Programmering i C

Kurt Nørmark © 2005
Institut for Datalogi, Aalborg Universitet

Sammendrag

Dette er et undervisningsmateriale om introducerende programmering i et imperativt sprog. Mere konkret er det et undervisningsmateriale om programmering i C.

Materialet er udviklet i 2003 og 2004 i forbindelse med undervisningen på den teknisk naturvidenskabelige basisuddannelse ved Aalborg Universitet. Materialet er struktureret som et sæt af slides, der danner basis for 10 forelæsninger i 'Programmering i C'. I 2004 er materialet endvidere sammenskrevet til et mere traditionelt tekstbogsformat.

Forud for 2005 semestret har materialet gennemgået en lettere redigering. Vi har fastholdt de 10 lektioner i materialet på trods af at kurset er reduceret til 8 egentlige lektioner, plus 2 yderligere lektioner, der afsættes til eksamensopgaven. I forbindelse med redigeringen i 2005 er der genereret en PDF udgave af det samlede materiale.

Kolofon: Dette materiale er lavet med LENO systemet. LENO er et XML sprog som er defineret ved brug af en XML DTD. Vi bruger LENO sammen med LAML, som tillader os at skrive LENO materiale i programmeringssproget Scheme. Det betyder at alle kildefilder i materialet er skrevet i Scheme, med brug af spejlet af LENO XML sproget, og spejlet af HTML. Det primære LENO sprog anvendes til at producere forskellige syn på slide sider. Et sekundært LENO sprog anvendes til at skrive det de tematiske noter (den sammenskrevne version). PDF udgaven er lavet med PDF Creator efter en let editering (sidenummerering og sideombrydning) i Microsoft Word.

Indhold

1. Variable og assignment

Dette er starten af det faglige indhold i første lektion af 'Programmering i C'. Før dette følger et antal mere praktiske slides, som vi ikke har medtaget i denne 'tematiske udgave' af materialet.

Vi starter med at se på variable og assignments. Dette er meget centrale elementer i enhver form for imperativ programmering. Vi slutter dette afsnit med et første kig på datatyper.

1.1. Assignment

Lektion 1 - slide 14

Tænk på variable som små kasser, hvori computeren kan placere forskellige værdier når programmet kører. Ændringer af kassernes værdier foregår gennem de såkaldte assignment kommandoer. På dansk vil vi ofte bruge betegnelsen *tildelinger* for assignments.

> Variable er pladser i lageret, hvis værdier kan ændres ved udførelse af assignment kommandoer.
>
> Alle variable skal erklæres før brug.

Her og i det følgende vil en mørkeblå boks med hvid skrift være præcise definitioner af faglige termer. Her definerer vi 'variabel' og 'assignment'.

> En *variabel* er en navngiven plads i computerens arbejdslager, som kan indeholde en værdi af en bestemt type.
>
> Et *assignment* er en kommando, som ændrer værdien af en variabel.

Tilsvarende vil gule bokse med sort skrift være syntaktiske definitioner. Syntaksen af et programmeringssprog beskriver lovlige strukturer af bestemte udtryksformer. Herunder ser vi at en type (et typenavn) kan efterfølges af et antal (her n) variable. Dette introducerer variablene, hvilket er nødvendigt inden disse kan anvendes til noget som helst. Det angiver også, hvilken slags (hvilken type) af værdier variablene kan antage.

```
type variable₁, variabel₂..., variableₙ;
variableᵢ = expression;
```

Syntaks 1.1

I sidste linie af syntaks boksen ser vi strukturen af et assignment: `var = udtryk`. Vi taler om en variabel på venstre side af = og et udtryk på højresiden. Udtrykket beregnes, og udtrykkets værdi placeres i variablen på venstresiden. C er speciel derved, at `var = udtryk` faktisk er et udtryk, som

har en værdi. I forhold til denne tankegang er = en operator, ligesom + og - . Mere om dette i afsnit 3.1.

I program 1.1 ser vi på et eksempel med variablene `course`, `classes`, `students` og `average_pr_class` alle af typen `int`. Dette er den røde del. Typen `int` betegner integers, altså heltal (negative såvel som positive). I den blå del ser vi et antal assignments til disse variable. De tre første er udprægede initialiseringer (se afsnit 1.2). I det sidste assignment tilskriver vi variablen `average_pr_class` kvotienten mellem `students` og `classes`. Bemærk her at I C er `students / classes` heltalsdivision: Indholdet af `students` (et heltal) heltalsdivideres med indholdet af `classes` (et andet tal). Som et eksempel på heltalsdivision har vi at 7 / 3 er 2. Mere om dette i kapitel 2. Tallet som er gemt i variablen `average_pr_class` udskrives i den grå del. `printf` kommandoer diskuteres i kapitel 4.

```
#include <stdio.h>

int main(void) {

  int course, classes, students, average_pr_class;

  course = 1;
  classes = 3;
  students = 301;

  average_pr_class = students / classes;

  printf("There are %d students pr. class\n", average_pr_class);

  return 0;
}
```

Program 1.1 *Et simplet program med variable og assignments.*

I program 1.2 viser vi en variant af programmet, hvor initialiseringerne af variablene er foretaget sammen med variabelerklæringerne. Sådanne erklæringer og initialiseringer er ofte bekvemme. De to programmer er iøvrigt ækvivalente.

```
#include <stdio.h>

int main(void) {

  int course = 1, classes = 3, students = 301, average_pr_class;

  average_pr_class = students / classes;

  printf("There are %d students pr. class\n", average_pr_class);

  return 0;
}
```

Program 1.2 *En variation af programmet.*

1.2. Initialisering

Lektion 1 - slide 15

Vi omtalte allerede initialiseringer i afsnit 1.1 Her vil vi gøre lidt mere ud af begrebet, og vi vil 'moralisere' over vigtigheden af at initialisere alle variable.

Enhver variabel bør eksplicit tildeles en startværdi i programmet

Ved *initialisering* forstås tilskrivning af en startværdi til en variabel

Initialisering er altså tilskrivning af initielle (start) værdier til de variable, som vi introducerer (erklærer). En god måde at anskueliggøre vigtigheden af initialiseringer kan ses gennem et eksempel, hvor vi undlader at gøre det. I program 1.3 ser vi et sådant eksempel, hvor variablen `classes` er uinitialiseret. Bortset fra den uinitialiserede variabel er programmet næsten identisk med program 1.1. I program 1.4 viser vi programmets output. Værdien af `classes` er et meget stort tal, som vi ikke kan genkende. Forklaringen er, at indholdet er `classes` er det tilfældige lagerindholde, som befinder sig det sted i RAM lageret, hvor variablen `classes` er blevet placeret.

I stedet for at få adgang til tilfældig data i uinitialiserede variable ville det have været bedre at få en særlig udefineret værdi tilbage, en fejlværdi, eller slet og ret at stoppe programmet som følge af at der tilgås en uinitialiseret variabel. Det sker ikke i C, men andre programmeringssprog er mere hjælpsomme i forhold til programmøren, som naturligvis ønsker at opdage den slags fejl i programmet.

```
#include <stdio.h>

int main(void) {

  int course, classes, students, average_pr_class;

  course = 1;
  /* classes uninitialized */
  students = 301;


  average_pr_class = students / classes;

  printf("There are %d students pr. class\n", average_pr_class);
  printf("course: %d, classes: %d, students: %d\n",
         course, classes, students);

  return 0;
}
```

Program 1.3 *Et program med en uinitialiseret variabel.*

```
There are 0 students pr. class
course: 1, classes: 1107341000, students: 301
```

Program 1.4 *Program output.*

Hvis man undlader at initialisere variable risikerer man at der opstår tilfældige fejl, når man kører programmet

1.3. Assignments af formen i = i + 1

Lektion 1 - slide 16

Vi vil nu se på assignments, hvor variablen på venstresiden af assignment symbolet = også optræder i udtrykket på højre side. Mere specifikt forekommer dette i `i = i + 1`.

Der er ofte behov for at tælle en variabel op eller ned.

Dermed optræder variablen både på venstre og højre side af assignmentsymbolet

Herunder forklarer vi trinvis hvad der sker når assignmentet `i = i + 1` udføres i en situation, hvor `i` indledningsvis er 7.

- Antag at **`i`** har værdien 7
- Beregning af **`i = i + 1`**
  - Udtrykket **`i + 1`** på venstre side af assignmentsymbolet beregnes til 8
  - Værdien af variablen **`i`** ændres fra 7 til 8

I program 1.5 har vi et lille program med variablene `i`, `j` og `k` . I de tre farvede assignments tælles disse op og ned. Først tælles `i` op fra 3 til 4 Dernæst tælles `j` ned fra 7 til 3 (idet `i` jo er 4). Endelig gøres `k` 3 gange større (idet `j` jo netop har fået værdien 3). Dermed ender k med at være 27. Når man kører programmet får man output som kan ses i program 1.6.

```c
#include <stdio.h>

int main(void) {

  int i = 3, j = 7, k = 9;

  i = i + 1;
  j = j - i;
  k = k * j;

  printf("i: %d, j: %d, k: %d\n", i, j, k);

  return 0;
}
```

Program 1.5 *Et program med forskellige optællinger og nedtællinger. annotation*

```
i: 4, j: 3, k: 27
```

Program 1.6 *Output fra ovenstående program.*

1.4. Et første blik på datatyper i C

Lektion 1 - slide 17

En datatype er en mængde af værdier. Dette vil vi se meget mere til i en senere del af materialet. I tabel 1.1 ser vi en oversigt over forskellige typer af heltal, reelle tal, tegn og tekststrenge. Vi har allerede mødt typen int tidligere. Disse typer ses i spil i program 1.7 Læg lige mærke til endelserne (suffixes) ala L i 111L. Dette betyder 111 opfattet som en **L**ong.

C understøtter forskellige fundamentale datatyper

	Konkrete typer	**Eksempler på værdier**
Heltal	int short long	10 5 111L
Reelle tal	double float long double	10.5 5.5F 111.75L
Tegn	char	'A' 'a'
Tekststrenge	char *	"Computer" "C"

Tabel 1.1

```
#include <stdio.h>

int main(void) {

  int i = 10;
  short j = 5;
  long k = 111L;

  double x = 10.5;
  float y = 5.5F;
  long double z = 111.75L;

  char c = 'A';
  char d = 'a';

  char *s = "Computer", *t = "C";

  return 0;
}
```

Program 1.7 *Illustration af konkrete typer i et C program.*

Hvis man indleder et tal med et 0 har man bedt om oktal notation. Tilsvarende, hvis man indleder et tal med 0x har man bedt om hexadecimal notation. Vi kommer stærkt tilbage til hvad dette betyder senere i materialet (se afsnit 16.5).

> Foruden de forskellige typer af tal understøtter C også heltal i oktal og hexadecimal notation (eksempelvis `0123` og `0x1bc`).

2. Udtryk og operatorer

Et imperativt program er bygget op af erklæringer, kommandoer og udtryk. Et funktionsorienteret program (ala et ML program) er har ingen kommandoer. I dette afsnit ser vi på udtryk i C.

2.1. Udtryk

Lektion 1 - slide 19

Et udtryk er et stykke program, som vi kan beregne med henblik på at producere en værdi. Denne værdi kan vi 'se på', gemme i en variabel, eller lade indgå i beregning af andre udtryk.

Udtryk i C er dog lidt mere komplicerede end ovenstående udlægning lader ane. C udtryk og C kommandoer lapper ganske kraftigt ind over hinanden. Denne problematik vil blive diskuteret i afsnit 3.1.

> Udtryk er sammen med kommandoer de vigtigste 'byggesten' i ethvert program

Et *udtryk* er et programfragment der beregnes med henblik på returnering af en værdi, som tilhører en bestemt type.

Et udtryk opbygges typisk af operatorer og operander.

Med efterfølgende antagelser vil vi i tabel 2.1 beregne en række udtryk (i venstre søjle). Udtrykkenes værdier vises i højre søjle. Typerne af udtrykkene vises i midten.

Antag at værdien af **`x`** er **`14`** og **`y`** er **`3`**:

Udtryk	Type	Værdi
`7`	Heltal	7
`x`	Heltal	14
`x + 6`	Heltal	20
`x / 3`	Heltal	4
`x % 3`	Heltal	2
`x <= y`	Boolean	*false*
`x + y * 3 >= y % x - 5`	Boolean	*true*

Tabel 2.1 *Forskellige udtryk med udtrykkenes typer og værdier.*

Typerne af et udtryk, som det fremgår af tabellen, udledes fra de erklærede typer af indgående variable, fra typerne af de indgående konstanter, og ud fra de erklærede returtyper af operatorer og funktioner.

På de følgende sider vil vi studere hvordan vi beregner udtryk, som er sammensat af flere deludtryk

2.2. Beregning af sammensatte udtryk

Lektion 1 - slide 20

Sammensatte udtryk er opbygget med brug af to eller flere operatorer. Det betyder, at det sammensatte udtryk kan beregnes på flere forskellige måder.

Problemstilling: Hvordan fortolkes udtryk som er sammensat af adskillige operatorer og operander?

Herunder giver vi to forskellige måder at håndtere beregningsrækkefølgen af sammensatte, og ofte komplicerede udtryk.

- Indsæt parenteser som bestemmer beregningsrækkefølgen
  - `x + y * 3 >= y % x - 5`
  - `(x + (y * 3)) >= ((y % x) - 5)`
- Benyt regler som definerer hvilke deludtryk (operander) der beregnes før andre
  - Multiplikation, division og modulus beregnes før addition og subtraktion
  - Addition og subtraktion beregnes før sammenligning

C prioriterer operatorer i 15 forskellige niveauer. Men det er også muligt - og ofte nødvendigt - at sætte parenteser omkring deludtryk

Det er altid muligt at sætte så mange parenteser, at beregningsrækkefølgen fastlægges. Som bekendt for de fleste beregnes udtryk med parenteser 'indefra og ud'. Sideordnede udtryk beregnes som regel fra venstre mod højre. Der findes programmeringssprog som beror på konsekvent,eksplicit parentessætning, nemlig sprog i Lisp familien. Det er dog en almindelig observation, at mange programmører ikke ønsker at sætte flere parenteser i udtryk end strengt nødvendigt. Dermed er scenen sat for prioritering af operatorer og associeringsregler. Dette ser vi på i det næste afsnit.

2.3. Prioritering af operatorer

Lektion 1 - slide 21

Udtryk som involverer operatorer med høj prioritet beregnes før udtryk med operatorer, der har lav prioritet. I tabel 2.2 viser vi en simplificeret tabel med nogle af de mest almindelige operatorer i C. Lidt senere, i tabel 2.3 gengiver vi den fulde tabel.

Niveau	Operatorer
14	+ unary - unary
13	* / %
12	+ -
10	< <= > >=
9	== !=
5	&&
4	||

Tabel 2.2 *En operatorprioriteringstabel for udvalgte operatorer.*

Af tabellen fremgår for eksempel at udtryk med multiplikative operatorer (altså *, /, og %) beregnes før udtryk med additive operatorer (+ og -).

2.4. Associering af operatorer

Lektion 1 - slide 22

Problemstilling: Prioriteringstabellen fortæller ikke hvordan vi beregner udtryk med operatorer fra samme niveau

Vi ser nu på udtryk som anvender to eller flere operatorer med samme prioritet. Et simpelt eksempel på et sådant udtryk er 1-2-3. Der er to mulige fortolkninger: (1-2)-3 som giver -4 og 1-(2-3) som giver 2. Den første svarer til venstre associativitet, den sidste til højre associativitet. I C og de fleste andre programmeringssprog er - venstre associativ. Det betyder at 1-2-3 er lig med (1-2)-3 som igen er lig med -4.

Lad os nu se på et andet tilsvarende eksempel:

- `10 - 1 - 9 / 3 / 2`
  - Operator prioriteringen fortæller at divisionerne foretages før subtraktionerne
  - Associeringsreglerne fortæller at der sættes parenteser fra venstre mod højre
  - `((10 - 1) - ((9 / 3) / 2))`
  - Resultatet er 8

Vi skriver et simpelt C program, program 2.1, som indeholder ovenstående udtryk. Vi viser også programmets output i program 2.2. Som forventet er `res1` og `res2` ens. `res3` er resultatet af den højre associative fortolkning, og er som sådan forskellige fra `res1` og `res2`.

```c
#include <stdio.h>

int main(void) {

  int res1, res2, res3;

  res1 =  10 - 1 - 9 / 3 / 2;

  res2 =  ((10 - 1) - ((9 / 3) / 2));
  /* left to right associativity */

  res3 =  (10 - (1 - (9 / (3 / 2))));
  /* right to left associativity */

  printf("res1 = %d, res2 = %d, res3 = %d\n", res1, res2, res3);


  return 0;
}
```

Program 2.1 *Et C program med udtrykket* ***`10 - 1 - 9 / 3 / 2`***.

```
res1 = 8, res2 = 8, res3 = 18
```

Program 2.2 *Output fra ovenstående program.*

De fleste operatorer i C associerer fra venstre mod højre

2.5. Operator tabel for C

Lektion 1 - slide 23

Niveau	Operatorer	Associativitet
16	`()` `[]` `->` `.` `++`*postfix* `--`*postfix*	*left to right*
15	`++`*prefix* `--`*prefix* `!` `~` `sizeof(type)` `+`*unary* `-`*unary* `&`*unary,prefix* `*`*unary,prefix*	*right to left*
14	`(type name)` *Cast*	*right to left*
13	`*` `/` `%`	*left to right*
12	`+` `-`	*left to right*
11	`<<` `>>`	*left to right*
10	`<` `<=` `>` `>=`	*left to right*
9	`==` `!=`	*left to right*
8	`&`	*left to right*
7	`^`	*left to right*
6	`|`	*left to right*
5	`&&`	*left to right*
4	`||`	*left to right*
3	`?:`	*right to left*
2	`=` `+=` `-=` `*=` `/=` `>>=` `<<=` `&=` `^=` `|=`	*right to left*
1	`,`	*left to right*

Tabel 2.3 *Den fulde operator prioriteringstabel for C.*

C by Dissection side 611

3. Assignment operatorer i C

Ifølge vore forklaringer i kapitel 1 dækker udtryk og kommandoer to forskellige begreber. Kommandoer (ala assignments) udføres for at ændre programmets tilstand (variablenes værdier). Udtryk beregnes for at frembringe en værdi.

I C er assignments operatorer, og dermed kan assignments indgå i udtryk.

3.1. Blandede udtryk og assignments i C

Lektion 1 - slide 25

> Udtryk i C er urene i forhold til vores hidtidige forklaring af udtryk
>
> Årsagen er at et assigment opfattes som et udtryk i C

Vi illustrerer problemstillingen konkret herunder.

- Antag at `i` har værdien 7
- Programfragmentet `a = i + 5` er et udtryk i C
  - Udtrykket beregnes som `a = (i + 5)`
    - Operatoren `+` har højere prioritet end operatoren `=`
  - Som en *sideeffekt* ved beregningen af udtrykket tildeles variablen `a` værdien 12
  - Værdien af udtrykket `a = i + 5` er 'højresidens værdi', altså 12.

I nedenstående program viser vi 'brug og misbrug' af assignments i C. Det røde assignment udtryk initialiserer variablene `a`, `b`, `c` og `d` til 7. Dette kan anses for en "fiks" forkortelse for 4 assignment kommandoer i sekvens, adskilt af semicolons.

Det blå fragment er mere suspekt. Reelt assignes a til værdien af `2 + 3 - 1` (altså 4). I forbifarten assignes `b` til 2, `c` til 3, og `d` til 1.

```
#include <stdio.h>

int main(void) {

  int a, b, c, d;

  a = b = c = d = 7;  /* a = (b = (c = (d = 7)))  */
  printf("a: %d, b: %d, c: %d, d: %d\n", a,b,c,d);

  a = (b = 2) + (c = 3) - (d = 1);
  printf("a: %d, b: %d, c: %d, d: %d\n", a,b,c,d);

  return 0;
}
```

Program 3.1 *Brug og misbrug af assignments i udtryk.*

Outputtet af program 3.1 vises i program 3.2.

```
a: 7, b: 7, c: 7, d: 7
a: 4, b: 2, c: 3, d: 1
```

Program 3.2 *Output fra ovenstående program.*

Assignment operatorerne har meget lav prioritering, hvilket indebærer at tildelningen af variablen altid sker efter beregningen af 'højresiden'.

Assignment operatorerne associerer fra højre mod venstre.

3.2. Increment og decrement operatorerne

Lektion 1 - slide 26

C understøtter flere forskellige specialiserede assignment operatorer. Principielt er disse ikke nødvendige. Vi kan altid begå os med de former for assignments vi introducerede i kapitel 1 og specielt i afsnit 1.3.

Vi starter med at se på increment og decrement operatorerne. Disse er højt prioriterede, hvilket betyder at de udføres før de fleste andre udtryk (med andre operatorer).

C understøtter særlige operatorer til optælling og nedtælling af variable.

Man kan også bruge ordinære assignments ala `i = i + 1` og `i = i - 1` til dette.

Vi viser herunder ++ i både prefiks og postfix form. Læg mærke til at vi diskuterer både værdien af udtryk ala `++i` og `i++`, og effekten på variablen `i`.

- Antag at variablen `i` har værdien `7`
- Prefix formen: `++i`
  - Tæller først variablen `i` op fra `7` til `8`
  - Returnerer dernæst værdien `8` som resultat af udtrykket
- Postfix formen: `i++`
  - Returnerer først værdien `7` som resultatet af udtrykket
  - Tæller dernæst variablen `i` op fra `7` til `8`

3.3. Eksempel: increment og decrement operatorerne

Lektion 1 - slide 27

I fortsættelse af diskussionen i forrige afsnit viser vi her et konkret eksempel på brug af increment og decrement operatorerne i et C program.

I den blå del tælles `i` op med 1, `j` ned med 1, og `k` op med 1. Det er principielt ligegyldigt om det sker med prefix eller postfix increment operatorer.

I den røde del indgår increment operatorerne i andre assignment operatorer, og det er derfor vigtigt at kunne afgøre værdierne af `--i`, `j++`, og `--k`.

Først tælles `i` ned til 3, som så assignes til `res1`. Dernæst assignes værdien af `j` (som på dette sted i programmet er 3 til) `res2`, hvorefter `j` tælles op til 4. Med dette ændres værdien af variable `j` altså fra 3 til 4. Endelig tælles `k` ned til 5, og den nedtalte værdi assignes til `res3`.

```
#include <stdio.h>

int main(void) {

  int i = 3, j = 4, k = 5;
  int res1, res2, res3;

  i++; j--; ++k;
  /* i is 4, j is 3, k is 6 */

  printf("i: %d, j: %d, k: %d\n", i,j,k);

  res1 = --i;
  res2 = j++;
  res3 = --k;

  printf("res1: %d, res2: %d, res3: %d\n", res1, res2, res3);
  printf("i: %d, j: %d, k: %d\n", i,j,k);

  return 0;
}
```

Program 3.3 *Illustration af increment og decrement operatorerne.*

Outputtet af programmet vises herunder. Vær sikker på at du kan følge ændringerne af de forskellige variable.

```
i: 4, j: 3, k: 6
res1: 3, res2: 3, res3: 5
i: 3, j: 4, k: 5
```

Program 3.4 *Output fra ovenstående program.*

Herefter opsummerer vi vores forståelse af increment og decrement operatorerne.

Sondringen mellem prefix og postfix relaterer sig til tidspunktet for op/nedtælling af variablen relativ til fastlæggelsen af returværdien.

Hvis man bruger increment- og decrementoperatorerne som kommandoer, er det ligegyldigt om der anvendes prefix eller postfix form.

3.4. Andre assignment operatorer

Lektion 1 - slide 28

Foruden increment og decrement operatorerne fra afsnit 3.3 findes der også et antal akkumulerende assignment operatorer, som vi nu vil se på. At akkumulere betyder at 'hobe sig op'. I forhold til variable og assignments hentyder betegnelsen til at vil oparbejder en værdi i variablen, som er beregnet ud fra den forrige værdi ved enten at tælle det op eller ned.

Ligesom increment og decrement operatorerne er de akkumulerende assignment operatorer i bund og grund overflødige. Vi kan sagtens klare os med den basale assignment operator, som beskrevet i kapitel 1. Der kan dog være en lille effektivitetsgevinst ved at bruge increment, decrement, og de akkumulerende assignment operatorer. Mere væsentligt er det dog, at mange programmører bruger disse; og vi skal naturligvis være i stand til at forstå programmer skrevet af sådanne programmører.

C understøtter *akkumulerende assignment operatorer* for +, -, *, / and flere andre binære operatorer

Herunder vises den generelle syntaks af de nye akkumulerende assignment operatorer. *op* kan f.eks. være + eller −.

```
variable op= expressions
```

Syntaks 3.1

I syntaks 3.2 viser vi den basale assignment operator, som beskrevet i afsnit 1.1, i en situation som er ækvivalent med syntaks 3.1.

```
variable = variable op (expression)
```

Syntaks 3.2

Vi illustrerer igen med et konkret C program. I det røde assignment tælles `i` én op. I det blå assignment `j` ned med værdien af i. I det lilla assignment ganges `k` med værdien af `j`. I alle tre tilfælde udfører vi altså et assignment til hhv. `i`, `j` og `k`.

```c
#include <stdio.h>

int main(void) {

  int i = 3, j = 7, k = 9;

  i += 1;       /*  i = i + 1; */
  j -= i;       /*  j = j - i; */
  k *= j;       /*  k = k * j; */

  printf("i: %d, j: %d, k: %d\n", i, j, k);

  return 0;
}
```

Program 3.5 *Et program med forskellige optællinger og nedtællinger - modsvarer tidligere vist program.*

Outputtet af programmet vises herunder.

```
i: 4, j: 3, k: 27
```

Program 3.6 *Output fra ovenstående program.*

Du kan få en oversigt over alle akkumulerende assignment operatorer i C i tabel 2.3, nærmere betegnet i prioritetsklasse 2 (anden række regnet fra bunden af tabellen).

4. Udskrivning og indlæsning

Udskrivning af tegn på skærmen og på filer er vigtig for mange af de programmer vi skriver. Tilsvarende er det vigtigt at kunne indlæse data via tegn fra tastaturet, eller fra tekstfiler. Indlæsning er mere udfordrende end udskrivning, idet vi skal fortolke tegnene som indlæses. Undertiden vil vi sige at disse tegn skal *parses*. Ved udskrivning skal vi formatere data, f.eks. tal, som et antal tegn.

Undervejs i kurset vender vi flere gange tilbage til problematikken. (Den mest grundige behandling gives i kapitel 45). Her og nu introducerer vi det mest nødvendige, så vi kan foretage nødvendig input og output i de programmer, vi skriver i den første del af kurset.

4.1. Udskrivning med printf

Lektion 1 - slide 30

Kommandoen printf udskriver tal, tegn, og tekststrenge i et format, som er foreskrevet af en kontrolstreng. Deraf nok også navnet, print**f** - '**f**' for formattering. For hver procent-tegn, konverteringstegn, i kontrolstrengen skal der være en parameter efter kontrolstrengen. Der er således en en-til-en korrespondance mellem konverteringstegn i kontrolstrengen og parametre, som overføres efter kontrolstrengen. (Vi kommer tilbage til parametre til funktioner i afsnit 12.7).

Kommandoen **`printf`** kan udskrive data af forskellige fundamentale typer.

Formateringen af output styres af en *kontrolstreng*, som altid er første parameter.

I program 4.1 ser vi straks på et eksempel, som er lavet i forlængelse af program 1.7. Værdien af hver af de forskellige variable i program 1.7 skrives ud med `printf`.

I den røde del ser vi at alle heltal kan udskrives med konverteringstegnet d. Vi ser endvidere at værdier i typen short kan angives med hd, hvor h kaldes en 'size modifier' i forhold til konverteringstegnet. Værdien af `k`, som er af typen long, udskrives med en 'size modifier' l til konverteringstegnet d.

I den blå del udskriver vi forskellige reelle tal. Værdien af `x` er af typen double, som udskrives med konverteringstegnet f. Floats kan udskrives med det sammen konverteringstegn. For `z` bruger vi konverteringstegnet e. Med dette beder vi om 'scientific notation'. Bemærk at det er nødvendigt at bruge size modifier L foran e, idet `z` er erklæret som en long double.

I den brune del udskriver vi tegn med konverteringstegnet c, og i den lilla del strenge med s. Dette er uden de store komplikationer.

```
#include <stdio.h>

int main(void) {

  int i = 10;
  short j = 5;
  long k = 111L;
  printf("i: %d, j: %3hd, k: %10ld\n", i, j, k);

  double x = 10.5;
  float y = 5.5F;
  long double z = 111.75L;
  printf("x: %f, y: %5.2f, z: %Le\n", x, y, z);

  char c = 'A';
  char d = 'a';
  printf("c: %c, d: %d\n", c, d);

  char *s = "Computer", *t = "C";
  printf("s: %s,\nt: %s\n", s, t);

  return 0;
}
```

Program 4.1 *Udskrivning af variable af forskellige typer med printf.*

Det er vigtigt at typen af den dataværdi, vi udskriver, svarer nøje til det anvendte konveteringstegn, som er tegnet efter %-tegnet. Output fra ovenstående program kan ses herunder.

```
i: 10, j:   5, k:         111
x: 10.500000, y:  5.50, z: 1.117500e+02
c: A, d: 97
s: Computer,
t: C
```

Program 4.2 *Output fra ovenstående program.*

Vi opsummerer her, ganske overordnet, egenskaberne og mulighederne i printf. Vi har mere at sige om dette i afsnit 18.2, afsnit 18.6 og afsnit 45.2.

- Egenskaber af **`printf`** kommandoen:
  - Indlejring af formaterede parametre i en fast tekst.
  - Én til én korrespondance mellem konverteringstegn og parametre efter kontrolstrengen.
  - Mulighed for at bestemme antal tegn i output, antal cifre før og efter 'komma', venstre/højrestilling, og output talsystem.
  - Antallet af udlæste tegn returneres

4.2. Indlæsning med scanf

Lektion 1 - slide 31

Vi ser her kort og relativt overfladisk på indlæsning med scanf.

Kommandoen **`scanf`** fortolker tegn med henblik på indlæsning af værdier af de fundamentale datatyper

Fortolkning og styring af input sker med en *kontrolstreng*, analogt til kontrolstrengen i **`printf`**.

Her følger et eksempel. Vi indlæser en float i variablen `x`. Dette gøres gentagne gange, i en while løkke. Indlæsningen stopper når der ikke længere er indlæst netop ét tal. I praksis vil løkken nok blive afbrudt, hvis man slet ikke indtaster data, hvilket vil medføre at scanf returnerer 0.

Læg endvidere mærke til hvorledes værdierne af heltallet `cnt` og det reelle tal `sum` udskrives med printf i eksemplet.

```c
/* Sums are computed. */

#include <stdio.h>

int main(void)
{
   int     cnt = 0;
   float   sum = 0.0, x;

   printf("The sum of your numbers will be computed\n\n");
   printf("Input some numbers:  ");
   while (scanf("%f", &x) == 1) {
      cnt = cnt + 1;
      sum = sum + x;
   }
   printf("\n%s%5d\n%s%12f\n\n",
      "Count:", cnt,
      "  Sum:", sum);
   return 0;
}
```

Program 4.3 *Indlæsning og summering af tal.*

- Egenskaber af **`scanf`** kommando:
  - De indlæste data indlæses i angivne lageraddresser
  - Antallet af gennemførte indlæsninger/konverteringer returneres

4.3. Et input output eksempel

Lektion 1 - slide 32

Vi afslutter dette afsnit med et eksempel, som stammer fra C by Dissection. Læg først og fremmest mærke til at vi indlæser `radius` med lf i kontrolstrengen i scanf. Dette svarer altså ikke helt til det tilsvarende konverteringstegn for doubles i printf, som jo kun er f. Vi viser en udgave hvor vi har fremhævet de enkelte bestanddele med farver, så det forhåbentlig bliver lidt lettere at se, hvordan det hele hænger sammen.

```c
#include <stdio.h>

#define PI 3.141592653589793

int main(void)
{
   double radius;

   printf("\n%s\n\n%s",
      "This program computes the area of a circle.",
      "Input the radius:  ");
   scanf("%lf", &radius);
   printf("\n%s\n%s%.2f%s%.2f%s%.2f\n%s%.5f\n\n",
      "Area = PI * radius * radius",
      "     = ",  PI, " * ",  radius, " * ",  radius,
      "     = ", PI * radius * radius );
   return 0;
}
```

Program 4.4 *Samme program med farvede fremhævninger af kontrolstreng og parametre til printf.*

Her ses output fra programmet:

```
This program computes the area of a circle.

Input the radius:  5.3

Area = PI * radius * radius
     = 3.14 * 5.30 * 5.30
     = 88.24734
```

Program 4.5 *Input til og output fra programmet.*

Flere detaljer om scanf følger senere i materialet. I afsnit 18.2 giver vi en nyttig oversigt over heltalstyper og scanf konverteringstegn. Tilsvarende oversigt bliver givet for flydende tal i afsnit 18.6. Endelig diskuterer vi nogle flere scanf detaljer i afsnit 45.3.

5. Kontrolstrukturer - Motivation og Oversigt

I dette og de følgende afsnit vil vi - vigtigst af alt - møde forgreninger og løkker. Sådanne kaldes kontrolstrukturer, fordi vi med disse kan kontrollere måden hvorpå 'programpegepinden' bevæger sig rundt i programmet. Vi vil også diskutere sekvens og sammensætning.

5.1. Oversigt over kommandoer

Lektion 2 - slide 2

Vi har i de forrige afsnit mødt forskellige kommandoer, som opsummeres herunder. Kontrolstrukturerne, som er fokus i dette og efterfølgende afnit, virker på kommandoer. Med andre ord, kontrolstrukturerne angiver hvorledes en række kommandoer udføres i forhold til hinanden.

Vi starter med en oversigt over de kommandoer vi har mødt indtil nu

- Assignments
  - Udtryk med en assignment operator efterfulgt af semicolon
- **`Printf`** og **`scanf`** kommandoerne
  - Disse er reelt funktionskald efterfulgt af semicolon
- Den tomme kommando
  - Blot et semicolon

5.2. Motivation: Primitiv kontrol med hop

Lektion 2 - slide 3

Vi vil motivere os selv for at studere moderne kontrolstrukturer ved at se på en primitiv kontrolstyring med goto kommandoen.

En goto kommando tillader hop fra ét sted i programmet til et andet.

Programmer med goto kommandoer kan være *meget vanskelige af forstå*.

Nedenstående program tildeler værdien 2 til variablen `pos` (og `res` bliver `j`) hvis `i` er mindre eller lig med `j`. Hvis `i` er større end `j` tildeles `pos` værdien 1, og `res` bliver `i`. De blå programdele i program 5.1 kaldes for labels.

Det er forholdsvis svært at følge kontrolforløbet af selv dette simple program, som bruger betinget og ubetinget flytning af kontrolpunktet med goto kommandoer. Vi vender tilbage til en meget bedre udgave af program 5.1 i program 8.2

```c
#include <stdio.h>

int main(void) {

  int i, j, pos, res;
  printf("Enter two integers: ");
  scanf("%d %d", &i, &j);

  if (i <= j) goto p1;
  pos = 1; res = i;
  goto p2;
  p1: pos = 2; res = j;

  p2: printf("pos: %d, res: %d\n", pos, res);

  return 0;
}
```

Program 5.1 *Et helt C program med to goto kommandoer.*

I næsten alle programmeringssituationer anbefales det at bruge kontrolstrukturer i stedet for goto kommandoer

Goto kommandoer er kun acceptable i undtagelsessituationer, eksempelvis for at komme ud af en 'dyb løkke'.

5.3. Oversigt over konstrolstrukturer

Lektion 2 - slide 4

Inden for kaster os over detaljerede kontrolstrukturer giver vi en oversigt over forskellige slags kontrolstrukturer.

De grundlæggende kontrolstrukturer kan klassificeres som sekventielle, sammensættende, udvælgende og gentagende

En *kontrolstruktur* styrer og kontrollerer rækkefølgen af udførelsen af et antal kommandoer

- **Sekventiel kontrol**
  - Kommandoer følger efter hinanden i den angivne rækkefølge
  - `kommando1; kommando2;`
- **Sammensætning**
  - En antal kommandoer sammensættes til én enkelt kommando
  - `{kommando1; kommando2;}`
- **Udvælgelse**
  - Én kommando udvælges blandt flere mulige.
  - Udvælgelsen foretages typisk ud fra værdien af et logisk udtryk.
  - `if (logiskUdtryk) kommando1; else kommando2;`

- **Gentagelse**
  - En kommando gentages et antal gange.
  - Antallet af gentagelser styres typisk af et logisk udtryk
  - `while (logiskUdtryk) kommando;`

Som sagt er udvælgende og gentagende kontrolstrukturer meget vigtige for langt de fleste af de programmer vi skriver. Sekventiel kontrol medtages for at få et komplet billede. Sammensætning kaldes også undertiden for *aggregering*. Det at *sammensætte dele til helheder* vil vi se i mange forskellige sammenhænge.

6. Logiske udtryk

Logiske udtryk bruges til at styre udvælgelse og gentagelse. Derfor er det vigtigt at vi har god styr på logiske udtryk. Dette vil vi få i dette afsnit.

6.1. Logiske udtryk

Lektion 2 - slide 6

Logiske udtryk kan naturligt sammenlignes med aritmetiske udtryk. Aritmetiske udtryk beregnes til talværdier. Logiske udtryk beregnes til sandhedsværdier, enten *sand* eller *falsk* (*true* eller *false* på engelsk). På samme måde som vi har set aritmetiske operatorer vil i dette afsnit studere logiske operatorer, som returnerer sandhedsværdier.

Vi skal vænne os til at håndtere sandhedsværdier på samme måde som vi håndterer tal, tegn, mv. Vi kan putte sandhedsværdier i variable, vi kan sammenligne dem mv. For en del af jer tager det måske lidt tid inden I finder den rette og naturlige melodi for håndteringen af sandhedsværdier.

Et *logisk udtryk* beregnes til en værdi i datatypen boolean.

Datatypen *boolean* indeholder værdierne *true* og *false*.

Her følger en række eksempler på logiske udtryk, som alle anvender heltalsvariablen `n`. Eksemplerne er vist inden for rammerne af et komplet C program i program 6.2.

```
int n = 6;

n == 5
n >= 5
n != 5
n >= 5 && n < 10
!(n >= 5 && n < 10)
n < 5 || n >= 10
```

Program 6.1 *Et uddrag af et C program med logiske udtryk.*

Først tester vi om `n` er lig med 5. `==` er altså en lighedoperator, som her returnerer false, idet `n` har værdien 6. Bemærk den afgørende forskel på assignment operatoren `=` og lighedsoperatoren `==`. I anden linie tester vi om `n` er større end eller lig med 5. Dette er sandt. I tredje linie tester vi om `n` er forskellig fra 5. Dette er også sandt. I fjerde line tester vi om `n` er større eller lig med 5 og mindre end 10. Bemærk `&&` svarer til 'logisk and'. (Se afsnit 6.2). Værdien er true igen. Udråbstegnet foran det tilsvarende udtryk betyder 'not', og det negerer altså værdien true til false. Det sidste logiske udtryk fortæller hvorvidt `n` er mindre end 5 eller større eller lig med 10. Når `n` stadig er 6 er dette naturligvis falsk. `||` betyder altså 'logisk eller'.

```
#include <stdio.h>

int main(void) {

  int n = 6, b;

  printf("%d\n", n == 5);

  if (n >= 5) printf("stor\n"); else printf("lille\n");

  b = n != 5; printf("%d\n", b);

  while (n >= 5 && n < 10) n++;
  printf("%d\n", n);

  if ( !(n >= 5 && n < 10) == (n < 5 || n >= 10))
    printf("OK\n");
  else printf("Problems\n");

  return 0;
}
```

Program 6.2 *De boolske udtryk vist i realistiske sammenhænge i et C program.*

Nedenstående er en vigtig detalje i C. Nul står for *false*, og ikke-nulværdier står alle for *true*.

I C repræsenteres *false* af en nulværdi, og *true* af en vilkårlig ikke-nulværdi

6.2. Sandhedstabeller

Lektion 2 - slide 7

Nu hvor vi er igang med at diskutere logiske udtryk benytter vi lejligheden til præcise definitioner af negering (not), konjunktion (and) og disjunktion (or) via sandhedstabellerne i hhv. tabel 6.1, tabel 6.2 og tabel 6.3. Negering håndteres af `!` operatoren i C. Tilsvarende hedder and operatoren `&&` i C, og or operatoren hedder `||`.

Da der kun findes to forskellige sandhedsværdier er det let at tabellægge alle forskellige muligheder for input og output af de nævnte operatorer. Bemærk her forskellen til heltal. Da der findes uendelig mange forskellige heltalsværdier vil tilsvarende definitioner af `+`, `-`, `*` og `/` være umulig.

A	!A
true	*false*
false	*true*

Tabel 6.1 *Sandhedstabel for not operatoren*

A	B	A && B	A || B
true	*true*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*

Tabel 6.2 *Sandhedstabel for and operatoren*

A	B	!(A && B)	!A || !B
true	*true*	*false*	*false*
true	*false*	*true*	*true*
false	*true*	*true*	*true*
false	*false*	*true*	*true*

Tabel 6.3 *Sandhedstabel for or operatoren*

Bemærk her at tabel 6.3 faktisk beviser DeMorgans boolske lov, som på et mere matematisk sprog siger at

- **not** (A **and** B) = **not** (A) **or not** (B).

I slide materialet viser vi en særlig udgave af operatorprioriteringstabellen i C, hvor de logiske og de sammenlignende operatorer er fremhævet. Se slide siden.

6.3. Short Circuit evaluering

Lektion 2 - slide 9

Vi vil nu beskæftige os med nogle binære, logiske operatorer, som under nogle omstændigheder ikke vil beregne værdien af den højre operand.

Direkte oversat 'kortsluttet beregning'

Short circuit evaluering betegner en beregning hvor visse operander ikke evalueres, idet deres værdi er uden betydning for resultatet

- `x && y`
  - y beregnes kun hvis x er *true*
- `x || y`
  - y beregnes kun hvis x er *false*

6.4. Short Circuit evaluering: eksempler

Lektion 2 - slide 10

Vi ser først på et eksempel, som er ganske kunstig, men dog illustrativ. Dernæst, i program 6.6 følger et 'real life' eksempel.

Vi viser eksempler på C programmer med short circuit evaluering

Herunder ses en række assignment operatorer som indgår i logiske udtryk. Dette kan lade sig gøre i C idet assignment operatorer returnerer en værdi, ligesom alle andre operatorer. Dette diskuterede vi i kapitel 3. En anden essentiel observation for nedenstående er at tallet nul spiller rollen som sandhedsværdien *false*, og at andre tal spiller rollen som *true*. Dette blev diskuteret i afsnit 6.1.

```
#include <stdio.h>

int main(void) {

  int i = 0, j = 0;

  (i = 1) && (j = 2);
  printf("%d %d\n", i, j);

  (i = 0) && (j = 3);
  printf("%d %d\n", i, j);

  i = 0 || (j = 4);
  printf("%d %d\n", i, j);

  (i = 2) || (j = 5);
  printf("%d %d\n", i, j);

  return 0;
}
```

Program 6.3 *Short circuit beregning i kombination med assignments.*

De to første assignments udføres begge, idet værdien af den første er 1 (true). Af de to næste assignment udføres kun det første. Årsagen er at `i` assignes til 0, som er false, hvilket forårsager undertrykkelse af beregningen af højre operand af &&. Det næste assignment skal fortolkes som `i = (0 || (j = 4))`. Husk igen af 0 er false. Højre operand af `||` skal derfor beregnes, jf. reglerne fra afsnit 6.3. Endelig gennemføres assignmentet til `j` ikke i sidste røde linie, idet venstre operand beregnes til 2 (true).

Med ovenstående observationer er program 6.4 ækvivalent med program 6.3.

```c
#include <stdio.h>

int main(void) {

  int i = 0, j = 0;

  (i = 1) && (j = 2);
  printf("%d %d\n", i, j);

  (i = 0);
  printf("%d %d\n", i, j);

  i = 0 || (j = 4);
  printf("%d %d\n", i, j);

  (i = 2);
  printf("%d %d\n", i, j);

  return 0;
}
```

Program 6.4 *Et ækvivalent program.*

Outputtet fra de to ovenstående (ækvivalente) programmer ses her:

```
1 2
0 2
1 4
2 4
```

Program 6.5 *Output fra ovenstående programmer.*

Og nu til et mere dagligdagsprogram. I den røde linie ses et logisk udtryk, hvor den venstre operator beskytter den højre. Det skal forstås således at `sqrt(x)` kun bliver beregnet hvis `x` er ikke-negativ. Dette er helt essentielt for at undgå en regnefejl i kvadratrodsfunktionen `sqrt`, som jo kun kan tage ikke-negative parametre.

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

double sqrt(double);

int main(void) {

  double x;

  printf("Enter a number: ");
  scanf("%lf", &x);

  if (x >= 0 && sqrt(x) <= 10)
    printf("x is positive, but less than or equal to 100.0\n");
  else printf("x is negative or greater than 100.0\n");

  return 0;
}
```

Program 6.6 *Short circuit beregning som beskyttelse mod fejlberegning.*

Prøvekør meget gerne programmet. Bemærk at på nogle C systemer skal der en ekstra option **`-lm`** på kompileren for at håndtere kvadratrodsfunktionen.

7. Sammensætning af kommandoer

Af hensyn til en ren strukturering efter sammensætning, udvælgelse og gentagelse har vi her et kort afsnit om sammensætning af flere kommandoer til én helhedskommando - blokke.

7.1. Sammensætning

Lektion 2 - slide 12

Vi bruger ofte betegnelsen *blokke* for en sammensætning af kommandoer til én kommando. I en blok af kommandoer kan vi endvidere frit erklære variable. Vi foretrækker dog at variable introduceres i starten af blokken, altså før den første kommando.

> En sammensat kommando, også kaldet en *blok*, er en gruppering af kommandoer til én kommando, hvori der i starten kan forekomme erklæringer af variable

Bestanddelene i en sammensat kommando indhegnes af 'tuborg klammer':

```
{declaration-or-command-list}
```

Syntaks 7.1 *En blok*

Det er ofte nødvendig at bruge sammensatte kommandoer, nemlig på pladser hvor kun én enkelt kommando er tilladt. Dette er f.eks. tilfældet for kroppene af if-else og while, som vi ser på i hhv. afsnit 8.1 og afsnit 9.1.

> Blokke er ofte nødvendige i forbindelse med udvælgende og gentagende kontrolstrukturer i C.
>
> Blokke kan indlejres i hinanden.

Vi viser herunder et eksempel på indlejring af blokke i hinanden. Den yderste blok er en fast bestanddel af `main` funktionen. I afsnit 20.1 nærmere betegnet program 20.1, vender vi tilbage til det faktum, at de samme navne `a`, `b` og `c` erklæres i de indlejrede blokke.

```
#include <stdio.h>

int main(void) {

  int a = 5, b = 7, c;

  c = a + b;

  {
    int a = 15, b = 17, c;

    {
      int a = 25, b = 27, c;
      c = a + b;
      printf("Inner: c er %d\n", c);
    }

    c = a + b;
    printf("Middle: c er %d\n", c);
  }

  c = a + b;
  printf("Outer: c er %d\n", c);

  return 0;
}
```

Program 7.1 *Tre indlejrede blokke med lokale erklæringer af heltal.*

8. Udvælgelse af kommandoer

Der er kun få programmer som slavisk følger ét enkelt spor uden forgreninger. Med andre ord er der kun få programmer der udelukkende betjener sig af sekventiel og sammensat kontrol, som beskrevet i hhv. afsnit 5.3 og afsnit 7.1.

Langt de fleste programmer skal kunne reagere forskelligt på en række hændelser. Derfor er *udvælgelse* - også kaldet *selektion* - af kommandoer et vigtigt emne. Vi studerer udvælgelse i dette kapitel.

8.1. Udvælgelse med if (1)

Lektion 2 - slide 14

Den mest basale kontrolstruktur til udvælgelse kaldes **if-then-else**. I den rene form, som i C afspejles af syntaks 8.1, vælges der mellem netop to muligheder. Valget styres af et logisk udtryk, som jo netop har to mulige udfald: *true* eller *false*. Som det fremgår af syntaksboken anvender C ikke nøgleordet **then**, og det er derfor lidt misvisende af tale om **if-then-else** kontrolstrukturer i C. Vi vil derfor blot tale om **if-else** strukturer.

En *selektiv kontrolstruktur* udvælger én kommando til udførelse blandt en mængde af muligheder

```
if (logicalExpression)
  command1
else
  command2
```

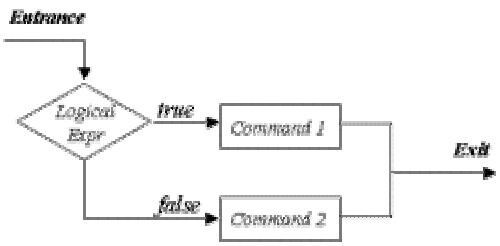
Syntaks 8.1 *En if-else kontrolstruktur.*

Bemærk parenteserne omkring *logicalExpression*, som altid skal være der. Parenteserne er en del af syntaksen for if-else i C. Tilsvarende forhold gør sig gældende for switch (syntaks 8.3), while (syntaks 9.1), og do (syntaks 9.2). Som et specialtilfælde kan else-delen være tom. I så fald er det ikke nødvendig at skrive nøgleordet 'else'. Vi taler i dette tilfældet om en *betinget kommando*. Vi siger altså at *command* er betinget af det logiske udtryk *logicalExpression* i syntaks 8.2. *Command* udføres kun hvis værdien af det logiske udtryk er sandt.

```
if (logicalExpression)
  command
```

Syntaks 8.2 *En betinget kommando.*

Herunder viser vi en kontrol flowgraf for en if-then-else kommando, som modsvarer syntaks 8.1. Sådanne grafer er nyttige til illustration er enkelte kontrolstrukturer i isolation, men de bliver dog for det meste uoverskuelige hvis vi forsøger at anvende dem på et helt program.



Figur 8.1 *Flow graf for if*

8.2. Udvælgelse med if (2)

Lektion 2 - slide 15

Vi fortsætter vores diskussion af if-else i dette afsnit. Betydningen af syntaks 8.1 kan sammenfattes i følgende punkter:

- Betydning af if-else:
  - Først beregnes det logiske udtryk
  - Hvis værdien er sand udføres command1
  - Hvis værdien er falsk udføres command2

Det er nu tid til at se på eksempler. I program 8.1 starter vi med et program, der finder og udskriver det største af to indlæste tal x og y. Bemærk hvordan vi i den røde del styrer kontrollen til enten at udføre assignmentet `max = y` eller `max = x`, afhængig af udfaldet af sammenligningen mellem `x` og `y`.

```
#include <stdio.h>

int main(void) {

  double x, y, max;

  printf("Enter two real numbers: ");
  scanf("%lf %lf", &x, &y);

  /* find the maximum of x and y */
  if (x < y)
    max = y;
  else
    max = x;

  printf("The maximum of x and y is %12.8f\n", max);

  return 0;
}
```

Program 8.1 *Et program som beregner det største af to doubles.*

Hernæst følger et andet eksempel, som vi allerede har varmet op i afsnit 5.2, nærmere betegnet i program 5.1. Her er tale om en omskrivning af goto programmet til at bruge en if-else kontrolstruktur.

Sammenlign nøje med program 5.1 og overbevis dig om if-else programmets bedre struktur og større læsbarhed.

```c
#include <stdio.h>

int main(void) {

  int i, j, pos, res;
  printf("Enter two integers: ");
  scanf("%d %d", &i, &j);

  if (i <= j){
    pos = 2; res = j;}
  else {
    pos = 1; res = i;}

  printf("pos: %d, res: %d\n", pos, res);

  return 0;
}
```

Program 8.2 *Goto programmet reformuleret med en if-else kontrolstruktur.*

Brug af kontrolstrukturer ala **`if-else`** er udtryk for *struktureret programmering*

Struktureret programmering var en af de store landvindinger inden for faget i 1970'erne.

8.3. 'Dangling else' problemet

Lektion 2 - slide 16

I dette afsnit vil vi beskrive en speciel sammensætning af if og if-else kontrolstrukturer, som foranlediger et tvetydighedsproblem.

Hvis **`if`** og **`if-else`** kontrolstrukturer indlejres i hinanden kan der opstå tvetydighedsproblemer

Problemet illustreres i eksempel program 8.3 herunder.

I den røde del signalerer indrykningen at else delen hører til den inderste if. I situationen hvor `a` er 3 og `b` er 4 bliver der ikke printet noget som helst i den røde del.

På trods af indrykningen i den blå del, er den blå del helt ækvivalent med den røde del. Som opsummeret herunder gælder der, at en else del altid knytter sig til den nærmeste if. Der skrives altså heller ikke noget ud i den blå del.

I den lilla del har vi indlejret `if (b==2) print(...)` i en blok, som er angivet med brune brackets. Dermed knyttes else delen til den yderste if. I denne situation udskrives et stort F.

```c
#include <stdio.h>

int main(void) {

  int a = 3, b = 4;

  if (a == 1)
    if (b == 2)
      printf("A\n");
    else
      printf("B\n");

  if (a == 1)
    if (b == 2)
      printf("C\n");
  else
    printf("D\n");

  if (a == 1){
    if (b == 2)
      printf("E\n");}
  else
    printf("F\n");

  return 0;
}
```

Program 8.3 *Illustration af dangling else problemet.*

Som beskrevet ovenfor vil program 8.3 blot udskrive et stort F.

> En **else** knytter sig altid til den inderste **if**.
>
> Det kan anbefales at bruge sammensætning med {...} hvis man er i tvivl om fortolkningen.

8.4. Udvælgelse med switch

Lektion 2 - slide 17

Switch er en anden kontrolstruktur til udvælgelse. Med switch vælger man mellem et vilkårligt antal muligheder. Valget foregår på basis af en heltallig værdi. Derfor er en switch ikke så kraftfuld som evt. indlejrede if-else konstruktioner. Mere om dette i afsnit 8.5.

```
switch (expression) {
  case const1: command-list1
  case const2: command-list2
  ...
  default: command-list
}
```

Syntaks 8.3 *Opbygningen af en switch kontrolstruktur.*

Vi beskriver betydningen af switch i de følgende punkter

- Udtrykket beregnes - skal være et heltal eller heltalsagtigt
- Hvis værdien svarer til en af konstanterne, flyttes kontrollen til den tilsvarende kommando
- Hvis værdien ikke svarer til en af konstanterne, og hvis der er en default, flyttes kontrollen til default kommandoen
- Hvis værdien ikke svarer til en af konstanterne, og hvis der ikke er en default, flyttes kontrollen til afslutningen af switchen

Der er ét væsentlig forhold, som skal bemærkes. Når et af 'casene' i en switch er udført, hopper man ikke automatisk til enden af switch konstruktionen. Derimod fortsætter udførelsen af de efterfølgende cases i switchen. Som regel er dette ikke hvad vi ønsker. Som illustreret i program 8.4 kan man afbryde dette 'gennemfald' med en break i hvert case. Dette er de blå dele i program 8.4.

```
#include <stdio.h>

int main(void) {

  int month, numberOfDays, leapYear;

  printf("Enter a month - a number between 1 and 12: ");
  scanf("%d", &month);
  printf("Is it a leap year (1) or not (0): ");
  scanf("%d", &leapYear);

  switch(month){
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      numberOfDays = 31; break;
    case 4: case 6: case 9: case 11:
      numberOfDays = 30; break;
    case 2:
      if (leapYear) numberOfDays = 29;
      else numberOfDays = 28; break;
    default: exit(-1);  break;
  }

  printf("There are %d days in month %d\n", numberOfDays, month);

  return 0;
}
```

Program 8.4 *Et program der beregner antal dage i en måned.*

Programmet ovenfor finder antallet af dage i en given måned. Hvis måneden er 1, 3, 5, 7, 8, 10 eller 12 er der altid 31 dage. Hvis måneden er 4, 6, 9 eller 11 er der 30 dage. I februar, den 2. måned, er der 28 eller 29 dage afhængig af om året er et skudår eller ej.

Vi har mere at sige om break i afsnit 9.7.

Vi ønsker næsten altid at afslutte et tilfælde i en switch med en **break**

8.5. If-else kæder

Lektion 2 - slide 18

Vi kan kun bruge switch til multivejsudvælgelse i det tilfælde at valget kan foretages på baggrund af en simpel, heltallig værdi.

I mange tilfælde er det nødvendigt at foretage en multivejsudvælgelse ud fra evaluering af en række betingelser. Når der er behov for dette kan vi bruge if-else kæder. Bemærk, at if-else kæder laves ved at indlejre en if-else kontrolstruktur i else-delen af en anden if-else struktur.

En **if-else** kæde er en **if-else** kontrolstruktur, hvor else-delen igen kan være en **if-else** struktur

Som et eksempel viser vi et program som omregner procentpoint til en karakter på 13-skalaen. Omregningen svarer til den officielle omregningstabel for skriftligt arbejde inden for naturvidenskab.

Det vil være meget akavet at skulle udtrykke omregningen med en switch. Dette vil nemlig kræve en case for hvert heltal, hvilket naturligvis er helt uacceptabelt. Det er langt mere naturligt at formulere logiske udtryk for de forskellige udfald.

```
#include <stdio.h>

int main(void) {

  int percent, grade;

  printf("How many percent? ");
  scanf("%d",&percent);

  if (percent >= 90)
    grade = 11;
  else if (percent >= 82)
    grade = 10;
  else if (percent >= 74)
    grade = 9;
  else if (percent >= 66)
    grade = 8;
  else if (percent >= 58)
    grade = 7;
  else if (percent >= 50)
    grade = 6;
  else if (percent >= 40)
    grade = 5;
  else if (percent >= 10)
    grade = 3;
```

```
  else grade = 0;

  printf("%d percent corresponds to the Danish grade %d\n\n",
          percent, grade);

  return 0;
}
```

Program 8.5 *Et program med en if-else kæde der beregner beregner en karakter ud fra et antal procentpoint.*

Vi har i program 8.5 anvendt en speciel indrykning af if-else strukturerne. En mere almindelig indrykning, som understreger måden hvorpå if-else er indlejret i else dele, er vist i program 8.6 herunder. Med denne indrykning havner vi dog hurtigt uden for den højre margin, og af denne årsag foretrækkes den flade indrykning i program 8.5.

```
#include <stdio.h>

int main(void) {

  int percent, grade;

  printf("How many percent? ");
  scanf("%d",&percent);

  if (percent >= 90)
   grade = 11;
  else if (percent >= 82)
        grade = 10;
       else if (percent >= 74)
             grade = 9;
            else if (percent >= 66)
                  grade = 8;
                 else if (percent >= 58)
                       grade = 7;
                      else if (percent >= 50)
                            grade = 6;
                           else if (percent >= 40)
                                 grade = 5;
                                else if (percent >= 10)
                                      grade = 3;
                                     else grade = 0;

  printf("%d percent corresponds to the Danish grade %d\n\n",
          percent, grade);

  return 0;
}
```

Program 8.6 *Alternativt og uønsket layout af if-else kæde.*

Læs mere om indrykning og 'style' i afsnit 3.20 side 106-108 i *C by Dissection*.

En **if-else** kæde kan generelt ikke programmeres med en **`switch`** kontrolstruktur

8.6. Den betingede operator

Lektion 2 - slide 19

If-else og switch er kontrolstrukturer, som styrer udførelsen af kommandoer. I mange engelske bøger bruges betegnelsen *statements* for kommandoer. I dette afsnit vender vi tilbage til udtryk, som vi studerede allerede i afsnit 2.1. Helt konkret ser vi på en if-then-else konstruktion som har status af et udtryk i C.

> I C findes en betinget operator med med tre operander som modsvarer en **`if-else`** kontrol struktur.
>
> En forekomst af en betinget operator er et udtryk, hvorimod en forekomst af **`if-else`** er en kommando.

Syntaksen af operatoren, som kaldes `?:`, er vist herunder. Ligesom if-else har den tre bestanddele, nemlig et logisk udtryk og to andre betanddele, som her begge er vilkårlige udtryk. Begge disse udtryk skal dog være af samme type.

```
logicalExpression ? expression1 : expression2
```

Syntaks 8.4 *Opbygningen af den betingede operator i C.*

Bemærk at den betingede operator tager tre operander i modsætning til de fleste andre operatorer, som enten er unære eller binære.

Her giver vi betydningen af `?:` operatoren relativ til syntaks 8.4.

- Først beregnes *logicalExpresssion*
- Hvis værdien er sand beregnes *expression1* ellers *expression2,* hvorefter den pågældende værdi returneres af **`?:`** operatoren

Vi viser herunder et eksempel på brug af den betingede operator. Givet et indlæst heltal i variablen `x` bestemmer det røde udtryk fortegnet af `x`. Det er enten strengen "negativ", "neutral", eller "positiv".

Ifølge tabel 2.3 associerer den betingede operator fra højre mod venstre. (Du finder den på prioritetsniveau 3 i tabellen). Det betyder at strukturen i udtrykket er følgende:

- `"x < 0 ? "negative" : (x == 0 ? "neutral" : "positive")`

```c
#include <stdio.h>

int main(void) {

  int x;

  printf("Enter an integer: ");
  scanf("%d", &x);

  printf("The sign of %d is %s\n\n",
         x,
         x < 0 ? "negative" : x == 0 ? "neutral" : "positive");

  return 0;
}
```

Program 8.7 *Et program der bestemmer fortegnet af et tal.*

Bemærk at det ikke ville have været muligt at have en if-else konstruktion inden i kaldet af printf. Kun udtryk kan optræde som parametre til funktioner. Og den røde del af program 8.7 er netop et udtryk.

Hermed slutter vores behandling af kontrolstrukturer og udtryk til udvælgelse. I næste afsnit ser vi på gentagelse.

9. Gentagelse af kommandoer

En af styrkerne ved en computer er dens evne til at gennemføre mange gentagelser på kort tid. Når vi programmerer bruger vi forskellige løkker til at kontrollere gentagelser af kommandoer. Undertiden bruget vi også betegnelsen *iterationer* i stedet for gentagelser.

I C er der tre forskellige løkker, som vi alle vil se på i dette afsnit. Vi starter med whileløkker.

9.1. Gentagelse med while (1)

Lektion 2 - slide 21

Hvis man kun vil lære én løkke konstruktion er det while løkken man bør kaste sig over. Som vi vil indse i afsnit 9.8 kan alle løkker i C relativt let omformes til whileløkker.

I en løkke er det muligt at gentage en kommando nul, én eller flere gange.

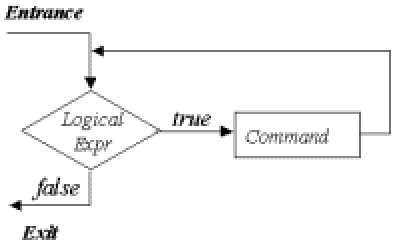
I en while løkke er det ubestemt hvormange gange kommandoen udføres.

Syntaksen af whileløkker minder om syntaksen af if, jf. syntaks 8.2 . Betydningen af en while kontrolstruktur er derimod helt forskellig fra en if kontrolstruktur.

```
while (logicalExpression)
  command
```

Syntaks 9.1 *Syntaksen af en while løkke i C.*

Vi viser en flowgraf for whileløkken herunder. Bemærk for det første muligheden for at køre rundt i løkken, altså det at kommandoen kan udføres et antal gange, under kontrol af det logiske udtryk. Bemærk også, at vi kan udføre whileløkken helt uden af udføre kommandoen (nemlig i det tilfælde at det logiske udtryk er falsk når vi møder det første gang).



Figur 9.1 *Flow graf for en while løkke*

9.2. Gentagelse med while (2)

Lektion 2 - slide 22

Vi beskriver her på punktform betydningen af en whileløkke.

- Betydningen af while:
  - Det logiske udtryk beregnes
  - Hvis værdien er *true* udføres kommandoen, og while løkken startes forfra
  - Hvis værdien er *false* afsluttes while løkken

Vores første eksempel er et klassisk program, nemlig *Euclids algoritme* til at finde den største fælles divisor af to positive heltal. Givet to heltal `small` og `large` ønsker vi altså at finde det største tal, som både går op i `small` og `large`.

```
#include <stdio.h>

int main(void) {
  int i, j, small, large, remainder;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  small = i <= j ? i : j;
  large = i <= j ? j : i;

  while (small > 0){
    remainder = large % small;
    large = small;
    small = remainder;
  }

  printf("GCD of %d and %d is %d\n\n", i, j, large);

  return 0;
}
```

Program 9.1 *Euclids algoritme - største fælles divisor - programmeret med en while løkke.*

Ved første øjesyn er det ikke umiddelbart klart, hvorfor algoritmen virker. Man mener, at netop ovenstående algoritme er det første eksempel i historien på en ikke-triviel beregningsmæssig opskrift (algoritme).

Lad os kort forklare programmet ud fra et eksempel. Lad os antage at vi skal finde den største fælles divisor af tallene 30 og 106. Dette tal betegnes som gcd(30,106), og det viser sig at være 2. gcd betyder "greatest common divisor". Variablen `small` starter altså med at være 30 og `large` er 106. I det første gennemløb af whileløkken beregnes resten af division af 106 med 30. Dette er 16, og dette tal vil generelt altid være mindre end divisoren 30. Hermed bliver `large` sat til 30 og `small` til 16.

Set over et antal gennemløb producerer programmet talrækken

- 106, 30, 16, 14, 2, 0

Den essentielle indsigt er nu at gcd(106,30) = gcd(30,16) = gcd(16,14) = gcd(14,2) = gcd(2,0) = 2. Dette indses ved et induktionsargument. Den sidste lighed følger af, at det største tal der både går op i 2 og 0 naturligvis er 2.

Med dette vil programmet finde, at det største tal der både går op i 106 og 30 er 2. Ydermere viser det sig, at programmet finder resultatet i et bemærkelsesværdigt lille antal skridt.

Herunder er vist en simpel variant af program 9.1 som udskriver information om mellemresultaterne.

```c
#include <stdio.h>

int main(void) {
  int i, j, small, large, remainder;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  small = i <= j ? i : j;
  large = i <= j ? j : i;

  printf("%d %d ", large, small);

  while (small > 0){
    remainder = large % small;
    large = small;
    small = remainder;
    printf("%d ", small);
  }

  printf("\n\nGCD of %d and %d is %d\n\n", i, j, large);

  return 0;
}
```

Program 9.2 *En udgave af Euclids algoritme som udskriver den beregnede talrække.*

9.3. Gentagelse med do (1)

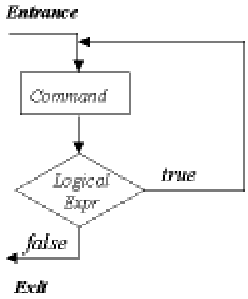
Lektion 2 - slide 23

I nogle situationer er vi interesserede i at arbejde med løkker, som sikrer mindst én gentagelse. I C er **do**-løkken af en sådan beskaffenhed. I andre sprog, ala Pascal, hedder sådanne løkker ofte **repeat until**.

En **do**-løkke sikrer altid mindst ét gennemløb af løkken

```c
do
  command
while (logicalExpression)
```

Syntaks 9.2 *Syntaksen af en do-løkke i C*



Figur 9.2 *Flow graf for en do løkke*

9.4. Gentagelse med do (2)

Lektion 2 - slide 24

Ligesom for de andre kontrolstrukturer vi har set vil vi beskrive betydningen på punktform, og vi vil se på et eksempel.

- Betydningen af **do**:
  - Kommandoen udføres
  - Det logiske udtryk beregnes
  - Hvis værdien er *true* overføres kontrollen til starten af **do**
  - Hvis værdien er *false* afsluttes do løkken

I nedenstående program viser vi en programstump, som vi kunne kalde "ask user". Programmet insisterer på et ja/nej svar, i termer af bogstavet 'y' eller 'n'. Vi antager her, for at simplificere opgaven, at brugeren kun indtaster ét tegn.

```
#include <stdio.h>

int main(void) {

  char answer, forget;

  do {
    printf("Answer yes or no (y/n): ");
    scanf("%c%c", &answer, &forget);
  } while (answer != 'y' && answer != 'n');

  printf("The answer is %s\n\n",
          answer == 'y'? "yes" : "no");

  return 0;
}
```

Program 9.3 *Et program som ønsker et 'yes/no' svar.*

Kroppen af løkken skal udføres mindst én gang. Hvis ikke svaret er tilfredsstillende skal vi udføre kroppen endnu en gang.

Bemærk at vi læser to tegn, `answer` og `forget`. `forget` introduceres for læse lineskiftet (RETURN), som er nødvendig for at få scanf til at reagere. Dette er ikke en perfekt løsning på problemet, men dog bedre end hvis kun ét tegn læses. Leg gerne selv med mulighederne.

9.5. Gentagelse med for (1)

Lektion 2 - slide 25

Ideen med den næste løkke, for-løkken, er i bund og grund at kunne kontrollere et fast og *på forhånd kendt antal gentagelser*. Det er imidlertid sådan, at for-løkker i C er mere generelle end som så.

> I mange sprog benyttes **`for`** kontrolstrukturen hvis man på forhånd kender antallet af gentagelser.
>
> **`for`** kontrolstrukturen i C kan benyttes således, men den bruges også ofte til mere generel gentagelse.

Her følger syntaksen og dernæst den punktvise betydning af for-løkken.

```
for(initExpression; continueExpression; updateExpression)
  command
```

Syntaks 9.3 *Opbygningen af en for-løkke i C.*

- Betydning af en for-løkke:
  - Trin 1: *initExpression* evalueres af hensyn til sideeffekter
  - Trin 2: *continueExpression* evalueres, og hvis den er *false* (0) er **`for`** løkken færdig
  - Trin 3: **`for`** løkkens *command* udføres
  - Trin 4: *updateExpression* evalueres, og der fortsættes med trin 2

I nedenstående eksempel udskriver vi alle tal fra og med `lower` til og med `upper`. Dette eksempel egner sig godt til en for-løkke, idet vi inden løkken starter ved at der skal bruges `upper - lower + 1` gentagelser.

```
#include <stdio.h>

int main(void) {

  int upper, lower, k;
```

```
  printf("Enter two integers, lower and upper: ");
  scanf("%d %d", &lower, &upper);

  for (k = lower; k <= upper; k++)
    printf("k = %d\n", k);

  return 0;
}
```

Program 9.4 *En typisk for løkke med et forudbestemt antal gentagelser.*

9.6. Gentagelse med for (2)

Lektion 2 - slide 26

Lad os illustrere den udvidede styrke af for-løkker i C. Altså styrken ud over at kunne kontrollere et på forhånd bestemt antal gentagelser.

> Det er muligt at styre en kompleks gentagelse som anvender flere kontrolvariable ved brug af en **`for`** løkke

Vi genprogrammerer Euclids Algoritme fra program 9.1, nu med brug af en for-løkke. Vi vil referere til de tre bestanddele af en for-løkke, jf. syntaks 9.3.

I den røde *initExpression* sørger vi for at variablene `sml` og `lg` initielt indeholder det største hhv. det mindste af de tal, som er indlæst. I den lilla *updateExpression* laver næste skridt i rækkeudviklingen, som vi beskrev i afsnit 9.2. I den blå *continueExpression* kontrollerer vi termineringen (afslutningen) af gentagelsen. Bemærk at alle tilstandsændringer og check foregår i init, continue og update delene. For løkken i program 9.5 er uden krop!

```
#include <stdio.h>

int main(void) {
  int i, j, sml, lg, rem;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  for(sml = i<=j?i:j , lg = i<=j?j:i;
      sml > 0;
      rem = lg % sml , lg = sml, sml = rem);

  printf("GCD of %d and %d is %d\n\n", i, j, lg);

  return 0;
}
```

Program 9.5 *Euclids algoritme - største fælles divisor - programmeret med en for-løkke 'uden krop'.*

Bemærk brugen af komma i de røde og blå dele. Komma er her en operator, som beregner udtryk i sekvens, og som returnerer værdien af det sidst evaluerede udtryk. Kommaoperatoren minder om semikolon, som skal placeres mellem (eller rettere efter) kommandoer i C. Læs om kommaoperatoren i afsnit 3.13 side 98 af *C by Dissection*. Kommaoperatoren har lavest mulig prioritering, jf. tabel 2.3 hvilket betyder at udtryk med alle andre operatorer end komma beregnes først.

Alle udtryk kan udelades, hvilket afstedkommer en uendelig løkke

Vi skitserer herunder situationen hvor *initExpression* og *continueExpression* begge er tomme. Dette afstedkommer en uendelig løkke - altså en løkke som ikke terminerer med mindre vi udfører et eksplicit hop ud af for-løkkens krop.

```
#include <stdio.h>

int main(void) {

  long i = 0;

  for( ; ; ++i)
    printf("%ld: Hi there...\n", i);

  return 0;
}
```

Program 9.6 *En for løkke, der optæller en variabel uendeligt.*

9.7. Break, continue og return

Lektion 2 - slide 27

Vi vil her se på mulighederne for at påvirke kontrolforløbet af løkker og visse udvælgende kontrolstrukturer.

I C kan man bruge goto og labels, som antydet i afsnit 5.2. Vi foretrækker dog mere specifikke og specialiserede hop kommandoer, som ikke kræver at vi introducerer labels. Vi ser nu på **break**, **continue** og **return**.

Anvendelse af generelle hop - goto - anses for meget dårlig programmeringsstil.

Brug af 'mere specialiserede hop' bruges oftere.

- `break`
  - anvendes til at springe ud af (den inderste) switch, while, do, eller for
- `continue`
  - anvendes til at afslutte kommandoen i en while, do eller for løkke
- `return`
  - anvendes til at afbryde, og returnere et resultat fra en funktion

Vi viser her et program, som illustrerer continue og break. Programmet indeholder en 'uendelig for-løkke'. Den røde del sørger for at gentagelser med lige *i* værdier overspringes, i det mindste således at *i* ikke skrives ud i printf kommandoen. Den blå del sørger for at afbryde løkken når værdien af *i* bliver større end 1000.

Samlet set vil program 9.7 udskrive alle ulige tal mellem 1 og 1000.

```
#include <stdio.h>

int main(void) {

  long i = 1;

  for( ; ; ++i){
    if (i % 2 == 0) continue;
    if (i > 1000) break;
    printf("%ld: Hi there...\n", i);
  }

  return 0;
}
```

Program 9.7 *Illustration af break og continue i en for løkke.*

Vi møder return kommandoen i kapitel 10, hvor emnet er funktioner.

9.8. Konvertering af do og for løkker til while

Lektion 2 - slide 28

I dette afsnit overbeviser vi os om, at både do- og for-løkker kan omskrives til whileløkker. Skabelonen for omskrivningerne ses i tabel 9.1.

Ethvert program med **`do`** og **`for`** løkker kan let omskrives til kun at bruge **`while`** løkker

Original	Transformation

`for (expr1; expr2; expr3)`
`  command`
	`expr1;`
	`while (expr2) {`
	`  command;`
	`  expr3;`
	`}`

`do`
`  command`
`while (expr)`
	`command;`
	`while (expr)`
	`  command;`

Tabel 9.1

Læg mærke til, at der for alle gentagende kontrolstrukturer gælder, at *kontrol udtrykket* er en *fortsættelsesbetingelse*.

Med andre ord afbrydes alle former slags løkker i C når kontroludtrykkets værdi bliver *false*.

10. Funktioner - Lasagne

Nøgleordet i dette og de følgende afsnit er *abstraktion*. I C udgøres den primære abstraktionsmekanisme af funktioner.

Ideen i dette kapitel er at introducere og motivere abstraktioner og funktioner via analogier fra gastronomien. Programmer kan opfattes som *opskrifter på beregninger*. Vi taler i denne sammenhæng om algoritmer. I den gastronomiske verden kender vi også til opskrifter, nemlig *opskrifter på det at lave mad*.

Vi studerer således indledningsvis en lasagne opskrift. God appetit!

10.1. Lasagne al forno

Lektion 3 - slide 2

Herunder viser vi en opskrift på lasagne al forno. Vi kan karakterisere den som en sekvens af trin, som vi antager vil kunne udføres af en *mad fortolker*. Uheldigvis har vi ikke (endnu) en automatiseret fortolker, som kan eksekvere opskriften. Vi må give den til en kyndig person (en kok, f.eks.) for hvem de enkelte trin giver mening. Vi vil her antage, at de enkelte trin er tydelige og klare for den person eller maskine, som effekturerer opskriften til en varm og lækker ret.

- Bland og ælt 500 g durum hvedemel, 5 æg, lidt salt og 2 spsk olie.
- Kør pastaen gemmen en pastamaskine i tynde baner.
- Smelt 3 spsk smør, og rør det sammen med 3 spsk mel.
- Spæd med med 5 dl mælk og kog sovsen under svag varme.
- Svits 2 hakkede løg med 500 g hakket oksekød, samt salt og pebber.
- Tilsæt 3 spsk tomatpuré, en ds. flåede tomater og 3 fed knust hvidløg.
- Kog kødsovsen 10 minutter.
- Bland nu lagvis pastabaner, kødsovs og hvid sovs. Drys med paramesanost.
- Gratiner retten i ovnen 15 minutter ved 225 grader.

Alle som har lavet lasagne vil vide, at denne ret laves af kødsovs, hvid sovs, og pasta. Vi ser det således som en svaghed, at disse *abstraktioner* ikke har fundet indpas i opskriften. Indførelse af abstraktioner vil være vores næste træk.

10.2. Struktureret lasagne

Lektion 3 - slide 3

Struktureret lasagne er nok ikke på menukortet i mange restauranter. Men strukturerede lasagneopskrifter, ala den vi viser i program 10.1 er almindelige i mange kogebøger.

I en mere struktureret opskrift indgår delopskrifter på lasagneplader, hvid sovs og kødsovs

```
Lav en dobbelt portion lasagneplader.

Lav en portion hvid sovs.

Lav en portion kødsovs.

Bland nu lagvis pastabaner, kødsovs og hvid sovs. Drys med paramesanost.

Gratiner retten i ovnen 15 minutter ved 225 grader.
```

Program 10.1 *Lasagne.*

De tre fremhævede linier aktiverer abstraktioner (delopskrifter) for hhv. det at lave lasagneplader (for vi køber dem nemlig ikke færdige i Bilka), hvid sovs, og kødsovs. Vi tænker på de sidste to trin som basale trin, der kan genkendes fra opskriften i afsnit 10.1.

Men vi skal naturligvis også beskrive, hvad det vil sige at lave plader, hvid sovs og kødsovs. Hver delopskrift *indkapsler* en række madlavningstrin. Vi viser de tre madlavningsabstraktioner i hhv. program 10.2, program 10.3 og program 10.4.

```
Bland og ælt 500 g durum hvedemel, 5 æg, lidt salt og 2 spsk olie;

Kør pastaen gemmen en pastamaskine i tynde baner;
```

Program 10.2 *Lav en dobbelt portion lasagneplader.*

```
Smelt 3 spsk smør, og rør det sammen med 3 spsk mel;

Spæd med med 5 mælk og kog sovsen under svag varme;
```

Program 10.3 *Lav en portion hvid sovs.*

```
Svits 2 hakkede løg med 500 g hakket oksekød,
samt salt og pebber;

Tilsæt 3 spsk tomatpuré, en ds. flåede tomater
og 3 fed knust hvidløg;

Kog kødsovsen 10 minutter;
```

Program 10.4 *Lav en portion kødsovs.*

Vi kunne forsøge at strække analogien lidt længere. For det første kunne vi forsøge os med en generalisering af en delopskrift til at dække forskellige retter med fælles kendetegn. En realistisk og brugbar generalisering kunne være omfanget af delretten (enkelt portion, dobbelt portion, etc.).

For det andet kunne vi forsøge os med opskrifter, som skulle bruge en given delopskrift mere end én gang. Dette er måske urealistisk i den gastronomiske verden, i det mindste relativ til denne forfatters begrænsede gastronomiske forestillingsevner.

Anvendelser af delopskrifter højner abstraktionsniveauet og muliggør genbrug af basale opskrifter.

Delopskrifter svarer til *procedurer* i programmeringssprog. Brug af parametre generaliserer en opskrift så den bliver mere alsidig.

10.3. Lasagne ala C

Lektion 3 - slide 4

Lad det være sagt klart og tydeligt allerede her - du får aldrig lasagne ud af din C compiler og fortolker. Alligevel forsøger vi os i program 10.5 med et C lignende program for den strukturerede lasagneopskrift, vi nåede frem til i afsnit 10.2.

I program 10.5 ser vi tre såkaldte funktionsprototyper lige efter include. Disse annoncerer at der er abstraktioner som hedder `make_lasagne_plates` `make_white_sauce`, og `make_meat_sauce`. Disse er nødvendige for at kunne anvende funktionerne i hovedopskriften. Så kommer *hovedprogrammet*, som altid i C hedder `main`. Efter denne ser vi implementeringen af de tre annoncerede funktioner.

At vi har forfattet program 10.5 på engelsk betyder naturligvis ikke alverden. Som regel foretrækker jeg at angive navne, kommentarer mv. i programmer på engelsk, for at opnå den bedste mulige helhed ift. selve programmeringssproget. Men det er udtryk for 'smag og behag'.

```
#include <stdio.h>

void make_lasagne_plates(void);
void make_white_sauce(void);
void make_meat_sauce(void)

int main(void) {
  make_lasagne_plates();
  make_white_sauce();
  make_meat_sauce();

  mix plates, meat sauce, and white sauce;
  sprinkle with paramesan cheese;
  bake 15 minutes at 225 degrees;

  return 0;
}

void make_lasagne_plates(void) {
  mix flour, egs, salt and oil;
  process the pasta in the pasta machine;
}

void make_white_sauce(void) {
  melt butter and stir in some flour;
  add milk and boil the sauce;
}
```

```
void make_meat_sauce(void){
  chop the onion, and add meat, salt and pebber;
  add tomatos and garlic;
  boil the sauce 10 minutes;
}
```

Program 10.5 *Et pseudo C program som laver lasagne.*

Det er hermed slut på det gastronomiske kvarter. I kapitel 11 fortsætter vi vores indledende udforskning af abstraktioner og funktioner i en meget simpel grafisk verden.

11. Tændstikgrafik abstraktioner

Tændstikgrafik anvendes her for simple stregtegninger af sammensatte figurer, f.eks. mennesker med hoved, arme, krop og ben.

11.1. Eksempel: En tændstikpige (1)

Lektion 3 - slide 6

I program 11.2 ser vi først en `main` funktion, som kalder en abstraktion, `prn_match_girl`. `prn_match_girl` udskriver konturerne af en pige i termer af hoved, arme, krop, ben ved at kalde `prn_head`, `prn_arms`, `prn_body` og `prn_legs`. Den ønskede tegning af pigen er vist herunder:

```
      *
    *   *
  *       *
  *       *
    *   *
      *
-------------
     /\
    /  \
   /    \
  /      \
 /        \
/          \
-------------
     /\
    /  \
   /    \
  /      \
 /        \
/          \
```

Program 11.1 *Det ønskede program output.*

Vi ønsker at tegne en 'tændstikpige' ved brug af simpel tegngrafik

```
int main(void){
  prn_match_girl();
  return 0;
}

void prn_match_girl(void){
  prn_head();
  prn_arms();
  prn_body();
  prn_legs();
}
```

Program 11.2 *Udskrivning af tændstikpige i termer af udskrivning af hoved, arme, krop og ben.*

Ovenstående er udtryk for en programmeringsidé som kaldes *programudvikling af trinvis forfinelse*. Vi anvender først `prn_match_girl` i `main`, uden at have denne til rådighed i forvejen. Når vi således har indset behovet for denne abstraktion, programmerer vi den som næste punkt på dagsordenen. Tilsvarende sker det med `prn_head`, `prn_arms`, `prn_body` og `prn_legs`.

Tegneprocessen udføres top-down. Det udtrykkes at en tændstikpige tegnes som et hoved, arme, krop og ben. I næste trin må vi definere, hvordan disse kropsdele tegnes.

11.2. Eksempel: En tændstikpige (2)

Lektion 3 - slide 7

Vi fortsætter den trinvise forfinelse af programmet som tegner tændstikpigen. Vi har brugt `prn_head`, `prn_arms`, `prn_body` og `prn_legs` i program 11.2, men vi har ikke lavet disse primitiver endnu. Det råder vi bod på i program 11.3.

Pigens hoved, arme, krop og ben tegnes ved kald af generelle geometriske tegne procedurer

```
void prn_head(void){
  prn_circle();
}

void prn_arms(void){
  prn_horizontal_line();
}

void prn_body(void){
  prn_reverse_v();
  prn_horizontal_line();
}

void prn_legs(void){
  prn_reverse_v();
}
```

Program 11.3 *Udskrivning af hoved, arme, krop og ben i termer af generelle geometriske figurer.*

Vi ser at et hoved tegnes som en cirkel, armene som en vandret streg, kroppen som et omvendt, lukket v, og benene som et omvendt v. Vi gør brug af tre nye primitiver, `prn_circle`, `prn_horizontal_line` og `prn_reverse_v`.

11.3. Eksempel: En tændstikpige (3)

Lektion 3 - slide 8

Der er nu tilbage at realisere tegningen af cirkler, vandrette streger og den omvendte v form. Dette gør vi på helt simpel vis i program 11.4.

Man kan spørge om vi virkelig ønsker så mange niveauer for den relativt simple overordnede tegneopgave, som vi startede i kapitel 11. Svaret er et klart ja.

Vi får for det første nedbrudt den komplekse og sammensatte tegneopgave i en række simple tegneopgaver. Generelt er det måden at beherske indviklede og komplicerede programmeringsudfordringer.

Genbrugelighed er en anden markant fordel. I tegningen af tændstikpigen bruges den omvendte V form to forskellige steder. Det er en helt åbenbar fordel at disse to steder kan deles om tegningen af denne form. Dette gøres ved at kalde `prn_reverse_v` fra både `prn_body` og `prn_legs`. Selve implementationen af `prn_reverse_v` findes naturligvis kun ét sted.

Cirkler, linier og andre former tegnes ved brug af primitiv tegngrafik

```c
#include <stdio.h>

void prn_circle(void){
  printf("      *          \n");
  printf("    *   *        \n");
  printf("  *       *      \n");
  printf("  *       *      \n");
  printf("    *   *        \n");
  printf("      *          \n");
}

void prn_horizontal_line(void){
  printf("------------    \n");
}

void prn_reverse_v(void){
  printf("     /\\         \n");
  printf("    /  \\        \n");
  printf("   /    \\       \n");
  printf("  /      \\      \n");
  printf(" /        \\     \n");
  printf("/          \\    \n");
}
```

Program 11.4 *Udskrivning af cirkler, linier og andre geometriske figurer.*

I program 11.4 ser det lidt underligt ud at der tilsyneladende udskrives en dobbelt backslash '\'. Dette skyldes at backslash er et såkaldt escape tegn, som påvirker betydningen af de efterfølgende tegn. Derfor noterer '\\' reelt et enkelt bagvendt skråstreg i en C tekststreng. Tilsvarende noterer '\n' et linieskift.

11.4. Eksempel: En tændstikpige (4)

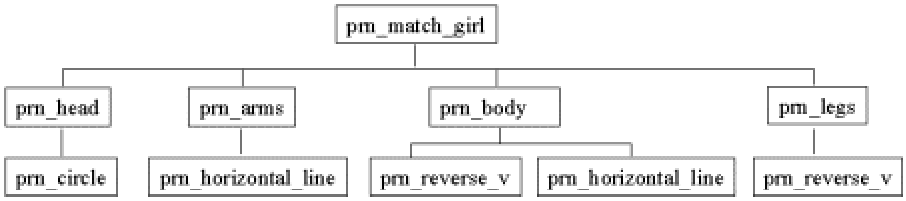
Lektion 3 - slide 9

I dette afsnit viser figur 11.1 et overblik over strukturen af de abstrationer, vi har indført i afsnit 11.1 - afsnit 11.3. Men først repeteres de essentielle observationer om den benyttede fremgangsmåde:

Programmet er lavet *top-down*.

Programmering ved *trinvis forfinelse*.

Programmet designes og implementeres i et antal niveauer med postulering og efterfølgende realisering af et antal procedurer



Figur 11.1 *En illustration of problemer og delproblemer i forbindelse med tegning af en tændstikpige*

Læg igen mærke til den hierarkiske nedbrydning og hvordan nogle abstraktioner bruges mere end ét sted.

11.5. Eksempel: En tændstikpige (5)

Lektion 3 - slide 10

I dette afsnit vil vi vise, hvordan vi konkret i C kan organisere de abstraktioner, som er præsenteret i de forrige afsnit. Som en central idé introducerer vi et separat *char graphics library*. I program 11.5 inkluderer vi dette bibliotek i den røde del. I den blå del skriver vi såkaldte funktionsprototyper for de funktioner, vi 'skubber foran os' i programmet. Dernæst følger abstraktionerne som printer hele pigen, hovedet, arme, krop og ben.

```
#include "char-graphics.h"

void prn_match_girl(void);
void prn_head(void);
void prn_arms(void);
void prn_body(void);
void prn_legs(void);

int main(void){
  prn_match_girl();
  return 0;
}

void prn_match_girl(void){
  prn_head();
  prn_arms();
  prn_body();
  prn_legs();
}

void prn_head(void){
  prn_circle();
}

void prn_arms(void){
  prn_horizontal_line();
}

void prn_body(void){
  prn_reverse_v();
  prn_horizontal_line();
```

```
}

void prn_legs(void){
  prn_reverse_v();
}
```

Program 11.5 *Tændstikpige programmet.*

I program 11.6 viser vi den såkaldte header fil for char graphics biblioteket. Det er denne fil vi inkluderede i den røde del af program 11.5.

```
/* Very simple graphical character-based library  */


/* Print a circle */
void prn_circle(void);

/* Print a horizontal line */
void prn_horizontal_line(void);

/* Print a reverse v shape */
void prn_reverse_v(void);
```

Program 11.6 *Filen char-graphics.h - header fil med prototyper af de grafiske funktioner.*

Herunder, i program 11.7 ser vi selve implementationen af det grafiske bibliotek. For hver funktionsprototype i program 11.6 har vi en fuldt implementeret funktion i program 11.7.

```
#include <stdio.h>

void prn_circle(void){
  printf("      *          \n");
  printf("    *   *        \n");
  printf("  *       *      \n");
  printf("  *       *      \n");
  printf("    *   *        \n");
  printf("      *          \n");
}

void prn_horizontal_line(void){
  printf("------------    \n");
}

void prn_reverse_v(void){
  printf("     /\\         \n");
  printf("    /  \\        \n");
  printf("   /    \\       \n");
  printf("  /      \\      \n");
  printf(" /        \\     \n");
  printf("/          \\    \n");
}
```

Program 11.7 *Filen char-graphics.c - implementationen af de grafiske funktioner.*

Endelig viser vi herunder hvordan vi først kan oversætte det grafiske bibliotek, og dernæst programmet som tegner tændstikpigen. Læg mærke til de forskellige compiler options, som giver udvidede fejlcheck.

```
* Compilation of the char-graphics.c library:

   gcc -ansi -pedantic -Wall -O -c char-graphics.c

* Compilation of the match-girl.c program:

   gcc -ansi -pedantic -Wall -O match-girl.c char-graphics.o
```

Program 11.8 *Oversættelse af programmerne - med mange warnings.*

Her følger nogle overordnede observationer af vores programorganisering:

> Funktioner skal erklæres før brug.
>
> Genbrugelige funktioner organiseres i separat oversatte filer.
>
> Prototyper af sådanne funktioner skrives i såkaldte header filer.

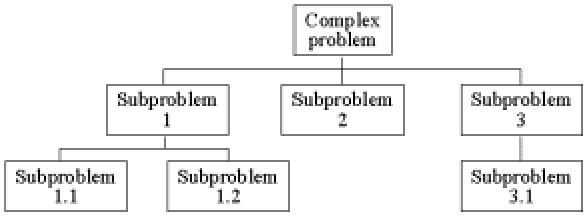
11.6. Del og hersk

Lektion 3 - slide 11

Efter at have overlevet det forholdsvis lange eksempel i de forrige afsnit er det nu tid til at gøre status.

Vi taler om *del og hersk problemløsning*. Ved at dele problemet op i mindre delproblemer hersker og behersker vi mange af de komplikationer, som vi støder på. Evnen til at dele problemer i mindre problemer er meget vigtig. Det er naturligvis tilfældet når vi prøver kræfter på store programmeringsopgaver. Men det gælder faktisk også ved løsning af de småopgaver, vi møder ved øvelserne i dette kursus. Jeg vil kraftigt opfordre alle til at tænke på dette næste gang I skriver et program!

> Komplekse problemer kan ofte opdeles i et antal simplere delproblemer.
>
> Delproblemernes løsninger kan ofte sammensættes til en løsning på det oprindelige problem.



Figur 11.2 *Opdelning af et kompleks problem i delproblemer*

- Del og hersk som top down programmering ved trinvis forfinelse:
  - Hvert delproblem P løses i en procedure
  - Procedurer, som er løsninger på delproblemer af P, placeres efter proceduren som løser P
    - Kræver erklæring af funktionsprototyper i starten af programmet
  - I kroppen af proceduren, som løser problemet P, programmeres en sammensætning af delproblemernes løsning

I næste kapitel vil vi se nærmere på funktionsbegrebet, herunder naturligvis funktioner i programmeringssproget C.

12. Procedurer og funktioner

Vi ser i dette kapitel på procedurer og funktioner generelt, og på funktionsbegrebet i C som dækker begge af disse.

12.1. Procedurer og funktioner

Lektion 3 - slide 13

Her følger kort vore definitioner af en procedure og en funktion:

En *procedure* er en abstraktion over en sekvens af kommandoer

En *funktion* er en abstraktion over et udtryk

Indkapsling er et helt basalt element i både procedurer og funktioner.

Et antal kommandoer sammensættes (aggregeres) i en procedure, og disse kan efterfølgende benyttes som én kommando i et procedurekald. Sammensætningen sker via en blok, som vi så på i kapitel 7.

I en funktion indkapsles ét udtryk, og indkapslingen spiller selv rollen som et udtryk, i et funktionskald.

- Procedure
  - En proceduredefinition indkapsler et antal kommandoer
  - Kaldet af en procedure er selv en kommando
- Funktion
  - En funktionsdefinition indkapsler netop ét udtryk
  - Kaldet af en funktion er selv et udtryk

Genbrugelighed og generalitet fremmes ved brug af parametre i funktioner. Vi ser konkrete eksempler på dette i afsnit 12.4 og de efterfølgende afsnit. Selve parametermekanismen diskuteres i afsnit 12.7.

12.2. Funktionsdefinitioner i C

Lektion 3 - slide 14

Både procedurer og funktioner, som defineret i afsnit 12.1 kaldes funktioner i C.

> C skelner ikke skarpt mellem procedurer og funktioner
>
> I C kaldes begge abstraktionsformer funktioner

Her følger den essentielle syntaks for en funktion i C:

```
type function_name (formal_parameter_list) {
  declarations
  commands
}
```

Syntaks 12.1 *En funktionsdefinition i C*

De *formelle parametre* er de parametre, som findes i selve funktionsdefinitionen. Når vi senere kalder funktionen angives *aktuelle parametre*, som erstatter de formelle parametre på en måde, som vi vil se nærmere på i forbindelse med eksemplerne i afsnit 12.4 - afsnit 12.7.

- Karakteristika:
  - Procedurer angiver *type* som **`void`**
  - Funktioner angiver *type* som en kendt datatype, f.eks. **`int`**, **`char`**, eller **`double`**
  - I procedurer og funktioner uden parametre angives *formal_parameter_list* som **`void`**

Mange C funktioner er reelt procedurer, som returnerer en eller anden værdi. Den returnerede værdi anvendes ofte til at signalere, om de indkapslede kommandoer 'gik godt' eller 'gik skidt'. I denne sammenhæng betyder returværdien 0 ofte, at alt er OK, medens en ikke-nulværdi som resultat er tegn på en eller anden fejl.

12.3. Funktionskald i C

Lektion 3 - slide 15

Når man kalder en funktion aktiveres kommandoerne i kroppen af funktionen på det vi kalder de aktuelle parametre. Et kald af funktionen står altså for de indkapslede kommandoer, hvor de aktuelle parametres værdier erstatter de formelle parametre. Dette kalder vi parameteroverførsel, og vi beskrive dette nærmere i afsnit 12.7.

```
function_name (actual_parameter_list)
```

Syntaks 12.2 *Et funktionskald i C*

- Karakteristika:
  - En funktion kan ikke kaldes før den er erklæret
  - Antallet af aktuelle parametre i kaldet skal modsvare antallet af formelle parametre i definitionen
  - En aktuel parameter er et udtryk, som beregnes inden den overføres og bindes til den tilsvarende formelle parameter

12.4. Eksempel: Temperaturomregninger

Lektion 3 - slide 16

Vi ser først på et eksempel på en ægte funktion. Hermed mener vi altså en funktion i den forstand vi definerede i afsnit 12.1, og som modsvarer det matematiske funktionsbegreb.

Funktionen `fahrenheit_temperature` tager som input en celcius temperatur, f.eks. 100 grader (kogepunktet ved normale forhold). Med dette output returnerer funktionen den tilsvarende Fahrenheit temperatur, 212 grader. Dette er den normale temperaturangivelse i Amerikanske lande, f.eks. USA.

Der udføres ingen kommandoer i C funktionen `fahrenheit_temperature`. Funktionen dækker med andre ord blot over en *formel*. Vi kalder ofte en formel for et *udtryk*, jf. afsnit 2.1.

```c
#include <stdio.h>

double fahrenheit_temperature(double celcius_temp){
  return (9.0 / 5.0) * celcius_temp + 32.0;
}

int main(void){

  printf("Freezing point: %6.2lf F.\n",
         fahrenheit_temperature(0.0));

  printf("Boiling point: %6.2lf F.\n",
         fahrenheit_temperature(100.0));

  return 0;
}
```

Program 12.1 *Et program som omregner frysepunkt og kogepunkt til Fahrenheit grader.*

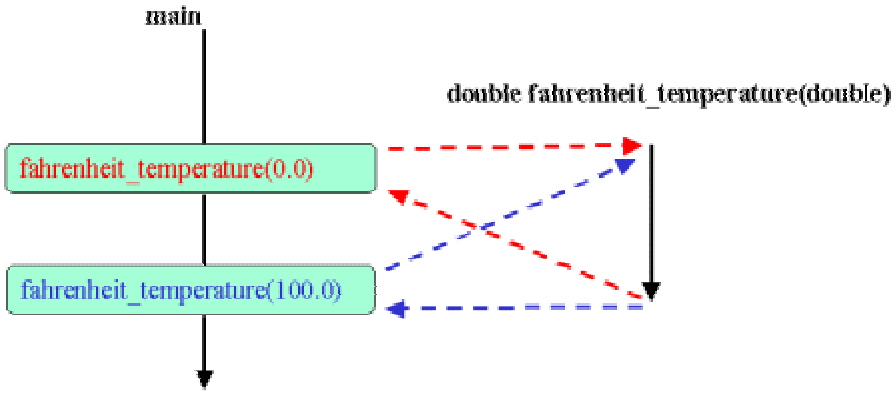
I programmet ser vi øverst, med fed sort skrift definitionen af `fahrenheit_temperature`. Bemærk udtrykket `(9.0 / 5.0) * celcius_temp + 32.0` , hvis værdi returneres fra funktionen med `return`. Bemærk også anvendelsen af navnet `celcius_temp`; Dette er den formelle parameter af funktionen.

I `main` ser vi to kald af `fahrenheit_temperature`. Dette er de røde og blå dele af program 12.1. Bemærk at disse dele er udtryk, og at de forekommer som parametre til `printf`.

Herunder, i figur 12.1 illustrerer vi det røde og blå kald af `fahrenheit_temperature` i program 12.1.

Kontrollen i `main` forløber som følger (følg pilene):

1. Indgang til `main` (lodret sort pil).
2. Det røde kald af `fahrenheit_temperature` (vandret rød pil til højre).
3. Første udførelse af `fahrenheit_temperature` (lodret sort pil).
4. Den røde returnering fra `fahrenheit_temperature` (opadrettet rød venstre pil).
5. Det blå kald af `fahrenheit_temperature` (vandret blå pil til højre).
6. Anden udførelse af `fahrenheit_temperature` (lodret sort pil).
7. Den blå returnering fra `fahrenheit_temperature` (blå pil mod venstre).
8. Udgang af `main` (lodret sort pil).



Figur 12.1 *To kald af fahrenheit_temperature funktionen*

Herunder, i program 12.2 viser vi et eksempel på brug af den omvendte funktion, som konverterer fahrenheit grader til celcius grader. Konkret programmerer vi en nyttig temperatur konverteringstabel. De første linier i udskriften er vist i program 12.3.

```
#include <stdio.h>

double celcius_temperature(double fahrenheit_temp){
  return (5.0 / 9.0) * (fahrenheit_temp - 32.0);
}

int main(void){
  double f;

  printf("%-20s %-20s\n", "Fahrenheit degrees", "Celcius degrees");

  for (f = 1.0; f <= 100.0; f++)
    printf("%-20.2lf %-20.2lf\n", f, celcius_temperature(f));

  return 0;
}
```

Program 12.2 *Et program som udskriver en nyttig temperaturkonverteringstabel.*

Bemærk kontrolstrengen i printf i program 12.2, såsom `%-20.2lf`. `lf` benyttes idet vi udskriver en double. `20.2` angiver antallet af cifre og antallet af cifre efter decimal point. Endelig angiver tegnet `-` and vi ønsker venstrestilling af output i feltet.

```
1.00                 -17.22
2.00                 -16.67
3.00                 -16.11
4.00                 -15.56
5.00                 -15.00
6.00                 -14.44
7.00                 -13.89
8.00                 -13.33
9.00                 -12.78
...                  ...
```

Program 12.3 *Partiel output fra ovenstående program.*

12.5. Eksempel: Antal dage i en måned

Lektion 3 - slide 17

Eksemplet i dette afsnit er en fortsættelse af eksemplet fra program 8.4. Bidraget i dette afsnit er en pæn indpakning af de relevante programdele i en funktion, og en lettere omskrivning, som går på at vi nu tager udgangspunkt i et månedsnummer og et årstal.

Bemærk lige kontrasten mellem program 8.4 og program 12.4. I det første program var vi i stand til at beregne antallet af dage i en måned som en del af `main`. I det sidste program har vi en veldefineret byggeklods, funktionen `daysInMonth`, som kan bruges så ofte vi vil. Dette er en meget mere modulær løsning. En løsning der på alle måder er mere tilfredsstillende.

Vi introducerer en funktion i programmet der beregner antal dage i en måned Vi bruger eksemplet til at diskutere overførsel af parametre til funktioner

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

int daysInMonth(int mth, int yr);
int isLeapYear(int yr);

int main(void) {

  int mth, yr;

  do{
    printf("Enter a month - a number between 1 and 12: ");
    scanf("%d", &mth);
    printf("Enter a year: ");
    scanf("%d", &yr);

    printf("There are %d days in month %d in year %d\n",
              daysInMonth(mth, yr), mth, yr);
  } while (yr != 0);

  return 0;
}

int daysInMonth(int month, int year){
  int numberOfDays;
  switch(month){
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      numberOfDays = 31; break;
    case 4: case 6: case 9: case 11:
      numberOfDays = 30; break;
    case 2:
      if (isLeapYear(year)) numberOfDays = 29;
      else numberOfDays = 28; break;
    default: exit(-1);  break;
  }
```

```
  return numberOfDays;
}

int isLeapYear(int year){
  int res;
  if (year % 400 == 0)
    res = 1;
  else if (year % 100 == 0)
    res = 0;
  else if (year % 4 == 0)
    res = 1;
  else res = 0;
  return res;
}
```

Program 12.4 *Et program der beregner antal dage i en måned med en funktion.*

I program 12.4 bemærker vi først, at vi har prototyper af funktionerne `daysInMonth` og `isLeapYear` (fremhævet med fed sort skrift). Dernæst følger `main`, som kalder `daysInMonth` (den blå del). Endelig følger definitionen af `daysInMonth` med rød skrift. Som input tager funktionen parametrene `month` og `year`, og som output returneres antallet af dage i den ønskede måned. Læg mærke til denne meget veldefinerede grænseflade mellem funktionen `daysInMonth` og dets omverden.

Vi har øget genbrugeligheden - og dermed 'værdien' af vort program

Vi ønsker at undgå brug af `scanf` og `printf` i genbrugelige procedurer og funktioner

12.6. Eksempel: GCD

Lektion 3 - slide 18

Vi vender tilbage til et andet eksempel fra kapitel 9, nemlig Euclids algoritme, som vi oprindelig mødte i program 9.1. Vi husker at Euclids algoritme finder den største fælles divisor af to tal. Altså det største tal, som både går op i det ene og det andet tal.

Mønstret fra dage-i-måned eksemplet, program 12.4, genfindes klart i program 12.5. Læg mærke til funktionen `gcd`'s grænseflader. På inputsiden er der helt naturligt to heltal, hvoraf vi ønsker at finde den største fælles divisor. På outputsiden resultatet, altså den største fælles divisor.

```c
#include <stdio.h>

int gcd(int, int);

int main(void) {
  int i, j, small, large;

  printf("Enter two positive integers: ");
  scanf("%d %d", &i, &j);

  small = i <= j ? i : j;
  large = i <= j ? j : i;

  printf("GCD of %d and %d is %d\n\n", i, j, gcd(large, small));

  return 0;
}

int gcd(int large, int small){
  int remainder;
  while (small > 0){
    remainder = large % small;
    large = small;
    small = remainder;
  }
  return large;
}
```

Program 12.5 *Et program der beregner største fælles divisor med en funktion.*

12.7. Parametre til C funktioner

Lektion 3 - slide 19

Som afslutning på dette afsnit ser vi lidt mere systematisk på parametre til funktioner i C.

Vi har mødt eksempler på parametre i både program 12.1, program 12.4 og program 12.5. I dette afsnit ser vi på den faktiske mekanisme til parameteroverførsel i C. Dette er de såkaldte værdiparametre, som på engelsk ofte betegnes som "call by value".

Parametre til en funktion bruges til at gøre funktionen mere generelt anvendelig

En parameter, som optræder i en funktionsdefinition, kaldes en *formel parameter*. En formel parameter er et navn.

En parameter, som optræder i et kald af en funktion, kaldes en *aktuel parameter*. En aktuel parameter er et udtryk, som beregnes inden det overføres.

- Regler for parameteroverførsel:
  - Antallet af aktuelle parametre i kaldet skal modsvare antallet af formelle parametre i definitionen
  - Typen af en aktuel parameter skal modsvare den angivne type af hver formel parameter
    - Dog muligheder for visse implicitte typekonverteringer
  - Parametre til funktioner i C overføres som *værdiparametre*
    - Der overføres en kopi af den aktuelle parameters værdi
    - Kopien bindes til (assignes til) det formelle parameternavn
    - Når funktionen returnerer ophører eksistensen af de formelle parametre, på samme måde som lokale variable i funktionen.

13. Eksempel: Rodsøgning

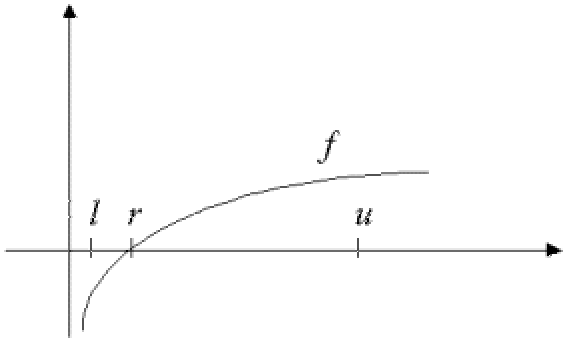
Vi gennemgår her et lidt mere avanceret eksempel, nemlig rodsøgning i en kontinuert funktion. Vores primære interesse er naturligvis problemløsning og funktioner, herunder topdown udvikling og programmering ved trinvis forfinelse.

13.1. Rodsøgning (1)

Lektion 3 - slide 21

Vi starter med den essentielle observation, der danner baggrund for den centrale rodsøgningsalgoritme:

Enhver kontinuert funktion *f*, som har en negativ værdi i *l* og en positiv værdi i *u* (eller omvendt) vil have mindst én rod *r* mellem *l* og *u*.



Figur 13.1 *En kontinuert funktion f med en rod mellem l og u*

I figur 13.1 ser vi tydeligt at der skal være mindst én rod mellem *l* og *u*, nemlig i *r*. Principielt kan der være flere. Hvis dette er tilfældet har vi ingen egentlig kontrol over, hvilken én vi finder.

Ved at indsnævre intervallet [*l*, *u*], og ved hele tiden at holde roden mellem *l* og *u*, kan vi hurtigt finde frem til en værdi *r* for hvilken *f(r) = 0*.

13.2. Rodsøgning (2)

Lektion 3 - slide 22

Den central algoritme, som finder en rod mellem *l* og *u* er vist i program 13.1. Bemærk, at vi forudsætter at *l* er mindre end (eller lig med) *u*, og at fortegnet af funktionens værdi i *l* og *u* er forskellige. Med andre ord skal situationen være som i figur figur 13.1.

```
double findRootBetween(double l, double u){
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 return middleOf(l,u);
}
```

Program 13.1 *Rodsøgningsfunktionen.*

I programmet flytter vi gentagne gange enten `l` eller `u` til midtpunktet . Bemærk her, at `l` og `u` er parametrene i funktionen `findRootBetween`, og når vi ændrer på `l` eller `u` er det kun den lokale værdi i funktionen der ændres. Dette er en konsekvens af brug af værdiparametre (se afsnit 12.7).

De røde, blå og lilla dele af `findRootBetween` i program 13.1 skal nu programmeres. I alle tre tilfælde er der tale om småfunktioner, der medvirker til at højne abstraktionsniveauet i `findRootBetween`. I program 13.2 viser vi mulige definitionen af disse funktioner.

```
int sameSign(double x, double y){
  return x * y >= 0.0;
}

double middleOf(double x, double y){
  return x + (y - x)/2;
}

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}
```

Program 13.2 *Funktionerne sameSign, middleOf og isSmallNumber.*

Vi kan nu samle alle dele i ét program. Alternativt kunne vi have organiseret 'stumperne' i to eller flere kildefiler, ligesom vi gjorde i tændstikpige eksemplet i afsnit 11.5.

Vi bemærker, at vi i `main` programmerer en løkke, som gentagne gange beder brugeren af programmet om intervaller, hvor vi søger efter rødder i funktionen $x^3 - x^2 - 41\,x + 105$. Som antydet ved vi at der er rødder i 5.0, 3.0 og -7.0. Det skyldes at $x^3 - x^2 - 41\,x + 105$ faktisk bare er lig med $(x - 5.0) \cdot (x - 3.0) \cdot (x + 7.0)$. Prøvekør selv programmet, og find rødderne!

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

double f (double x){
  /* (x - 5.0) * (x - 3.0) * (x + 7.0) */
  return (x*x*x - x*x - 41.0 * x + 105.0);
}

int sameSign(double x, double y){
  return x * y >= 0.0;
}

double middleOf(double x, double y){
  return x + (y - x)/2;
}

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}

double findRootBetween(double l, double u){
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 return middleOf(l,u);
}

int main (void){
    double x, y;
    int numbers;

    do{
      printf("%s","Find a ROOT between which numbers: ");
      numbers = scanf("%lf%lf", &x, &y);

      if (numbers == 2 && !sameSign(f(x),f(y))){
          double solution = findRootBetween(x,y);
          printf("\nThere is a root in %lf\n", solution);
        }
      else if (numbers == 2 && sameSign(f(x),f(y)))
          printf("\nf must have different signs in %lf and %lf\n",
                  x, y);
      else if (numbers != 2)
          printf("\nBye\n\n");
    }
    while (numbers == 2);
}
```

Program 13.3 *Hele rodsøgningsprogrammet.*

14. Rekursive funktioner

En rekursiv funktion er kendetegnet ved, at den kalder sig selv. Men som det mest væsentlige, er rekursive funktioner udtryk for en problemløsning, hvor delproblemerne ligner det overordnede problem.

Vi introducerer rekursion i dette kapitel. I en senere lektion, startende i kapitel 32, vender vi tilbage til rekursion med fornyet styrke.

14.1. Rekursive funktioner

Lektion 3 - slide 24

Først gør vi os de essentielle iagttagelser om rekursion, problemer som har rekursivt forekommende delproblemer:

> En rekursiv funktion kalder sig selv
>
> Rekursive funktioner er nyttige når et problem kan opdeles i delproblemer, hvoraf nogle har samme natur som problemet selv

Som det første eksempel programmerer vi fakultetsfunktionen. Hvis vi af bekvemmelighed her kalder fakultetsfunktionen for *f*, gælder at $f(n) = n \cdot (n-1) \cdot (n-2) \cdot \ldots 2 \cdot 1$. Dette kan også udtrykkes rekursivt som følger:

- $f(n) = n \cdot f(n-1)$ for $n > 0$
- $f(0) = 1$

Denne rekursive formel er udtrykt direkte i C funktionen `factorial` i program 14.1. Det rekursive kald er fremhævet med rødt.

```
#include <stdio.h>

unsigned long factorial(unsigned long n){
  if (n == 0)
    return 1;
  else return n * factorial(n - 1);
}

int main(void) {
  unsigned long k;

  for (k = 1; k <= 12; k++)
    printf("%-20lu %20lu\n", k, factorial(k));

  return 0;
}
```

Program 14.1 *Et program med en rekursivt defineret fakultetsfunktion.*

Som de fleste nok ved i forvejen vokser fakultetsfunktionen ganske voldsomt. Så voldsomt at vi højst kan beregne `factorial(12)`, selv med tal i typen `long`. Vi viser outputtet af program 14.1 i program 14.2.

```
1                                       1
2                                       2
3                                       6
4                                      24
5                                     120
6                                     720
7                                    5040
8                                   40320
9                                  362880
10                                3628800
11                               39916800
12                              479001600
```

Program 14.2 *Output fra ovenstående program.*

Som endnu et eksempel viser vi en rekursiv udgave af rodsøgningsfunktionen `findRootBetween` i program 14.3. Vi mødte den iterative (gentagende) udgave i program 13.1. Gentagelsen blev i dette program lavet med en while løkke.

```
double findRootBetween(double l, double u){
  if (isSmallNumber(f(middleOf(l,u))))
     return middleOf(l,u);
  else if (sameSign(f(middleOf(l,u)), f(u)))
     return findRootBetween(l, middleOf(l,u));
  else if (!sameSign(f(middleOf(l,u)), f(u)))
     return findRootBetween(middleOf(l,u),u);
  else exit (-1);
}
```

Program 14.3 *En rekursiv version af funktionen findRootBetween.*

Vi lægger mærke til, at der ikke anvendes nogen form for løkker i `findRootBetween` som programmeret i program 14.3. Gentagelsen bestyres af at funktionen kalder sig selv et tilstrækkeligt antal gange, indtil vi er tæt nok på roden.

15. Korrekthed af programmer

Vi slutter denne lektion af med et kapitel om programkorrekthed.

15.1. Assertions

Lektion 3 - slide 26

Ved brug af logiske udtryk kan vi ofte få et program til at checke sig selv. Vi kalder ofte sådanne udtryk for *assertions*. Hvis et assertion beregnes til *false* (altså 0 i C) stoppes programmet med en informativ besked om, hvad der er galt.

Det er muligt at 'dekorere' et program med logiske udtryk (assertions), som fortæller os om programmet opfører sig som vi forventer

- To specielt interessante assertions:
  - **Præbetingelse af en funktion**: Fortæller om det giver mening at kalde funktionen
  - **Postbetingelse af en funktion**: Fortæller om funktionen returnerer et korrekt resultat

Vi illustrerer ideen om assertions med en forkert og en korrekt implementation af en kvadratrodsfunktion. Først, i program 15.2, en underlig og meget forkert implementation af my_sqrt som blot returnerer konstanten 7.3 uanset input.

Den røde prebetingelse i program 15.1 udtrykker, at parameteren til `my_sqrt` skal være ikke-negativ. Den blå postbetingelse udtrykker, at kvadratet af resultatet, altså res^2, skal være meget tæt på det oprindelige input `x`. Vi kan ikke forvente at ramme `x` eksakt, idet der jo altid vil være risiko for afrundingsfejl når vi arbejder med reelle tal på en computer.

Med den viste implementation af `my_sqrt` vil postbetingelsen (den blå assertions) fejle for de fleste input.

```
#include <stdio.h>
#include <math.h>
/* #define NDEBUG 1 */
#include <assert.h>

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}

double my_sqrt(double x){
  double res;
  assert(x >= 0);
  res = 7.3;
  assert(isSmallNumber(res*res - x));
  return res;
}

int main(void) {

  printf("my_sqrt(15.0): %lf\n", my_sqrt(15.0));
  printf("my_sqrt(20.0): %lf\n", my_sqrt(20.0));
  printf("my_sqrt(2.0): %lf\n", my_sqrt(2.0));
  printf("my_sqrt(16.0): %lf\n", my_sqrt(16.0));
  printf("my_sqrt(-3.0): %lf\n", my_sqrt(-3.0));

  return 0;
}
```

Program 15.1 *En forkert implementation af my_sqrt.*

Den udkommenterede konstant `NDEBUG` kan bruges til at styre, om assertions rent faktisk efterchecks. Hvis kommentarerne fjernes, vil assertions ikke blive beregnet.

Lad os også vise en korrekt implementation af en kvadratrodsfunktion. Vi benytter rodsøgningsprogrammet fra afsnit 13.2 til formålet. Den centrale observation er, at vi kan finde kvadratroden af k ved at finde en rod i funktionen $f(x) = x^2 - k$ i intervallet 0 og k. Det er nøjagtigt hvad vi gør i program 15.2 .

Bemærk de røde og blå dele i program 15.2, som er hhv. prebetingelse og postbetingelse af `my_sqrt`, ganske som i den forkerte udgave af kvadratrodsfunktionen i program 15.1. Den lilla del er funktionen `f`, som vi diskuterede ovenfor.

```
#include <stdio.h>
#include <math.h>
/* #define NDEBUG 1 */
#include <assert.h>

int isSmallNumber(double x){
  return (fabs(x) < 0.0000001);
}

double displacement;

double f (double x){
  return (x * x - displacement);
}

int sameSign(double x, double y){
  return x * y > 0.0;
}

double middleOf(double x, double y){
  return x + (y - x)/2;
}

double findRootBetween(double l, double u){
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 return middleOf(l,u);
}


double my_sqrt(double x){
  double res;
  assert(x >= 0);
  displacement = x;
  res = findRootBetween(0,x);
  assert(isSmallNumber(res*res - x));
  return res;
}

int main(void) {
```

```
  printf("my_sqrt(15.0): %lf\n", my_sqrt(15.0));
  printf("my_sqrt(20.0): %lf\n", my_sqrt(20.0));
  printf("my_sqrt(2.0): %lf\n", my_sqrt(2.0));
  printf("my_sqrt(16.0): %lf\n", my_sqrt(16.0));

  return 0;
}
```

Program 15.2 *En brugbar implementation af my_sqrt implementeret via rodsøgningsfunktionen.*

Vi kan også illustrere assertions i selve rodsøgningsfunktionen `findRootBetween`, som oprindelig blev vist i program 13.1. Dette er illustreret i program 15.3 herunder.

I den røde prebetingelse af `findRootBetween` udtrykker vi at `l` skal være mindre end eller lige med `u` og at fortegnet af `f(l)` og `f(u)` skal være forskellige. I den blå postbetingelse kræver vi at `f(res)` er meget tæt på 0. Dette udtrykker at funktionen `findRootBetween` er korrekt på de givne input.

```
double findRootBetween(double l, double u){
 double res;
 assert(l <= u); assert(!sameSign(f(l), f(u)));
 while (!isSmallNumber(f(middleOf(l,u)))){
   if(sameSign(f(middleOf(l,u)), f(u)))
     u = middleOf(l,u);
   else
     l = middleOf(l,u);
 }
 res = middleOf(l,u);
 assert(isSmallNumber(f(res)));
 return res;
}
```

Program 15.3 *En udgave af findRootBetween med præbetingelse og postbetingelse.*

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)

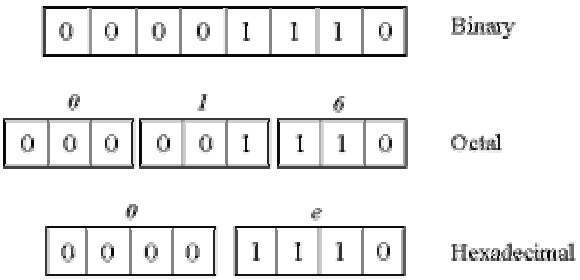
Lektion 4 - slide 6

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.

17. Typer

I lektionen, som starter med dette kapitel, studerer vi typer i C. Vi har allerede ved flere lejligheder set på typer. I denne lektion vil vi dog være mere systematiske end vi hidtil har været. Vi vil også møde flere nye ting, som har med typer at gøre i C.

17.1. Typer

Lektion 5 - slide 2

Følgende definition fortæller i bund og grund, at en type er en mængde af værdier. Så simpelt er det faktisk.

En *type* er en mængde af værdier med fælles egenskaber

Der er følgende tre gode årsager til at skrive programmer i programmeringssprog, som understøtter typer.

- Typer forbedrer programmets læsbarhed og forståelighed
- Typer gør det muligt for compileren at generere bedre og mere effektiv kode
- Typer gør det muligt at opdage fejl i programmet - typisk under oversættelsen

Nedenstående program er tænkt som illustration af det første og det sidste af disse punkter.

```
#include <stdlib.h>

double f (double p1, int p2){
  if (p2 == 0)
    return p1 * 2.0;
  else
    return p1 - p2;
}

int main(void){
  int i = 1, j = 5;
  double d = 2.0, e = 6.0;

  printf("The first result is %lf\n", f(e-d, i+j));
  printf("The second result is %lf\n", f(d-e, 0));

  return 0;
}
```

Program 17.1 *Et C program med typer.*

Først læsbarhed: Vi kan se, at funktionen `f` altid vil få overført to tal, nemlig først en double (`p1`) og en `int` (`p2`). Vi kan også se at funktionen altid vil returnere en værdi i typen `double`. Denne information er nyttig, når vi skal læse og forstå funktionen `f`.

Alle *udtryk* har en type, som kan bestemmes allerede på det tidspunkt at programmet oversættes. I program program 17.1 er typen af `e-d` `double`, `i+j` er `int`, og typen af `f(e-d, i+j)` er `double`. Hvis ikke disse typer passer ind, det sted, hvor udtrykket er placeret, vil compileren udskrive en fejlbesked.

Hvis vi kalder `f` som `f(i+j, e-d)` har vi principielt en fejlsituation, idet typen af første parameter er et heltal (`int`), og typen af den anden er et reelt tal (`double`). Det er imidlertid således, at C compileren implicit konverterer `i+j` til en `double`, og `e-d` til en `int`. Disse implicitte konverteringer vil vi diskutere i afsnit 19.1.

Det skulle være let at verificere, at programmet ovenfor giver output, som det fremgår herunder.

```
The first result is -2.000000
The second result is -8.000000
```

Program 17.2 *Output fra ovenstående program.*

> En type udgør en klassificering af data som fortæller compilereren hvordan programmøren ønsker at *fortolke data*

Ud over den fortolkning, som typerne bidrager med, finder der yderlig en type fortolkning sted i C. I de to `printf` kald i program 17.1 fortæller konverterings tegnene **`%lf`** at værdien skal fortolkes som en 'long float', hvilket vil sige en `double`. Hvis vi i stedet skrev **`%i`** vil vi få udskrevet et helt andet resultat, som fortolker bitmønstret i de udskrevne udtryks værdier som heltal.

17.2. Typer i C

Lektion 5 - slide 3

Vi opdager ikke så mange typefejl på 'compile time' som vi kunne have ønsket os, når vi programmerer i C. C er meget villig til at omfortolke værdier fra én type til værdier i en anden type. Vi tænker på dette som *løshed* i typesystemet i C.

> ANSI C har en relativ løs skelnen mellem værdier i forskellige typer

- Eksempler på løshed i C's typesystem:
  - Typen boolean er indlejret i de numeriske typer
  - Typen **`char`** er reelt et heltalsinterval
  - Enumeration typer er reelt heltalstyper
    - Vi møder enumeration typer senere i denne lektion

Det traditionelle C sprog har en endnu mere løs omgang med typer end ANSI C

18. Fundamentale C datatyper

I dette kapitel ser vi primært på de fundamentale taltyper. Vi vil også se hvordan taltyperne passer ind i en lidt større sammenhæng. Endvidere vil vi også studere enumeration typer i dette kapitel.

18.1. Oversigt over typer i C

Lektion 5 - slide 5

Herunder giver vi et bud på en hierarkisk klassificering af typerne i C.

- Typer i C
  - Typen **`void`** svarende til den tomme mængde uden værdier.
  - Skalar typer
    - Aritmetiske typer
      - Heltalstyper (integral types)
        - **`short`**, **`int`**, **`long`**, **`char`**
        - enumeration typer
      - Floating-point typer
        - **`float`**, **`double`**, **`long double`**
    - Pointer typer
  - Sammensatte typer (aggregerede typer).
    - Array typer
    - Record typer (**`struct`**ure types)

På det øverste niveau skelner vi mellem skalartyper og sammensatte typer. Endvidere har vi også typen `void` på dette niveau. Typen `void` benyttes ofte når vi vil signalere 'ingen værdi', f.eks. når en funktion ikke tager parametre eller ikke returnerer nogen værdi.

En 'skalar' (engelsk: 'scalar') er en værdi karakteriseret udelukkende ved sin størrelse (having only magnitude). Skalartyperne består af taltyperne, som vi har meget at sige om i resten af dette kapitel. Pointertyper klassificeres også som skalartyper, og de introduceres i kapitel 22.

Vi har endnu ikke set på de sammensatte typer, arrays og records (structs). Arrays behandles mere dybdegående i kapitel 24. Records er et af de sene emner i dette kursus. Vi møder det først i kapitel 39.

18.2. Heltalstyper

Lektion 5 - slide 6

Vi har måske ikke så meget nyt at sige om heltal på dette tidspunkt. Vi vil dog pege på, at heltal findes i tre længder: **`int`**, **`long int`** og **`short int`**. Endvidere findes både signed og unsigned udgaver af disse. På jævn dansk betyder dette med og uden fortegn. Dette kan kombineres til seks

forskellige typer, som vi viser i tabel 18.1. Tabellen medtager også typen **`char`**, som vi allerede i kapitel 16 indså blot repræsenteres som heltal.

Type	Kort typenavn	Suffix	`printf` conv. tegn	`scanf` conv. tegn	Eksempel
`signed int`	`int`	*intet*	`%d` eller `%i`	`%d` eller `%i`	*-123*
`unsigned int`	`unsigned`	u eller U	`%u`	`%u`	*123U*
`signed long int`	`long`	l eller L	`%li`	`%li`	*-123456L*
`unsigned long int`	`unsigned long`	lu eller LU	`%lu`	`%lu`	*123456LU*
`signed short int`	`short`	*intet*	`%hi`	`%hi`	*-12*
`unsigned short int`	`unsigned short`	*intet*	`%hu`	`%hu`	*12U*
`char`	`char`	-	`%c`	`%c`	*'a'* eller *97*

Tabel 18.1

Tabellen ovenfor viser fuld og kort typenavn samt suffix tegn så vi kan typebestemme (de fleste) talkonstanter. Vi viser også hvilke konverteringstegn der skal bruges i printf og scanf for de forskellige typer. Bemærk illustrationen af suffix tegn i den højre eksempel kolonne.

Herunder viser vi typerne fra tabel 18.1 i et konkret C program.

```
#include <stdio.h>

int main(void) {
  int i = -123;
  unsigned ui = 123U;
  long l = -123456L;
  unsigned long ul = 123456LU;
  short s = -12;
  unsigned short us = 12U;
  char c = 97;

  printf("int: %i, unsigned: %u, long: %li, unsigned long: %lu\n",
         i, ui, l, ul);
  printf("short: %hi, unsigned short: %hu, char: %c\n",
         s, us, c);

  printf("Enter integer numbers: ");
  scanf(" %i %u %li %lu %hi %hu %c", &i, &ui, &l, &ul, &s,
        &us, &c);

  printf("int: %i, unsigned: %u, long: %li, unsigned long: %lu\n",
         i, ui, l, ul);
  printf("short: %hi, unsigned short: %hu, char: %c\n", s, us, c);

  return 0;
}
```

Program 18.1 *Et C program som illustrerer ovenstående.*

I program 18.2 viser vi hvordan man finder ud af bytestørrelsen af de forskellige typer fra tabel 18.1. Outputtet fra programmet kan ses i program 18.3. Da C ikke har standardiseret taltyperne kan man opleve andre output, hvis programmet kører på en anden maskine.

```
/* Compute the size of some fundamental types. */

#include <stdio.h>

int main(void)
{
   printf("\n");
   printf("Here are the sizes of some integral types:\n\n");

   printf("           int:%3d bytes\n", sizeof(int));
   printf("      unsigned:%3d bytes\n", sizeof(unsigned));
   printf("          long:%3d bytes\n", sizeof(long));
   printf(" unsigned long:%3d bytes\n", sizeof(unsigned long));
   printf("         short:%3d bytes\n", sizeof(short));
   printf("unsigned short:%3d bytes\n", sizeof(unsigned short));
   printf("          char:%3d byte \n", sizeof(char));

   printf("\n");
   return 0;
}
```

Program 18.2 *Et program der 'udregner' bytestørrelsen af heltalstyperne.*

```
Here are the sizes of some integral types:

           int:  4 bytes
      unsigned:  4 bytes
          long:  4 bytes
 unsigned long:  4 bytes
         short:  2 bytes
unsigned short:  2 bytes
          char:  1 byte
```

Program 18.3 *Output fra programmet.*

Endelig viser vi program 18.4 hvordan man kan få oplyst mindste og største værdier i forskellige heltalstyper. De røde konstanter i program 18.4 stammer fra header filen `limits.h`. Når man kører programmet på min maskine får man grænserne, som vises i program 18.5.

```c
#include <stdio.h>
#include <limits.h>
int main(void) {

  printf("Minimum int:            %12i  Maximum int:           %12i\n",
         INT_MIN, INT_MAX);

  printf("Minimum unsigned int:   %12u  Maximum unsigned int:  %12u\n",
         0, UINT_MAX);

  printf("Minimum long:           %12li  Maximum long:          %12li\n",
         LONG_MIN, LONG_MAX);

  printf("Minimum unsigned long:  %12lu  Maximum unsigned long: %12lu\n",
         0, ULONG_MAX);

  printf("Minimum short:          %12hi  Maximum short:         %12hi\n",
         SHRT_MIN, SHRT_MAX);

  printf("Minimum unsigned short: %12hu  Maximum short:         %12hu\n",
         0, USHRT_MAX);

  return 0;
}
```

Program 18.4 *Et program der tilgår konstanter i limits.h.*

```
Minimum int:             -2147483648  Maximum int:             2147483647
Minimum unsigned int:              0  Maximum unsigned int:    4294967295
Minimum long:            -2147483648  Maximum long:            2147483647
Minimum unsigned long:             0  Maximum unsigned long:   4294967295
Minimum short:                 -32768  Maximum short:               32767
Minimum unsigned short:            0  Maximum short:               65535
```

Program 18.5 *Output fra programmet.*

18.3. Enumeration types (1)

Lektion 5 - slide 7

Enumeration types kan anvendes i situationer, hvor vi har brug for et mindre antal navngivne heltal, som spiller bestemte og veldefinerede roller i et program.

En *enumeration type* er en endelig mængde af heltal som er knyttet til enumeration konstanter

En *enumeration konstant* (enumerator) er et navn, som på mange måder ligner en variabel

Som sædvanlig viser vi den syntaktiske komposition af nye sproglige elementer. Det viste er type erklæringer, hvor vi under betegnelsen `enum tag` henviser til en type med bestemte navne, der har heltalsværdier.

```
enum tag {name1, name2, ... namei}

enum tag {name1=expr1, name2=expr2, ... namei=expri}
```

Syntaks 18.1 *Syntaktisk definition af to mulige former af enumeration typer i C*

Lad os forstå enumeration typer gennem et eksempel, se program 18.6. Under betegnelsen `enum days` gemmer sig en type, som netop består af navnene `sunday`, `monday`, ... `saturday`. Værdien af navnet `sunday` er 0, værdien af `monday` er 1, ..., og værdien af `saturday` er 6.

Typen `enum days` bruges i program 18.6 i funktionen `next_day_of`. Denne funktion tager en dag som input, og returnerer den efterfølgende dag som output. Bemærk 'cyklen' der består i, at `next_day_of(saturday)` er lig med `sunday`.

```
enum days {sunday, monday, tuesday, wednesday, thursday,
           friday, saturday};

enum days  next_day_of(enum days  d){
  enum days next_day;
  switch (d){
    case sunday: next_day = monday;
      break;
    case monday: next_day = tuesday;
      break;
    case tuesday: next_day = wednesday;
      break;
    case wednesday: next_day = thursday;
      break;
    case thursday: next_day = friday;
      break;
    case friday: next_day = saturday;
      break;
    case saturday: next_day = sunday;
      break;
  }
  return next_day;
}
```

Program 18.6 *En enumeration type* ***enum days*** *og en funktionen* ***next_day_of***.

I program 18.7 vises en procedure (som en void C funktion), der givet en dag som input udskriver denne dag på standard output via `printf`.

Implementationen af `prnt_day` illustrerer, at vi ikke har direkte adgang til det symbolske navn af en enumeration konstant. Hvis vi blot udskriver værdien `sunday` vil vi se værdien 0, og ikke "sunday". Derfor må vi lave en `switch` kontrolstruktur, som på baggrund af en **`enum days`** værdi udskriver en passende tekst.

Lad mig også her nævne, at man ikke kan læse enumerators med `scanf`. Man skal indlæse heltal, og evt. internt typekonvertere disse en enumeration værdi. Typekonvertering (type casting) bliver diskuteret i afsnit 19.2. (Under opgaveregning ved øvelserne i 5. lektion 2004 så jeg en del studerende, som havde forventet at kunne indlæse enumerators via scanf. Men den går altså ikke).

```
void prnt_day(enum days d){
  switch (d) {
    case sunday: printf("Sunday");
       break;
    case monday: printf("Monday");
       break;
    case tuesday: printf("Tuesday");
       break;
    case wednesday: printf("Wednesday");
       break;
    case thursday: printf("Thursday");
       break;
    case friday: printf("Friday");
       break;
    case saturday: printf("Saturday");
       break;
  }
}
```

Program 18.7 *En funktion der udskriver det symbolske navn på en dag.*

Der findes en version af programmet, hvor de to funktioner fra program 18.6 og program 18.7 er sat sammen med et hovedprogram, `main`. Af pladshensyn er den ikke vist her. Se Det samlede program - inklusive main på den tilhørende slide.

I et C program bidrager anvendelse af enumeration typer primært til større læsbarhed - og lettere programforståelse

18.4. Enumeration types (2)

Lektion 5 - slide 8

I dette afsnit ser vi på et antal regler som knytter sig til enumeration typer i C. Reglerne knytter sig til syntaks 18.1. Og vi ser på endnu et eksempel.

- Regler om betydningen af enumeration typer og konstanter
  - Enumeration konstanter har samme status som variable og må som sådan kun defineres én gang i det samme scope
  - I det første tilfælde tildeles *name1* værdien *0*, *name2* værdien *1*, etc.
  - I det andet tilfælde bestemmer programmøren hvilke heltalsværdier de enkelte enumeration konstanter tildeles
    - Der er mulighed for at to eller flere konstanter i samme enumeration type har samme værdi

I eksemplet herunder anvender vi to enumeration typer, `enum grade_simple` og `enum grade_13`. Disse afspejler karaktererne på de to skalaer bestået/ikke bestået og den danske 13 skala.

Vi ser at enumeratorerne på 13 skalaen bliver afbildet på de heltal, som karaktererne naturligt står for. Eksempelvis skriver vi `nul_nul = 0` for at binde enumeratoren `nul_nul` til 0.

```
#include <stdio.h>

enum grade_simple {not_passed, passed};

enum grade_13 {nul_nul = 0, nul_tre = 3, fem = 5, seks,
               syv, otte, ni, ti, elleve, tretten = 13};

enum grade_simple convert_grade_13_to_simple_grade
                                        (enum grade_13 g){
  enum grade_simple result;

  if (g <= fem)
    result = not_passed;
  else
    result = passed;

  return result;
}

void prnt_grade_simple(enum grade_simple g){
  switch (g) {
    case not_passed: printf("Not passed");
      break;
    case passed: printf("Passed");
      break;
  }
}

int main(void){

  int grade_number;

  printf("Enter '13 skala' grade: ");
  scanf(" %d", &grade_number);

  prnt_grade_simple(
         convert_grade_13_to_simple_grade(grade_number));
  printf("\n");

  return 0;
}
```

Program 18.8 *Et eksempel på et program som bruger enumeration typer til karakterskalaer.*

I karaktereksemplet i program 18.8 kan man klart diskutere, om introduktionen af de symbolske navne for karakterværdierne på 13 skalaen gør en nævneværdig forskel. En konsekvent anvendelse beskytter os dog mod at bruge meningsløse karakterværdier, så som -1, 4, 12 og 14.

Anvendelsen af `not_passed` og `passed` (for ikke-bestået og bestået) i stedet for 0 og 1 bidrager klart til bedre læsbarhed og forståelighed. Vi kan sige at det bringer vores program lidt tættere på virkelighedens begrebsverden.

18.5. Enumeration typer i andre sprog

Lektion 5 - slide 9

Der findes programmeringssprog, der på en mere klar og ren måde end C understøtter enumeration typer. Der findes også nyere sprog, som slet ikke har særlig understøttelse af enumeration typer.

Enumeration typer i C er anderledes end tilsvarende typer i de fleste andre programmeringssprog

- Enumeration typer i Pascal-lignende sprog:
  - Værdien af enumeration konstanterne er nye, entydige værdier - ikke heltal
  - Ordningen af enumeration konstanterne er af betydning
    - *Efterfølger* og *forgænger* funktioner af enumeration konstanter giver mening

Java har ikke overtaget enumeration typerne fra C

18.6. Floating point typer (1)

Lektion 5 - slide 10

I dette afsnit giver vi en oversigt over floating point typer i C, efter samme mønster som vi diskuterede for heltalstyper i afsnit 18.2.

Type	Suffix	`printf` conv. tegn	`scanf` conv. tegn	Eksempel
`float`	f eller F	`%f`	`%f`	*5.4F*
`double`	*intet*	`%f`	`%lf`	*5.4*
`long double`	l eller L	`%Lf`	`%Lf`	*5.4L*

Tabel 18.2

Bemærk lige `scanf` konverterings- og modifier tegnene `%lf` for indlæsning af double. I forhold til det tilsvarende `printf` konverteringstegn `%f` kan brug af `%lf` (i betydningen `l`ong `f`loat) virke underlig.

Nedenstående program viser mulighederne i tabel 18.2 i et konkret C program.

```c
#include <stdio.h>

int main(void) {

  float f = 123.456F;
  double d = 123.456;
  long double ld = 123.456L;

  printf("float: %f, double: %f, long double: %Lf\n", f, d, ld);

  printf("Enter new values: ");
  scanf(" %f %lf %Lf",&f, &d, &ld);
  printf("float: %f, double: %f, long double: %Lf\n", f, d, ld);

  return 0;
}
```

Program 18.9 *Et C program som illustrerer ovenstående.*

Ligesom i program 18.2 viser vi herunder hvordan vi kan bestemme bytestørrelsen af typerne `float`, `double` og `long double`. Det selvforklarende output fra programmet vises i program 18.11. Som omtalt tidligere er det ikke sikkert, at du får det samme output hvis du kører programmet på din computer.

```c
/* Compute the size of some fundamental types. */

#include <stdio.h>

int main(void)
{
   printf("\n");
   printf("Here are the sizes of some floating types:\n\n");

   printf("      float:%3d bytes\n", sizeof(float));
   printf("     double:%3d bytes\n", sizeof(double));
   printf("long double:%3d bytes\n", sizeof(long double));

   printf("\n");
   return 0;
}
```

Program 18.10 *Et program der 'udregner' bytestørrelsen af float typerne.*

```
Here are the sizes of some floating types:

      float:  4 bytes
     double:  8 bytes
long double: 12 bytes
```

Program 18.11 *Output fra programmet.*

Ligesom i program 18.4 viser vi herunder grænserne for værdierne i typerne `float`, `double` og `long double`. Bemærk at der er tale om mindste og største *positive* tal i typerne. Output af programmet program 18.12 ses i program 18.13.

```c
#include <stdio.h>
#include <float.h>
int main(void) {

  printf("Minimum float:        %20.16e  \nMaximum float:        %20.16e\n\n",
         FLT_MIN, FLT_MAX);

  printf("Minimum double:       %20.16le \nMaximum double:       %20.16le\n\n",
         DBL_MIN, DBL_MAX);

  printf("Minimum long double:  %26.20Le \nMaximum long double:  %26.20Le\n\n",
         LDBL_MIN, LDBL_MAX);


  return 0;
}
```

Program 18.12 *Et program der tilgår konstanter i floats.h.*

```
Minimum float:         1.175494e-38
Maximum float:         3.402823e+38

Minimum double:        2.2250738585072014e-308
Maximum double:        1.7976931348623157e+308

Minimum long double:   3.36210314311209350626e-4932
Maximum long double:   1.18973149535723176502e+4932
```

Program 18.13 *Output fra programmet.*

I outputtet fra program 18.12, som er vist herover, kan vi notere os at vi i `printf` kaldene har taget hensyn til den *præcision*, som understøttes af hhv. `float`, `double` og `long double`. Præcision er et af de emner vi ser på i næste afsnit, afsnit 18.7.

18.7. Floating point typer (2)

Lektion 5 - slide 11

Efter nu at have brugt tid og kræfter på at forstå de tre floating point typer `float`, `double` og `long double` vil vi nu se på *notation* for flydende tal, *præcisionen* af værdierne i de tre typer, *intervallet* der dækkes af typerne.

- *Notation*
  - Decimal: 12345.6789
  - Eksponential: 0.1234567e-89
  - En flydende tal konstant skal enten indeholde *decimal punktet* eller *eksponential delen* (eller begge)
- *Præcision*: Antallet af signifikante cifre i tallet
  - **`float`**: typisk 6 cifre
  - **`double`**: typisk 15 cifre
  - **`long double`**: typisk 20 cifre

- *Interval* (range): Største og mindste mulige *positive* værdier
  - **`float`**: Typisk 10^{-38} til 10^{38}
  - **`double`**: Typisk 10^{-308} til 10^{308}
  - **`long double`**: Typisk 10^{-4932} til 10^{4932}

Dette afslutter vores dækning af de fundamentale heltals og floating point typer i C. Relativ til oversigten i afsnit 18.1 mangler vi at dække pointertyper og sammensatte typer. Pointertyper tager vi hul på i kapitel 22. Arrays behandles i kapitel 24, kapitel 25 og kapitel 40. Structs kommer på banen i kapitel 39.

19. Typekonvertering og typedef

Der er undertiden behov for at konvertere en værdi i én type til en værdi i en anden type. I nogle situationer er det programmøren, som ønsker dette. I så fald taler vi om en *eksplicit typekonvertering*, eller i teknisk jargon for en *cast*. I andre tilfælde er det C compileren der føler trang til at foretage en typekonvertering. Disse omtales som *implicitte typekonverteringer*. I afsnit 19.1 ser vi først på de implicitte typerkonverteringer.

Som et andet emne i dette kapitel ser vi på brug af `typedef` til omnavngivning af eksisterende typer.

19.1. Implicit typekonvertering

Lektion 5 - slide 13

Vi giver en ganske kortfattet oversigt over tre former for implicit typekonvertering herunder.

C foretager en række typekonverteringer 'bag ryggen på programmøren'

- *"Integral promotion"*
  - Udtryk af heltalstype hvori der indgår **`short`** eller **`char`** værdier konverteres til en værdi i typen **`int`**.
  - Eksempel: Hvis x og y er af typen **`short`** er værdien af x + y **`int`**.
- *"Usual arithmetic conversions" - widening*
  - Konvertering af en mindre præcis værdi til en tilsvarende mere præcis værdi således at begge operander får samme type
  - Der mistes ikke information og præcision
- *Narrowing - demotion*
  - Konvertering af mere præcis værdi til en mindre præcis værdi
  - Der mistes information

Integral promotion benyttes bl.a. til at få beregninger på `shorts` og `chars` til at forgå inden for værdier i typen int.

Brug af widening er bekvem, når vi skriver udtryk der involverer værdier i to forskellige typer. Eksempelvis `d + i`, hvor `d` er en `double` og `i` er en `int`. Maskinen understøtter sandsynligvis ikke en addition af netop disse to typer, så derfor konverteres den mest snævre operand, her `i`, til typen af den mest brede operand. Altså konverteres værdien af `i` til en `double`, hvorefter der kan udføres en addition af to `double` værdier.

Narrowing udføres f.eks. når en `double` værdi assignes til en integer variable.

De tre forskellige former for typekonverteringer er illustreret i program 19.1.

```
#include <stdio.h>

int main(void) {

  short s = 12; char c = 'a';
  double d = 123456.7; float f = 4322.1;
  int i;

  printf("c - s = %i is converted to int\n", c - s);
  /* The type of c - s is promoted to int */

  printf("d + f = %f is converted to a double\n", d + f);
  /* f is converted to double before adding the numbers */

  i = d;
  printf("In assigning d to i %f is demoted to the int %i\n", d, i);
  /* d is converted to an int - information is lost */

  return 0;
}
```

Program 19.1 *Eksempler på implicitte typekonverteringer.*

19.2. Eksplicit typekonvertering

Lektion 5 - slide 14

Som programmører har vi en gang imellem brug for at gennemtvinge en konvertering af en værdi fra en type til en anden. Ofte er konverteringen dog "spil for galleriet", i den forstand at vi blot garanterer over for kompileren at værdien kan opfattes som værende af en bestemt type. I disse situationer finder der ingen reel forandring sted som følge af en typekonvertering.

I C er det muligt for programmøren at konvertere værdien af et udtryk til en anden type ved brug af en såkaldt *cast operator*

En typecast noteres ved at foranstille det udtryk, hvis værdi ønskes konverteret, i en typecase operator. En typecast operator er et typenavn i parentes.

At en typecast faktisk er en operator kan man overbevise sig om ved at studere niveau 14 i tabel 2.3.

```
(typeName) expression
```

Syntaks 19.1 *Syntaktisk definition af casting - eksplicit typekonvertering i C*

I program 19.2 viser vi hvordan værdien af `'A'+10` kan konverteres til en `long`. Vi ser også hvordan værdien af `long` variablen `y` kan konverteres til en `int`. Bemærk at `y` konverteres før additionen, på grund af den indbyrdes placering af typecast og addition i tabel 2.3. Endelig ser vi hvordan værdien af udtrykket `x=77`, altså heltallet 77, kan konverteres til en `double` før assignmentet til `z`.

```
#include <stdio.h>

int main(void) {

  long y;
  float x;
  double z;

  y = (long) ('A' + 10);
  x = (float) ((int) y + 1);
  z = (double) (x = 77);

  printf("y: %li, x: %f, z: %f", y, x, z);
  return 0;
}
```

Program 19.2 *Et program med eksempler på casts.*

I det næste eksempel, program 19.3, ser vi hvordan funktionen `next_day_of`, som oprindelig blev vist i program 18.6, kan omskrives. I den omskrevne, og meget kortere udgave af `next_day_of`, benytter vi to casts. Den blå konvertering transformerer en `enum days` værdi til en `int`. Til dette heltal adderer vi 1 og vi uddrager resten ved division med 7. Tænk lige over hvorfor... Den røde konvertering bringer det resulterende heltal tilbage som en `enum days` værdi.

```
enum days next_day_of(enum days d){
  return (enum days) (((int) d + 1) % 7);
}
```

Program 19.3 *Funktionen* ***`next_day_of`*** *omskrevet med brug af casts.*

19.3. Navngivne typer med typedef

Lektion 5 - slide 15

I nogle situationer kan vi have lyst til at give en eksisterende type et nyt navn. Dette gør sig specielt gældende for typer med ubekvemme betegnelser, så som `enum days`, `enum grade_13`, og `enum simple_grade`.

I C er det muligt at tildele en type et bestemt navn

I tilfælde af komplicerede typer kan dette øge læsbarheden af et program

Syntaksen for typedef er enkel. Blot skal man lige bemærke, at det nye typenavn angives til sidst.

```
typedef oldType newType
```

Syntaks 19.2 *Syntaktisk definition af casting - eksplicit typekonvertering i C*

Vi viser herunder hvordan program 18.6 og 18.7 kan omskrives til program 19.4. Ændringerne er ikke revolutionerende, men læg alligevel mærke til de blå og røde steder i forhold til program 19.4.

```
#include <stdio.h>

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

days  next_day_of(days  d){
  return ( days ) (((int) d + 1) % 7);
}

void prnt_day(days  d){
  switch (d) {
    case sunday: printf("Sunday");
       break;
    case monday: printf("Monday");
       break;
    case tuesday: printf("Tuesday");
       break;
    case wednesday: printf("Wednesday");
       break;
    case thursday: printf("Thursday");
       break;
    case friday: printf("Friday");
       break;
    case saturday: printf("Saturday");
       break;
  }
}

int main(void){
```

```
  days  day1 = saturday,  another_day;
  int i;

  printf("Day1 is %d\n", day1);

  printf("Day1 is also "); prnt_day(day1); printf("\n");

  another_day = day1;
  for(i = 1; i <= 3; i++)
    another_day = next_day_of(another_day);

  printf("Three days after day1: ");  prnt_day(another_day);
  printf("\n");

  return 0;
}
```

Program 19.4 *En omskrivning af ugedags programmet som benytter typedef.*

Det kan måske forvirre lidt, at vi bruger `days` både som navnet på den nye type og som en del af navnet på den eksisterende type. Teknisk set er *tag navnet* `days` i `enum days` og det nye typenavn `days` i to forskellige navnerum. Derfor generer de ikke hinanden.

20. Scope og storage classes

Scope begrebet hjælper os til at forstå, hvor et bestemt navn i program er gyldigt og brugbart. Storage classes er en sondring i C, som bruges til at skelne mellem forskellig lagringer og forskellige synligheder af variable, funktioner, mv.

20.1. Scope

Lektion 5 - slide 17

Scope reglerne bruges til at afgøre i hvilke programdele et navn er gyldigt.

På dansk kan vi bruge ordet *'virkefelt'* i stedet for *'scope'*.

I store programmer bruges det samme navn ofte i forskellige scopes med forskellige betydninger.

Scope af et navn er de dele af en programtekst hvor navnet er kendt og tilgængeligt

Følgende beskriver den vigtigste af alle scoperegler af alle i C.

- De mest basale scoperegler i C - storage class **`auto`**
  - Scopet af et navn er den blok hvori den er erklæret

- Introducerer *huller i scope*

Den bedste måde at forstå reglerne, som blev omtalt herover, er at bringe dem i spil på et godt eksempel. Et sådant er vist i program 20.1. Nogle vil nok genkende dette eksempel fra afsnit 7.1 (program 7.1) hvor vi diskuterede sammensatte kommandoer.

```
#include <stdio.h>

int main(void) {

  int a = 5, b = 7, c;

  c = a + b;

  {
    int b = 10, c;

    {
      int a = 3, c;
      c = a + b;
      printf("Inner:  a + b = %d + %d = %d\n", a, b, c);
    }

    c = a + b;
    printf("Middle: a + b = %d + %d = %d\n", a, b, c);
  }

  c = a + b;
  printf("Outer:  a + b = %d + %d = %d\n", a, b, c);

  return 0;
}
```

Program 20.1 *Illustration af scope i tre indlejrede blokke.*

Der vises tre blokke i `main`, som indlejres i hinanden. I den yderste blok har vi variablene `a`, `b` og `c`. I den mellemste blok har vi `b` og `c`, som skygger for `b` og `c` fra den yderste blok. I den inderste blok har vi `a` og `c`, som igen skygger for hhv. `a` (fra yderste blok) og `c` (fra den mellemste blok). Skygningen medfører det vi kalder et "hul i scopet".

Bemærk at vi kan se fra indre blokke ud i ydre blokke, men aldrig omvendt.

Du skal kunne forklare det output, som kommer fra program 20.1. Hvis du kan, har du givetvis styr på blokke og storage class auto. Vi har mere at sige om sidstnævnte - automatiske variable i C - i næste afsnit.

```
Inner:  a + b = 3 + 10 = 13
Middle: a + b = 5 + 10 = 15
Outer:  a + b = 5 + 7 = 12
```

Program 20.2 *Output fra programmet.*

20.2. Storage class auto

Lektion 5 - slide 18

Der findes et lille antal forskellige former for såkaldte storage classes i C, som er vigtige for fastlægningen af scope af de forskellige navngivne enheder i programmet.

Storage class af variable og funktioner medvirker til bestemmelse af disses scope

Storage classes angives som en modifier før selve typenavnet i en erklæring:

```
storageClass type var1, var2, ..., vari
```

Syntaks 20.1 *Syntaktisk definition af storage classes i variabel erklæringer*

Variable med storage class *auto* kaldes automatiske variable

En *automatisk variabel* er lokal i en blok, dvs den skabes når blokken aktiveres og nedlægges når blokken forlades

Essentielt set illustrerede vi allerede storage class auto i program 20.1 i afsnittet ovenfor.

- Storage class **`auto`**
  - Default storage class i blokke, herunder kroppe af funktioner
    - Hvis man ikke angiver storage class af variable i blokke er den således **`auto`**
  - Man kan angive storage class **`auto`** eksplicit med brug af nøgleordet **`auto`**

`register` storage class er en variant af **`auto`** storage class

Sammenfattende kan vi sige, at storage class auto sammenfatter lokale variable, som oprettes ved indgang i en blok og nedlægges ved udgang af en blok.

20.3. Storage class static af lokale variable

Lektion 5 - slide 19

En automatisk variabel ophører med at eksistere når den omkringliggende blok ophører med at eksistere. Dette kan f.eks. ske når en funktion returnerer.

Hvis vi ønsker at en lokal variabel overlever blok ophør eller funktionsreturnering kan vi gøre den global (ekstern, se afsnit 20.4). Men vi kan også lave en lokal variabel af storage class static.

I blokke er storage class **`static`** et alternativ til storage class **`auto`**

En *statisk variabel* i en blok beholder sin værdi fra en aktivering af blokken til den næste

- Storage class static - lokale variable i en blok
  - Skal angives med **`static`** som storageClass
  - Variabel initialiseringen udføres ved første aktivering

Lad os illustrere statiske lokale variable med følgende eksempel. Vi ser en funktion `accumulating_f`, som kaldes 10 gange fra `main`. I alle kald overføres parameteren 3 til `accumulating_f`.

I funktionen `accumulating_f` ser vi med rødt den statiske variabel `previous_result`, hvori vi gemmer den forrige returværdi fra kaldet af `accumulating_f`. Kun når funktionen kaldes første gang initialiseres `previous_result` til 1. (Dette er en vigtig detalje i eksemplet). Hvis ikke `previous_result` er 1 multipliceres `previous_result` med funktionens input, og denne størrelse returneres.

Output fra funktionen vises i program 20.4. Hvis vi sletter ordet **`static`** fra program 20.3 ser vi let, at alle de 10 kald returnerer tallet 1.

```
#include <stdio.h>

int accumulating_f (int input){
  int result;
  static int previous_result = 1;

  if (previous_result == 1)
    result = input;
  else
    result = previous_result * input;

  previous_result = result;
  return result;
}

int main(void) {
  int i;

  for (i = 0; i < 10; i++)
    printf("accumulating_f(%d) = %d\n", 3, accumulating_f(3));

  return 0;
}
```

Program 20.3 *Illustration af statiske lokale variable - en funktion der husker forrige returværdi.*

```
accumulating_f(3) = 3
accumulating_f(3) = 9
accumulating_f(3) = 27
accumulating_f(3) = 81
accumulating_f(3) = 243
accumulating_f(3) = 729
accumulating_f(3) = 2187
accumulating_f(3) = 6561
accumulating_f(3) = 19683
accumulating_f(3) = 59049
```

Program 20.4 *Output fra programmet.*

Statiske lokale variable er et alternativ til brug af globale variable.

En statisk lokal variabel bevarer sin værdi fra kald til kald - men er usynlig uden for funktionen.

Man kunne få den tanke at en variabel som er statisk ikke kan ændres. Dette vil dog være underligt, idet betegnelsen 'variabel' jo næsten inviterer til ændringer. Men 'static' i C har ikke denne konsekvens. Hvis vi i C ønsker at gardere os mod ændringer af en variable skal vi bruge en modifier som hedder `const`.

20.4. Storage class extern

Lektion 5 - slide 20

Nogle variable og funktioner ønskes at være tilgængelige i hele programmet. For variable opnås dette ved at placere disse uden for funktionerne, og uden for **`main`**. Sådanne variable, og funktioner, siges at have external linkage, og de er tilgængelige i hele programmet.

Dog gælder stadig reglen, at en variabel eller funktion ikke kan bruges før det sted i programmet, hvor den er introduceret og defineret. For funktioner bruger vi ofte de såkaldte prototyper til at annoncere en funktion, som vi endnu ikke har skrevet.

Variable og funktioner med storage class *extern* kaldes eksterne variable

En *ekstern variabel* eller funktion er global tilgængelig i hele programmet

- Storage class **`extern`**
  - Default storage class af variable som erklæres uden for funktioner
  - Default storage class af alle funktioner er også **`extern`**
  - *"External linkage"*

20.5. Storage class static af eksterne variable

Lektion 5 - slide 21

Den sidste storage class vi vil introducere hedder også static, men den virker på eksterne variable. Statiske eksterne variable kan kun ses i den kildefil, hvori de er erklæret.

Statiske eksterne variable har ikke noget at gøre med statiske lokale variable ala afsnit 20.3.

For globale variable virker storage class **`static`** som en scope begrænsning i forhold til storage class **`extern`**.

Statiske globale variable er private i den aktuelle kildefil.

En *statisk variabel* på globalt niveau er kun synlig i den aktuelle kildefil

- Storage class static - globale variable
  - Skal angives med **`static`** som storage class
  - Synlighed i den aktuelle kompileringsenhed - kildefil
  - Kun synlig fra det punkt i filen hvor variablen er erklæret
  - *"Internal linkage"*

Funktioner kan også være statiske, og dermed private i en kildefil.

Synlighedskontrol er vigtig i store programmer.

Synlighedskontrol er meget vigtig i objekt-orienteret programmering.

21. Introduktion til arrays

I dette kapitel vil vi introducere arrays. Vi vil se på de væsentligste overordnede egenskaber af arrays, og vi vil se på nogle forhold, som er specifikke for C.

21.1. Arrays

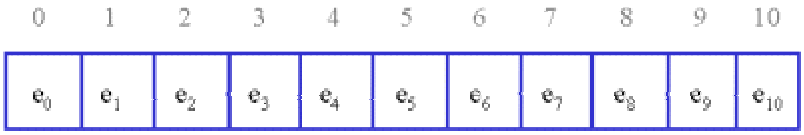
Lektion 6 - slide 2

Et array er en tabel med effektiv tilgang til elementerne via heltallige indeks numre

Vi viser herunder et array med 11 elementer. Disse kaldes her e_0, e_1, ..., e_{10}.

Med *array elementerne* mener vi det som befinder sig inden i kasserne i figur 21.1. Eksempelvis er e_0 et element i tabellen. I et konkret program kunne e_0 være et heltal. Elementerne er det lagrede indhold af tabellen. Alle elementerne i et array skal være af samme type.

Med *array indekserne* mener vi de forskellige pladser i et array. Inspireret af gængs matematisk notation kaldes indekser også ofte for *subscripts*. Indekserne i arrays i C starter altid med 0. Hvis der i alt er 11 elementer i tabellen er det højeste indeks således 10. Indekserne lagres ikke i tabellen. Det er kun en pladsbetegnelse. Da alle kasserne i tabellen har samme størrelse kan vi let slå direkte ned på et element med et bestemt indeks nummer.



Figur 21.1 *Et array med 11 elementer indiceret fra 0 til 10*

Herunder beskriver vi de vigtigste egenskaber af et array.

- Alle elementer er af samme type
- Index angiver en plads i tabellen
- Elementerne i et array `a` tilgås med *subscripting* notation: `a[i]`
- Når et array er skabt har det faste nedre og øvre indeksgrænser
  - Den nedre indeksgrænse er altid *0* i C.
- Elementerne lagres *konsekutivt* (umiddelbart efter hinanden).
  - Dermed er det simpelt at beregne hvor et element med et bestemt index er lagret
  - Effektiv tilgang via index

Lad os understrege to ting her. For det første at et array er lagret i ét sammenhængende lagerområde. For det andet at grænserne for, og dermed antallet af elementer i et array, ikke kan ændres når først arrayet er skabt.

21.2. Arrays i C

Lektion 6 - slide 3

Vi træder nu et skridt tilbage, og søger en motivation for at have arrays i vore programmer. Det gør vi i program 21.1 ved at erklære, initialisere og udskrive 11 variable, navngivet `tabel0`, ..., `tabel10`.

Det fremgår tydeligt, at program 21.1 er "træls". Det er uholdbart eksplicit at skulle erklære alle disse variable enkeltvis, at initialisere disse enkeltvis (på trods af at initialiseringen er systematisk), og at udskrive disse enkeltvis. Vi ønsker en bedre løsning, hvor vi i ét slag kan erklære `table0`, ..., `table10`. Endvidere ønsker vi at kunne initialisere og udskrive variable i en for-løkke.

```
#include <stdio.h>

int main(void) {

  double table0, table1, table2, table3, table4,
         table5, table6, table7, table8, table9, table10;

  table0 = 0.0;
  table1 = 2 * 1.0;
  table2 = 2 * 2.0;
  table3 = 2 * 3.0;
  table4 = 2 * 4.0;
  table5 = 2 * 5.0;
  table6 = 2 * 6.0;
  table7 = 2 * 7.0;
  table8 = 2 * 8.0;
  table9 = 2 * 9.0;
  table10 = 2 * 10.0;

  printf("%f, %f, %f, %f, %f, %f, %f, %f, %f, %f, %f\n",
         table0, table1, table2, table3, table4,
         table5, table6, table7, table8, table9, table10);

  return 0;
}
```

Program 21.1 *Motivation for arrays - et stort antal simple variable.*

I program 21.2 viser vi et program som svarer til program 21.1. I programmet herunder erklærer vi på det røde sted 11 doubles, som benævnes `table[0]`, ... `table[10]`. Variablen `table` er erklæret som et array. På de to blå steder initialiserer og udskriver vi elementer i `table`.

```c
#include <stdio.h>

int main(void) {

  double table[11];
  int i;

  for(i = 0; i < 11; i++)
    table[i] = (double)(2 * i);

  for(i = 0; i < 11; i++)
    printf("%f ", table[i]);

  printf("\n");

  return 0;
}
```

Program 21.2 *Introduktion af arrays.*

Vi viser herunder en variant af program 21.2 hvor vi på det røde sted viser en direkte, manifest repræsentation af tabellen. Dette er nyttigt hvis ikke vi på en systematisk måde kan initialisere tabellen i en løkke.

```c
#include <stdio.h>

int main(void) {

  double table[11] =
    {0.0, 5.0, 8.0, 9.9, 10.2, 8.5, 99.9, 1.0, -5.2, 7.5, 9.4};

  int i;

  for(i = 0; i < 11; i++)
    printf("%f ", table[i]);

  printf("\n");

  return 0;
}
```

Program 21.3 *Eksplicit array initialisering.*

Med disse eksempler og beskrivelserne har vi introduceret de vigtigste egenskaber ved arrays i C. I de efterfølgende kapitler ser vi på arrays i forbindelse med pointere. I kapitel 27 og efterfølgende kapiteler ser vi på arrays i forbindelse med tekststrenge.

22. Pointers

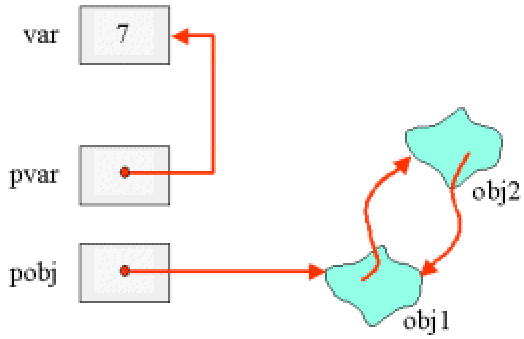
I dette kapitel diskuterer vi pointere i C.

22.1. Introduktion til pointere

Lektion 6 - slide 5

Herunder viser vi, at pointere kan bruges som alternative adgangsveje til variable. Vi viser også, at pointere kan bruges som adgangsvej til dataområder, som er placeret uden for variablene. Dette er pointerne til de grønne områder i figur 22.1. De grønne dataområder i figuren vil typisk være frembragt med dynamisk allokering med `malloc` eller `calloc`. Mere om dette i afsnit 26.2.

Pointere er uløseligt knyttet til programmering i C



Figur 22.1 *En illustration af to pointer variable.* ***pvar*** *er en pointer til en integer, som peger på adressen af var.* ***pobj*** *peger på en objekt obj1, som ikke er indeholdt i nogen variabel. obj1 peger på et andet objekt, som 'cyklisk' peger tilbage på obj1.*

Den normale måde at tilgå data er gennem navngivne variable.

Pointere udgør en alternativ og fleksibel tilgangsvej til data.

22.2. Pointerbegrebet i C

Lektion 6 - slide 6

Inden vi ser på eksempler beskriver vi aspekter af pointere i nedenstående punkter.

En *pointer* er en værdi som betegner en adresse på et lagret dataobjekt

- Værdier af variable kan tilgås via pointere til variablene
  - Adresseoperatoren **`&`** giver os adressen på variablen - hvilket er en pointer til indholdet i variablen
- Pointere til andre pladser i lageret returneres af de primitiver, der allokerer lager dynamisk
  - **`calloc`** og **`malloc`**
- Vi visualiserer ofte en pointer som en pil, som peger på et dataobjekt eller en variabel
  - Angivelse af lageradresser ville udgøre en mere maskinnær visualisering
- Pointere har typer, alt efter hvilke typer af værdier de peger på
  - Pointere til integers, pointere til chars, ...
  - Pointer til **`void`** er en *generisk pointer type* i C
- Dataobjekter, som ikke er indeholdt i nogen variabel, tilgås via pointere
- Pointere, som ikke peger på noget dataobjekt, skal have værdien **`NULL`**

22.3. Pointer variable

Lektion 6 - slide 7

Variable der indeholder pointere til dataobjekter af typen **`T`** skal erklæres af typen **`T *`**

I program 22.1 erklærer vi variablene `i`, `ptr_i`, `j`, `ptr_j`, `c` og `ptr_c`. Alle på nær `j` og `ptr_j` initialiseres sammen med erklæringerne.

Hvis der står en **`*`** foran en variabel i en erklæring betyder det at variablen skal indeholder en pointer. Konkret i program 22.1 skal `ptr_i` og `ptr_j` være pointers til heltal (`int`). `ptr_c` skal være en pointer til en char.

Assignmentet `ptr_j = &j` lægger adressen af `j` ind i `ptr_j`.

Assignmentet `ptr_i = ptr_j` lægger (kopierer) den pointer, som `ptr_j` indeholder, over i `ptr_i`. I figur 22.2 viser vi situationen efter at program 22.1 er kørt til ende.

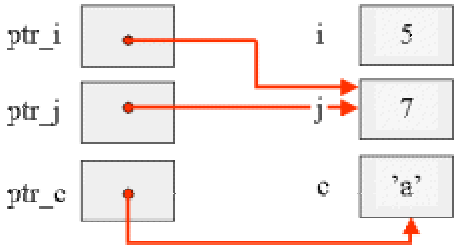
Bemærk at pointere visualiseres som pile, men reelt er en pointer en lagreradresse.

```
int i = 5, *ptr_i = &i, j = 7, *ptr_j;
char c = 'a', *ptr_c = &c;

ptr_j = &j;

ptr_i = ptr_j;
```

Program 22.1 *Erklæring og initialisering af et antal variable, hvoraf nogle er pointer variable.*



Figur 22.2 *Grafisk illustration af variablene i, j, c og deres tilsvarende pointervariable. Læg mærke til værdien af ptr_i, som er sat til at pege på samme værdi som ptr_j, altså variablen j.*

Vi viser i program 22.2 et helt C program, som indeholder fragmentet fra program 22.1. Outputtet fra programmet ses i program 22.3. Vær sikker på at du kan relatere outputtet fra program 22.2 med situationen i figur 22.2.

```
#include <stdio.h>

int main(void) {
  int i = 5, *ptr_i = &i, j = 7, *ptr_j;
  char c = 'a', *ptr_c = &c;

  ptr_j = &j;

  ptr_i = ptr_j;

  printf("i=%d, ptr_i=%p, *ptr_i = %i\n", i, ptr_i,  *ptr_i);
  printf("j=%d, ptr_j=%p, *ptr_j = %i\n", j, ptr_j,  *ptr_j);
  printf("c=%c, ptr_c=%p, *ptr_c = %c\n", c, ptr_c,  *ptr_c);

  return 0;
}
```

Program 22.2 *Hele programmet inklusive udskrivning af variablenes værdier.*

```
i=5, ptr_i=ffbef534, *ptr_i = 7
j=7, ptr_j=ffbef534, *ptr_j = 7
c=a, ptr_c=ffbef52f, *ptr_c = a
```

Program 22.3 *Output fra programmet.*

Bemærk i udskriften ovenfor, at pointere, udskrevet med konverteringstegnet `%p` i program 22.2, er vist i hexadecimal notation. Vi studerende hexadecimale tal i afsnit 16.5.

22.4. Addresse og dereferencing operatorerne

Lektion 6 - slide 8

Vi sætter i dette afsnit fokus på to nye operatorer, som knytter sig til pointere, nemlig adresse operatoren **`&`** og dereferencing operatoren **`*`**.

Udtrykket `&var` beregner til *adressen af variablen* `var`.

Udtrykket `*ptrValue` beregner *den værdi, som* `ptrValue` *peger på.*

Operatorerne `*` og `&` er unære prefix operatorer med høj prioritet

Dereferencing operatoren følger pointeren, og tilgår dermed det som pilen peger på.

I programmet herunder, som er helt identisk med program 22.2, viser de blå steder anvendelser af dereferencing operatoren, og de brune steder anvendelser af adresse operatoren.

```
int i = 5, *ptr_i = &i, j = 7, *ptr_j;
char c = 'a', *ptr_c = &c;

ptr_j = &j;

ptr_i = ptr_j;

printf("i=%d, ptr_i=%p, *ptr_i = %i\n", i, ptr_i,  *ptr_i);
printf("j=%d, ptr_j=%p, *ptr_j = %i\n", j, ptr_j,  *ptr_j);
printf("c=%c, ptr_c=%p, *ptr_c = %c\n", c, ptr_c,  *ptr_c);
```

Program 22.4 *Fremhævelse af adresse og dereferencing udtryk programmet fra forrige side.*

Oparatoren `*` kaldes også for *indirection operatoren*

For alle variable `v` gælder at udtrykket `*&v` er ækvivalent med udtrykket `v`

23. Call by reference parametre

I dette korte, men vigtige kapitel, introducerer vi en ny måde at bruge parametre på; Nemlig ved at overføre pointere som parametre. Kort sagt er der tale om parametre, som kan bruges til at ændre værdien af eksisterende variable. Det er også tale om en parametermekanisme som ofte er meget økonomisk i både tid og plads.

23.1. Call by reference parametre

Lektion 6 - slide 10

Ved brug af pointere kan vi realisere *call by reference parametre* i C

Vi skriver først et program, som ikke udnytter pointere og adresser på variable. Dermed bliver vi motiveret for at bruge pointere som parametre. Vi kan også bruge program 23.1 til at erindre hvordan værdiparametre (call by value parametre), diskuteret i bl.a. afsnit 12.5, virker.

```c
#include <stdio.h>

void swap(int p, int q){
   int   tmp;

   tmp = p;
   p = q;
   q = tmp;
}


int main(void)
{
   int   a = 3, b = 7;

   printf("%d  %d\n", a, b);      /* 3  7 is printed */
   swap(a, b);
   printf("%d  %d\n", a, b);      /* 3  7 is printed */
   return 0;
}
```

Program 23.1 *Et forsøg på ombytning af to variables værdi uden pointere - virker ikke.*

I program 23.1 herover ser vi med blåt definitionen af en funktion `swap`, og med rødt kaldet af `swap`. Vi overfører de to aktuelle parametre `a` og `b`, som modsvarer de to formelle parametre `p` og `q` i `swap`. I `swap` ombytter vi de to formelle parametres værdier, men der sker ingen ændring af de to aktuelle parametres værdier. Med andre ord, værdierne af `a` og `b` ændres ikke i kaldet af `swap`.

```c
#include <stdio.h>

void swap(int *p, int *q){
   int   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}


int main(void)
{
   int   a = 3, b = 7;

   printf("%d  %d\n", a, b);      /* 3  7 is printed */
   swap(&a, &b);
   printf("%d  %d\n", a, b);      /* 7  3 is printed */
   return 0;
}
```

Program 23.2 *Funktionen swap og et hovedprogram, som kalder **swap** på to variable.*

Programmet herover, program 23.2, er ændret lidt i forhold til program 23.1. I den blå del - funktionsdefinitionen - har vi indført to pointer parametre `p` og `q`. Både `p` og `q` skal pege på heltal. I

den røde del, hvor vi kalder `swap`, overfører vi adressen på hhv. `a` og `b` som aktuelle parametre. Bemærk brugen af adresseoperatoren **`&`**.

I kroppen af definitionen af `swap` (den blå del) lægger vi `*p` over i `temp`. Dette indebærer at vi følger pointeren `p` for at tilgå den værdi, som `p` peger på. I det konkrete kald er det værdien af `a`, altså 3 som assignes til `tmp`. I det andet assignment bliver `*q`, som konkret er værdien af `b`, lagt over i `temp`. I det tredje assignment bliver `*q`, hvilket er `a`, assignet til værdien af `temp`. Altså lægger vi 3 i `b`. Du kan se outputtet i program 23.3.

Vi ser at vi i denne udgave af programmet opnår en ombytning af værdierne i `a` og `b`. Vi har altså skrevet en funktion, som gennem parametrene er i stand til at ændre værdier af variable, som er uden for selve funktionen. Dette er essensen af *call by reference parametre*.

```
3   7
7   3
```

Program 23.3 *Output fra programmet.*

Call by reference parametre opnås når pointere overføres som *call by value* parametre.

Vi har allerede mange gange gjort brug af *call by reference parametre* i **`scanf`**.

Lad os påpege én ekstra ting. I f.eks. assignmentet `*p = *q` i program 23.2 skelner vi mellem *venstre værdien* og *højre værdien* af et udtryk. I den engelske litteratur omtales dette ofte som l-values og r-values. `*p` optræder som venstreside værdi i assignmentet. Ergo får vi med `*p` fat i den *plads*, som pointeren `p` peger på. `*q` optræder på højresiden af assignmentet. Her får vi med `*q` fat i den værdi, som *q peger på.

24. Pointers og arrays

Der er et tæt samspil mellem pointere og arrays i C. Dette er et udpræget kendetegn ved C. Samspillet gælder ikke generelt for andre programmeringssprog.

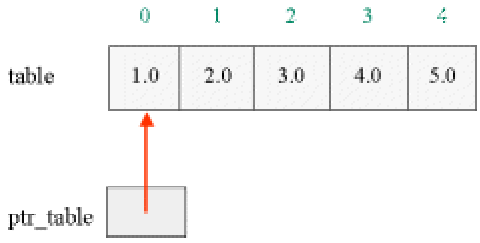
24.1. Pointers og arrays (1)

Lektion 6 - slide 12

Et array identificeres i C som en pointer til det første element.

Enhver operation der kan udføres med array subscripting kan også udføres med pointere.

I figur 24.1 illustrerer vi at `table` identificeres af en pointer til det første element i arrayet. For at illustrere dette klart har vi her lavet en ekstra variabel, `ptr_table`. Denne variabel er ikke nødvendigt idet udtrykket `&table[0]`, giver os den ønskede pointer. Husk her, at udtrykket `&table[0]` skal læses som adressen på plads 0 i `table`.



Figur 24.1 *Et array med fem elementer og en pointer til første element*

Hvis du skifter til 'slide view' af materiale (eller hvis du blot følger dette link) kommer du til en trinvis udvikling af figur 24.1. Følg de enkelte trin, og vær sikker på du forstår pointerens bevægelse hen over tabellen. I program 24.1 viser vi programmet for de trin, som gennemføres på figur 24.1 i den trinvise udvikling.

```
#include <stdio.h>

int main(void) {

  int i;
  double table[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
  double *ptr_table;

  ptr_table = &table[0];

  *ptr_table = 7.0;

  ptr_table = ptr_table + 1;
  *ptr_table = *(ptr_table-1) * 5;

  ptr_table += 3;
  (*ptr_table)++;

  printf("Distance: %i\n", ptr_table - &table[0]);
  for(i=0; i<5; i++) printf("%f ", table[i]);
  printf("\n");

  return 0;
}
```

Program 24.1 *Et komplet C program, som illustrerer opdateringerne af table.*

24.2. Pointers og arrays (2)

Lektion 6 - slide 13

Vi vil her se flere eksempler på elementær pointertilgang til elementer i et array.

I program 24.2 herunder har vi et array med fem reelle tal (doubles). I den blå del etablerer vi en pointer til det første element i arrayet. I den røde del bruger vi denne pointer til at ændre de første to elementer i tabellen. Udskriften i program 24.3 afslører de ændringer vi har lavet via pointeren `pa`.

```c
#include <stdio.h>

int main(void) {

  int i;
  double a[5] = {5.0, 7.0, 9.0, 3.0, 1.0}, *pa = &a[0];

  /* Print the elements in a */
  for(i = 0; i < 5 ; i++) printf("%f ", a[i]);  printf("\n");

  /* Modify elements in a via the pointer pa */
  *pa = 13.0;
  pa++;
  *pa = 19.5;

  /* Print the elements in a again */
  for(i = 0; i < 5 ; i++) printf("%f ", a[i]);  printf("\n");

  return 0;
}
```

Program 24.2 *Hele det ovenstående program.*

```
5.000000 7.000000 9.000000 3.000000 1.000000
13.000000 19.500000 9.000000 3.000000 1.000000
```

Program 24.3 *Output fra programmet.*

I vores motivation for arrays så vi på et simpelt program, som erklærer, initialiser og udskriver et array. Det var program 21.2. Herunder, i program 24.4 viser vi en alternativ version af program 21.2 som bruger pointere til gennemløb (initialisering og udskrivning) af elementerne. Outputtet af programmmet vises i program 24.5.

```c
#include <stdio.h>

#define TABLE_SIZE 11

int main(void) {

  double table[TABLE_SIZE], *table_ptr = &table[0], *tp;
  int i;

  for(tp = table_ptr, i=0; tp < &table[TABLE_SIZE]; tp++, i++)
    *tp = (double)(2 * i);

  for(tp = table_ptr; tp < &table[TABLE_SIZE]; tp++)
    printf("%f\n",  *tp);

  printf("\n");

  return 0;
}
```

Program 24.4 *Et program hvor tabellen `table` tilgås via en pointer. Med rødt og blåt har vi fremhævet de aspekter, som svarer til detaljerne med samme farver i det oprindelige array program: De røde erklæringer af tabellen, og de blå tilgange til array elementerne. Med brunt ser vi et udtryk som udgør betingelsen for afslutningen af forløkkerne. Med lilla ser vi en 'optælling' af pointeren.*

```
0.000000
2.000000
4.000000
6.000000
8.000000
10.000000
12.000000
14.000000
16.000000
18.000000
20.000000
```

Program 24.5 *Output fra programmet.*

24.3. Pointers og arrays (3)

Lektion 6 - slide 14

Inden vi ser på et 'rigtigt eksempel', bubblesort i afsnit 24.4, vil vi her fastslå en ækvivalens mellem array indeks notation og pointer (aritmetik) notation. Vi vender tilbage til pointeraritmetik i afsnit 24.5.

Lad ligesom i program 24.2 `a` være et array, og lad `pa` være en ekstra pointer variabel som peger på det første element i `a`:

- `*(pa+i)` og `a[i]` har samme værdi, nemlig tabellens `i`'te element
- `pa`, `a` og `&a[0]` har samme værdi, nemlig en pointer til det første element i arrayet

Når et array overføres som parameter til en funktion er det reelt en pointer til arrayet som overføres

Bubble sort programmet på næste side giver flere eksempler på dette.

24.4. Eksempel: Bubble sort

Lektion 6 - slide 15

Foreløbig slut med de helt små kunstige eksempler. Vi ser her på et program der kan sortere et array af heltal. Vi bruger bubblesort algoritmen, som er én af de forholdsvis velkendte sorteringsalgoritmer.

Boblesortering er en klassisk, men ikke særlig effektiv sorteringsalgoritme

Kernen i sorteringsalgoritmen vises herunder. `j` løber fra bagenden af vores array tilbage til `i`. `i` bliver gradvis større i takt med at den ydre forløkke gør fremskridt. Billedlig talt bobler de lette elementer (de mindste elementer) op mod forenden af tabellen.

```
void bubble(int a[] , int n)     /* n is the size of a[] */
{
   int   i, j;

   for (i = 0; i < n - 1; ++i){
     for (j = n - 1; j > i; --j)
       if (a[j-1] > a[j])
          swap(&a[j-1], &a[j]);
   }
}
```

Program 24.6 *Function* ***bubble*** *som laver 'bubble sort' på arrayet a som har n heltalselementer.*

Herunder, i program 24.7 ser vi et komplet program, hvori `bubble` funktionen fra program 24.6 er indsat næsten direkte. (Den brune del, kaldet af `prn_array`, er dog tilføjet).

I main ser vi med blåt et udtryk der måler størrelsen (antallet af elementer) i tabellen `a`. Læg godt mærke til hvordan dette sker. `sizeof` er en operator, som giver størrelsen af en variabel eller størrelsen af en type. Størrelsen læggers over i variablen `n`.

Med rødt viser vi kaldet af `bubble` på `a` og `n`. Tabellen `a` overføres som en reference til arrayet. Dette er fordi `a` i bund og grund blot er en pointer til tabellen, som omtalt i afsnit 24.1.

Forneden i program 24.7 ser vi funktionen `swap`, som vi udviklede i afsnit 23.1. `swap` laver de primitive skridt i sorteringsarbejdet, nemlig naboombytningerne i boble processen.

```
#include <stdio.h>

void   bubble(int a[], int n);
void   prn_array(char* s, int a[], int n);
void   swap(int *p, int *q);

int main(void)
{
   int   a[] = {7, 3, 66, 3, -5, 22, -77, 2};
   int   n;

   n = sizeof(a) / sizeof(int);
   prn_array("Before", a, n);
   bubble(a, n);                 /* bubble sort */
   prn_array(" After", a, n);
   putchar('\n');
   return 0;
}

void bubble(int a[] , int n)      /* n is the size of a[] */
{
   int   i, j;
```

```
   for (i = 0; i < n - 1; ++i){
     for (j = n - 1; j > i; --j)
       if (a[j-1] > a[j])
          swap(&a[j-1], &a[j]);
     prn_array("During", a, n);
   }
}

void prn_array(char* s , int a[] , int n)
{
   int   i;

   printf("\n%s%s%s", "   ", s, " sorting:");
   for (i = 0; i < n; ++i)
      printf("%5d", a[i]);
   putchar('\n');
}

void swap(int *p, int *q)
{
   int   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Program 24.7 *Hele 'bubble sort' programmet - med løbende udskrift af arrayet.*

På den slide, som er tilknyttet dette afsnit, har vi illustreret de enkelte skridt i boblesorteringen.

Det er let at set at boblesorteringsprogrammet fortager op til n^2 sammenligninger og ombytninger.

De gode sorteringsalgoritmer kan nøjes med i størrelsesordenen $n\ log(n)$ sammenligninger og ombytninger.

24.5. Pointeraritmetik

Lektion 6 - slide 16

Vi skal her se, at man kan bruge de aritmetiske operatorer i udtryk hvor der indgår pointere.

```
T *p, *q;
int i;
```

Program 24.8 *Erklæring af to pointere p og q samt en int i.*

Variablene `p`, `q` og `i` vist herover sætter scenen for punkterne herefter.

- Følgende former for pointeraritmetik er lovlige i C:
  - Assignment af pointere af samme type: `p = q`
  - Assignment af pointer til `NULL` eller 0: `p = 0` eller `p = NULL`
  - Addition og subtraktion af integer til pointer: `p + i` og `p - i`
  - Subtraktion af to pointere: `p - q`
  - Sammenligning af to pointere: `p < q`
  - Sammenligning af pointere med *0* eller `NULL`:
    `p == 0`, `p == NULL`, `p != NULL` eller `p > NULL`

Når vi i f.eks. `p + i` adderer et heltal til en pointer bliver pointeren optalt i logisk enheder, ikke i bytes. Med andre ord er C smart nok til at addere et passende antal bytes til `p`, som svarer til størrelsen af typen `T` i erklæringerne i program 24.8.

Vi viser et praktisk eksempel på pointeraritmetik i et program vi har lånt fra bogen 'The C Programming Language'. I funktionen `my_strlen` benytter det røde og det blå udtryk pointeraritmetik. Funktionen beregner antallet af tegn i en C tekststreng. Vi har ikke endnu diskuteret tekststrenge. Det vi ske i kapitel 27.

```
int my_strlen(char *s){
  char *p = s;

  while (*p != '\0')
    p++;

  return p-s;
}
```

Program 24.9 *En funktion som tæller antallet af tegn i en streng.*

I program 24.10 viser vi funktionen i konteksten af et komplet C program.

```c
#include <stdio.h>

/* Program from 'The C programming language' by
   Kernighan and Ritchie. */

#define STR1 "monkey"
#define STR2 "programming in C"

int my_strlen(char *s){
  char *p = s;

  while (*p != '\0')
    p++;

  return p-s;
}

int main(void) {

  printf("Length of \"%s\" = %i\n", STR1, my_strlen(STR1));
  printf("Length of \"%s\" = %i\n", STR2, my_strlen(STR2));

  return 0;
}
```

Program 24.10 *Hele programmet.*

24.6. Index out of bounds

Lektion 6 - slide 17

Når man programmerer med arrays er der altid fare for at man kan adressere uden for arraygrænserne. I de fleste sprog vil man få en fejlbesked fra det kørende program, hvis dette sker. I C skal man ikke regne med at få en sådan advarsel. Typisk vil man blot tilgå (læse eller skrive) data uden for arrayet. Dette kan have alvorlige konsekvenser.

> For et array **`a[N]`** er det programmørens ansvar at sikre, at indexes forbliver i intervallet **`[0..N-1]`**

Vi ser nedenfor på et typisk eksempel, hvor lidt uforsigtighed i en forløkke (det røde sted) betyder at vi adresserer `a[5]`. Husk på, at det kun giver mening at referere `a[0]`, ..., `a[4]`.

```c
double table[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

for(i = 0; i <= 5; i++)     /* index out of bounds for i = 5 */
{
  table[i] += 5.0;
  printf("Element %i is: %f\n", i, table[i]);
}
```

Program 24.11 *Et eksempel på indicering uden for indeksgrænserne i et array.*

Når man kører programmmet kan der ske lidt af hvert. Typisk vil man få fat i et bitmønster i `a[5]` (otte bytes), som fortolkes som et double. Dette kan give 'sjove' resultater.

> Det kørende C program opdager ikke nødvendigvis at indekset løber over den øvre grænse.
>
> Programmets opførsel er *udefineret* i sådanne tilfælde.

25. Arrays af flere dimensioner

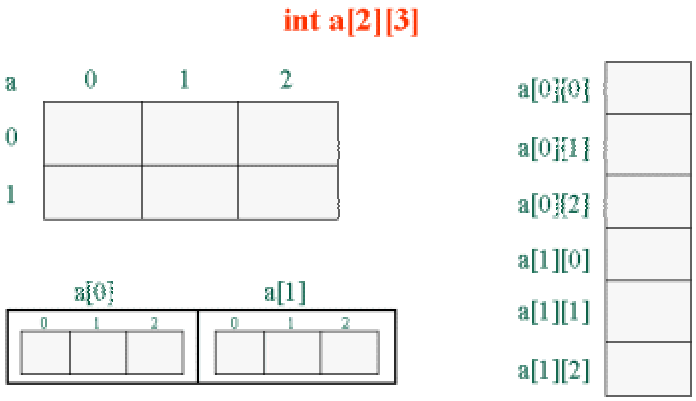
Efter at vi har set på arrays med elementer af simple typer vil vi nu studere arrays hvor elementerne er arrays. Sådanne arrays kan forstås som arrays af to eller flere dimensioner.

25.1. To dimensionelle arrays

Lektion 6 - slide 19

> Arrays af to dimensioner er et array hvor elementtypen er en anden arraytype

I figur 25.1 ses tre forskellige fortolkninger af `a[2][3]`. Øverst til venstre vises vores mentale model af en table med to dimensioner. Nederst til venstre vises *arrays i arrays* forståelsen, som er den korrekte begrebsmæssige forståelse i forhold til C. Til højre vises hvordan arrayet håndteres i lageret; Vi kan sige at arrayet er lineariseret. I forhold til den viste tabel af to dimensioner sker der en rækkevis lagring.



Figur 25.1 *Illustrationer af et to dimensionelt array. Øverst til venstre illustreres forståelse af de to dimensioner, med to rækker og tre søjler. Vi kunne naturligvis lige så godt have byttet om på rækker og søjler. Nederst til venstre ses en forståelse der understreger at a er et array med to elementer, som begge er arrays af tre elementer. Længst til højre ses den maskinnære forståelse, 'row major' ordningen.*

Vi tænker normalt på `a` som en to-dimensionel tabel med to rækker og tre søjler.

Reelt er elementerne lagret lineært og konsekutivt, i *row major* orden.

Elementer som kun afviger med én i det sidste indeks er naboelementer.

25.2. Processering af arrays med to dimensioner

Lektion 6 - slide 20

Næste emne er helt naturligt hvordan vi gennemløber et array af flere dimensioner.

Det er naturligt at bruge to indlejrede `for` løkker når man skal processere to dimensionelle arrays

Alternativt er det muligt at gennemløbe den samlede tabel lineært, med udgangspunkt i en pointer til det første element.

De to gennemløb omtalt ovenfor er illustreret i program 25.1.

```
double a[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}};

for (i = 0; i < 2; i++)
  for (j = 0; j < 3; j++)
    sum1 += a[i][j];

for (k = 0; k < 2 * 3; k++)
  sum2 += *(&a[0][0] + k);
```

Program 25.1 *Erklæring, initialisering og to identiske gennemløb af et to dimensionelt array.*

For god ordens skyld viser vi program 25.1 som et komplet C program i program 25.2.

```c
#include <stdio.h>

int main(void) {

  int i, j, k;
  double sum1 = 0.0, sum2 = 0.0;
  double a[2][3] = {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}};

  for (i = 0; i < 2; i++)
    for (j = 0; j < 3; j++)
      sum1 += a[i][j];

  for (k = 0; k < 2 * 3; k++)
    sum2 += *(&a[0][0] + k);

  printf("First method:  the sum is: %f\n", sum1);
  printf("Second method: The sum is: %f\n", sum2);

  return 0;
}
```

Program 25.2 *Det samlede program.*

C understøtter arrays af vilkårlig mange dimensioner

Tre- og flerdimensionelle arrays er umiddelbare generaliseringer af to dimensionelle arrays

26. Statisk og dynamisk lagerallokering

Variable kræver lagerplads når programmet kører. Vi siger, at der *allokeres lager* til vore vore variable. Når vi i kildeprogrammet har erklæret en variabel vil køretidssystemet klare lagerallokeringen af variablen uden yderligere programøranstrengelser. Dette er *statisk lagerallokering*.

I C er det også muligt, når programmet kører, at bede om lager, som ikke er indeholdt i nogen variable. Når dette sker vil vi få overdraget en pointer til begyndelsen af det pågældende lagerområde. Dette kaldes *dynamisk lagerallokering*.

26.1. Statisk lagerallokering

Lektion 6 - slide 22

I kurset indtil nu har vi benyttet *statisk lagerallokering* for alle data

Med *statisk lagerallokering* afsættes lagerplads implicit når den omkringliggende blok aktiveres. Lagerplads inddrages implicit når blokken forlades.

I programmet herunder allokeres `tab1` og `tab2` statisk i `main`. Til `tab1` afsættes plads til `N` doubles. `N` er en symbolsk konstant, som er defineret til 500. Til `tab2` afsættes plads til `N*K` doubles, altså konkret 500 doubles (eller typisk 4000 bytes). Der allokeres naturligvis også lager til `i`.

```
#include <stdio.h>
#define N  500
#define K   10

int main(void) {
  int i;
  double tab1[N];       /* allocation of N doubles */
  double tab2[N*K];     /* allocation of N*K doubles */

  /* do something with tab1 and tab2 */
  return 0;
}
```

Program 26.1 *Illustration af statisk lagerallokering i C.*

Når `main` er kørt færdig frigives lagerpladsen, som er afsat til `tab1`, `tab2` og `i`.

Statisk lagerallokering er knyttet til erklæringen af variable.

Udtryk som indgår i indeksgrænser for et array skal være *statiske udtryk*

26.2. Dynamisk lagerallokering (1)

Lektion 6 - slide 23

I mange tilfælde ved vi først på programmets køretid hvor meget lager vi har brug for. Dette kan f.eks. afhænge af et tal, som er indlæst tidligere i programkørslen. Derfor er dynamisk lagerallokering ikke let at komme uden om.

Der er ofte behov for en mere *fleksibel* og *dynamisk* form for lagerallokering, hvor det kørende program eksplicit allokerer lager.

Med *dynamisk lagerallokering* afsættes lagerplads eksplicit, når programmet har behov for det.

I program 26.2 vises et forsøg på at at allokere passende plads til `a` i forhold til `n`, som er et indlæst heltal.

Ideen med at indlæse `n` i en ydre blok i forhold til den blok, hvor arrayet allokeres er god. Men vi skal ikke regne med, at den virker i C. ANSI C kræver, at `n` er et konstant udtryk, hvilket vil sige at der kun må indgå konstanter i udtrykket. `sizeof` operatoren kan også indgå.

```
#include <stdio.h>

int main(void) {

  int n;
  printf("Enter an integer size:\n");
  scanf("%d",&n);

  { int i;
    double a[n];

    for (i = 1; i < n; i++)
      a[i] = (double)i;

    for (i = 1; i < n; i++)
      printf("%f ", a[i]);
    printf("\n");

  }

  return 0;
}
```

Program 26.2 *Et kreativt forsøg på dynamisk allokering - ikke ANSI C.*

I stedet for at bruge ideen i program 26.2 skal vi anvende enten `malloc` eller `calloc`, som eksemplificeret i program 26.3. På det røde sted allokerer vi plads til `n` heltal (`int`s). Vi får en pointer til lagerområdet. Læg mærke til at `a` er en pointer til en `int`. Det nyallokerede lagerområde bruges i de to efterfølgende forløkker. På det blå sted deallokerer vi igen lagerområdet. Det betyder at vi overgiver lageret til C køretidssystemet, som så efterfølgende kan bruge det til andre formål. F.eks. kan lageret genbruges hvis vi lidt efter igen beder om nyt laver via `calloc`.

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

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

   printf("\n%s",
      "An array will be created dynamically.\n\n"
      "Input an array size n followed by n integers:  ");
   scanf("%d", &n);
   a = calloc(n, sizeof(int));  /* get space for n ints */
   for (i = 0; i < n; ++i)
      scanf("%d", &a[i]);
   for (i = 0; i < n; ++i)
      sum += a[i];
   free(a);                     /* free the space */
   printf("\n%s%7d\n%s%7d\n\n",
      " Number of elements:", n,
      "Sum of the elements:", sum);
   return 0;
}
```

Program 26.3 *Et program der foretager dynamisk allokering af et array.*

Vi viser afslutningsvis et program, som dynamisk allokerer plads til det array, som vi illustrerede i figur 25.1. Allokeringen sker på det røde sted. I dette eksempel frigiver vi ikke lagerpladsen, idet lageret bruges helt til programafslutningen.

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

#define N 2
#define M 3

int main(void) {
  int *pint, i, j;

  pint = malloc(N*M*sizeof(int));

  for (i=0; i<N; i++)
   for (j=0; j<M; j++)
     *(pint + M*i + j) = (i+1) * (j+1);

  for (i = 0; i < M * N; i++){
    printf("%4i ", *(pint + i));
    if ((i+1) % M == 0) printf("\n");
  }

  return 0;
}
```

Program 26.4 *Et program der dynamisk allokerer et to-dimensionelt array.*

26.3. Dynamisk lagerallokering (2)

Lektion 6 - slide 24

Herunder opsummerer vi dynamisk lagerallokering.

- Dynamisk lagerallokering
  - Eksplicit *allokering* med **`calloc`** eller **`malloc`** fra fra **`stdlib.h`**
  - Eksplicit *deallokering* med funktionen **`free`**
  - Risiko for *dangling references*
    - Hvis en pointer til et dynamisk allokeret objekt følges efter at objektet er deallokeret med **`free`**

I mange moderne sprog styres lager deallokering af en *garbage collector*, som frigiver lagerplads, når det ikke længere kan tilgås af programmet.

Understøttelse af lageradministration med en garbage collector er et meget spændende emne. En diskussion af dette falder dog uden for rammerne af disse noter.

27. Tekststrenge i C

Denne lektionen er om tekststrenge, som både har berøringspunkter med arrays og pointers. Programmering med tekststrenge er vigtig i enhver form for tekstbehandling. Lektionen udspænder kapitel 27 - kapitel 31.

27.1. Strenge og tekststrenge

Lektion 7 - slide 2

Vi starter forholdsvis generelt med at definere strenge og tekststrenge.

En *streng* er en sekvens af data af samme type

En *tekststreng* er en streng af datatypen tegn

I de fleste programmeringssprog noteres tekststrenge ved brug af dobbelte anførselstegn. Dermed adskiller strenge sig fra tegn (se afsnit 16.3) der i mange sprog noteres ved enkelte anførselstegn. Se endvidere afsnit 27.6.

- Almen notation for tekststrenge:
  - `"En tekststreng"`
- Den tomme streng
  - Strengen der ikke indeholder data
  - Strengen af længde *0* - den kortest mulige streng
  - Den tomme tekststreng noteres naturligt som `""`

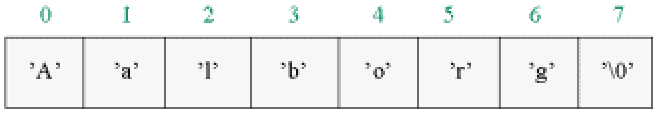
27.2. Tekststrenge og arrays

Lektion 7 - slide 4

Efter den korte, generelle introduktion til tekststrenge i afsnit 27.1 ser vi nu konkret på tekststrenge i C. Den vigtigste detalje er konventionen om, at der skal bruges et særligt tegn, nultegnet, for at afslutte en tekststreng. Nultegnet er det tegn i ASCII alfabetet (afsnit 16.1 og program 16.2) som har talværdien 0. Det er med andre ord det første tegn i tabellen.

En tekststreng i C er et *nulafsluttet* array med elementtypen `char`.

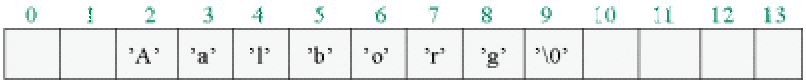
I figur 27.1 herunder ser vi tekststrengen `"Aalborg"`, som det repræsenteres i et array. Læg mærke til tegnet i 7'ende position: nul tegnet. Uden dette tegn ved vi ikke hvor tekststrengen afsluttes i lageret.



Figur 27.1 *En illustration af en nulafsluttet tekststreng i C*

- Det afsluttende nultegn kaldes en *sentinel*
  - Oversættes direkte som en 'vagt'
  - Eksplicit markering af afslutningen af tekststrengen
  - Som et alternativt kunne vi holde styr på den aktuelle længde af tekststrengen

Herunder, i figur 27.2 viser vi tekststrengen `"Aalborg"` placeret midt i et array. Med en `char` pointer som peger på celle 2 i tabellen har vi stadig fat i `"Aalborg"`.



Figur 27.2 *En tekststreng som er placeret 'i midten' af et array of char*

En tekststreng i C skal ikke nødvendigvis udfylde hele det omkringliggende array

27.3. Initialisering af tekststrenge

Lektion 7 - slide 5

Der er flere forskellige måder at initialisere en tekststreng. Herunder, i program 27.1 viser vi essentielt tre forskellige. Den røde måde benytter manifest array notation, som blev diskuteret i program 21.3 i afsnit 21.2.

Den blå og den lilla måde er overfladisk set sammenfaldende. Her benytter vi en strengkonstant i dobbeltquotes, som diskuteret i afsnit 27.1. Den blå måde bruger en array variabel, og den lilla en pointervariabel. I afsnit 27.5 diskuterer vi nogle vigtige forskelle mellem den blå og lilla form, hvad angår mulighederne for mutation af tegn i strengene.

Med den brune måde indsætter vi enkelttegn i et array pr. assignment.

Læg mærke til at den røde og brune måde kræver eksplicit indsættelse af nultegnet. Nultegnet skal ikke indsættes eller noteres eksplicit, når vi bruger strengkonstanter.

```
char str_1[] = {'A', 'a', 'l', 'b', 'o', 'r', 'g', '\0'};

char str_2[] = "Aalborg";

char *str_3 = "Aalborg";

char str_4[8];
str_4[0] = 'A';  str_4[1] = 'a';  str_4[2] = 'l';
str_4[3] = 'b';  str_4[4] = 'o';  str_4[5] = 'r';
str_4[6] = 'g';  str_4[7] = '\0';
```

Program 27.1 *Forskellige initialiseringer af tekststrenge.*

Ved initialisering via tekstkonstanter tilføjes nultegnet automatisk af compileren

27.4. Tekststrenge og pointere

Lektion 7 - slide 6

Vi skal nu se på forholdet mellem tekststrenge og pointere. I bund og grund er dette forhold afledt af at en tekststreng er et array (afsnit 27.2), og at et array er en pointer til det første element i arrayet (afsnit 24.1).

En tekststreng opfattes i C som en pointer til det første tegn

Dette følger direkte af den generelle sammenhæng mellem arrays og pointere

Herunder, i program 27.2 ser vi et eksempel på et program, der tilgår tekststrenge via pointere. Overordnet set kopierer programmet tegnene i `"Aalborg"` ind midt i strengen `str`.

```
#include <stdio.h>

int main(void) {
  char str_aal[] = "Aalborg";
  char str[14];

  char *str_1, *str_2;
  int i;

  /* fill str with '-' */
  for(i = 0; i < 14; i++)
    str[i] = '-';

  /* let str_1 and str_2 be running pointers */
  str_1 = str;  str_2 = str_aal;

  /* copy str_all into the middle of str */
  str_1 += 2;
  for( ; *str_2 != '\0'; str_1++, str_2++)
```

```
    *str_1 = *str_2;

  /* terminate str */
  *str_1 = '\0';

  printf("%s\n", str);

  return 0;
}
```

Program 27.2 *Et program der ved brug af pointere kopierer strengen "Aalborg" ind midt i en anden streng.*

Lad os forklare programmet i lidt større detalje. I starten af `main` ser vi to variable `str_all` og `str`, som er strenge. Vi initialiserer `str_all` til `"Aalborg"`. `str` initialiseres i den første forløkke til udelukkende at indeholde tegnene `'-'`.

Dernæst assignes `str_1` og `str_2` til pointere til de første tegn i hhv. `str` og `str_all`. Med rødt ser vi at `str_1` føres to positioner frem i arrayet med `'-'` tegn. Dernæst kommer en forløkke, som kopierer tegnene i `str_all` over i `str`. Dette sker gennem de pointere vi har sat op til at gennemløbe disse arrays. Før vi udskriver `str` med `printf` indsætter vi eksplicit et `'\0'` tegn i `str`.

Herunder, i program 27.3 ser vi outputtet fra program 27.2.

```
--Aalborg
```

Program 27.3 *Output fra programmet.*

27.5. Ændringer af tekststrenge

Lektion 7 - slide 7

Vi arbejder i dette afsnit videre på program 27.1. Det er vores mål at forklare under hvilke omstændigher vi kan ændre i de tekststrenge, som indgår i program 27.1.

Tekststrenge, til hvilke der er allokeret plads i et array, kan ændres (muteres)
Tekststrenge, der er angivet som en strengkonstant refereret af en pointer, kan ikke ændres

Mutation er et biologisk inspireret ord, som betyder *at ændre (i en del af) noget*. Ordet benyttes ofte når vi foretager ændringer i en del af en eksisterende datastruktur, eksempelvis et array.

Vores pointe bliver illustreret i program 27.4. Svarende til de farvede aspekter i program 27.1 foretager vi fire ændringer i strengen `"Aalborg"`.

Kun den lilla ændring giver problemer. `str_3` peger på en tekststreng, som ikke er allokeret som et array i det program vi har skrevet. `str_3` peger derimod på en konstant tekststreng, som er en del af programmet. Denne tekststreng er lagret sammen med programmet, på et sted vi ikke kan ændre i strengen.

```
char str_1[] = {'A', 'a', 'l', 'b', 'o', 'r', 'g', '\0'};
*(str_1 + 1) = 'A';

char str_2[8] = "Aalborg";
*(str_2 + 1) = 'A';

char *str_3 = "Aalborg";
*(str_3 + 1) = 'A';                          /* Ulovligt */

char str_4[8];
str_4[0] = 'A';  str_4[1] = 'a';  str_4[2] = 'l';
str_4[3] = 'b';  str_4[4] = 'o';  str_4[5] = 'r';
str_4[6] = 'g';  str_4[7] = '\0';
*(str_4 + 1) = 'A';
```

Program 27.4 *Et program der ændrer det andet tegn i Aalborg fra 'a' til 'A'.*

Moralen vi kan uddrage af ovenstående er at mutation af en tekststreng kræver at denne tekststreng er allokeret statisk (afsnit 26.1) eller dynamisk (afsnit 26.2) i programmet. Vi kan ikke ændre i konstante tekststrenge, til hvilke vi kun har en pointer, og som er angivet i selve programmet.

27.6. Tekststrenge i forhold til tegn

Lektion 7 - slide 8

Vi vil her påpege en væsentlig forskel mellem notationen for tegn og tekststrenge.

> Tekststrenge er sammensat af tegn
>
> De to værdier `'a'` og `"a"` meget forskellige

Forskellen på notationen for tegn og tekststrenge kan eksemplificeres som følgende:

- `'a'`
  - Et enkelt tegn af typen char
  - Reelt heltallet *97*
- `"a"`
  - Et array med to elementer
  - Nulte element er tegnet `'a'` og første element er tegnet `'\0'`

27.7. Den tomme streng og NULL

Lektion 7 - slide 9

I lighed for forskellen på `'a'` og `"a"` vil vi nu pege på forskellen mellem `NULL` og den tomme tekststreng.

> Man skal kunne skelne mellem *den tomme tesktstreng* og en **`NULL`** pointer
>
> **`NULL`** og `""` er meget forskellige

NULL og den tomme streng kan karakteriseres som følger:

- **`NULL`**
  - **`NULL`** er en pointer værdi
  - **`NULL`** værdien bruges for en pointer, der ikke peger på en plads i lageret
  - Reelt heltallet 0
- Den tomme streng `""`
  - Den tomme string `""` er en streng værdi
  - `""` er et array med ét element, nemlig `'\0'` tegnet

28. Leksikografisk ordning

Dette kapitel handler om ordning og lighed mellem strenge.

28.1. Leksikografisk ordning af strenge

Lektion 7 - slide 11

En ordning af tekststrenge kan afledes af ordningen blandt de tegn, som tekststrengen er bygget op af. I dette afsnit diskuterer vi ordningen af tekststrenge på det generelle plan, uafhængig af C.

> Ordningen af tegn inducerer en ordning af tegnstrenge (tekststrenge)
>
> Den normale alfabetiske ordning af tekststrenge spiller en vigtig rolle ved opslag i leksika, telefonbøger, mv.
>
> Vi vil her definere hvad det betyder af strengen *s* ***er mindre end*** strengen *t*

Herunder definerer vi betydningen af at en streng *s* er mindre end en streng *t*.

- Lad *e* betegne den tomme streng `""` og lad *s* og *t* være to tekststrenge
- *s* < *t* hvis og kun hvis der findes tegn **`c`** og **`d`** samt to kortere strenge *u* og *v* så
    - *s* = *e*
      t = **`c`** *u* *eller*
    - *s* = **`c`** *u*
      t = **`d`** *v*
        - **`c`** < **`d`**
        - **`c`** = **`d`** og *u* < *v*

Der er to tilfælde i definitionen. Første tilfælde siger at den tomme streng er mindre end enhver ikke-tom streng.

Det andet tilfælde har to undertilfælde, som begge udtaler sig om to ikke-tomme strenge *s* og *t*. Det første undertilfælde siger at *s* < *t* hvis første tegn i *s* er mindre end første tegn i *t*. Det andet undertilfælde siger at hvis første tegn i *s* er lig med første tegn i *t*, afgøres vurderingen *s* < *t* rekursivt af *u* < *v*, hvor *u* er halen af *s* og *v* er halen af *t*.

Lad os se på nogle eksempler. De følgende fire udsagn er alle sande.

1. `"Andersen" < "Hansen"`
2. `"abe" < "kat"`
3. `"abehat" < "abekat"`
4. `"abe" < "abekat"`

Det første tilfælde forklares ved at `'A' < 'H'`.

Det andet tilfælde argumenteres ganske tilsvarende, nemlig ved at `'a' < 'k'`.

I det tredje tilfælde skræller vil tegnene `'a'`, `'b'`, og `'e'` af begge strenge og vurderer dermed `"hat"` i forhold til `"kat"`.

I det sidste tilfælde skræller vi også tegnene `'a'`, `'b'`, og `'e'` af begge strenge og vurderer den tomme streng `""` i forhold til `"kat"`.

28.2. Lighed af strenge (1)

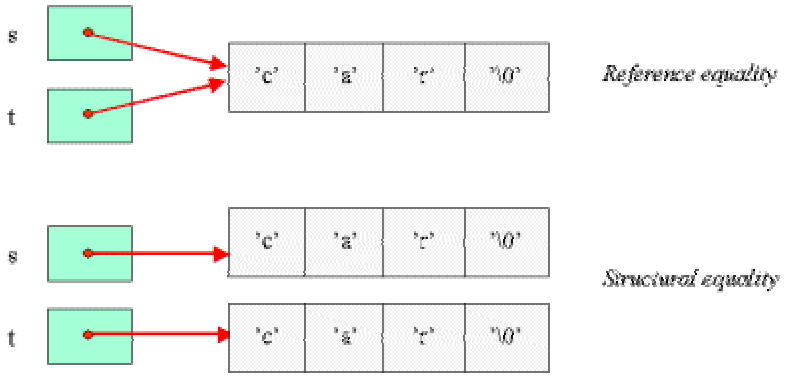
Lektion 7 - slide 12

I dette afsnit ser vi på et eksempel, som generelt illustrerer at lighed mellem to ting kan have flere forskellige betydninger.

Lighed mellem skalartyper (specielt aritmetiske typer), jf. afsnit 18.1, volder sjældent problemer. Lighed mellem sammensatte typer, såsom arrays og strenge, kan have flere forskellige betydninger.

Den lære vi drager i dette afsnit for strenge kan generaliseres til andre sammensatte datastrukturer, f.eks. lister (afsnit 41.1) og records/structs (kapitel 39).

Der er to mulige fortolkninger af lighed af to strenge *s* og *t*



Figur 28.1 *En illustration af reference lighed og strukturel lighed mellem* **`char *`** *variable*

- Pointerværdierne indeholdt i de to variable **`s`** og **`t`** er ens
  - **`s`** og **`t`** peger på det samme array af tegn
  - Hvis vi ændrer på tegnene i **`s`** ændres tegnene i **`t`** automatisk
  - *Reference lighed*
- **`s`** og **`t`** udpeger forskellige arrays, med tegn der parvis er ens
  - Hvis vi ændrer på tegnene i **`s`** ændres intet i **`t`**
  - *Strukturel lighed*

Hvis to strenge er *reference ens* er de også *strukturelt ens*, men ikke nødvendigvis omvendt

Ovenstående observation følger naturligt ved at se på figur 28.1. Hvis to strenge `s` og `t` er reference ens vil det naturligvis være tilfældet at tegnene i `s` og `t` er parvis ens. Hvis derimod to strenge `s` og `t` er strukturelt ens, kan vi ikke sluttet at `s` og `t` udpeges af identiske pointere.

28.3. Funktionen strcmp fra string.h

Lektion 7 - slide 14

Funktionen `strcmp` fra standard biblioteket i C er nyttig i forbindelse med vurdering af leksikografisk ordning og strukturel lighed når vi programmerer i C.

Functionen **`strcmp`** fra **`string.h`** implementerer den leksikografiske ordning samt

strukturel lighed på tekststrenge i C

- Tre mulige output af **`strcmp(str1,str2)`**:
  - ***Positivt heltal***: **`str1`** > **`str2`**
  - ***Nul***: **`str1`** = **`str2`** (**`str1`** og **`str2`** er ens i strukturel forstand)
  - ***Negativt heltal***: **`str1`** < **`str2`**

Det vil ofte give bedre programmer hvis vi vurderer to strenge med en funktion, som giver et boolsk (true/false) resultat. Med sådanne kan vi skrive `string_leq(s,t)` i stedet for `strcmp(s,t) == -1`. Og i en selektiv kontrolstruktur kan man skrive

```
if (string_equal(s,t))
  ...
else
  ...
```

i stedet for

```
if (strcmp(s,t) == 0)
  ...
else
  ...
```

Hvis vi ønsker funktioner ala `string_leq` og `string_equal` må vi selv programmere dem, hvilket naturligvis er ganske enkelt.

Vi kender ikke de konkret returværdier af et kald af `strcmp`. Returværdien er altså implementationsafhængig. Dette illustreres klart i program 28.2.

Programmet herunder, program 28.1, viser eksempler på leksikografisk ordning målt med `strcmp`.

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

void pr(char*, char*, int);

int main(void) {

  int b1 = strcmp("book", "shelf");
  int b2 = strcmp("shelf", "book");
  int b3 = strcmp("","book");
  int b4 = strcmp("book","bookshelf");
  int b5 = strcmp("book", "book");
  int b6 = strcmp("7book", "book");
  int b7 = strcmp("BOOK", "book");

  pr("book", "shelf", b1);
```

```
  pr("shelf", "book", b2);
  pr("", "book", b3);
  pr("book", "bookshelf", b4);
  pr("book", "book", b5);
  pr("7book","book", b6);
  pr("BOOK", "book", b7);

  return 0;
}

void pr(char *s, char *t, int r){
 printf("strcmp(\"%s\",\"%s\") = %i\n", s,t,r);
}
```

Program 28.1 *Et program der illustrerer den leksikografiske ordning af tegn i C.*

Output fra program 28.1 kan ses i program 28.2.

```
strcmp("book","shelf") = -17
strcmp("shelf","book") = 17
strcmp("","book") = -98
strcmp("book","bookshelf") = -115
strcmp("book","book") = 0
strcmp("7book","book") = -43
strcmp("BOOK","book") = -32
```

Program 28.2 *Output fra programmet.*

29. Tidligere eksempler

I dette kapitel vil vi genprogrammere et par eksempler fra hhv. afsnit 16.10 og afsnit 18.3.

Vores ærinde er at illustrere hvordan vi kan drage fordel af at overføre tekststrenge som parametre og af at returnere en tekststreng som resultatet af en funktion. Dette vil ofte give mere generelle og mere genbrugbare funktioner.

29.1. Konvertering mellem talsystemer

Lektion 7 - slide 16

I en tidligere lektion har vi programmeret funktioner der konverterer tal mellem forskellige talsystemer

De involverede funktioner læste/skrev input/output med **`getchar`** og **`putchar`**

Det er mere alsidigt at skrive funktioner, der modtager eller returnerer tekststrenge

Herunder, i program 29.1 viser vi en funktion, som konverterer et tal `n` i talsystemet med basis `base` til et decimalt tal. Da tal i f.eks. 16-talsystemet kan gøre brug af alfabetiske cifre ala `'a'` og `'f'` kan vi ikke overføre et heltal som parameter til `to_decimal_number`. Det er derfor bekvemt at overføre en tekststreng, som opfattes som et tal i `n`-talsystemet. Den oprindelige version af funktionen kan ses i program 16.8, `read_in_base`.

```
/* Convert the string n to a decimal number in base and return it.
   Assume that input string is without errors */
int to_decimal_number(char *n, int base){
  int ciffer_number, res = 0;
  char *ciffer_ptr = &n[0], ciffer = *ciffer_ptr;

  do {
    if (ciffer >= '0' && ciffer <= '9')
      ciffer_number = ciffer - '0';
    else if (ciffer >= 'a' && ciffer <= 'z')
      ciffer_number = ciffer - 'a' + 10;
    else ciffer_number = -1;   /* error */

    if (ciffer_number >= 0 && ciffer_number < base)
      res = res * base + ciffer_number;

    ciffer_ptr++; ciffer = *ciffer_ptr;
  }
  while (ciffer != '\0');

  return res;
}
```

Program 29.1 *En funktion der konverterer et tal `n` i `base` talsystemet (en streng) til et decimalt tal.*

I program 29.1 ser vi på det røde sted at `n` er en tekststreng `(char *)`. På det blå sted initialiserer vi en pointer, `ciffer_ptr`, der skal bruges til at løbe igennem strengen `n`. Vi dereferencer straks `ciffer_ptr`, og lægger tegnet over i `char` variablen `ciffer`. Vi bruger `ciffer` i en formel (udtryk), som svarer til formlen i program 16.8. Den brune del opdaterer `ciffer_ptr` og `ciffer`. Den lilla del holder udkig efter nulafslutningen i `n`.

I program 29.2 ses funktionen `to_decimal_number` i et komplet C program. Læg specielt mærke til hvordan vi kalder `to_decimal_number` i `printf`.

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

int read_in_base(int);

int main(void) {
  int i, n, base;
  char *the_number[20];

  for (i = 1; i <= 5; i++){
    printf("Enter number base (a decimal number)"
           "and a number in that base: ");
    scanf("%d %s", &base, the_number);
```

```
    printf("The decimal number is: %d\n",
            to_decimal_number(the_number, base));
  }

  return 0;
}

/* Convert the string n to a decimal number in base and return it.
   Assume that input string is without errors */
int to_decimal_number(char *n, int base){
  int ciffer_number, res = 0;
  char *ciffer_ptr = &n[0], ciffer = *ciffer_ptr;

  do {
    if (ciffer >= '0' && ciffer <= '9')
      ciffer_number = ciffer - '0';
    else if (ciffer >= 'a' && ciffer <= 'z')
      ciffer_number = ciffer - 'a' + 10;
    else ciffer_number = -1;   /* error */

    if (ciffer_number >= 0 && ciffer_number < base)
      res = res * base + ciffer_number;

    ciffer_ptr++; ciffer = *ciffer_ptr;
  }
  while (ciffer != '\0');

  return res;
}
```

Program 29.2 *Hele programmet.*

29.2. Udskrivning af enumeration konstanter

Lektion 7 - slide 17

I proceduren `prnt_day`, program 18.7, så vi hvordan vi kan lave og udskrive værdierne af enumerators. Herunder, i program 29.3, ser vi hvordan vi kan få hånd på *print værdien* af en enumerator via funktionen `print_name_of_day`. Læg mærke til at `print_name_of_day` returnerer en streng, hvorimod `prnt_day` fra program 18.7 returnerer `void`.

I en tidligere lektion har vi programmeret funktioner der udskriver navnene på enumeration konstanter

De involverede funktioner skrev output med **`printf`**

Det er mere alsidigt at skrive funktioner, returnerer tekststrenge

```c
/* Return the symbolic name of day d */
char *print_name_of_day(days  d){
  char *result;
  switch (d) {
    case sunday: result = "Sunday";
       break;
    case monday: result = "Monday";
       break;
    case tuesday: result = "Tuesday";
       break;
    case wednesday: result = "Wednesday";
       break;
    case thursday: result = "Thursday";
       break;
    case friday: result = "Friday";
       break;
    case saturday: result = "Saturday";
       break;
  }
  return result;
}
```

Program 29.3 *En funktion som returnerer en symbolsk ugedag (en streng) givet dagens nummer.*

30. Biblioteket string.h

I standard biblioteket findes en række funktioner, der arbejder på tekststrenge. Knap 20 ialt. Du skal inkludere `string.h` for at få adgang til disse. Læs om streng funktionerne i appendix A i *C by Dissection*

30.1. Sammensætning af strenge

Lektion 7 - slide 19

Dette afsnit handler om `strcat` og `strcpy`. Fælles for disse og tilsvarende funktioner er den koncise, korte, og lidt kryptiske navngivning, som er en reminicens fra både C og Unix traditionen.

> Funktionen **`strcat`** indsætter en streng i bagenden af en anden streng forudsat der er plads efter nultegnet

Vi viser herunder, i program 30.1, et program, som benytter `strcat` til at sammensætte strengene `"Aalborg"` og `"University"` til `"Aalborg University"`. Læg mærke til at vi bliver nødt til at sammesætte de to bestanddele med et eksplit mellemrum: `" "`.

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

int main(void) {

  char s[25] = "Aalborg";
  char t[] = "University";
  char *res;

  res = strcat(strcat(s," "),t);
  printf("%s: %i chars\n", res, strlen(s));

  return 0;
}
```

Program 30.1 *Et program der illustrerer* **`strcat`** *og* **`strlen`**.

Strengen `s` indeholder god plads til `"Aalborg University"`: 25 tegn ialt. Med to indlejrede kald af `strcat`, på det røde sted, sammensættes strengene til det ønskede resultat. På det blå sted ses et kald af `strlen`, som tæller antallet af tegn i en streng (ikke medregnet nultegnet).

Herunder observerer vi hvordan `strcpy` virker i forhold til `strcat`.

Funktionen **`strcpy`** ligner **`strcat`**, blot indsætter den en streng i begyndelsen af en anden streng

30.2. En alternativ funktion til strcpy

Lektion 7 - slide 20

Som beskrevet ganske kort i afsnit 30.1 kopierer `strcpy` tegnene fra den anden parameter ind i strengen, som overføres i den første parameter.

Som de fleste andre funktioner i **`string.h`** allokerers der ikke lager i **`strcpy`**

Her vil vi programmere en streng-kopierings funktion der allokerer plads til en ny kopi

Vort udgangspunkt er således en observation om, at `strcpy` arbejde inden for allerede allokeret lager. Dette er et generelt kendetegn for mange funktioner i C standard library.

Herunder, i program 30.2 programmerer vi en streng kopieringsfunktion, som allokerer ny og frisk plads til den streng, som overføres som parameter. Den eksisterende streng kopieres over i det nyallokerede lager, således at den eksisterende streng og den nye streng er strukturelt ens (jf. afsnit 28.2).

```
/* Copy s to a fresh allocated string and return it */
char *string_copy(const char *s){

 static char *new_str;

 new_str = (char *)malloc(strlen(s)+1);
 strcpy(new_str,s);

 return new_str;
}
```

Program 30.2 *Et funktioner der allokerer plads til og kopierer en streng.*

På det røde sted ser vi selve lagerallokeringen. Vi bruger dynamisk lagerallokering med `malloc`, jf. afsnit 26.2. Vi assigner pointeren `new_str` til det nye lagerområde. Dernæst foretages kopieringen af `s` over i `new_str` med `strcpy`. Vi erklærer `new_str` som `static` (det lilla sted), idet `new_str` skal returneres som resultat fra funktionen `string_copy`. Dette er ikke strengt nødvendigt, men dog en god ide når vi returnerer en array fra en funktion.

Herunder ser vi på en variant af `str_copy` fra program 30.2. I denne variant er det essentielt at vi på det røde sted erklærer `new_str` som `static`. Årsagen er, at vi i denne variant af programmet statisk har allokeret lokalt lager på køretidsstakken, som ønskes 'med ud fra funktionen' pr. returværdi. I program 30.2 allokerede vi dynamisk lager i det område, vi plejer at benævne som *heapen*.

Hvis vi glemmer 'static' i erklæringen af `new_str` i program 30.3 vil vi få en *dangling reference*, jf. afsnit 26.3. Compileren vil muligvis advare dig, hvis du glemmer `static`.

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


/* Copy s to a locally allocated string and return it */
char *string_copy(const char *s){

 static char new_str[8];   /* static is essential */
 strncpy(new_str,s,7);

 return new_str;
}

int main(void) {

  char s[] = "Aalborg University", *t;
  t = string_copy(s);
  strcpy(s,"---");      /* destroy s */

  printf("The original is: %s.\nThe copy is: %s\n", s, t);

  return 0;
}
```

Program 30.3 *En variant af string_copy der opbygger kopien i lokalt lager. Vi har valgt kun at kopiere 7 tegn.*

Som en mindre detalje i program 30.3 overskriver vi på det sorte sted delvist `s`, som lige er kopieret over i `t`. Dermed kan vi erkende, at `s` og `t` ikke er lig med hinanden i reference forstand, jf. afsnit 28.2.

30.3. Substring ved brug af strncpy

Lektion 7 - slide 21

Vi viser i dette afsnit en typisk anvendelse af `strncpy`, som udtrækker en delstreng af en tekststreng. Bemærk at det ikke er `strcpy`, men `strncopy` vi diskuterer.

> Der er ofte behov for at udtrække en substreng af en anden streng
>
> Dette kan gøres med **`strncpy`**

Strengen vi betragter er `str`, som indeholder `"The Aalborg University basis year"`. På det sorte sted kopierer vi et antal `'\0'` tegn ind i strengen `target`. Dette sikrer i den sidste ende nulafslutningen af `target`. Det røde sted er central for dette eksempel. Bemærk først udtrykket `str+4`, som udpeger `"Aalborg University basis year"`. Den sidste parameter, 18, i kaldet af `strncpy` kopierer netop 18 tegn. Samlet set kopieres `"Aalborg University"` over i `target`.

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

#define LEN 25

int main(void) {

  char str[] = "The Aalborg University basis year";
  char target[25];
  int i;

  for(i = 0; i < LEN; i++) target[i] = '\0';

  strncpy(target, str+4, 18);

  printf("The substring is: %s\nLength: %i\n",
         target, strlen(target));

  return 0;
}
```

Program 30.4 *Et program der udtrækker en substreng af en streng. Det er essentielt at target initialiseres med nultegn, idet det viser sig, at strncpy ikke overfører et nultegn.*

31. Andre emner om tekststrenge

I dette kapitel diskuterer vi fortrinsvis arrays af strenge, herunder det array af strenge som kan overføres fra operativsystemet til `main` i et C program, når vi starter programmet fra en command prompt (shell). Vi tangerer også input/output af string i `printf`/`scanf`.

31.1. Arrays af tekststrenge

Lektion 7 - slide 23

> Et array af tekststrenge kan enten forstås som en to dimensionel **`char`** tabel eller som en pointer til en **`char`** pointer

Herunder, i program 31.1, skal vi først lægge mærke til erklæringen af `numbers`. Vi ser at `numbers` er et array af strenge. `numbers` kan også opfattes som en to-dimensionelt char array. `numbers` initialiseres i samme åndedrag det erklæres, med brug af den notation vi så første gang i program 21.3 i afsnit 21.2.

```
char *numbers[] = {"one", "two", "three"};
char ch1, ch2, ch3, ch4;

ch1 = **numbers;
ch2 = numbers[0][0];
ch3 = *(*(numbers+1) + 1);
ch4 = numbers[2][2];
```

Program 31.1 *Et program der allokerer og tilgår et array af tre tekststrenge.*

Når vi skal følge de fire assignments efter erklæringen af `numbers` i program 31.1 er det meget nyttigt at se på figur 31.1. Figuren viser `numbers`, som referer til et array med tre pointere til strengene `"one"`, `"two"`og `"three"`.

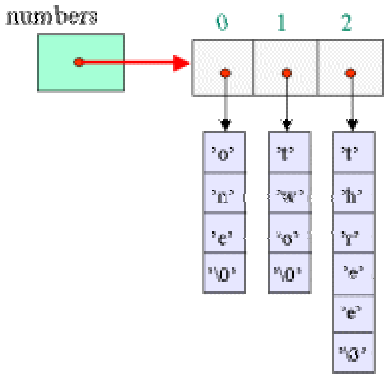
`*numbers` følger den røde pointer i figur 31.1. `**numbers` følger den første sorte pointer til tegnet `'o'`.

`numbers[0]` er cellen som udpeges af den røde pointer. Dette er strengen "one". `numbers[0][0]` er således tegnet 'o'.

`numbers+1` peger på det andet element i det vandrette array. `*(numbers+1)` peger på strengen "two". `*(numbers+1) + 1` peger på strengen "wo". `*(*(numbers+1) + 1)` peger således på tegnet 'w'.

`numbers[2]` peger på strengen "three". `numbers[2][2]` peger på tegnet 'r'.

Det komplette C program i program 31.2 bekræfter gennem outputtet program 31.3 at vore ræsonnementer ovenfor er korrekte.



Figur 31.1 *En illustration af variablen* **`numbers`** *fra ovenstående program*

```
#include <stdio.h>

int main(void) {

  char *numbers[] = {"one", "two", "three"};
  char ch1, ch2, ch3, ch4;

  ch1 = **numbers;
  ch2 = numbers[0][0];
  ch3 = *(*(numbers+1) + 1);
  ch4 = numbers[2][2];

  printf("ch1 = %c, ch2 = %c, ch3 = %c, ch4 = %c\n", ch1, ch2, ch3, ch4);

  return 0;
}
```

Program 31.2 *Hele programmet.*

```
ch1 = o, ch2 = o, ch3 = w, ch4 = r
```

Program 31.3 *Output fra programmet.*

I boksen herunder viser vi hvordan vi kunne have erklæret `numbers` som et to-dimensonelt array.

Variablen **`numbers`** kunne alternativt erklæres og initialiseres som **`char numbers[][6] = {"one", "two", "three"}`**

Den anden af dagens opgaver handler om et to dimensionelt array af tekststrenge

31.2. Input og output af tekststrenge

Lektion 7 - slide 24

Pointen i dette afsnit er reglerne for, hvordan `"%s"` virker i `printf` og `scanf`. Vi bemærker, at I `scanf` springer vi over white space (mellemrum, eksempelvis). Dernæst læser og opsamler `scanf` tegn indtil der igen mødes white space.

Leg gerne med program 31.4 og bliv dermed fortrolig med `scanf` på strenge.

Input af teststrenge med **`scanf`** har specielle regler

Output af tekststrenge med **`printf`** virker som forventet

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

int main(void) {
  char input[100];

  do {
    printf("Enter a string: ");
    scanf("%s", input);
    printf("You entered \"%s\"\n", input);
  }  while (strcmp(input, "exit"));

  return 0;
}
```

Program 31.4 *Et program der indlæser og udskriver en tekststreng med* ***`scanf`*** *og* ***`printf`***.

31.3. Programparametre

Lektion 7 - slide 25

Vi slutter lektionen om tekststrenge af med en kig på, hvordan vi kan overføre parametre til `main` fra operativsystemets command prompt eller shell.

Operativsystemet overfører et array af strenge til `main` funktionen med information om, hvordan programmet er kaldt

Dette giver muligheder for at overføre programparametre som tekststrenge

På det røde sted i program 31.5 ser vi parametrene til main. `argc` vil indeholde antallet af programparametre. `argv` vil indeholde et array af tekststrenge, ala strukturen i figur 31.1.

I program 31.6 ser vi med fed sort skrift en aktivering af **`a.out`** (som er det oversatte program) på tre programparametre. Outputtet fra program 31.5 er vist med rødt i program 31.6.

```c
/* Echo the command line arguments. */

#include <stdio.h>

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

   printf("\n  argc = %d\n\n", argc);
   for (i = 0; i < argc; ++i)
      printf("   argv[%d] = %s\n", i, argv[i]);
   putchar('\n');
   return 0;
}
```

Program 31.5 *Et program der udskriver de overførte programparametre.*

```
[normark@localhost strings]$ a.out programming in C

  argc = 4

   argv[0] = a.out
   argv[1] = programming
   argv[2] = in
   argv[3] = C
```

Program 31.6 *Input til (fed) og output (rød) fra programmet.*

32. Rekursion

Rekursive funktioner er uundværlige til bearbejdning af rekursive datastrukturer. Rekursive datastrukturer forekommer ofte - f.eks. både som lister og træer. Rekursiv problemløsning via del og hersk algoritmer repræsenterer en vigtig ide i faget. Derfor er det vigtigt for alle, som skal programmere, at kunne forholde sig til rekursion.

32.1. Rekursion

Lektion 8 - slide 2

Herunder udtrykker vi den essentielle indsigt som gør rekursion attraktiv. Vi indså allerede i afsnit 11.6 at del og hersk problemløsning er af stor betydning. I en nødeskal går del og hersk problemløsning ud på at opdele et stort problem i mindre problemer, løse disse mindre problemer, samt at kombinere løsningerne på de små problemer til en løsning af det oprindelige, store problem.

Anvendelse af del og hersk princippet involverer altså *problemopdelning* og *løsningskombination*.

> *Rekursion* er en problemløsningside, som involverer løsning af delproblemer af samme slags som det overordnede problem

Vi omgiver os med rekursive strukturer - både i vores hverdag og når vi arbejder på en computer. Blade i naturen udviser rekursive strukturer. Filer og kataloger på en computer udviser rekursive strukturer. Vi vil se på et antal endnu mere hyppige hverdagseksempler i afsnit 32.2 og afsnit 32.3.

Vi argumenter herunder for sammenhængnen mellem rekursive (data)strukturer og rekursive processer. Rekursive processer kan f.eks. programmeres med rekursive funktioner i C.

- Rekursive strukturer
  - I en rekursiv struktur kan man genfinde helheden i detaljen
  - I datalogisk sammenhæng arbejder vi ofte med rekursive datastrukturer
- Rekursive processer
  - Rekursive strukturer processeres naturligt rekursivt
  - Rekursive processer programmeres med rekursive procedurer eller funktioner

32.2. Hverdagsrekursion (1)

Lektion 8 - slide 3

I dette og næste afsnit ser vi på mere eller mindre naturlige rekursive fænomener som er kendt fra vores hverdag.

Man kan opnå et rekursivt billede hvis et spejl spejler sig et i et andet spejl

Tilsvarende rekursive billeder fåes hvis man filmer et TV med kamera, som samtidigt viser signalet fra kameraet

Hvis man er så heldig at have et badeværelse med to spejle på modsat rettede vægge kan man næsten ikke undgå at observere et rekursivt billede.

Hvis man ejer et videokamera og viser signalet på et fjernsyn, som samtidig filmes med kameraet giver dette også et rekursivt billede. Ganske som billedet vist i figur 32.1.

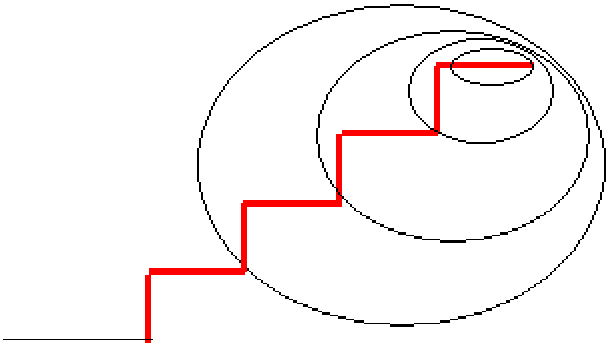
Figur 32.1 *Et billede af en del af en computerskærm, optaget med et web kamera, hvis billede samtidigt vises på skærmen. Det rekursive billede opstår fordi vinduet på skærmen filmes af kameraet, som viser det optagede billede som en del af vinduet, som hermed påvirker hvad kameraet filmer, som hermed påvirker billedet, osv. Efter nogle få niveuaer fortoner effekten sig i et meget forvrænget billede.*

32.3. Hverdagsrekursion (2)

Lektion 8 - slide 4

Når jeg går op af en trappe kan jeg vælge at tænke og gå rekursivt! Jeg tager et skridt, og går dernæst op af trappen som udgøres af de resterende trin foran mig. Jeg er med på at man nok skal være (dat)faglig idiot for at bestige trapper på denne måde. Men i princippet kunne det være en mulig - og langt op ad trappen - en fornuftig tankegang...

Det er muligt - men næppe naturligt - at opfatte en trappe som en rekursiv struktur



Figur 32.2 *En trappe opfattet som en rekursiv struktur.*

- En rekursiv trappe:
  - En trappe kan være 'tom' - flad gulv
  - En trappe kan bestå af et trin og en tilstødende trappe

I afsnit 41.1 ser vi på lineære lister, som i realiteten er ensdannede med trappen ovenfor. Mange centrale liste funktioner programmeres rekursivt, ud fra en forestilling om at en liste består af et hoved (svarende til et trin) og en hale (en resterende del af trappen) som selv er en liste (trappe).

32.4. Referencer

[-] ECIU materiale om rekursion - bedst i IE med SVG, Flash og Java plugins
http://www.cs.auc.dk/~normark/eciu-recursion/html/recit.html

33. Basal rekursion i C

I dette afsnit ser vi på det mere tekniske aspekt af rekursive funktioner i C. Dette er for øvrigt stort set det perspektiv som dyrkes i kapitel 11 af *C by Dissection*.

33.1. Basal rekursion (1)

Lektion 8 - slide 6

Programmerne i program 33.1 og program 33.2 er næsten funktionelt identiske. Begge gentager - på en forholdsvis besværlig måde - udskriften 'f: i', hvor i er et lille heltal.

I program 33.2 benytter vi en funktion, som kalder sig selv rekursivt. I program 33.1 har vi håndlavet et antal versioner af `f`, som kalder hinanden. I forhold til både program 33.1 og program 33.2 ville det være lettere at skrive en simpel for-løkke. Men programmerne illustrerer hvordan vi kan bruge rekursion til simple former for gentagelser i stedet for at bruge løkker.

En funktion i C er rekursiv hvis den i nogle programtilstande kalder sig selv direkte eller indirekte

```c
#include <stdio.h>

void f1(int i){
  printf("f1: %i\n", i);
}

void f2(int i){
  printf("f2: %i\n", i);
  f1(i-1);
}

void f3(int i){
  printf("f3: %i\n", i);
  f2(i-1);
}

int main(void) {
  printf("main\n");
  f3(3);
  return 0;
}
```

Program 33.1 *Funktionen* **`main`** *som kalder en funktion* **`f3`**, *som kalder* **`f2`**, *og som kalder* **`f1`**.

```c
#include <stdio.h>

void f(int i){
  if (i >= 1){
    printf("f: %i\n", i);
    f(i-1);
  }
}

int main(void) {
  printf("main\n");
  f(3);
  return 0;
}
```

Program 33.2 *En tilsvarende kæde af rekusive kald.*

For at sikre at programmet afsluttes, skal der eksistere et grundtilfælde (en programtilstand) hvor funktionen undlader at kalde sig selv rekursivt

Grundtilfældet i en rekursiv definition svarer til et delproblem, som ofte er så simpelt¨, at det kan løses umiddelbart uden yderlig problemopsplitning.

33.2. Basal rekursion (2)

Lektion 8 - slide 7

I dette afsnit vil vi i større detalje undersøge hvad der sker når en funktion kalder sig selv rekursivt. Vi vil se på stakken af aktiveringer (activation records), og vi vil i den forbindelse se på kommandoer som udføres før og efter de rekursive kald.

Vi viser her en variation af programmet, som afslører flere interessante aspekter af rekursion

I program 33.3 ser vi `main`, som kalder den rekursive funktion `f`. Kaldene af `f`, herunder det rekursive kald, svarer til de blå dele i programmet.

I funktionen `f` indlæses der et tegn med `scanf` inden det rekursive kald af `f`. Efter det rekursive kald af f udskrives det indlæste tegn.

```
#include <stdio.h>

void f(int i){
  char ch, skipch;
  if (i >= 1){
    printf("Enter a character: ");
    scanf("%c%c", &ch, &skipch);
    f(i-1);
    printf("We read: '%c'\n", ch);}
  else
    printf("No more recursive calls\n");
}

int main(void) {
  f(4);
  return 0;
}
```

Program 33.3 *Læsning på vej op ad bakken/stakken - Skrivning på vej ned.*

Herunder, i program 33.4 viser vi en dialog med program 33.3 hvor tegnene `'a'`, `'b'`, `'c'` og `'d'` indlæses i den lokale variabel `ch`. Hert nyt tegn indlæses i et separat rekursivt kald af f. Bemærk at der på denne måde kan sameksistere et antal `ch` variable - én pr. kald af `f`.

```
Enter a character: a
Enter a character: b
Enter a character: c
Enter a character: d
Enter a character: e
No more recursive calls
We read: 'e'
We read: 'd'
We read: 'c'
We read: 'b'
We read: 'a'
```

Program 33.4 *Input til og output fra programmet.*

Hvis du skifter til den slide, som svarer til dette afsnit, kan du følge udvikling af kaldstakken, som modsvarer kørslen af program 33.3.

Ved at følge udviklingen af kaldstakken skal du bide mærke i hvordan variablene `i` og `ch` optræder i alle kald af `f`. Man kan kun tilgå variable i den øverste ramme på stakken. Variable under toppen af stakken kan først tilgås når kaldene svarende til de øvre rammer er kørt til ende.

Læg også mærke til at tegnene `a`, `b`, `c` og `d`, som indlæses *på vej op ad stakken* udskrives i modsat rækkefølge *på vej ned ad stakken*.

34. Simple eksempler

I dette afsnit vil vi se på række simple eksempler på rekursion. Vi repeterer fakultetsfunktionen, rodsøgning og strengsammenligningen fra tidligere lektioner. Som nye eksempler vil vi se på en Fibonaccital funktion og en funktion til potensopløftning.

34.1. Eksempler fra tidligere lektioner

Lektion 8 - slide 9

Fakultetsfunktionen mødte vi i afsnit 14.1 i forbindelse med vores introduktion til rekursiv funktioner. I program 14.1 illustreres det hvordan den velkendte formel for fakultetsfunktionen (se teksten i afsnit 14.1) kan implementeres som en funktion skrevet i C.

Rodsøgningsfunktionen fra kapitel 13 er et andet godt og simpelt eksempel som appellerer til rekursiv tankegang. Den iterative løsning på rodsøgningsproblemet fra program 13.1 er vist som en tilsvarende rekursiv løsning i program 14.3. Personlig finder jeg at den rekursive løsning er mere elegant, og på et lidt højere niveau end den iterative løsning.

I program 34.1 viser vi en rekursiv løsning på strengsammenligningsproblemet, jf. afsnit 28.1, som har været en øvelse i en tidligere lektion af kurset. I en if-else kæde håndterer vi først en række randtilfælde samt situationerne hvor det første tegn i `s1` og `s2` er forskellige. I det sidste tilfælde kalder vi funktionen rekursivt (det røde sted). Vi håndterer her et delproblem, svarende til det oprindelige problem, på de dele af `s1` og `s2` som ikke omfatter de første tegn i strengene.

```
int mystrcmp(const char* s1, const char* s2){
  int result;

  if (*s1 == '\0' && *s2 == '\0')
    result = 0;
  else if (*s1 == '\0' && *s2 != '\0')
    result = -1;
  else if (*s1 != '\0' && *s2 == '\0')
    result = 1;
  else if (*s1 < *s2)
    result = -1;
  else if (*s1 > *s2)
    result = 1;
  else       /* (*s1 == *s2) */
    result = mystrcmp(s1+1, s2+1);

  return result;
}
```

Program 34.1 *Rekursiv udgave af* ***strcmp*** *- fra opgaveregningen i forrige lektion.*

I det følgende vil vi se på endnu flere eksempler som involverer rekursion.

34.2. Fibonacci tal (1)

Lektion 8 - slide 10

Der er et antal 'sikre hits', som næsten altid findes blandt eksemplerne, som bruges til at illustrere rekursive løsninger. Beregningen af Fibonacci tal er en af disse. Et Fibonaccital er summen af de to foregående Fibonacci tal. Starten kan i princippet være vilkårlig. I vore eksempler er fib(0) = 0 og fib(1) = 1.

Det er ikke svært at programmerer en funktion, som returnerer det n't Fibonacci tal. I program 34.1 viser vi en flot rekursiv udgave, som er en ganske direkte nedskrivning af definitionen givet herover.

```
long fib(long n){
  long result;

  if (n == 0)
    result = 0;
  else if (n == 1)
    result = 1;
  else
    result = fib(n-1) + fib(n-2);

  return result;
}
```

Program 34.2 *Funktionen fib der udregner det n'te Fibonaccital.*

Læg mærke til at hvert kald af `fib` forårsager to nye kald af `fib`. Denne observation er kritisk når vi skal forstå, hvorfor et kald af `fib`, f.eks. `fib(45)`, tager adskillige minutter at gennemføre.

På slide udgaven af dette materiale linker vi til en komplet program, som udskriver de 100 første Fibonaccital. Programmet er også her. Dette program vil aldrig køre til ende på grund af ovenstående observation. Køretiden af `fib` udvikler sig eksponentielt. I praksis betyder dette, at der er en øvre grænse for hvilke kald af `fib` der overhovedet kan beregnes med metoden i program 34.2.

Vi viser en let modificeret udgave af `fib` funktionen i program 34.3 herunder. I denne udgave tæller vi antallet af additioner, som udføres af et kald af `fib` samt alle de deraf afledte rekursive kald. Læg mærke til at tælleren `plus_count` er en global variabel (det røde sted). Variablen tælles op for hver addition (det blå sted), og den nulstilles for hvert nyt ydre kald af `fib` (det brune sted).

```
#include <stdio.h>

long plus_count = 0;

long fib(long n){
  long result;

  if (n == 0)
    result = 0;
  else if (n == 1)
    result = 1;
  else {
    result = fib(n-1) + fib(n-2);
    plus_count++;
  }

  return result;
}

int main(void) {
  long i, fib_res;

  for(i = 0; i < 42; i++){
    plus_count = 0;
    fib_res = fib(i);
    printf("Fib(%li) = %li (%li)\n", i, fib_res, plus_count  );
  }


  return 0;
}
```

Program 34.3 *En udgave af programmet som holder regnskab med antallet af additioner.*

Når vi kører program 34.3 ser vi at værdien af variablen `plus_count` vokser i samme takt som værdierne af `fib` funktionen anvendt på større og større inputværdier. Herunder viser vi et udsnit af output fra program 34.3.

```
Fib(0) = 0 (0)
Fib(1) = 1 (0)
Fib(2) = 1 (1)
Fib(3) = 2 (2)
Fib(4) = 3 (4)
Fib(5) = 5 (7)
Fib(6) = 8 (12)
Fib(7) = 13 (20)
Fib(8) = 21 (33)
...
Fib(36) = 14930352 (24157816)
Fib(37) = 24157817 (39088168)
Fib(38) = 39088169 (63245985)
Fib(39) = 63245986 (102334154)
Fib(40) = 102334155 (165580140)
Fib(41) = 165580141 (267914295)
```

Vi cutter listen på det sted, hvor udførelsen af `fib` bliver virkelig langsom. Vi ser og mærker tydeligt, at arbejdet i `fib` vokser eksponentielt i forhold til parameteren `n`.

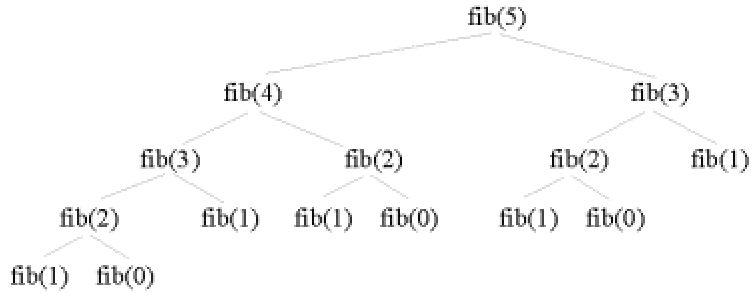
34.3. Fibonacci tal (2)

Lektion 8 - slide 11

Vi vil nu i yderligere detalje forstå hvorfor `fib` er så langsom. I figur 34.1 ser vi kaldene af `fib(5)` . Hvert kald af `fib` resulterer i to nye kald, på nær kald af `fib(0)` og `fib(1)`. Antallet af knuder et træet `fib(n)` bliver astronomisk stor når `n` vokser. Da der udføres én addition pr. indre knude i træet vokser beregningsbyrden tilsvarende. Reelt er det dog vigtigt at forstå, at langt de fleste beregninger er *genberegninger* af allerede beregnede værdier af `fib(n)`.

Funktionen **`fib`** foretager store mængder af unødvendige genberegninger

Antallet af additioner vokser eksponentielt i forhold til parameteren **`n`**



Figur 34.1 *En illustration af beregningsprocessen af fib(5)*

- Mere effektive løsninger
  - En simpel iterativ løsning
  - En rekursiv løsning, som kortslutter genberegningerne

Vi ser på to mere effektive løsninger i næste afsnit.

34.4. Fibonacci tal (3)

Lektion 8 - slide 12

Den mest oplagte - men også den mindst interessante - effektivisering er programmering af `fib` med en forløkke. Denne udgave er vist i program 34.4.

```
#include <stdio.h>

long fib(long n){
  long small, large, temp, m;

  for (small = 0, large = 1, m = 0;
       m < n;
       m++){
    temp = small;
    small = large;
    large = large + temp;
  }

  return small;
}

int main(void) {
  long i;

  for(i = 0; i < 100; i++){
    printf("Fib(%li) = %li\n", i, fib(i) );
  }


  return 0;
}
```

Program 34.4 *En iterativ udgave af fib programmeret med en forløkke.*

I program 34.4 har vi variablene `small` og `large`, som på et hvert tidspunkt indeholder hhv. `fib(n)` og `fib(n+1)` for et eller andet heltal `n`. Variablene `small` og `large` opdateres stort set på samme måde som de to variable `p` og `q` i proceduren `swap`, jf. program 23.2.

Herunder ser vi en udgave af `fib`, som minder meget om program 34.2. Forskellene fremgår af de blå, røde og lilla programsteder. Vi har indført et array, `memo`, som indeholder allerede beregnede værdier af `fib(n)`. I stedet for at genberegne `fib(n)` hentes det ud af tabellen `memo` (det røde sted). Vi skal naturligvis også huske at putte nyberegnede værdier af `fib(n)` ind i `memo` (det lilla sted). Bemærk at rækkefølgen af tilfældene i if-else kæden er vigtig at iagttage.

```c
#include <stdio.h>


long fib(long n){
  long result;
  static long memo[100];   /* elements pre-initialized to 0 */

  if (n == 0)
    result = 0;
  else if (n == 1)
    result = 1;
  else if (memo[n] != 0)
    result = memo[n];
  else {
    result = fib(n-1) + fib(n-2);
    memo[n] = result;
  }

  return result;
}

int main(void) {
  long i;

  for(i = 0; i < 100; i++)
    printf("Fib(%li) = %li\n", i, fib(i));

  return 0;
}
```

Program 34.5 *En memoriseret udgave af fib.*

Hermed afslutter vi vores interesse for Fibonacci tal.

34.5. Potensopløftning (1)

Lektion 8 - slide 13

Potensopløftning handler i bund og grund om at gange et tal med sig selv et antal gange. Vi starter med at se på en simpel, ligefrem og rekursiv potensopløfningsfunktion. I afsnit 34.6 ser vi på en meget mere interessant og meget bedre udgave.

Den beregningsmæssige idé er vist herunder. Bemærk at vi f.eks. tillægger 2^{-3} betydning, nemlig som $1/2^3$.

- Beregningsidé:
  - **potens > 0:** $tal^{potens} = tal^{potens-1} \cdot tal$
  - **potens = 0:** $tal^{0} = 1.0$
  - **potens < 0:** $1.0 / tal^{-potens}$

Idéen er direkte og meget ligefrem implementeret i program 34.6 herunder.

```
double power(double number, int pow){
  double result;

  if (pow == 0)
    result = 1.0;
  else if (pow > 0)
    result = number * power(number, pow - 1);
  else
    result = 1.0 / power(number, -pow);

  return result;
}
```

Program 34.6 *Den simple power funktion.*

34.6. Potensopløftning (2)

Lektion 8 - slide 14

Løsningen i program 34.6 var ganske naiv. Det er måske overraskende at der kan udtænkes en rekursiv løsning som er meget mere effektiv uden at være ret meget mere kompliceret. Ideen i den nye løsning er beskrevet på matematisk vis herunder.

- Beregningsidé:
  - ***potens > 0, potens er lige:*** $tal^{potens} = (tal^{potens/2})^{2}$
  - ***potens > 0, potens er ulige*** $tal^{potens} = tal^{potens-1} \cdot tal$
  - ***potens = 0:*** $tal^{0} = 1.0$
  - ***potens < 0:*** $1.0 / tal^{-potens}$

Den essentielle indsigt er at opløftning i en lige potens *p* kan udføres ved at opløfte til potensen *p/2* efterfulgt at en kvadrering. I stedet for at udføre *p* multiplikationer skal vi nu kun udføre arbejdet forbundet ved opløfning i *p/2* potens efterfulgt af en kvadrering. Hvis *p/2* igen er lige kan vi igen halvere. Hvis *p/2* er ulige ved vi at *(p/2)-1* er lige. Så "tricket" kan typisk gentages flere gange - rekursivt. Kvadrering består naturligvis af en enkelt multiplikation.

Programmet herunder er en direkte og ligefrem implementation af ovenstående formler.

```
double power(double number, int pow){
  double result;

  printf("power(%lf,%i)\n", number, pow);
  if (pow == 0)
    result = 1.0;
  else if (pow > 0 && even(pow))
    result = sqr(power(number,pow/2));
  else if (pow > 0 && odd(pow))
    result = number * power(number, pow - 1);
  else
    result = 1.0 / power(number, -pow);

  return result;
}
```

Program 34.7 *Den hurtige power funktion.*

På en separate slide i materialet leverer vi dokumentation for fordelene ved program 34.7 i forhold til program 34.6. Det sker ved et billedserie som viser de rekursive kald i `power(2.0,10)`. Vi viser også et komplet program som kalder både `power` ala program 34.6 og `power` ala program 34.6. Udskriften af programmet viser antallet af multiplikationer i en lang række kald. Det fremgår klart, at antallet af multiplikationer i den "smarte udgave" vokser meget langsomt i forhold til potensen. Væksten viser sig at være logaritmisk, idet vi i mindst hver andet kald halverer potensen.

35. Hvordan virker rekursive funktioner?

Før det næste eksempel indskyder vi her et ganske kort afsnit, som i tekst gør rede for hvordan rekursive funktioner virker internt.

35.1. Implementation af rekursive funktioner

Lektion 8 - slide 17

De tre første punkter gør rede for stakken af activation records, som vi har set i både udviklingen af de simple (og kunstige) rekursive kald af f fra afsnit 33.2 og udviklingen af power(2.0,10) omtalt i afsnit 34.6.

- Hvert funktionskald kræver afsættelse af en *activation record* - lager til værdier af aktuelle parametre, lokale variable, og noget "internt bogholderi".
- En kæde af rekursive funktionskald kræver én *activation record* pr. kald/inkarnation af funktionen.
- Man skal være varsom med brug af rekursion som kan afstedkomme mange (tusinder) af rekursive kald.
    - Dette kan føre til 'run time stack overflow'.
- Der findes en speciel form for rekursion hvor lagerforbruget kan optimaliseres

- Denne form kaldes halerekursion
- Halerekursion er tæt forbundet med iterative løsninger (brug af while eller for kontrolstrukturer).

Det sidste punkt (og de to underpunkter) nævner *halerekursion* (tail recursiveness på engelsk). Vi går ikke i detaljer med halerekursion i dette materiale. Jeg henviser til materialet Functional Programming in Scheme som diskuterer emnet forholdsvis grundigt.

Enhver løkke (ala while, do, for) kan let omprogrammeres med en (hale)rekursiv funktion.

Nogle former for rekursion kan kun meget vanskeligt omprogrammeres med løkker.

36. Towers of Hanoi

Ligesom Fibonacci tal er Towers of Hanoi en absolut klassiker i diskussion og introduktion af rekursion.

Towers of Hanoi illustrerer en beregningsmæssig opgave med eksponentiel køretid. Derved ligner dette program vores første program til beregning af Fibonacci tal (afsnit 34.2). Ligheden rækker dog ikke langt. Det var helt enkelt at finde en effektiv algoritme til udvikling af talrækken af Fibonacci tal. Der er ingen genveje til løsning af Towers of Hanoi problemet.

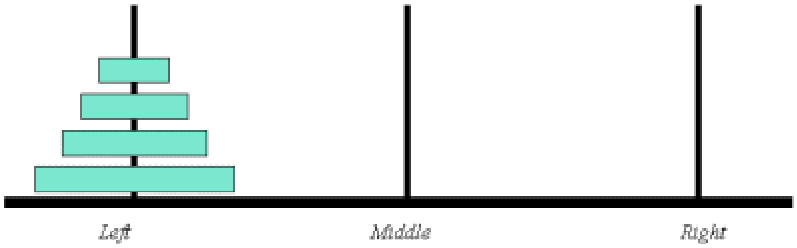
36.1. Towers of Hanoi (1)

Lektion 8 - slide 19

Problemet bag Towers of Hanoi er at flytte skiver fra et tårn til en anden med en tredje som 'mellemstation'. De nærmere regler er beskrevet herunder.

Towers of Hanoi er en gammel asiatisk overlevering om munke, der skulle flytte 64 skiver fra én søjle til en anden

Ifølge legenden ville templet og verden omkring det falde inden arbejdet kunne fuldføres



Figur 36.1 *En illustration af Towers of Hanoi med fire skiver.*

- Problemet:
  - Flyt stakken af skiver fra venstre stang til højre stang med brug af midterste stang som 'mellemstation'
  - Skiverne skal flyttes én ad gangen
  - En større skive må aldrig placeres oven på en mindre skive

36.2. Towers of Hanoi (3)

Lektion 8 - slide 21

Vi løser Towers of Hanoi problemet ved udpræget brug af en del og hersk strategi, jf. afsnit 11.6. Delproblemerne appellerer til rekursiv problemløsning, idet delproblemerne til forveksling ligner det oprindelige problem.

> Vi programmerer en løsning på problemet som kun viser hvilke flytninger der skal foretages.
>
> Bogen har en løsning, som også viser hvordan tårnene ser ud efter hver flytning.

Funktionen `hanoi` herunder flytter `n` skiver fra tårn `a` til tårn `b` med brug af tårn `c` som mellemstation. Som det fremgår af program 36.2 er typen `tower` en enumeration type som dækker over værdierne `left`, `middle` og `right`. Enumerationtyper blev behandlet i afsnit 18.3.

```
/* Move n discs from tower a to tower b via tower c */
void hanoi(int n, tower a, tower b, tower c){
  if (n == 1)
    move_one_disc(a,b);
  else {
    hanoi(n-1,a,c,b);
    move_one_disc(a,b);
    hanoi(n-1,c,b,a);
  }
}
```

Program 36.1 *Den centrale funktion i "Towers of Hanoi".*

Lad os nu forstå løsningen i program 36.1. Hvis `n` er lig med 1 er løsningen triviel. Vi flytter blot skiven fra tårn `a` til `b`. Dette gøres med `move_one_disc`. Hvis `n` er større end 1 flyttes først `n-1`

skiver fra `a` til hjælpetårnet `c` (det første røde sted). Dernæst flyttes den nederste store skive fra `a` til `b` med `move_one_disc` (det andet blå sted). Endelig flyttes de `n-1` skiver på fra `c` til `b`, nu med `a` som hjælpetårn (det andet røde sted).

Hele programmet er vist i program 36.2. Vi ser først enumeration typen `tower` og dernæst `hanoi` funktionen beskrevet herover. Funktionen `move_one_disc` rapporterer blot flytningen ved at udskrive data om flytning på standard output (skærmen). Variablen `count`, som er static i funktionen (jf. afsnit 20.3), tæller antallet af flytninger, blot for at tilfredsstille vores nysgerrighed om opgavens omfang. Funktionen `tower_out`, som kaldes fra `move_one_disc`, løser problemet med at udskrive en `tower` værdi. Denne problematik er tidligere illustreret i bl.a. program 18.7. Funktionen `main` prompter brugeren for antallet af skiver, og dernæst kaldes `hanoi`.

```
#include <stdio.h>

enum tower {left, middle, right};
typedef enum tower tower;

void move_one_disc(tower a, tower b);
char *tower_out(tower t);


/* Move n discs from tower a to tower b via tower c */
void hanoi(int n, tower a, tower b, tower c){
  if (n == 1)
    move_one_disc(a,b);
  else {
    hanoi(n-1,a,c,b);
    move_one_disc(a,b);
    hanoi(n-1,c,b,a);
  }
}

void move_one_disc(tower a, tower b){
  static long count = 0;
  count++;
  printf("%5i. Move disc from  %6s  tower to  %6s  tower\n",
         count, tower_out(a), tower_out(b));
}

char *tower_out(tower t){
  static char *result;
  switch (t){
    case left: result = "LEFT"; break;
    case middle: result = "MIDDLE"; break;
    case right: result = "RIGHT"; break;
  }
  return result;
}

int main(void) {
  int number_of_discs;

  printf("How many discs: ");
  scanf("%d", &number_of_discs);
```

```
  hanoi(number_of_discs, left, right, middle);

  return 0;
}
```

Program 36.2 *Hele programmet.*

En kørsel af program 36.2 med fire skiver giver følgende output:

```
 1. Move disc from    LEFT  tower to  MIDDLE  tower
 2. Move disc from    LEFT  tower to   RIGHT  tower
 3. Move disc from  MIDDLE  tower to   RIGHT  tower
 4. Move disc from    LEFT  tower to  MIDDLE  tower
 5. Move disc from   RIGHT  tower to    LEFT  tower
 6. Move disc from   RIGHT  tower to  MIDDLE  tower
 7. Move disc from    LEFT  tower to  MIDDLE  tower
 8. Move disc from    LEFT  tower to   RIGHT  tower
 9. Move disc from  MIDDLE  tower to   RIGHT  tower
10. Move disc from  MIDDLE  tower to    LEFT  tower
11. Move disc from   RIGHT  tower to    LEFT  tower
12. Move disc from  MIDDLE  tower to   RIGHT  tower
13. Move disc from    LEFT  tower to  MIDDLE  tower
14. Move disc from    LEFT  tower to   RIGHT  tower
15. Move disc from  MIDDLE  tower to   RIGHT  tower
```

Program 36.3 *Output fra programmet ved flytning af fire skiver.*

Som det tydeligt fremgår af næste afsnit, vil outputtet fra program 36.2 føles som uendelig langt hvis `hanoi` kaldes med et stort heltal `n` som input.

36.3. Towers of Hanoi (4)

Lektion 8 - slide 22

Som allerede observeret forårsager hvert ikke-trivielt kald af `hanoi` to yderligere kald. Derved fordobles arbejdet for hver ekstra skive, som er involveret. Dette er eksponentiel vækst.

> Det er let at se at antallet af flytninger *F(n)* vokser eksponentielt med antallet af diske i tårnet.
>
> Mere præcist ser vi at $F(n) = 2^n - 1$.

Vi kan kun håndtere et arbejde, som vokser eksponentielt, for meget begrænsede problemstørrelser. I tabel 36.1 illustreres dette konkret. I søjlen 10^{-9} antager vi at vi kan flytte 1000000000 skiver pr. sekund. I søjlen 10^{-3} antager vi at vi kun kan flytte 1000 skiver pr. sekund. Uanset dette ser vi at arbejdet tager tilnærmelsesvis uendelig lang tid at udføre, når `n` går mod 100.

n	**2^n**	**$10^{-9} \cdot 2^n$**	**$10^{-3} \cdot 2^n$**
5	32	$3.2 \cdot 10^{-8}$ sek	$3.2 \cdot 10^{-2}$ sek
25	$3.4 \cdot 10^7$	0.03 sek	9.3 timer
50	$1.1 \cdot 10^{15}$	13 dage	35702 år
65	$3.7 \cdot 10^{19}$	1170 år	$1.17 \cdot 10^9$ år
80	$1.2 \cdot 10^{24}$	$3.8 \cdot 10^7$ år	$3.8 \cdot 10^{13}$ år
100	$1.2 \cdot 10^{39}$	$4.0 \cdot 10^{13}$ år	$4.0 \cdot 10^{20}$ år

Tabel 36.1 *En table der viser antallet af flytninger og køretider af hanoi funktionen. Køretiderne i tredje kolonne er angivet under antagelsen af vi kan flytte 10^9 skiver pr. sekund. Køretiderne i fjerde kolonne er angivet under antagelsen af vi kan flytte 10^3 skiver pr. sekund.*

37. Quicksort

Som det sidste indslag i lektionen om rekursion ser vi her på Quicksort. Quicksort sorterer et array, ligesom det var tilfældet med Bubblesort i afsnit 24.4.

Quicksort er en klassisk rekursiv sorteringsprocedure. Tilmed er Quicksort bredt anerkendt som én af de bedste sorteringsteknikker overhovedet.

Hvis man skal bruge Quicksort i C skal man anvende `qsort` fra standard C biblioteket. Programmet, som udvikles i denne lektion er funktionel, men ikke optimal i praktisk forstand. Det er svært at programmere Quicksort med henblik på god, praktisk anvendelighed.

37.1. Quicksort (1)

Lektion 8 - slide 24

I dette afsnit vil vi med ord beskrive den algoritmiske idé i Quicksort. I afsnit 37.2 viser vi den overordnede rekursive sorteringsfunktion i C. I afsnit 37.3 diskuterer vi den centrale detalje i algoritmen, som er en iterativ array opdelningsfunktion. Endelig, i afsnit 37.4 vurderer vi god Quicksort er i forhold til Bubblesort.

Quicksort er et eksempel på en rekursiv sorteringsteknik der typisk er meget mere effektiv end f.eks. boblesortering

- Algoritmisk idé:
  - Udvælg et vilkårligt *deleelement* blandt tabellens elementer
  - Reorganiser tabellen således at den består af to dele:
    - En venstre del hvor alle elementer er mindre end eller lig med deleelementet
    - En højre del hvor alle elementer er større end eller lig med deleelementet
  - Sorter både venstre og højre del af tabellen rekursivt, efter samme idé som netop beskrevet

Reorganiseringen af elementerne i punkt to herover kan opfattes som en *grovsortering* af tabellen i små og store elementer, relativ til deleelementet.

Quicksort er et andet eksempel på en rekursiv del og hersk løsning

En stor del af arbejdet består i opdelning af problemet i to mindre delproblemer, som egner sig til en rekursiv løsning

Sammensætningen af delproblemernes løsninger er triviel

37.2. Quicksort (2)

Lektion 8 - slide 25

Herunder viser vi C funktionen `quicksort`. Som det ses er der to rekursive kald `quicksort` i hver aktivering af `quicksort`. Baseret på vores erfaring med Fibonacci tal fra afsnit 34.2 og Towers of Hanoi fra afsnit 36.2 kunne det godt få alarmklokkerne til at ringe med tonerne af 'langsommelighed'. Det viser sig dog at være ubegrundet. Mere om dette i afsnit 37.4.

```
/* Sort the array interval a[from..to] */
void quicksort(element a[], index from, index to){
  index point1, point2;

  if (from < to){
    do_partitioning(a, from, to, &point1, &point2);
    partitioning_info(a, from, to, point1, point2);
    quicksort(a, from, point1);
    quicksort(a, point2, to);
  }
}
```

Program 37.1 *Den rekursive quicksort funktion.*

Det er ligetil at forstå program 37.1 ud fra opskriften - den algoritmiske idé - i afsnit 37.1. Kaldet af `do_partitioning` udvælger dele elementet og foretager opdelningen af arrayet i en venstre del

med små elementer og en højre del med store elementer. Med andre ord bestyrer `do_partitioning` grovsorteringen.

Givet en veludført opdelning kan sorteringsopgaven fuldføres ved at sortere de små elementer i den første røde del. Dernæst sorteres de store elementer i den anden røde del. Dette gøres naturligvis rekursivt, fordi denne opgave er af præcis samme natur som det oprindelige problem. Bemærk her, at når de to rekursive sorteringer er fuldført, er der ikke mere arbejdet at der skal udføres i den oprindelige sortering. Der er altså intet kombinationsarbejde, som vi ofte ser i forbindelse med del og hersk problemløsning.

37.3. Quicksort (3)

Lektion 8 - slide 26

Vi vil nu se på den centrale 'detalje' i `quicksort`, nemlig grovsorteringen. Dette udføres af et kald af `do_partitioning`, som vises i program 37.2.

Funktionen `do_partitioning` opdeler den del af arrayet `a` som er afgrænset af indekserne `from` og `to`. Resultatet af opdelningen er to indekser, der føres tilbage fra `do_partitioning` via to call by reference parametre `point_left` og `point_right`.

```
/* Do a partitioning of a, and return the partitioning
   points in *point_left and *point_right */
void do_partitioning(element a[], index from, index to,
                     index *point_left, index *point_right){
 index i,j;
 element partitioning_el;

 i = from - 1; j = to + 1;
 partitioning_el = a[from];
 do{
   do {i++;} while (a[i] < partitioning_el);
   do {j--;} while (a[j] > partitioning_el);
   if (i < j) swap(&a[i],&a[j]);
 } while (i < j);

 if (i > j) {
   *point_left = j; *point_right = i;}
 else {
   *point_left = j-1; *point_right = i+1;}
}
```

Program 37.2 *Funktionen do_partitioning der opdeler tabellens elementer op i store og små.*

Lad os løbe hen over `do_partitioning` i nogen detalje. På den tilknyttede slide findes der en billedserie, som støtter forståelsen af opdelningen. Den brune del udvælger et deleelement blandt tabellens elementer. Et hvilket som helst element kan vælges. Vi vælger blot `a[from]`, altså det første. Indekserne `i` og `j` gennemløber nu tabellen fra hver sin side i den sorte while-løkke. `i` tvinges frem af den røde do-løkke, og `j` tvinges tilbage af den blå do-løkke. For hvert gennemløb af

den ydre while-løkke foretages en ombytning af en *uorden* i tabellen. Den lilla del forneden sørger for justeringen af de endelige delepunkter.

Som man kan se er der en del detaljer at tage sig af i `do_partitioning`. Det er svært at tune disse, så man får en god, praktisk og velfungerende implementation af Quicksort ud af det.

37.4. Quicksort (4)

Lektion 8 - slide 27

I dette afsnit vil vi kort indse, at vores bekymring om køretiden fra afsnit 37.2 ikke er begrundet. Selv om Quicksort kalder sig selv to gange er vi meget langt fra at udføre et eksponentielt arbejde.

En god implementation plejer at være blandt de allerbedste sorteringsmetoder - men der er ingen garanti

- Køretid i bedste tilfælde:
  - I størrelsesordenen $n \cdot log_2\,(n)$ sammenligninger og ombytninger.
- Køretid i værste tilfælde
  - I størrelsesordenen n^2 sammenligninger og ombytninger.

Argumentet for bedste tilfælde køretiden på $n \cdot log(n)$ er følgende. Hvert kald af `do_partitioning` har en køretid proportional med afstanden `to - from`. Hvis vi derfor kalder `do_partitioning` på et array af størrelsen n udfører vi et arbejde som svarer til n. Vi siger også undertiden at arbejdet er $O(n)$.

Antag nu, at `do_partitioning` deler tabellen på midten i hvert rekursivt kald. Disse opdelninger kan foretages $log(n)$ gange. Samlet set (over alle kald på et givet niveau) udføres et arbejde i `do_partitioning` på n på hvert af de $log(n)$ niveauer. Derfor er køretiden af Quicksort samlet set $n \cdot log(n)$.

Hvis `do_partitioning` skævdeler tabellen i 1 og $(n-1)$ elementer opnås køretiden i værste tilfælde. Dermed får vi n niveauer hver med et samlet opdelningsarbejde på n. Med dette bliver køretiden af Quicksort på $n \cdot n$.

n	n^2	$n \cdot log_2\,(n)$
100	10.000	664
1000	1.000.000	9.966
10.000	100.000.000	132.887
100.000	10.000.000.000	1.660.964

Tabel 37.1 *En tabel der illustrerer forskellen på væksten af n^2 og $n \cdot log_2\,(n)$*

Det kan vises at en sortering med i størrelsesordenen $n \cdot log\ (n)$ sammenligninger og ombytninger er optimal.

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

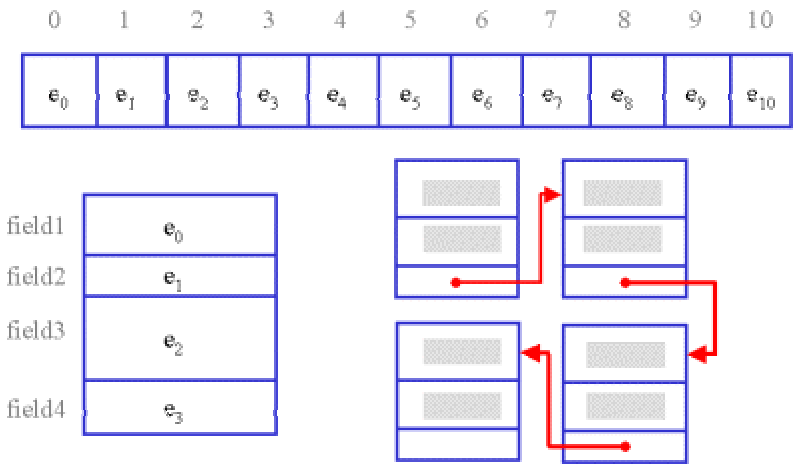
Vi ser først overordnet på datastrukturer.

> 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

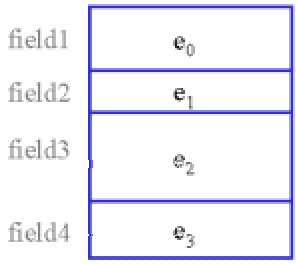
Både arrays og structures kan opfattes som tabeller. Den mest betegnende forskel er dog, at structures tillader 'elementer' af forskellige typer i tabellen. Vi husker fra afsnit 21.1 at alle elementer i et array skal være af samme type.

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)

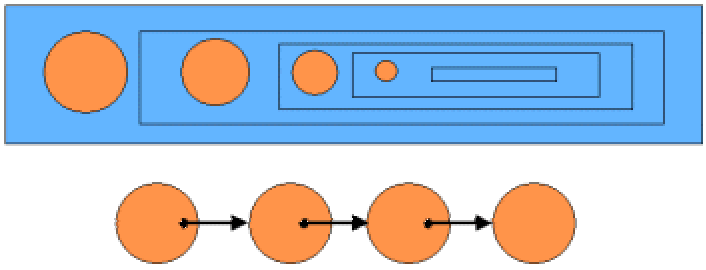
Lektion 9 - slide 18

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.

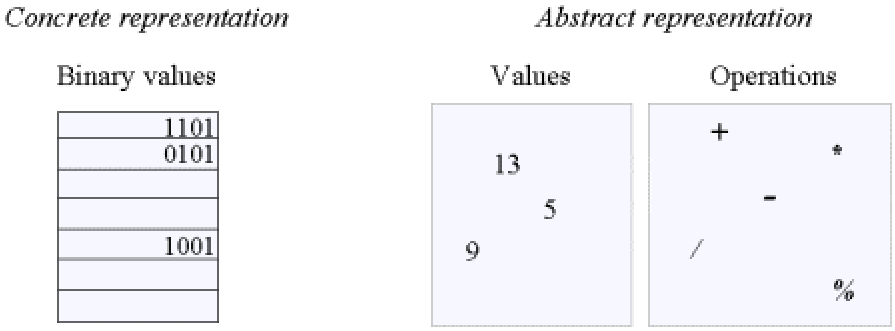
42.2. Tal som en abstrakt datatype

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.

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

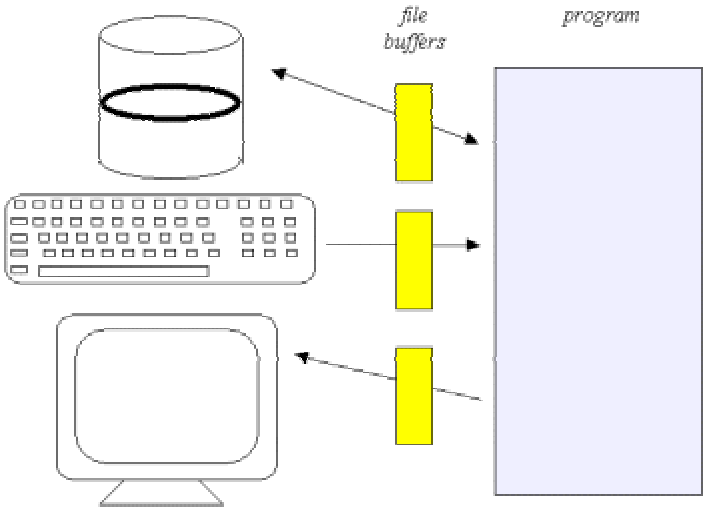
44.2. Filbuffering

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:

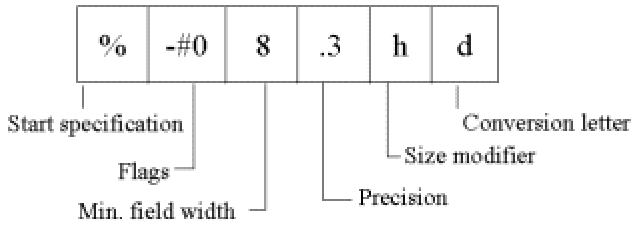
- Pretty printing
- Konvertering af værdier fra forskellige typer til tekst
- Kontrol af det tekstuelle format

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 **s**`scanf` 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 bibliotektet. 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`.

```
#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.*