerne ind i datatypen, hvilket bringer os fra records til ***klasser***
- ***Synlighed*** af objektets egenskaber (data og operationer) afklares
  - Det er attraktivt at flest mulige egenskaber holdes ***private***
- Objektets ***grænseflade*** til andre objekter udarbejdes omhyggeligt
  - Grænsefladen udgøres af en udvalgt mængde af operationer

Der er mange gode måder er lære om objekt-orienteret programmering. Lad mig nævne at jeg har skrevet et undervisningsmateriale om objekt-orienteret programmering i Java - i samme stil som indeværende materiale.