C - Programmering

Grundbog i programmering

Henrik Kressner
Denne og andre kan findes på: https://synkro.dk/bog

Indholdsfortegnelse

Indledning....3
1 I gang med C....5
1.2 Variabler....12
1.3 Intelligens....17
1.6 Afrunding....38
1.7 Opgaver....41
2 Grundbegreber....42
2.1 Blokke....42
2.2 Datatyper....47
2.3 Typekonvertering....49
2.4 Formatering....53
2.5 Operatorer og operander....57
2.6 If ... else....62
2.8 Funktioner....71
2.9 Øvelse gør mester....77
3 Array, pointere og filer....83
3.1 Array....83
3.2 Strenge....89
3.3 Pointere....98
3.4 Enumeration....106
3.5 Arrays med flere dimensioner....108
3.6 Filer....113
3.7 Et eksempel....117
3.8 Opgaver....119
4 Strukturer og lister....120
4.1 Begrebet struktur....120
4.2 Array af struct's....123
4.3 Headerfiler....126
4.4 Kædede lister....130
4.5 Dobbeltkædede lister....137
4.6 Fra en kædet liste til en fil....140
4.7 Opgaver....145
5. Praktiske ting....146
5.1 Rekursion....146
5.2 Include filer....151
5.3 Mere om variabler og rækkevidde....156
5.6 Kodebrydning....163
5.6.1 Afrunding....167

5.7 Opgaver....169
Appendix A Operatorer....171
A.1 Aretmetiske operatorer....171
A.2 Relations- og betingelsesoperatorer....171
A.3 Bitvise operatorer....172
A.4 Præcendens....173
Appendix B Standardbiblioteker for C....174
B.1 <assert.h>....174
B.2 <ctype.h>....174
B.3 <errno.h>....174
B.4 <float.h>....176
B.5 <limits.h>....177
B.6 <math.h>....179
B.7 <stddef.h>....180
B.8 <stdio.h>....180
B.9 <stdlib.h>....184
B.10 <string.h>....186
B.11 <time.h>....187

Indledning

Denne bog er en videreudvikling/tilpasning af den jeg udgav i 2001 med titlen C/C++, men her er kun C delen. Bogen udgives on-line som pdf fil, og kan frit benyttes og kopieres af alle som de ønsker det, delvis eller i uddrag. Ved kopiering skal der dog mindst kopieres en fuld side med header og footer, og der må ikke fjernes noget fra siderne.

Forudsætninger:

– Grundlæggende IT-kundskab, hvilket betyder, du skal kunne bevæge dig rundt i directory strukturen, du skal kunne starte programmer og kunne benytte en tektseditor.

Kapitel 1 er en surf ind i C's forundelige verden. Kapitlet er skrevet for ”at lokke husarene til”. Der stryges let hen over mange ting, og mange der ikke følger et kursus vil nok tage en længere pause efter dette kapitel, for ligesom at synke det hele. Under alle omstændigheder kan du skrive en del småprogrammer, efter at have gennemgået kapitel 1.

Kapitel 2 dykker ned i C's grundbegreber, startende med blok begrebet. Derefter gennemgås datatyper og typekonvertering. Output formatering gennemgås også, selv om det nok mest har historisk interesse, da alt jo nu om dage foregår i et visuelt miljø. Operatorer og operander gennemgås grundigt, og til sidst gennemgås begrebet funktioner.

Kapitel 3 Drejer sig egentlig kun om pointere, men da array's er et specialtilfælde af pointere gennemgås disse også, i den tætte sammenhæng, som de to begrebers familiebånd indikerer. Der gennemgås også hvordan man kan komme til at læse fra, og skrive til filer.

Kapitel 4 Gennemgår begreberne structurer og lister, herunder specielt kædede lister. Kapitlet er vigtigt at forstå for den der ønsker at arbejde med objektorienteret programmering i eksempelvis C++.

Kapitel 5 Runder af med lidt praktiske ting, af den slags mange elever finder irriterende. De er lagt i dette kapitel for sig selv, i håb om dem der springer det over, finder tid og ro til alligevel at gennemgå det senere.

Sidst i bogen er et appendix, hvor der findes praktiske informationer i form af tabeller og standardbiblioteker.

Udeståender:

Der udestår først og fremmest en gennemgang af layout, så det hele bliver meget nemmere at læse, der skal der nok også lidt mere hjælp til at starte med at få kompileret det første program.

Der er sket en løbende opdatering af output i eksemplerne, fra udgave 1.3.

Som altid er kritik velkommen.

Til sidst vil jeg gerne takke alle dem der har kritiseret mig, man kan jo kun blive bedre ved at lytte til sine kritikere, hvilket dog ikke er ensbetydende med, jeg altid er enig med dem.

Henrik Kressner
kressner@synkro.dk

1 I gang med C

Da dette er en begynderbog starter vi helt fra bunden her i første kapitel med nogle småprogrammer, der er beregnet til at lade begynderen få noget til at virke.

Men inden du går i gang med at programmere, eller at kode som det også kaldes, skal du have fat i nogle værktøjer. Jeg vil ikke anbefale begynderen at vælge et kraftigt moderne IDE (Integrated Devolopment Inveriomemt) , det er min erfaring begynderen ender op med at side og klikke med musen. I en tid ser det godt ud, men efter nogle timer vil begynderen uvægerligt køre fast i en stribe af blindgyder, hvor det kræver dybt kendskab til C++ for at komme videre. I den situation giver mange op.

Det første du skal beslutte er derfor, hvilket værktøj du vil vælge. Jeg vil anbefale du vælger et kommandolinie baseret værktøj. Der findes flere ude på webbet du bare kan downloade, søg på "C programmerin" evt. suppleret med ”compiler”, så vil du finde en flere værktøjer du kan vælge mellem. Fortvivl ikke hvis du har købt et kostbart IDE baseret vindue værktøj, det kan ofte bruges. Du skal åbne et tekstvindue (kaldes DOS eller kommando prompt i MS verdenen) og finde ud af hvad kompileren hedder, det står i manualen. (Den hedder ofte noget med CC.EXE, eller CPP.EXE i MS verdenen)

Hvis du er så heldig at have valgt Linux som operativsystem, hertil regnes alle Apples og Android produkter, er det lige ud af landevejen. Benyt en tekstorienteret editor, eksempelvis vi, elvis eller nano, og brug den medfølgende C og/eller C++ kompiler. Hvis der ikke er en compiller installeres er det virkelig nemt at finde en på nettet.

Hvis din Linux er Debian baseret, herunder Ubuntu, skal du blot skrive:

```
$ sudo apt-get install build-essential ↵
```

"↵ " Betyder du skal trykke på Enter tasten.

1.1 Det første program

Vi starter med at lave et lille klassisk begynderprogram, der har en opgave her i livet, at skrive ”Jeg tænker, ergo er jeg” på skærmen.

```
/* Filnavn = hallo1.c
   Dette program udskriver,
   "Jeg tænker, ergo er jeg"
   på skærmen.
*/

#include <stdio.h>

int main()
{
       printf("Hallo verden");
       return 0;
}
```

```
$ cc hallo1.c
$ ./a.out
Hallo verden$
```

Figur 1.1.1

For at få programmet til at køre, skal du starte med at taste det ind i en teksteditor. Det kan være `vi`, `elvis` eller `nano` hvis du kører Linux, eller det kan være `notepad` hvis du bruger Windows, men du skal huske at gemme som et tekst dokument.

Det er ikke altid nødvendigt, men mange oversættere (kaldet kompilere på engelsk) forlanger at filer der skal oversættes skal have et filextendet som c eller cpp. (extendet = c betyder der er tale om en c kildetekst, hvis der står cpp er der tale om en C++ kildetekst)

Når du taster programmet ind, skal du være opmærksom på at C og C++ er "case sensitive", hvilket på dansk betyder, at der er forskel på små og store bogstaver. Programmet skal altså tastes ind nøjagtigt som det står, ellers må du forvente at få fejlmeddelelser når du forsøger at oversætte kildeteksten til noget computeren forstår.

Når programmet er tastet ind, og gemt som en tekstfil med extendet c, kan du begynde at

oversætte programmet. Til dette formål skal du bruge en oversætter, kaldet en compiler på engelsk.
Fremover vil jeg her i bogen benævne oversætteren med dens engelske betegnelse (compiler), og benævne det at oversætte med den engelske betegnelse. (compile) Man bruger altså en compiler til at compilere en kildetekst (kaldet sourcecode på engelsk), for at generere et eksekverbart program.

Hvis du kører Linux, og ikke har tastet forkert, kan kompileringsprocessen se således ud:

```
$ cc hallo1.c
$
```

Derefter vil du opdage, at der er kommet en ny fil i samme directory som du foretog kompileringen i, denne fil bærer navnet a.out. Dette er den eksekverbare fil. Programmet kan startes ved at skrive:

```
$ ./a.out
```

Som vist på Figur 1.1.1

Hvis du ønsker den eksekverbare fil skal have et andet navn end a.out kan du omdøbe den i kompileringsprocessen, ved at bruge o switchen. (o for output)

```
$ cc hallo1.c -o hallo1
$
```

Derved vil output filen hedde hallo1 i stedet for a.out. hallo1 eksekveres ved at skrive:

```
$ ./hallo1
```

Det er særdeles normalt, for begyndere som for øvede, ja selv for specialister, at kompileringsprocessen ikke går som beskrevet ovenfor. Ofte kommer der en byge af

meddelelser. Disse kan deles op i to grupper, warnings (advarsler) og errors (fejl). Errors er altid alvorlige, og medfører at kompileringen ikke blev fuldført, hvorfor der ikke er nogen eksekverbar fil. (Hvis du ser en, er det en tidligere fil)

Warnings er ofte noget man kan se bort fra, men det er sjusket programmering at gøre det. Derfor, så længe du får errors eller warnings er der kun en ting at gøre, gå ind i kildeteksten, find fejlen og ret den.

Selvfølgelig er det surt at lede kildeteksten igennem, blot for at finde ud af man har brugt et stort P i stedet for et lille p, eller fordi man er kommet til at trykke på mellemrum et uheldigt sted i kildeteksten, men sådan er det at programmere. Selv erfarne programmører ærgrer sig gang på gang over de har tastet forkert, så der er kun en ting at gøre, find fejlen og ret den, kompileren har altid ret. (Der findes fejl i kompilere, men begynderen skal altid gå ud fra, at det er ham/hende der har lavet en fejl. Årsagen er, denne bog indeholder så enkle eksempler, at stort set ingen kompilere slipper forbi de første test hvis disse programmer ikke kan kompileres uden fejl)

Lad os se på hvorfor programmet gør som det gør, når det altså først er blevet korrekt kompileret.

Programmet starter med /*. Dette betyder, at alt hvad der kommer efter skal ignoreres af kompileren, indtil den møder */. Dette bruges til at indsætte kommentarer i kildeteksten. Det betyder at linierne:

```
/* Filnavn = hallo1.c
   Dette program udskriver,
   "Hallo verden"
   på skærmen.
*/
```

helt kan udelades uden konsekvenser for kompileringen. Det er altid en god ide at sætte et

passende antal kommentar ind i sine kildetekster, det hjælper når du på et senere tidspunkt skal ind og se hvad du lavede i et program du skrev for længe siden.

Altså første regel for programmører: Indsæt kommentarer i kildeteksten. På nuværende tidspunkt må du hellere sætte for mange kommentarer ind, end for få. Begrænsningens kunst kommer helt af sig selv når du får rutinen.

Efter kommentaren kommer linien:

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

Det betyder at kompileren skal inkludere en fil ved navn stdio.h. Hvor filen befinder sig bestemmes af kompileren's opsætning. Hvis kompileren ikke kan finde filen, skal årsagen søges i installation eller opsætning af kompileren. For langt de fleste kompilere er dette ikke et problem, giver det alligevel problemer bør du bruge nettet til at finde hjælp.

Den næste linie af betydning, (tomme linier ignoreres altid af kompileren) er linien:

```
int main()
```

Dette er starten på en funktion ved navn main. Denne funktion skal altid være til stede i et C program. De to parenteser er til en såkaldt argumentliste, som vi ikke vil bruge endnu. Al eksekvering starter altid med main.

Funktionen main() har sin kode inde mellem de efterfølgende tuborgklammer {}. (Kaldes tuborgklammer fordi de minder om et tuborg skilt hvis man lægger dem ned) Den første klamme kaldes starttuborg, den sidste kaldes sluttuborg.

Mellem start og slut tuborg står der:

```
    printf("Hallo verden");
```

Og det er her hele funktionaliteten ligger. På linien står der faktisk, kald funktionen ved navn `printf`, med argumentet "Jeg tænker, ergo er jeg" For at afslutte linien skal der være et semikolon til sidst. Hvis du glemmer et semikolon vil du ofte få en stribe af fejlmeddelelser når du kompilerer, da det medfører at resten af programmet bliver håbløst at kompilere. Et glemt semikolon vil ofte give den første fejl i linien efter den linie hvor det mangler.

Funktionen `printf` er en funktion fra stdio.h biblioteket, som vi jo inkluderede først i programmet. Funktionen udskriver det der står i argumentlisten på standard input/output enheden, der normalt er skærmen.
Den sidste linier return 0; lader vi være ind til videre, vi kommer tilbage til hvad den gør.

I de fremtidige eksempler vil vi kun vise kompileringsprocessen set fra UNIX. Hvis du har valgt noget andet, må du rådføre dig med manualen, men husk Apple og Android ER UNIX.

Du har muligvis bemærket en irriterende ting ved dit første program, kommandolinien fortsætter lige efter programmet er stoppet. Det kan ændres blot ved at indsætte en instruks om et linieskift. Instruksen om linieskift ser således ud: \n (backslash n), og den skal indsættes lige efter det sidste vi vil have skrevet ud på linien.

Dette er vist i programmet hallo2.c på Figur 1.1.2.

```c
/* Filnavn = hallo2.c
   Dette program udskriver,
   "Jeg tænker, ergo er jeg"
   på skærmen, efterfulgt af
   et linieskift.
*/

#include <stdio.h>

int main()
{
       printf("Hallo verden\n");
       return 0;
}
```

```
$ cc hallo2.c
$ ./a.out
Hallo verden
$
```

Figur 1.1.2

Det er muligt at dele argumentet "Jeg tænker, ergo er jeg" op i to linier, det er dog vigtigt at hver linie afsluttes korrekt, hvilket betyder at linien skal afsluttes med gåseøjne. (") Dette er vist i programmet hallo3.c på Figur 1.1.3.

Det der står inde mellem gåseøjnene kaldes ofte en streng (fra engelsk string), fordi der er tale om en streng af tekst, til forskel fra eksempelvis tal. Man kan sige, at tekst og tal er to forskellige typer af data.

```c
/* Filnavn = hallo3.c
   Dette program udskriver,
   "Jeg tænker, ergo er jeg"
   på skærmen.
*/

#include <stdio.h>

int main()
{
        printf("Hallo "
        "verden\n");
        return 0;
}
```

```
$ cc hallo3.c
$ ./a.out
Hallo verden
$
```

Figur 1.1.3

Efter denne begyndelse kan det være en god ide, hvis du stopper lidt op, og begynder at lege med det du har prøvet ind til nu. Prøv eksempelvis at placere \n andre steder inden for gåseøjnene, for at se hvad der sker. Prøv også at løse disse opgaver.

Opgave 1.1.1 Skriv et program der udskriver dit fornavn og dit efternavn på en linie.

Opgave 1.1.2 Skriv et program der udskriver dit fornavn på en linie, og dit efternavn på efterfølgende linie.

Opgave 1.1.3 Skriv et program der skriver dit efternavn ud på den første linie, og dit fornavn ud på linien efter.

Opgave 1.1.4 Prøv at udkommentere linien:

#include <stdio.h>

i programmet hallo1.c. Find ud af hvor compileren melder fejl, og hvorfor.

1.2 Variabler

Variabler er et sted hvor man kan opbevare og manipulere data. Vi vil senere komme mere ind på begrebet, lige nu vil vi blot vise et eksempel på hvordan variabler kan bruges.

I programmet var1.c på Figur 1.2.1, er der indført en variabel ved navn minVariabel. Den erklæres i linien:

```
int minVariabel;
```

Bemærk at der derefter er indsat en tom linie. Det er kutyme at man indsætter en tom linie efter variabelerklæringer. For kompileren er det betydningsløst, men det gør det nemmere for os mennesker, at læse programmet. Efter erklæringen tilskrives variablen i linien:

```
minVariabel = 1;
```

```
/* Filnavn = var1.c */
#include <stdio.h>

int main()
{
	int minVariabel;

	minVariabel = 1;
	printf("%d\n", minVariabel);
	return 0;
}
```

```
$ cc var1.c
$ ./a.out
1
$
```

Figur 1.2.1

Efter denne linie har variablen ved navn `minVariabel` indholdet tallet et. Dette fremgår også når vi udskriver indholdet af `minVariabel` i linien:

```
printf("%d\n", minVariabel);
```

Bemærk det besynderlige `"%d\n"` først i printf's argument. Det er en formatteringsstreng, og nok en af årsagerne til, at så mange opgiver C programmering på dette sted. Det kan ligne volapyk, men når man først har lært det er det ganske smart. Procent d betyder, at nu kommer der et heltal, \n betyder det samme som før, nemlig indsæt et linieskift. Alt i alt betyder formatteringen, at der skal udskrives et heltal, efterfulgt af et linieskift.

Vi kunne i stedet have skrevet:

```
printf("1\n");
```

Det lærte vi jo lige før, men problemet er at computeren opfatter det vi ser som tallet et, som værende tegnet 1, hvilket set med computerens øjne er ganske forskelligt. Årsagen er, at en computer internt bruger tal til både at repræsentere tegn, og til at regne med. Vi skal altså finde en måde at forklare computeren, hvornår vi mener tal, og hvornår vi mener tegn. Det sker i linien:

```
int minVariabel;
```

Her erklæres `minVariabel` som værende en type kaldet `int`. (int er en forkortelse for integer = heltal) Det vil sige, at `minVariabel` kan indeholde et helt tal, og intet andet.

Da `minVariabel` er en variabel af typen heltal (int), og da variabler indeholder data, så må `int` være en datatype. Det kan altsammen virke noget abstrakt på nuværende tidspunkt, men vi vil komme tilbage til begrebet typer ved flere lejligheder, så frygt ikke

datatyperne, de er her for at hjælpe dig. (Du ved da bare ikke endnu :)

Programmet var2.c på Figur 1.2.2 viser at man kan manipulere med indholdet af en variabel. I dette tilfælde starter vi med at tilskrive variablen `minVariabel` med heltallet 1, så skriver vi det ud på skærmen. Når det er gjort kører programmet videre, og lægger tallet 3 til hvad der nu står i forvejen, hvorefter det overraskende resultat udskrives på skærmen.

```
/* Filnavn = var2.c */


#include <stdio.h>

int main()
{
        int minVariabel;

        minVariabel = 1;
        printf("%d\n", minVariabel);
        minVariabel = minVariabel + 3;
        printf("%d\n", minVariabel);
        return 0;
}
```

```
$ cc var2.c ↵
$ ./a.out ↵
1
4
$
```

Figur 1.2.2

En anden datatype er `char`, som er en forkortelse for character, eller tegn på dansk. Ved at sætte flere af dem sammen dannes et array, som kan fremstille en tekststreng. Det kan se ud som vist på Figur 1.2.3.

```c
/* Filnavn = var3.c */

#include <stdio.h>

int main()
{
	char tekst[] = "Jeg tænker, ergo er jeg";

	printf("%s\n", tekst);
	return 0;
}
```

```
$ cc var3.c
$ ./a.out
Jeg tænker, ergo er jeg
$
```

Figur 1.2.3

Endelig kan man kombinere disse tekniker til et lille program der lægger to tal sammen, og udskriver resultatet på skærmen.

```c
/* Filnavn = var4.c */
#include <stdio.h>

int main()
{
	int a,b,c;
	char tekst[] = "summen er : ";

	a = 3;
	b = 2;
	c = a + b;
	printf("%s%d\n", tekst, c );
	return 0;
}
```

```
$ cc var4.c
$ ./a.out
summen er : 5
$
```

Figur 1.2.4

%s%d\n Betyder, at først skal der udskrives en streng, derefter skal der udskrives et heltal. Linien sluttes af med et linieskift.

I stedet for linien:

```
printf("%s%d\n", tekst, c );
```

Kunne man have skrevet:

```
printf("%s%d\n", tekst, a + b );
```

Det vil virke på præcis samme måde.

Opgave 1.2.1 Skriv et program der trækker to tal fra hinanden, og afleverer resultatet på skærmen.

Opgave 1.2.2 Skriv et program med de to variabler kmITimen og tid. Programmet skal udskrive hvor langt man er kommet, efter at have kørt tid timer med hastigheden kmITimen.

1.3 Intelligens

En computer er ikke så intelligent som mange gør den til, men et af de træk der kan opfattes som værende intelligens, er if ... else strukturen. Det betyder kort og godt, at hvis (if) et eller andet er sandt, så udføres en ting, ellers (else) udføres en anden ting. Det hele afhænger altså af, om programmøren kan stille nogle brugbare spørgsmål af den type, der kun kan svares ja eller nej til.

```c
/* Filnavn = if1.c */

#include <stdio.h>

int main()
{
	int test = 5;

	if (test > 5)
		printf("test er > 5");
	else
		printf("test er <= 5");
	printf("\n");
	return 0;
}
```

```
$  cc if1.c
$ ./a.out
test er <= 5
$
```

Figur 1.3.1

Programmet if1.c er næsten selvforklarende. Hvis indholdet af variablen test er større end 5 udskrives linien:

test er > 5

Hvis indholdet af variablen test ikke er større end 5 udskrives den linie ikke, i stedet udskrives linien:

```
test er <= 5
```

Den sidste linie udføres altid, den benyttes til at udføre et linieskift inden programmet forlades. Man kunne i stedet have valgt at placere \n bagerst i begge printf, det ville ikke gøre nogen forskel, bortset fra at tilfredsstille forfatterens lyst til at spare en byte i hukommelsen.

En ny teknik i programmet er linien:

```
int test = 5;
```

Linien erklærer variablen test til at være af typen int, derefter tilskrives variablen med værdien 5. Dette kaldes også at initiere en variabel. Nogen gange er der praktisk, andre gange er det spild af tid at gøre det.

Det kan nogen gange være en dårlig ide altid at initiere sine variabler, samtidigt med at man erklærer dem. Årsagen er, at moderne C kompilere vil fremsætte warnings hvis en variabel er erklæret, men ikke tilskrevet. Det betyder, at programmøren bliver gjort opmærksom på, hvis han/hun har en ubrugt variabel. Sådanne variabler bør fjernes, da der ellers opstår rod i kildeteksten, og det gør det svært senere at finde rundt i kildeteksten.

Rigtige programmører har orden i deres kildetekster. Hvis der ikke er orden i kildetksten opstår spaghettikode, hvilket betyder at koden er sammenfiltret som spaghetti, og derfor problematisk at rette i.

Nu kunne det jo godt være man kunne tænke sig, at udføre mere end en ting (linie) hvis betingelsen er sand, det klares ved at indføre blokke. Blokke starter altid med en starttuborg, og slutter altid med en sluttuborg.

```c
/* Filnavn = if2.c */

#include <stdio.h>

int main()
{
	int test = 6;

	if (test > 5)
	{
		printf("Efter at have udført\n");
		printf("en nærmere undersøgelse\n");
		printf("kan det fastslåes, at\n");
		printf("værdien af variablen test\n");
		printf("er større end 5.\n");
	}
	else
		printf("test er <= 5");
	printf("\n");
	return 0;
}
```

```
$ cc if2.c
$ ./a.out
Efter at have udført
en nærmere undersøgelse
kan det fastslåes, at
værdien af variablen test
er større end 5.

$
```

Figur 1.3.2

Ved at kombinere if udtrykket med en blok, kan der udføres mere end en linie, hvis betingelsen er sand. Det samme gælder for else, hvis man ønsker at udføre mere end en linie hvis else er sand. Det er vigtigt at bemærke to ting.

Det er semikolon der viser at en linie er afsluttet. Det er en fejl at afslutte if (..) linien med et semikolon. Det ville betyde, at hvis betingelsen er sand, så skal der ikke udføres noget.

Hvis man ønsker at gøre brug af else, der rent faktisk er valgfri, så skal else komme lige efter if er færdig. Der må altså ikke være to linier efter if, hvis der ikke er tuborg omkring, fordi der vil opstå en kompilerfejl i else linien.

Følgende vil give en kompilerfejl.

```
if (test > 5)
        printf("Efter at have udført\n");
        printf("en nærmere undersøgelse\n");
else    /* I denne linie vil der komme en kompilerfejl */
        printf("test <= 5\n");
```

Det er lidt nemmere at se, hvis vi sætter nogle indrykninger der passer bedre med hvad der sker i programmet:

```
if (test > 5)
        printf("Efter at have udført\n");
printf("en nærmere undersøgelse\n");
else    /* I denne linie vil der komme en kompilerfejl */
        printf("test <= 5\n");
```

Ved at bruge indrykninger bliver det tydeligt, at uanset værdien af variablen test, så vil linien:

```
printf("en nærmere undersøgelse\n");
```

blive udført. Det betyder igen, at kompileren bliver forvirret når den kommer til linien:

```
else    /* ... */
```

fordi den ikke kan finde nogen if at knytte else til.

Hvis der ikke er behov for else, kan den helt udelades, det kan se ud som vist i programmet if3.c på Figur 1.3.3.

```c
/* Filnavn = if3.c */

#include <stdio.h>

int main()
{
        int test = 5;

        if (test > 5)
                printf("test er > 5\n");
        return 0;
}
```

```
$ cc if3.c
$ ./a.out
$
```

Figur 1.3.3

Så længe indholdet af variablen test er mindre end 6, så bliver der ikke udskrevet noget, derfor er der ikke noget output. Prøv at ændre værdien på test, og se hvad der sker.

If fungerer altså ud fra følgende princip:

```
if (betingelse)
     er sand gør dette
else
     gør dette, hvis der er en else
```

Tilbage bliver at definere hvad en betingelse i C er. For nuværende er det nok at kende følgende betingelser.

a == b	Er a lig med b?
a != b	Er a forskellig fra b?
a > b	Er a større end b?
a < b	Er a mindre end b?
a <= b	Er a mindre, eller lig med b?

a == b	Er a lig med b?
a >= b	Er a større end, eller lig med b?

Hvis svaret på spørgsmålet er ja, så er betingelsen sand, ellers er betingelsen ikke sand. Der er ikke noget der hedder måske.

Bemærk at ved undersøgelse af om to variabler er ens, skal der bruges to lighedstegn, det er forkert at bruge et. Hvis der kun bruges et lighedstegn vil der ikke blive testet korrekt. Vi vil senere komme tilbage til hvorfor dette er tilfældet, når vi går dybere ind i if.

Opgave 1.3.1 Skrive et program med de to heltalsvariabler a og b. Tilskriv de to variabler med to forskellige tal, og få programmet til at udskrive indholdet af den variabel der har den største værdi

1.4 Rundt og rundt

En grundliggende programmeringsteknik er gentagelse, dette opnås med løkker. Med gentagelse menes, at en ting (en programdel) udføres igen og igen, indtil et tilfredsstillende resultat er opnået. Dette resultat skal kunne udtrykkes som en betingelse, der kan være enten sand (true) eller falsk/usand (false).

Programmet for1.c på Figur 1.4.1 er et eksempel på hvordan gentagelse kan opnås i C.

```c
/* Filnavn = for1.c */

#include <stdio.h>

int main()
{
	int i;

	for (i = 1; i < 10; i = i + 1)
		printf("%s%d\n", "i er nu = ", i);
	return 0;
}
```

```
$ cc for1.c ▯
$ ./a.out ▯
i er nu = 1
i er nu = 2
i er nu = 3
i er nu = 4
i er nu = 5
i er nu = 6
i er nu = 7
i er nu = 8
i er nu = 9
$
```

Figur 1.4.1

For at forstå hvordan løkken fungerer, løber vi den igennem trin for trin. Til at starte med erklærer vi en variabel af typen int, og kalder den for `i` Derefter møder vi linien:

```c
for (i = 1; i < 10; i = i + 1)
```

Det medfører at variablen `i` bliver sat til værdien 1. Når det er gjort testes det om indholdet i variablen `i` er mindre end 10, hvilket er tilfældet. Det medfører at linien:

```c
printf("%s%d\n", "i er nu = ", i);
```

udføres. Derefter lægges der en til den værdi der står i variablen i. Løkken er nu gennemløbet første gang, og der skal tages en beslutning om løkken skal gentages. Hvis betingelsen i < 10 er sand, så gentages løkken, og da variablen `i` må have værdien to, så er betingelsen sand, og løkken gentages.

På et eller andet tidspunkt har variablen `i` nået værdien 9 og printf er udført. Derefter

lægges der en til variablen `i`, der nu har værdien 10, hvilket ikke er mindre end 10. Det betyder at betingelsen er usand, og løkken forlades uden at printf udføres.

Hele arbejdet i programmet ligger i linien:

```
for (i = 1; i < 10; i = i + 1)
```

Det betyder, at til at starte med sættes variablen `i` lig med værdien 1. Derefter gennemføres den efterfølgende linie, så længe i er mindre end 10. Hver gang programmet er løbet en gang rundt i løkken lægges der en til i. Eller sagt på en anden måde.

```
for (initialisering; betingelse; tilskrivning)
```

Det betyder, at det der står foran det første semikolon udføres når løkken startes, og ikke igen, medmindre løkken startes igen. Det der står mellem de to semikolon er en betingelse, som vi kender den fra `if` sætningen. Så længe betingelsen er sand, fortsættes der rundt i løkken. Når betingelsen (forhåbentlig) på et tidspunkt bliver usand stopper løkken. Hver gang der tages en tur rundt i løkken udføres det der står bag sidste semikolon, i dette tilfælde lægges der en til variablen `i`.

Hvis løkken aldrig bliver sand, har vi et program der hænger. Det er noget enhver programmør af og til kommer ud for. Hvis (når) du oplever det, kan du prøve at trykke samtidigt på CTRL og C tasten, det plejer at stoppe vildfarne programmer. I MS verdenen kan man ofte stoppe et vildfaren program med taskmanageren, i UNIX og dermed Linux verdenen kan man altid stoppe et vildfaren program, bare ved at slå det ihjæl med kill kommandoen.

Hvis man ønsker mere end en ting udført for hvert gennemløb af løkken, skal det der ønskes udført samles i en blok, som vist i programmet for2.c på Figur 1.4.2.

```c
/* Filnavn = for2.c */

#include <stdio.h>

int main()
{
        int i;
        int a;

        for (i = 1; i < 10; i = i + 1)
        {
                a = 10 * i;
                printf("%s%d\n", "a er nu = ", a);
        }
        return 0;
}
```

```
$ cc for2.c
$ ./a.out
a er nu = 10
a er nu = 20
a er nu = 30
a er nu = 40
a er nu = 50
a er nu = 60
a er nu = 70
a er nu = 80
a er nu = 90
$
```

Figur 1.4.2

Vi har blot udvidet programmet med linien:

```c
a = 10 * i;
```

Hvorved vi har opnået at få den lille tabel frem på skærmen.

FORSLAG: Prøv at ændre programmet for2.c så det tæller op til 100.

Funktionen i = i + 1 er faktisk så hyppigt brugt, at den har fået en mere smart måde at gøre det samme på. (at lægge en til den værdi der nu står i en variabel)

i = i + 1 er det samme som at skrive i++. Dette kan udføres med alle talvariabler i C. Hvis vi bruger den teknik i programmet for2.c, og bruger et kneb mere, så har vi programmet for3.c på Figur 1.4.3.

```c
/* Filnavn = for3.c */

#include <stdio.h>

int main()
{
	int i;

	for (i = 1; i < 10; i++)
		printf("%s%d\n", "a er nu = ", i * 10);
	return 0;
}
```

```
$ cc for3.c
$ ./a.out
a er nu = 10
a er nu = 20
a er nu = 30
a er nu = 40
a er nu = 50
a er nu = 60
a er nu = 70
a er nu = 80
a er nu = 90
$
```

Figur 1.4.3

Vi har opnået det samme som før, men variablen a er blevet sparet væk. I stedet for variablen `a`, indsætter vi det udtryk der giver a sin værdi i printf sætningen. (Det er det andet kneb)

En `for` løkke kan have en indbygget `if` sætning, hvis der er behov for det. Programmet for4.c er et eksempel på en ofte anvendte kombination af sætninger:

```
/* Filnavn = for4.c */

#include <stdio.h>

int main()
{
        int i;

        for (i = 1; i < 10; i++)
              {
              if (i == 5)
                     printf("%s\n", "i er nu fem\n");
              }
        return 0;
}
```

```
$ cc for4.c
$ ./a.out
i er nu fem
$
```

Figur 1.4.4

Bemærk, det er ikke nødvendigt at bruge blokke (tuborgparanteser) for at opnå virkningen i dette eksempel, de er blot sat for at starte en diskution om emnet. Blokke er næremre beskrevet i kapitel.

Set med løkkens øjne er der kun en programlinie inde i løkken, at den så indeholder en betingelse med tilhørende linier, det kan for løkken ikke "se". Selv om man udvidde if sætningen med en else del vil det ikke gøre nogen forskel, blot reglerne for if sætningen overholdes.

Det er aldrig en fejl at bruge en blok, men i dette tilfælde er den blot overflødig.

Bemærk at vi hele tiden har lavet indrykninger i kildeteksten, hver gang vi er gået et skridt dybere ned i programmet, det skaber overblik. Indrykning udføres bedst ved at bruge tabulering, men kompileren er ganske ligeglad.

Set med kompilerens øjne kunne programmet for4.c se således ud.

```
/* Filnavn = for4.c */
#include <stdio.h>
int main(){  int i;for (i = 1; i < 10; i = i++)if (i == 5)printf("%s\n", "i er nu
fem\n");}
```

Kompileren er ligeglad, men vi mennesker kan have ret svært ved at se hvad der foregår, specielt i en fejlfindingssitiuation.

En anden løkkestruktur i C er while løkken. Programmet while1.c er et eksempel på hvordan while kan fungere:

```
/* Filnavn = while1.c */

#include <stdio.h>

int main()
{
	int i = 3;

	printf("Klar til affyring!\n");
	while (i > 0)
	{
		printf("%d\n", i);
		i--;
	}
	printf("FYR!\n");
	return 0;
}
```

```
$ cc while1.c
$ ./a.out
Klar til affyring!
3
2
1
FYR!
$
```

Figur 1.4.5

Programmet while1.c tæller baglæns ned fra tre, og stopper når variablen i får værdien nul eller mindre.

Programmet kan måske gøres lidt smartere, ved at udskifte linierne:

```
printf("%d\n", i);
i--;
```

med linien:

```
printf("%d\n", i--);
```

Så kan der også spares et par tuborg.

En interessant detalje er betingelsen i > 0. Man kunne jo i stedet have valgt betingelsen i != 0, der jo vil give det samme resultat. Årsagen til jeg vælger uligheden fremfor en lighed i en stop betingelse er, at den mængde der er sand for uligheden, er mange gange større en mængden af værdier for variablen i der gør ligheden sand. Det har ingen betydning i dette eksempel, men hvad nu hvis vi ændrer linien i--; til linien i = i - 3;?

Det kan jo tænkes vi en dag ville tælle ned i step på tre. I den situation vil betingelsen ikke blive opfyldt, og programmet vil "hænge". Vi vil altså sidde med et gennemtestet program der lige pludseligt "hænger", fordi det er gået ind i en uendelig løkke. Derfor holder jeg af uligheder i stop betingelser.

While strukturen ligner for strukturen en hel del, while strukturen er måske lidt simplere efter nogens opfattelse, da initialiseringen og tilskrivningen er pillet ud for sig selv, men i bund og grund er der ingen forskel i deres funktionalitet. Til gengæld er der forskel på den sidste løkke struktur i C, nemlig do ... while.

``/* Filnavn = do_while1.c */
 	
#include <stdio.h>
 	
int main()
{
        int i = 3;
 	
        printf("Klar til affyring!\n");
        do
              printf("%d\n", i--);
        while (i > 0);
        printf("FYR!\n");
        return 0;
}```
	$ cc do_while1.c ↵
	$ ./a.out ↵
	Klar til affyring!
	3
	2
	1
	FYR!
	$

Figur 1.4.6

Der er en stor forskel på do ... while strukturen og for og while strukturene. I do ... while strukturen vil løkken altid gennemløbes mindst en gang. I dette tilfælde kan det være uheldigt hvis variablen i er initialiseret til nul, det vil betyde at programmet foretager en nedtælling under nul, hvilket er usmart i denne sammenhæng. I andre sammenhænge kan det være fatalt for programmet. Hvis variablen i initialiseres til værdien nul i while1.c programmet, sker der ingen nedtælling. (Prøv det)

Opgave 1.4.1 Omskriv programmet for1.c således at det bruge en while løkke i stedet for en for løkke.

Opgave 1.4.2 Omskriv programmet for1.c således at det bruge en do ..while løkke i stedet for en for løkke.

Opgave 1.4.3 Omskriv programmet while1.c således at det bruge en for løkke i stedet for en while løkke.

Opgave 1.4.4 Skriv et program med en variabel der tælles op fra 1 til 100. Når variablen får værdien 50, skal der udskrives, "nu er vi halvvejs" på skærmen.

Opgave 1.4.5 Udvid programmet fra 1.4.4 således at det skriver, "nu er vi næsten færdige" på skærmen når variablen bliver 90.

Opgave 1.4.6 Udvid programmet fra 1.4.5 således at det tæller baglæns ned til nul på skærmen, når variablens værdi er større end 90.

1.5 Input

Indtil nu har vi ladet vores programmer bruge indkodede data. I dette afsnit vil vi begynde at hente data fra standard input, der normalt er tastaturet. Men for at gøre det lidt nemmere at teste bruger vi piping. Hvis du ikke ved hvad piping er, kan du læse mere om det i en bog om operativsystemer. Men her er en kort forklaring.

I UNIX er der et lille program ved navn cat, i MS verdenen er det en kommando der hedder type, den udskriver indholdet af det efterfølgende argument på skærmen. Hvis du skriver:

```
$ cat  fil.txt ¿
```

vil indholdet af filen ved navn fil.txt blive udskrevet på skærmen. Dette output kan du omdirigere, således at det bliver input til et andet program, i dette tilfælde udgør det input til programmet input1.c i kompileret, altså eksekverbar udgave. Omdirigeringen opnås ved at sætte en lodret streg efter det filnavn man vil have ud. Det kan se således ud:

```
$ cat input1.c | a.out ¿
```

Man kan sige at programmet bruger sig selv som test.

```
/* Filnavn = input1.c */

#include <stdio.h>

int main()
{
	int taeller = 0;

	while (getchar() != EOF)
		taeller++;
	printf("%s%d", "Der var ", taeller);
	printf("%s\n", " byte i filen.");
	return 0;
}
```

```
$ cc input1.c -o input1
$ cat  input1.c | ./input1
Der var 193 byte i filen.
$
```

Bemærk: Tallet 193 kan variere, det afhænger af hvor mange tegn du har brugt i dit program.

Figur 1.5.1

Programmet input1.c starter med at erklære og initiere variablen taeller. Derefter går programmet ind i løkken med betingelsen:

```
getchar() != EOF
```

`getchar()` er en funktion fra `stdio.h` biblioteket. Den har den egenskab, at den henter det næste tegn (char) fra standard input. På grund af den omdirigering vi har lavet tidligere, undgår du at sidde og taste en hel masse tegn ind, det kommer helt af sig selv gennem omdirigeringen.

Når der er hentet et tegn fra standardinput, undersøges det om tegnet er lig med EOF, der betyder End Of File, altså slut på fil. Det vil sige, at programmet løber rundt i løkken lige så mange gange som der er byte i filen, mens taeller tæller antallet. Til sidst er der bare en ting at gøre, nemlig at udskive resultatet af optællingen i linierne:

```
printf("%s%d", "Der var ", taeller);
printf("%s\n", " byte i filen.");
```

Prøv programmet af ved at tælle antallet af byte i den eksekverbare fil.

Ved at lave en mindre ændring i programmet kan vi lave et program der skriver alt input direkte til skærme. Prøv at kigge på koden til programmet input2.c.

```c
/* Filnavn = input2.c */

#include <stdio.h>

int main()
{
	char ind;

	while ((ind = getchar()) != EOF)
		printf("%c", ind);
	return 0;
}
```

```
$ cc input2.c
$ cat input2.c | ./a.out
/* Filnavn = input2.c */


#include <stdio.h>


int main()
{
	char ind;


	while ((ind = getchar()) != EOF)
		printf("%c", ind);
	return 0;
}
$
```

Figur 1.5.2

Programmet minder en del om input1.c, men i stedet for at tælle gemmer vi inputtet, et tegn ad gangen, for derefter straks at skrive det ud på standard output, som er skærmen så længe vi ikke gør noget for at ændre det.

Bemærk linien:

```c
while ((ind = getchar()) != EOF)
```

Den er ny, og der sker også noget nyt her.

Det der først sker er, at variablen `ind` bliver tilskrevet med værdien der kom fra standard input. Når det er gjort, testes om det der kom ind er lig med End Of File. Hvis det ikke er tilfældet fortsætter vi rundt i løkken, skriver tegnet ud, og henter et nyt tegn, hvorefter det testes om det tegn er EOF o.s.v. For at opnå denne virkning er vi nød til at sætte parentes om udtrykket `(ind = getchar())` ellers ville det der kommer fra standard input først være blevet testet om det er EOF, derefter ville variablen ind blive tilskrevet med resultatet af denne test, og det resultat er enten sandt eller falsk, hvilket vil være noget vrøvl at putte ind i variablen ind i denne sammenhæng.

En anden ny ting i dette eksempel er erklæringen:

```
char ind;
```

Her indfører vi datatypen `char`. Char er en forkortelse for character, eller tegn på dansk. Det vil sige variabler af denne type kan indeholde tegn. Det bliver vi mindet om i linien:

```
printf("%c", ind);
```

Hvor procent c angiver, at der skal udskrives et tegn på skærmen. Ved at erklære en char, og initiere den med et bogstav, så kan vi nu lave et lille program der tæller forekomsten af et bestemt bogstav i et givet input, dette er gjort i programmet input3.c på Figur 1.5.3.

```c
/* Filnavn = input3.c */

#include <stdio.h>

int main()
{
        char ind;
        char test = 'a';
        int  taeller = 0;

        while ((ind = getchar()) != EOF)
              if (ind == test)
                      taeller++;
        printf("%s%d\n", "Bogstavet a findes : ", taeller);
        return 0;
}
```

```
$ cc input3.c -o input3
$ cat input3.c | ./input3
Bogstavet a findes : 11
$
```

Figur 1.5.3

Bemærk linien:

```c
char test = 'a';
```

Det er ikke en streng vi arbejder med, men et enkelt tegn, derfor vil det være en fejl at sætte bogstavet a ind mellem gåseøjne. I stedet skal der bruges et apostrof på begge sider af a, for at compileren skal opfatte det som et tegn.

Resten er lige ud af landevejen. Programmet løber rundt i løkken sålænge det ikke har mødt EOF. Mens det løber rundt optælles variablen taeller med en, hver gang vi møder bogstavet a i inputtet. Når programmet møder EOF stoppes løkken, og resultatet udskrives.

Opgave 1.5.1 Skriv et program der er en blanding af input1.c og input2.c. Det betyder programmet skal udskrive input på skærmen, tælle antallet af tegn, og udskrive det til sidst.

Opgave 1.5.2 Skriv et program der tæller antallet af mellemrum i input. Resultatet skal udskrives inden programmet forlades.

1.6 Afrunding

Dette har været en kort gennemgang af nogle enkle programmeringsteknikker, for at få begynderen i gang med at programmere. Inden du går videre vil jeg anbefale du kigger lidt på opgaverne sidst i dette kapitel, inden du går videre til kapitel 2.

Som et lille krydderi er her et par eksempler der kan nydes som de er.

Første er der programmet rente1.c

```
/* Filnavn = rente1.c */

#include <stdio.h>

int main()
{
        int terminer, i;
        float rente, startsum, slutsum;

        terminer = 10;
        rente = 3.5;
        startsum = slutsum = 1000;
        for (i = 1; i <= terminer; i++)
              slutsum = slutsum * (1 + rente/100);
        printf("%s%d", "Efter ", terminer);
        printf("%s%f\n", " terminer\ner summen ", slutsum);
        return 0;
}
```

```
$ cc rente1.c
$ ./a.out
Efter 10 terminer
er summen1410.598877
$
```

Figur 1.6.1

Programmet indfører en ny datatype kaldet float. Det betyder flydende tal, eller kommatal som vi normalt siger på dansk. Hvis vi havde valgt at erklære variablerne rente og slutsum som int, ville resultatet af divisionen rente/100 være blevet nul, på grund af afrunding, og vi kunne ikke bruge programmet til ret meget. Bemærk i øvrigt at der

bruges et punktum som decimalseperator. (Ikke komma som vi bruger i Danmark)
Programmet rente1.c foretager en simpel renteberegning uf fra formlen:

$$\text{Slutsum} = \text{Startsum}(1 + \text{rentefod})^{i}$$

Vi kan altså udføre et trivielt stykke matematik som potensopløftning ved gentagelse.

Programmet sekperdoegn.c der er vist på Figur 1.6.2, viser hvordan man på ganske besværlig vis, kan udregne antallet af sekunder i et døgn. For at gøre beregningen tydelig, er der valgt at bruge fire variabler. Man kan argumentere for at variablerne `sek, min` og `timer` er overflødige, men de er med til at gøre det nemmere at læse programmet.

Den kvikke læser vil straks kunne se, at i stedet for at skrive:

```
for (h = 0; h < timer; h++)
  for (j = 0; j < min; j++)
     for (i = 0; i < sek; i++)
       sekunder++;
```

Kunne man i stedet have skrevet:

```
sekunder = sek * min * timer;
```

Ved at flette løkkerne ind i hinanden har vi altså opnået en multiplikation.

Løkker kan bruges til meget mere end der er vist i dette kapitel, men det vil vi komme meget mere ind på senere.

`/* Filnavn = sekperdoegn.c */`
 	
`#include <stdio.h>`
 	
`int main()`
`{`
`    int sek = 60;`
`    int min = 60;`
`    int timer = 24;`
`    int sekunder = 0;`
`    int h, i , j;`
 	
`    for (h = 0; h < timer; h++)`
`        for (j = 0; j < min; j++)`
`            for (i = 0; i < sek; i++)`
`                sekunder++;`
`    printf("%s\n", "Antal sekunder på et døgn er: ");`
`    printf("%d\n", sekunder);`
`    return 0;`
`}`
	$ cc sekperdoegn.c
	$ ./a.out
	
	Antallet af sekunder på et døgn er:
	86400
	$

Figur 1.6.2

1.7 Opgaver

Opgave 1.7.1 Skriv et program der tæller baglæns fra 10 til nul.

Opgave 1.7.2 Tilpas programmet rente1.c fra afsnit 1.6, således at det benytter en while løkke.

Opgave 1.7.3 Tilpas programmet rente1.c fra afsnit 1.6, således at det benytter en do ... while løkke.

Opgave 1.7.4 Tilpas programmet sekperdoegn.c fra afsnit 1.6, således at det samtidigt benytter en while løkke, en for løkke og en do ... while løkke.

Opgave 1.7.5 Skriv et program der udskriver et vandret histogram på skærmen bestående af asterikser (*), en for hver af et tal der kan være i intervallet 0 .. 70.

Opgave 1.7.6 Tilpas 1.7.5 således, at historgrammet udskrives lodret.

Opgave 1.7.7 Skriv et program der beregner hvor langt der er kørt efter 1, 2, ... 9 timer, med en fast hastighed af 100 km/timen. Resultatet skal udskrives på skærmen.

Opgave 1.7.8 Skriv et program der udskriver antallet af bogstaver i et input. Hint: Hvis det ikke er et mellemrum, må du betragte det som et bogstav.

Opgave 1.7.9 Udvid programmet fra 1.7.8, således at det det udskriver forholdet mellem bogstaver og mellemrum, i et givet input.

2 Grundbegreber

Efter at have fået et førstehåndsindtryk af C i kapitel 1, vil vi nu bevæge os over i nogle grundliggende programmeringsteknikker. Først er der blokke.

2.1 Blokke

Blokke bruges til at fortælle kompileren, hvilke dele af programmet der hører sammen. En blok starter altid med en { (kaldet starttuborg), og slutter altid med en } (kaldet sluttuborg).

Foreløbigt har vi benyttet blokke uden at tænke nærmere over deres funktion, hvilket har begrænset os en smule. Prøv at kigge på koden i blok1.c i Figur 2.1.1.

```
/* Filnavn = blok1.c */

#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 5; i++)
        printf("%s", "Udskrives 5 gange.\n");
    printf("%s", "Udskrives en gang\n");
    return 0;
}
```

```
$ cc blok1.c
$ ./a.out
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives en gang
$
```

Figur 2.1.1

Det fremgår klart af kildeteksten, hvad der forventes at ske, og det er da også det oplagte der sker. Men hvad nu, hvis der i stedet havde stået som vist i programmet blok2.c?

```c
/* Filnavn = blok2.c */

#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 5; i++)
        printf("%s", "Udskrives 5 gange.\n");
        printf("%s", "Udskrives også 5 gange\n");
    return 0;
}
```

```
$ cc blok2.c
$ ./a.out
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives 5 gange.
Udskrives også 5 gange
$
```

Tabel 2.1.2

Her vil det oplagte ikke ske. Forvirringen skyldes indrynkning og teksten i linien:

```c
printf("%s", "Udskrives også 5 gange\n");
```

der i denne sammenhæng er misvisende. Det skyldes, at `for` løkken betragter den efterfølgende linie som det der skal udføres, og en sådan linie skal afsluttes med et semikolon. Hvis den efterfølgende linie er et startblokmærke (starttuborg), så vil alt der er inde i blokken blive udført en gang, hver gang løkken tager en omgang.

Hvis vi vil have det oplagte i blok2.c til at ske, skal der indsættes blokmærker for at fortælle compileren, hvad der skal udføres for hver omgang i løkken. Det kan se ud som vist på Figur 2.1.3.

```c
/* Filnavn = blok3.c */

#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 5; i++)
    {
        printf("%s", "Udskrives 5 gange.\n");
        printf("%s", "Udskrives også 5 gange\n");
    }
    return 0;
}
```

```
$ cc blok3.c
$ ./a.out
Udskrives 5 gange.
Udskrives også 5 gange
Udskrives 5 gange.
Udskrives også 5 gange
Udskrives 5 gange.
Udskrives også 5 gange
Udskrives 5 gange.
Udskrives også 5 gange
Udskrives 5 gange.
Udskrives også 5 gange
$
```

Tabel 2.1.3

Nogle programmører foretrækker at placere startblokmærket på samme linie som løkken. Jeg mener det er mere overskueligt, hvis man placerer start og slut blokmærkene lodret over hinanden. Her i bogen vil jeg dog placere startblokmærket på samme linie hvor løkken starter, hvis det er praktisk af pladshensyn.

Det korte af det lange er, at de to følgende eksempler begge er helt legitime måder at bruge blokke på. Den første er ret udbredt, den anden er det forhåbentligt ikke, da den er noget uoverskuelig.

```c
for (i = 0; i < 5; i++)  {
     printf("%s", "Udskrives 5 gange.\n");
     printf("%s", "Udskrives også 5 gange\n");
}

for (i = 0; i < 5; i++)
{
          printf("%s", "Udskrives 5 gange.\n");
     printf("%s", "Udskrives også 5 gange\n");    }
```

Men compileren kan ikke kende det ene fra det andet.

Blokke kan placeres frit inde i koden, det betyder at det er helt legalt at skrive følgende:

```
for (i = 0; i < 5; i++)
{
     printf("%s", "Udskrives 5 gange.\n");
     {
     }
     printf("%s", "Udskrives også 5 gange\n");
}
```

Altså at have en tom blok inde i en blok. Selv om det ikke giver nogen mening i dette eksempel, kan man bruge det for at markere hvor man har til hensigt at udvidde programmet.

Der må ikke være uparrede blokmærker mellem en start og en slutblok. Følgende brug af blokke er forkert, og vil afstedkomme en compilerfejl.

```
for (i = 0; i < 5; i++)
{
     printf("%s", "Udskrives 5 gange.\n");
     {    /* Fejl: Startblok uden slutblok */
     printf("%s", "Udskrives også 5 gange\n");
}
```

Blokke har også en anden vigtig egenskab, i det de definerer rækkevidden, kaldet scope på engelsk. Begrebet rækkevidde dækker over, hvor langt væk en variabel kan ses, i forhold til hvor den blev erklæret. Programmet scope1.c er et ganske dårligt skrevet program, men i denne sammenhæng illustrerer det begrebet rækkevidde:

```c
/* Filnavn = scope1.c */

#include <stdio.h>

int main()
{
    int a = 10;

    printf("%d\n", a);
    {
        int a = 20;

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

```
$ cc scope1.c
$ ./a.out
10
20
10
$
```

Tabel 2.1.4

Årsagen til programmet er dårligt skrevet er, at variablen a erklæres to gange. Hvis der ikke havde været brugt blokmærker, ville dette have afstedkommet en kompilerfejl. De to variabler, der begge hedder a, er to forskellige variabler, det skyldes at de er erklæret i hver sin blok. Hvis a ikke blev erklæret inde i den inderste blok, så ville variablen a stadig være synlig for printf, men det vil være en anden a, den med værdien 10, hvorfor output ville have set således ud:

```
$ scope1
10
10
10
$
```

Derfor er det en rigtig dårlig ide at have flere variabler med samme navn, kun adskilt af deres rækkevidde.

Når en blok oprettes inde i en anden blok kaldes det nesting. Det der gælder i en blok, gælder også i de blokke som nestes, men det modsatte er ikke tilfældet, som vi så i programmet scope1.c.

Rækkevidden er årsagen til jeg ønsker blokmærkerne stående med samme indrykning over hinanden, det gør efter min opfattelse rækkevidden tydeligere.

2.2 Datatyper

Datatyper bruges til at fastslå, hvilken type af data vi ønsker at opbevare i variabler af den givne type. Grundliggende findes der 4 datatyper i C, dertil vil der komme dem vi selv kan oprette, men det kommer vi først til meget senere. De fire grundtyper er:

char	Bruges til at opbevare data af typen tegn. (abc .. &%/ .. 123)
int	Bruges til at opbevare hele tal. Det interval der dækkes afhænger af den enkelte kompiler.
float	Bruges til at opbevare kommatal. Det interval der dækkes afhænger af den enkelte kompiler.
double	Bruges til at opbevare kommatal med den dobbelte præsision af float. Det interval der dækkes afhænger af den enkelte kompiler.

Tabel 2.2.1

Typen `int` kan suppleres med `short` eller `long`.

```
short int a;
```

Hvis man erklærer variablen a som ovenfor betyder det, at variablen `a` fylder mindre eller det samme i hukommelsen som en almindelig `int`. Hvis man i stedet erklærer variablen a således:

```
long int a;
```

Betyder det at variablen `a` fylder det samme eller mere end en en almindelig `int` i hukommelsen. Der er altså ikke nogen særlig klar speceficering af hvad der fylder hvor meget, så du må læse i manualen til din kompiler hvor meget de forskellige typer fylder der. Når man bruger long eller short kan int helt udelades.

Typerne char og int kan suppleres med signed eller unsigned, hvilket betyder med og uden fortegn. Hvis eksempelvis en char er repræsenteret i computeren med 8 bit, så kan en unsigned char repræsentere værdier fra 0 til 255, og en signed char kan repræsentere fra -128 til 127.

Hvis man ikke har styr på en types begrænsning kan komme til at skrive et program i stil med forever1.c:

```
/* Filnavn = forever1.c
   Dette program vil køre uendeligt
   da stopbetingelsen aldrig vil blive
   opfyldt. Programmet kan normalt stoppes,
   ved at trykke samtidigt på contro og
   C tasten.
*/

int main()
{
     char i;

     for (i = 200; i < 300; i++)
          printf("%s%d\ni = ", i);
     return 0;
}
```

Årsagen til stopbetingelsen aldrig bliver opfyldt er, at en char kan (normalt) kun tælle fra 0 .. 255. Når variablen i har værdien 255, og der bliver lagt en til, så vil den nye værdi for i være 0 (nul). Da variablen i aldrig bliver større end 255, så vil løkken aldrig stoppe. Nogle kompilere vil advare mod denne slags fejl, men du kan ikke regne med det.

Opgave 2.2.1 Skriv et program der undersøger hvornår en signed int skifter fortegn

Opgave 2.2.2 Skriv et program der undersøger, hvor stort et tal der kan repræsenteres med en int i din kompiler.

2.3 Typekonvertering

Når der er flere typer til rådighed, er der også mulighed for at begå fejl ved at bruge en forkert type, eller ved at tilskrive variabler fra en type, til variabler af en anden type.

Programmet typekonv1.c viser hvad der sker hvis man forsøger at tilskrive en variable af typen int med en variabel af typen float. Heltalsvariablen smider alt bag kommaet væk, der sker ingen afrunding, kun en bortsmidning.

```
/* Filnavn = typekonv1.c */

#include <stdio.h>

int main()
{
    int a;
    float pi = 22.0/7.0;

    a = pi;
    printf("%d\t%f\n", a, pi);
    return 0;
}
```

```
$ cc typekonv1.c
$ ./a.out
3       3.142857
$
```

Figur 2.3.1

Hvis resultatet af divisonen var blevet mindre end 1.0000 ville a have fået værdien nul. Der sker altså ingen afrunding når en int tilskrives med en float eller double, men alt bag kommaet (der normalt er et punktum) smides væk. Hvis linien:

```
float pi = 22.0/7.0;
```

Blev ændret til:

```
float pi = 22/7;
```

ville vi have haft det omvendte problem, nemlig at variablen pi ville få resultatet af en heltalsdivision, hvilket i dette tilfælde er 3.0000. Årsagen er, at 22 er et heltal, hvorimod 22.0 er et kommatal, og når kompileren ser der står 22/7, så mener compileren at den skal dividere to heltal.

I C er typen char også en taltype. Den er typisk på 8 bit, og int typen er ofte bredere. Det betyder at man kan skrive:

```
int a = 10;
char b = a;
```

Men hvis indholdet af variablen a er større end det der kan rummes i en char, (typisk 255) så bliver de overskydende bit blot smidt væk. Det betyder i ovennævnte eksempel, at hvis a = 256 ville b blive nul, hvis a = 257 ville b = 1 o.s.v. Programmet typekonv2.c viser hvordan man kan bruge en char til at tælle med, på en lidt usædvanlig måde:

```c
/* Filnavn = typekonv2.c */

#include <stdio.h>

int main()
{
    char a;

    for (a = 'A'; a <= 'Z'; a++)
        printf("%c\t%d\n", a, a);
    return 0;
}
```

```
$ cc typekonv2.c
$ ./a.out
A    65
B    66
C    67
...
Z    90
$
```

Figur 2.3.2

Her starter vi med at sætte variablen `a` lig med bogstavet A, som har talværdien 65. Herefter lægges der en til værdien i a for hver gang løkken tager en omgang, sålænge a's værdi er mindre eller lig med værdien af bogstavet Z. Med dette lille program har vi faktisk lavet en ASCII tabel, det vil sige, en tabel over hvilke talværdier hvert bogstav har.

Hvis man har en type i brug, og skal konvertere den til en anden type, kan det gøres med en cast. Programmet cast1.c viser hvad der sker, hvis man forsøger at skrive et heltal ud som en float, og hvad man kan gøre for at kunne udskrive et heltal som en float.

```c
/* Filnavn = cast1.c */

#include <stdio.h>

int main()
{
    int a = 12;

    printf("%f\n", a);
    printf("%f\n", (float) a);
    return 0;
}
```

```
$ cc cast1.c
$ ./a.out
0.00000
12.00000
$
```

Figur 2.3.3

Linien:

```
printf("%f\n", a);
```

giver tilsyneladende et forkert output. Årsagen er, at vi fortæller compileren, at nu kommer der et float der skal udskrives på skærmen. Den hopper compileren på, og gør det så godt den nu kan, hvilket set med vores øjne er forkert. Men det er bare en programmørfejl. Dette problem kan løses ved at udføre en cast, man kan sige at vi "kaster" en variabel af en type, over i en variabel af en anden type. Det er det der sker i linien:

```
printf("%f\n", (float) a);
```

hvor variablen `a` bliver "kastet" over i en float. Selve (float) er faktisk en unær operator, men andre typer end float kan også kastes, som vi vil se senere.

Opgave 2.3.1 Skriv et program der omregner en Fahrenheit temperatur til Celcius grader. Programmet skal udskrive en tabel over omregningen i intervallet 0 til 100, i step på 10. Hint: C = (5/9) gange (F - 32)

Opgave 2.3.2 Skriv et program der indkoder et tekstinput efter Cæsarkoden. Hint: Cæsarkoden betyder at a = b, b = c ... z = a, A = B ... Undlad at konvertere de danske bogstaer.

2.4 Formatering

Hidtil har vi brugt formattering med forsigtighed, mest for ikke at skræmme begynderen løbende bort, men nu skal vi altså til det der ser meget værre ud end det er. Formattering bruges til at formattere output på en pæn måde. Det bruges også til at definere typen på det output vi har.

Lad os se på to eksempler der ligner hinanden, programmerne format1.c og format2.c.

`/* Filnavn format1.c */`
 	
`#include <stdio.h>`
 	
`int main()`
`{`
`        int i = 65;`
 	
`        printf("%d\n", i);`
`        return 0;`
`}`
	$ cc format1.c
	$ ./a.out
	65
	$

`/* Filnavn format2.c */`
 	
`#include <stdio.h>`
 	
`int main()`
`{`
`        int i = 65;`
 	
`        printf("%c\n", i);`
`        return 0;`
`}`
	$ cc format2
	$ ./a.out
	A
	$

Figur 2.4.1

De to programmer divergerer udelukkende på formatteringen i output, programmet format1.c har en printf sætning der ser således ud:

```
printf("%d\n", i);
```

Og programmet format2.c har en printf sætning der ser således ud:

```
printf("%c\n", i);
```

Forskellen er, at format1.c beder om at få variablen i udskrevet som et digit (%d) mens format2.c beder om at få variablen i udskrevet som et tegn (%c). Da talværdien af bogstavet store A på de fleste computere er 65, udskrives bogstavet A.

Procent d (%d) bruges altså til at definere at output skal være et digit. C byder på flere formatteringsmuligheder, her er en liste.

\a	Udskriver bell. (klokke/beep lyd)	\v	Vertikal tabulering
\b	Backspace (Slet bagud)	\\	backslash
\f	Formfeed (Ny side)	\?	Spørgsmålstegn
\n	newline (Indsæt linieskift)	\´	Apostrof
\0	Linie slut (EOL)	\"	gåseøje
\r	Carrige return (cursor helt t.v)	\ooo	Oktal nummer
\t	Horisontal tabulering	\xhh	Hex nummer

Tabel 2.4.1

Lad os se på et program der bruger bell (\a) og tabulering (\t). Programmet format3.c udskriver en tabel der konvergerer cm til tommer. For at få tallene til at stå lidt pænt, indsætter vi en tabulering mellem de to decimalvariabler.

`/* Filnavn = format3.c */`
 	
`#include <stdio.h>`
 	
`int main()`
`{`
`    int i;`
`    float tomme = 2.54;`
 	
`    for (i = 1; i < 5; i++)`
`        printf("%d\t%f\n", i, tomme * i);`
`    printf("%s\n\a", "Slut.");`
`    return 0;`
`}`
	$ cc format3.c
	$ ./a.out
	1 2.5400
	2 5.0800
	3 7.6200
	4 10.1600
	Slut.
	(Beep)
	$

Figur 2.4.2

Ved at bruge lidt formattering kan vi få vores output til at se pænere ud, det kan nu stadig gøres bedre. Vi kan nemlig kombinere formateringen med en information om hvor mange tegn der skal udskrives. Eksempelvis betyder `%3d` at der skal udskrives et decimaltal med 3 tegns bredde. Hvis vi skriver `%5.2f` betyder det, at der skal udskrives et kommatal, med 5 tegn før kommaet (der er et punktum) og to tegn efter kommaet. Ved at bruge det kan vi få output til at se lidt kønnere ud, som i programmet format4.c:

```c
/* Filnavn = format4.c */

#include <stdio.h>

int main()
{
    int i;
    float tomme = 2.54;

    for (i = 1; i < 5; i++)
        printf("%d\t%6.2f\n", i, tomme * i);
    printf("%s\n\a", "Slut.");
    return 0;
}
```

```
$ ccformat4.c
$ ./a.out
1         2.54
2         5.08
3         7.62
4        10.16
Slut.
(Beep)
$
```

Figur 2.4.3

Det kan virke lidt pudsigt, men en formatering fylder præcis en char, selv om den kræver to tegn at skrive på skærmen. Vi har tidligere set at en char kan tilskrives således:

```c
char input = 'A';   /* Legalt */
```

Vi kan tilskreve variablen med et formateringstegn i stedet, det vil se således ud:

```c
char input = '\n';  /* Legalt */
```

Det er helt legalt, fordi et formateringstegn er en char. Det ville derimod være en fejl at skrive:

```c
char input = 'ab';  /* FEJL */
```

Der må ikke være mere end et tegn i en char.

Opgave 2.4.1 Skriv et program der udskriver den lille tabel pænt på skærmen.

Opgave 2.4.2 Skriv et program der gennemser alt i input. Hver gang programmet møder et newline tegn skal det give et beep.

2.5 Operatorer og operander

Operatorer er nogle hyggelige småfyre, der udfører operationer på nogle operander, og derefter afleverer et resultat. Nogle typiske operatorer er plus (+), minus (-), gange (*) og division (/). Fælles for disse er, at de er binære operatorer. De kaldes binære fordi de tager to operander. Unære operatorer tager en operand og tertiære operatorer tager 3 operander. I appendiks A er der en liste over alle operatorer.

Vi har tidligere kigget på den unære operator ++, den bruges til at lægge en til en variabels værdi, variablen er operanden. Operatoren kan anvendes postfix og prefix. Hvis operatoren anvendes pretfix, betyder det at den står foran variablen, således:

```
++a;
```

Hvis operatoren er postfix er den placeret bag variablen, således:

```
a--;
```

Hvis man bruger prefixnotationen betyder det, at variablen bliver opdateret, før den indgår i en evt yderligere operation. Hvis notationen er postfix opdateres variablen efter operationen.

Programmet notation.c viser forskellen mellem prefix og postfix notation:

```c
/* Filnavn = notation1.c */
/* Dette program viser forskellen */
/* på prefix og postfix notation  */

#include <stdio.h>

int main()
{
    int a = 10;
    int b = 10;

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

```
$ cc format3.c 
$ ./a.out 
10
11
11    11
$
```

Figur 2.5.1

Som det ses er værdien af både a og b 11 når programmet er færdigt. Forskellen er, at a bliver udskrevet på skærmen før der bliver lagt en til a, variablen b bliver opdateret før den bliver skrevet ud på skærmen.

Notationsformen kan virkelig give overskuelighedsproblemer, hvis man tilskriver en variabel og samtidig tester på den.

```c
    int a = 10;
    int b = 10;

    if (++a > 10)   /* Betingelse er sand: a = 11 når der 
testes */
    if (b++ > 10)   /* Betingelsen er falsk: b = 10 når der 
testes */
```

Det er bestemt ikke forkert at bruge nogen af de to tekniker, man skal bare vide hvad det er man laver.

En anden nyttig operator er modolusoperatoren (%), den afleverer den rest der opstår ved en division. Dette er specielt nyttigt til at tælle, hvor mange gange et tal går op i et andet, da resten netop er nul i det tilfælde. Programmet op2.c demonstrerer netop dette.

```c
/* Filnavn = op2.c */

#include <stdio.h>

int main()
{
    int i = 0;

    while (i < 100)
    {
        if (i % 10 == 0)
            printf("%s%2d\n", "i = ", i);
        i++;
    }
    return 0;
}
```

```
$ cc op2.c
$ ./a.out
i =  0
i =  10
i =  20
i =  30
i =  40
i =  50
i =  60
i =  70
i =  80
i =  90
$
```

Figur 2.5.2

Den interessante linie er:

```c
if (i % 10 == 0)
```

Der står: Hvis den rest der opstår, når i bliver divideret med 10 er nul, så udskriv i.

Når man programmerer kommer man ofte ud for at skrive noget i retning af:

```c
a = a + 2;
```

Derfor er der i C en måde at gøre det på mere enkelt, det ser således ud:

```c
a += 2;
```

Ved at benytte den teknik, kan programmet op2.c omskrives som vist i programmet op3.c.

```
/* Filnavn = op3.c */

#include <stdio.h>

int main()
{
    int i = 0;

    while (i < 100)
    {
       printf("%s%2d\n", "i = ", i);
       i += 10;
    }
    return 0;
}
```

```
$ cc op3.c
$ ./a.out
i =  10
i =  20
i =  30
i =  40
i =  50
i =  60
i =  70
i =  80
i =  90
i = 100
$
```

Figur 2.5.3

Man kan jo også tage skridtet fuldt ud, og spare en programmeringslinie helt væk, som vist i programmet op3a.c. Bemærk parentesen rundt om deludtrykket

```
(i += 10)
```

Den er nødvendigt fordi operatoren < (mindre end) har højere præcedens end += operatoren. Det betyder, at hvis parentesen ikke var er der, så vil der først bliv testet om 10 er mindre end 100), derefter vil der blive adderet til i. De 10 altid er mindre end 100, vil programmet i givet fald ende op i en uendelig løkke.

```
/* Filnavn = op3a.c */

#include <stdio.h>

int main()
{
    int i = 0;

    while ((i += 10) < 100)
        printf("%s%2d\n", "i = ", i);
    return 0;
}
```

```
$ cc op3a.c
$ ./a.out
i =  10
i =  20
i =  30
i =  40
i =  50
i =  60
i =  70
i =  80
i =  90
i = 100
$
```

Figur 2.5.4

Teknikken kan bruges med alle operatorer. I appendix A er en liste over mulighederne.

Opgave 2.5.1 Skriv et program der tæller fra et til og med 10. Værdierne skal udskrives for hver omgang i løkken. Programmet skal bruge en for løkke og opgraderingen af løkkekontrolvariablen skal være postfix.

Opgave 2.5.2 Tilpas programmet fra opgave 2.5.1 således at kontrolvariablen tilskrives prefix. Hvad sker der?

Opgave 2.5.3 Skriv et program der kan omregne tid, opgivet i minutter, til timer og minutter.

Opgave 2.5.4 Skriv et program der omregner tid, opgivet i dage, til timer.

2.6 If ... else

Med if kan man tage beslutninger. If kan udviddes med blokke, hvis man vil have mere end en ting udført, ud fra en given betingelse. Egentlig er det forkert at bruge begrebet betingelse, da if faktisk blot undersøger om der står nul eller ikke nul inde i parantesen. Lidt forenklet ser if strukturen således ud

```
if (udtryk)
     gør dette;
else
     gør dette;
```

Hvor udtryk er en beregning der kan give et heltals resultat. Hvis resultatet er forskellig fra nul er det sandt, hvis resultatet er lig med nul er det falsk. Det betyder at følgende er en legitim men nok lidt pudsig måde at bruge if på:

```
if (a - b)
     ....
```

Der står nemlig if resultatet af beregningen a minus b er forskelig fra nul, så skal næste linie udføres. Det betyder at der lige så godt kunne have stået:

```
if (a != b)
     ....
```

En anden sjov virkning er, man kan skrive:

```
int i;

if (i)
     printf("%s\n", "i er ikkenul");
else
     printf("%s\n", "i er nul");
```

Dette skyldes at if ikke tjekker op mod nogen sand/falsk værdi, men tjekker rent

nummerisk om der står nul eller ikke nul inde i "betingelsen". Det betyder, at programmet op2.c kan skrives, som det ses i programmet op2a.c

```
/* Filnavn = op2a.c */

#include <stdio.h>

int main()
{
    int i = 0;

    while (i < 100)
    {
        if ( !(i % 10) )
            printf("%s%2d\n", "i = ", i);
        i++;
    }
    return 0;
}
```

```
$ cc op2a.c 
$ ./a.out 
i =  0
i =  10
i =  20
i =  30
i =  40
i =  50
i =  60
i =  70
i =  80
i =  90
$
```

Figur 2.6.1

Igen er det værd at bemærke parentesen rundt om

```
(i % 10)
```

Den er nødvendig på grund af operatorenes præcedens. Hvis du kommer i tvivl om hvornår du skal sætte parentes rundt om et udtryk, for ikke at blive ”forstyret” af operatorer med højere præcedens, så brug parenteser rundt om dine udtryk. Parenteser er også operatorer, og de har den højeste præcedens.

Ofte kan det være nødvendigt at have mere end en betingelse for at udføre en opgave, det kaldes at neste. (to nest = at bygge rede) En nestet if-konstruktion er vist her:

```
if (a == b)
     if (c == d)
     {
          Gør dette
          Gør dette
     }
```

Vi kan i stedet benytte AND operatoren, således at det hele udføres ud fra en sammensat betingelse.

```
if (a == b && c == d)
     {
          Gør dette
          Gør dette
     }
```

De to & tegn udgør tilsammen AND operatoren. If sætningen udføres i dette tilfælde kun hvis a = b og c = d. Hvis vi i stedet vil udføre noget, hvis blot en ud af to betingelser er sande, kan det se således ud.

```
if (a == b || c == d)
     {
          Gør dette
          Gør dette
     }
```

De to lodrette streger er OR operatoren (OR betyder eller). Det der er inde i if sætningen udføres altså hvis a = b eller hvis b = c.

Dette sidste eksempel kunne også være lavet således:

```
if (a == b)
     {
          Gør dette
          Gør dette
     }
if (c == d)
     {
          Gør dette
          Gør dette
     }
```

Men man kan altså spare noget skriveri, ved at bruge AND og OR operatorene.

Når man får noget erfaring med at bruge if strukturen, så vil man også begynde at neste en del. I den forbindelse skal man være opmærksom på den følgende situation, hvor det er svært at gennemskue hvad der sker.

```
/* Filnavn = if1.c
   Dette program virker ikke
   som man kunne forledes til
   at tro.
*/

#include <stdio.h>

int main()
{
    int a = 6;
    int b = 5;

    if (a <= b)
        if (b == 5)
            printf("%s\n", "b = 5");
    else   /* Her sker fejlen */
        printf("%s\n", "a > b");
}
```

```
4 cc if1.c
$ ./a.out

$
```

Figur 2.6.2

Egentlig er der ikke nogen fejl i programmet, set med compilerens øjne. Fra dens synspunkt af er det helt naturligt, at hvis b er lig med fem, så udskrives der b = 5 på skærmen. Hvis b ikke er lig med fem, så står der jo en else, men den er knyttet til den inderste if, programmet kommer kun så dybt ind, hvis a <= b. Det er en typisk fejl man kan opleve når man nester, p.gr.a. indrykningerne kan et menneske få den ide, at else udføres hvis a er større end b, hvilket er forkert. Hvis du kommer i tvivl om hvilke dele der hører sammen når du nester, så brug blokke. Blokke skader aldrig, men kan være med til at skabe overblik. Efterhånden som du får mere erfaringen kan du begynde at skrive mere smarte programmer, men prøver du det på nuværende tidspunkt, kører du nok bare sur i det, og dropper til sidst programmeringen.

Med nestning kan man altså nemt rode sig ud i en værre redelig, men man kan ikke undgå at bruge nestning. Så mit råd til nye programmører er som altid: KIS (Keep It Simple), eller sagt på en anden måde:

Enkle systemer laver enkle fejl, komplicerede systemer laver komplicerede fejl.

En lidt speciel ting i C er, at man også kan skrive:

```
c = (a > b) ? a : b;
```

Det betyder, at hvis (if/?) a er større end b, så sæt c = a, ellers sæt c = b. Det er blot en lidt speciel måde at bruge if strukturen på. Erfarne programmører bruger teknikken, men jeg vil ikke anbefale begynderen at bruge tid på den teknik, før han begynder at kede sig og trænger til nye udfordringer.

Opgave 2.6.1 Skriv et program der indeholder to løkker, inden i hindanen. Begge løkker skal tælle fra et til og med 10. Når kontrolvariablen i den inderste løkke er lig med kontrolvariablen i den yderste løkke, skal det udskrives på skærmen. Inden du eksekverer programmet bør du måske spørge dig selv, hvor mange gange vil der blive skrevet til skærmen?

Opgave 2.6.2 Skriv et program der undersøger hvor store tal der kan være i en double på dit system.

2.7 switch

Hvis man ønsker at træffe en beslutning, ud fra en pallette af muligheder, kan man bygge sit program op af mange if sætninger. En del af et program der kan kende en person ud fra et nr. kan se således ud:

```
if (ind == 1)
     printf("%s\n", "Hej Peter");
if (ind == 2)
     printf("%s\n", "Hej Charlotte");
if (ind == 3)
     printf("%s\n", "Hej Hans");
```

Det er en ganske normal situation. Man har et input, ud fra det ønsker man at ens program skal foretage noget. En god gammeldags menu er et godt eksempel. Når man får en masse linier med if sætninger, så kan det hurtigt blive uoverskueligt, derfor er der udviklet en konstruktion der kaldes switch. (switch = kontakt)

```
/* Filnavn = switch1.c */

#include <stdio.h>

int main()
{
    char ind;

    printf("\t%s\n", "1 = Peter");
    printf("\t%s\n", "2 = Charlotte");
    printf("\t%s\n", "3 = Hans");
    printf("%s\n", "Vælg 1 ... 3");
    ind = getchar();
    switch (ind)
    {
       case '1': printf("%s\n", "Hej Peter");
            break;
       case '2': printf("%s\n", "Hej Charlotte");
            break;
       case '3': printf("%s\n", "Hej Hans");

       default:  printf("%s\n", "Dig kender jeg ikke.");
    }
    return 0;
}
```

```
$ cc switch1.c
$ ./a.out
    1 = Peter
    2 = Charlotte
    3 = Hans
Vælg 1 ... 3
3
Hej Hans
$
```

Figur 2.7.1

Der et par nye tricks i programmet switch1.c. Programmet skriver en lille menu på skærmen, beder brugeren om input, og gemmer det i variablen ind. Det sker i linien:

```
ind = getchar();
```

Vi har tidligere benyttet denne teknik, men hvis du ikke er typen der eksperimenterer lidt, så er det første gang vi henter noget direkte fra brugerens indtastning.

`getchar()` er en funktion i `stdio.h` biblioteket. Den fungerer på den måde, at programmet venter til der bliver indtastet et tegn på tastaturet, det tegn returneres som en int værdi fra funktionen, i dette eksempel gemmes værdien i variablen ind, som der derefter testes på i switchen. getchar() kan godt virke lidt underlig første gang man bruger den, det er nemlig muligt at indtaste mange tegn, inden man trykker på enter, og getchar() gør intet før der er trykket på enter. Men det skal du ikke tænke så meget over lige nu.

(Værd opmærksom på, at der kun kan indtastes et tegn som svar på menuen, yderligere indtastninger vi blive ignoreret)

Linien:

```
switch (ind)
```

Betyder, at nu starter switch strukturen, og den tester på variablen ind. Bemærk; der er ingen semikolon efter denne linie, og det er en fejl at sætte et.

Alt hvad der skal håndteres af switchen skal være inde i den efterfølgende blok. Der skal bruges en blok i forbindelse med en switch, også selv om der kun er en linie inde i switchen.

Derefter følger linien

```
case '1': printf("%s\n", "Hej Peter");
```

I den linie står der faktisk, hvis ind = '1' (tegnet 1, ikke tallet et) så udskriv Hej Peter til skærmen. Den efterfølgende linie med break; medfører, at der breakes ud af switchen. Hvis linien ikke hvar der ville switchen fortsætte med at teste i den næste case linie.

Den sidste linie i switchen indeholder default. Default er engelsk for noget i retning af standard. I denne samenhæng medfører det, at hvis ingen af de ovenstående betingelser er opfyldt, så udføres det der stå i default. Lige som else i sammenhæng med if, kan default

udelades.

Hvis en case i en switch er sand, så udføres den efterfølgende linie. Hvis man ønsker at udføre mere end en linie per case, skal man bruge blokke.

Opgave 2.7.1 Skriv et program der ud fra et input mellem 1 .. 9, udskriver et månedsnavn. Januar er måned nr. 1.

2.8 Funktioner

En funktion er faktisk bare en blok med et navn, og med mulighed for at overføre parametre til og fra blokken. Vi kan sammenligne en funktion med "black box" (sort boks) berebet. En black box gør et stykke arbejde for os uden at vi behøver overveje, hvordan den bærer sig ad. Hvis vi løfter telefonen og ringer til en ven, sker der et ud af to, enten får vi forbindelse, eller også gør vi ikke. Vi kan altså se det samlede telefonsystem, med kabler og telefoncentraler som en sort kasse, vi giver den et input (telefonnr) og den giver os enten en forbindelse, eller en fejlmelding.

I pseudokode kan det se således ud:

```
void ring()
{
     Løft telefonen
     if (klartone)
          ring nr.
     else
          tilkald reperatør
     return 0;
}
```

Man behøver ikke at være et geni, for at finde ud af hvad koden går ud på, og hvordan den arbejder.

I C skal en funktion bestå af følgende dele:

```
returtype funktionsnavn(argument erklæringer)
{
     erklæringer og udtryk
     return udtryk;
}
```

Returtype er enten void, en af typerne fra tabel 2.2.1, eller en brugerskabt type, som vi ikke har arbejdet med endnu. Udeladelse af returtypen er en fejl, og vil medføre en fejlmeddelelse under kompileringen.

Hvis returtypen er void betyder det, at funktionen ikke returnerer nogen type. (void betyder tomhed) Hvis void vælges som returtype, er det en fejl at benytte return inde i funktionen.

Funktionsnavn er et hvilket som helst navn, begyndene med et tal, eller et bogstav. Længden er i princippet uendelig, men de fleste kompilere benytter kun et antal af de første tegn. (Kontroler med din kompilermanuel, hvor mange tegn i funktionsnavne din kompiler kan håndtere.) Det er en god ide at give funktioner navne der afspejler deres funktion. Det er let at gætte sig til hvad en funktion ved navn givMigEtTilfædigtTal gør, men det er besværligt at finde ud af hvad en funktion ved navn yX3entry gør.

Det er ikke tilladt at have mellemrum i et funktionsnavn, benyt i stedet store bogstaver i starten af hvert ord, det virker udmærket.

Argumenterklæringerne er navne og typer på de data der ønskes overført til funktionen. Disse data er at opfatte som variabler der er lokale for denne funktion.

Selve metodekroppen er indeholdt i en blok.

Returtypen skal være samme type som den erklærede, den kan evt. konverteres med en cast, ellers opstår en compilerfejl. Det er en fejl at have et return udtryk, hvis returtypen er erklæret som void.

Programmet funktion1.c viser hvordan man kan bruge en funktion.

```
/* Filnavn = funktion1.c */

#include <stdio.h>

void minFunktion()
{
    printf("%s\n", "Skriver fra minFunktion()");
}

int main()
{
    printf("%s\n", "Skriver fra main()");
    minFunktion();
    return 0;
}
```

```
$ cc funktion1.c 
$ ./a.out 
Skriver fra main()
Skriver fra minFunktion()
$
```

Figur 2.8.1

Programmet har to funktioner, main som vi kender, og minFunktion som kaldes fra main funktionen. Bemærk at minFunktion kommer før main i kildeteksten, det er nødvendigt hvis main skal kunne "se" funktionen. Hvis vi ønsker at placere en kaldt funktion efter den er kaldt, skal der laves en prototype til funktionen. Det er vist i programmet funktion2.c:

```c
/* Filnavn = funktion2.c */

#include <stdio.h>

void minFunktion();

int main()
{
    printf("%s\n", "Skriver fra main()");
    minFunktion();
    return 0;
}

void minFunktion()
{
    printf("%s\n", "Skriver fra minFunktion()");
}
```

```
$ cc funktion2.c
$ ./a.out
Skriver fra main()
Skriver fra minFunktion()
$
```

Figur 2.8.2

En prototype er altså en erklæring af en funktion der er implementeret et andet sted.

Hvis vi nu ønskede at funktionen skulle aflevere nogle data til den funktion der kalder den, så kan det gøres som i programmet funktion3.c:

```c
/* Filnavn = funktion3.c */

#include <stdio.h>

int adder(int operand1, int operand2)
{
    return operand1 + operand2;
}

int main()
{
    int a = 2;
    int b = 2;
    int c = 0;

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

```
$ cc funktion3.c
$ ./a.out

2+2=4
$
```

Figur 2.8.3

Funktionen adder, udfører en addetion af de to argumenter operand1 og operand2. De to argumenter er variabler der er lokale for funktionen adder, og kan altså på ingen måde ses uden for funktionen. (Funktionen skal jo virke som en black box, altså må vi ikke udefra kunne røre ved det der sker inde i den.)

Funktionen adder returnerer data af typen int, det sker i linien:

```c
return operand1 + operand2;
```

Derfor kan vi nede i main skrive: c = adder(a,b); Det betyder at variablen c derefter indeholder returværdien af funktionen adder. Vi kan helt undvære variablen c, ved at lade adder returnere sin værdi direkte inde i printf. Dette er vist i programmet funktion4.c:

```c
/* Filnavn = funktion4.c */

#include <stdio.h>

int adder(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 2;
    int b = 2;

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

```
$ cc funktion4.c
$ ./a.out
2+2=4
$
```

Figur 2.8.4

Vi har hele tiden indsat linien return 0; sidst i main funktionen. Det kan virke lidt mærkeligt, da det ikke ser ud som om der er noget sted for main at aflevere sin returværdi. Men det er der, main afleverer sin returværdi til operativsystemet, og her er det kutyme, at når et program afleverer værdien nul, så er programmet eksekveret uden fejl, hvis det afleverer en værdi forskellig fra nul, så er returværdien en fejlkode. Hvis du vil vide mere om hvordan dit operativsystem håndterer returværdien fra et program, skal du lede efter en "programers guide" til dit operativsystem.

Opgave 2.8.1 Skriv en funktion der returnerer differencen af to heltal.

Opgave 2.8.2 Skriv en funktion der returnerer produktet af to tal.

Opgave 2.8.3 Skriv en funktion der returnerer divisionen af to float.

2.9 Øvelse gør mester

Nu hvor vi har været rundt om de mest basale begreber er det på tide at stoppe lidt op, og synke det hele. Programmering er et håndværk, og som andre håndværk lærer man det ved at udføre det. Derfor bør du nu bruge det vi har gennemgået til at skrive nogle småpropgrammer inden du går videre.

Lad os skrive et program der kan finde ud af hvilke år der er skudår. For at teste programmet udvidder vi det, så programmet tester for en række år.

Først skal vi lige repetere regelen for et skudår. Hvis et år er deleligt med 4, men ikke med 100, er det et skudår, dog er det også et skudår selv om det er deleligt med 100, blot det er deleligt med 400.

Vi kan altså starte med at teste om året er deleligt med 400, hvis det er tilfældet, er det helt sikkert et skudår, hvis ikke må vi teste noget mere. Det er vist i programmet skudaar1.c

```c
/* Filnavn = skudaar1.c */

#include <stdio.h>

int skudaar(int aar)
{
    if ( (aar % 400) == 0)
        return 1;
    else if ( (aar % 4) == 0)
        if ( (aar % 100) != 0)
            return 1;
    return 0;
}

int main()
{
    int i;

    for (i = 1999; i < 2009; i++)
        if (skudaar(i))
            printf("%d%s\n", i , " er et skudår ");
        else
            printf("%d%s\n", i , " er ikke et skudår ");
    return 0;
}
```

```
$ cc skudaar1.c
$ ./a.out
1999 er ikke et skudår
2000 er et skudår
2001 er ikke et skudår
2002 er ikke et skudår
2003 er ikke et skudår
2004 er et skudår
2005 er ikke et skudår
2006 er ikke et skudår
2007 er ikke et skudår
2008 er et skudår
$
```

Figur 2.9.1

I funktionen skudaar er der flere return. Når programmet under eksekveringen kommer til et return, så stopper eksekveringen i den funktion, og returværdien afleveres. Ved at benytte flere return ”slipper” man nemt ud af funktionen, uden at skulle konstruere en struktrur der gør det for en.
Man kunne selvfølgelig også konstatere, at betingelsen for et skudaar kan skives som:

Hvis 400 går op i årstallet, eller hvis (4 går op, men 100 ikke gør det), så er det et skudår.

Dette er benyttet i programmet skudaar2.c

```
/* Filnavn = skudaar2.c */

#include <stdio.h>

int skudaar(int aar)
{
    if ( (aar % 400) == 0 || (aar % 4) == 0 &&
         (aar % 100) != 0)
        return 1;
    else
        return 0;
}

int main()
{
    int i;

    for (i = 1999; i < 2009; i++)
        if (skudaar(i))
            printf("%d%s\n", i , " er et skudår ");
        else
            printf("%d%s\n", i , " er ikke et skudår ");
    return 0;
}
```

```
$ cc skudaar2.c 
$ ./a.out 
1999 er ikke et skudår
2000 er et skudår
2001 er ikke et skudår
2002 er ikke et skudår
2003 er ikke et skudår
2004 er et skudår
2005 er ikke et skudår
2006 er ikke et skudår
2007 er ikke et skudår
2008 er et skudår
$
```

Figur 2.9.2

Bemærk at der kun er sket ændringer i skudaar funktionen, main ikke blevet rørt på nogen måde. Det er en af fordelene ved at bruge funktioner, man placerer logisk sammenhængende funktioner i blokke for sig selv, og giver dem nogen sigende navne, det går programmeringen lettere.

Det giver også muligheder for at øge abstraktionsniveauet, idet vi kan skrive en

pseudokode der hovedsageligt benytter funktionernes navne. Pseudokoden for skudårsprogrammet kan se således ud.

```
For (aar = 1999; aar < 2009; aar++)
     if (skudaar(aar))
          udskriv det er et skudår
     ellers
          udskriv at det ikke er et skudår
```

Vi kan øge abstraktionsniveauet en tand mere, ved at benytte begreberne true og false i stedet for tal. Selv om if kun undersøger om betingelsen = nul, så kan vi jo godt gøre det lidt nemmere for os mennesker.

```
/* Filnavn = skudaar3.c */

#include <stdio.h>

#define TRUE 1
#define FALSE 0

int skudaar(int aar)
{
    if ( (aar % 400) == 0 || (aar % 4) == 0 &&
         (aar % 100) != 0)
        return TRUE;
    else
        return FALSE;
}

int main()
{
    int i;

    for (i = 1999; i < 2009; i++)
        if (skudaar(i))
            printf("%d%s\n", i , " er et skudår ");
        else
            printf("%d%s\n", i , " er ikke et skudår ");
    return 0;
}
```

```
$ cc skudaar3.c
$ ./a.out
1999 er ikke et skudår
2000 er et skudår
2001 er ikke et skudår
2002 er ikke et skudår
2003 er ikke et skudår
2004 er et skudår
2005 er ikke et skudår
2006 er ikke et skudår
2007 er ikke et skudår
2008 er et skudår
C:\
```

Figur 2.9.3

I linierne

```
#define TRUE 1
#define FALSE 0
```

erklæres to konstanter. De kaldes konstanter fordi deres værdi skal være

konstant, og de skrives altid med store bostaver, for at adkille dem fra variabler.

TRUE og FALSE fungerer som globale konstanter, i det de kan ses derfra hvor de er erklæret, og til slutningen af den fil de er erklæret i.

De følgende opgaver relaterer til det gennemgåede, men du bør ikke lade dig begrænse af dem, prøv selv at finde på noget, du har på nuværende tidspunkt lært nok til at du faktisk kan skrive ganske store programmer.

Opgave 2.9.1 Skriv et program der indeholder en funktion til at teste om et tal er lige. Test funktionen ved et teste alle tal fra 0 til 10.

Opgave 2.9.2 Skriv et program der indeholder funktionen `int menu()`. Funktionen skal udskrive en menu på skærmen, og vente på en indtastning. Når brugeren har indtastet sit valg, skal funktionen returnere valget.

Opgave 2.9.3 Skriv et program der indeholder funktionen `int hvorLangt(int hastighed, int tid)`. Funktionen skal returnere hvor langt man er kommet, efter `tid` kørsel med `hastighed` km/t.

Opgave 2.9.4 Skriv et program der udskriver hvor langt man er kommet efter 14 minutters kørsel med 60 km/t, 32 minutters kørsel med 80 km/t, og til sidst 1½ times kørsel med 47 km/t.

Tip: Brug funktionen fra opgave 2.9.3.

Opgave 2.9.5 Skriv et program der omregner antal dage, til antal år og dage. Se bort fra skudår, og regn med der er 365 dage på et år.

Opgave 2.9.6 Skriv et program der kan udregne hvor mange dage der er mellem to årstal. Programmet skal regne i hele år, og tage højde for skudår.

Tip: Genbrug skudårfunktionen fra skudaar3.c.

3 Array, pointere og filer

I mange situationer er der behov for at kunne gennemløbe store mængder data på en nem måde. Hvis man eksempelvis ønsker at hæve priserne på alt det man sælger med 10%, så er det en fordel hvis det kan gøres, uden at skulle tilgå en lang stribe af variabler, med navne man måske ikke engang kender. Her kommer begreberne array og pointere (pegepinde) ind i billedet.

3.1 Array

Et array er en samling variabler af en bestemt type. Alle variablerne har det samme navn, det eneste der adskiller dem er deres nr. (kaldet indeks) Det kan sammenlignes med en boligadresse, hvis der bor 129 familier på en villavej, der hedder bitvej, så kan man jo påstå at Hansen bor på bitvej nr 1, Petersen bor bitvej nr 2 o.s.v. I stedet for at skrive bitvej nr 1, vil man i C skrive:

```
bitvej[1]
```

Dermed bliver bitvej[2] der hvor Petersen bor o.s.v. Tallet to (2) er det index der peger på det 2'de element i arrayet. Vi kunne også have kaldt vores variabler Hansen, Petersen o.s.v., men det er ikke praktisk i alle sammenhænge.

Hvis der bor 129 på bitvej, så skal vi have et array der er dimensioneret således, at der kan være 129 husnumre i det. I C vil numrende gå fra 0 (nul) til og med 128, hvis vi dimensionerer arrayet til 129. Det skyldes at et array altid starter med det nulte element. For at gøre programmerne nemmere at læse, kan det være en god ide at dimensionere sit array et element større end nødvendigt, så kan man nemlig starte med det første element, (der rent teknisk er det andet element) men lad os se på nogle eksempler.

Forestil dig du har et lager med 3 forskellige dipedutter. Hvis du benytter et array som den datastruktur du gemmer antallet af hver dipedut der er på lager, så kan et lille program som array1.c være en beskeden begyndelse til din lagerhåndtering.

```c
/* Filnavn = array1.c */

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

int main()
{
    int a;
    int i;
    int dippedut[3];

    dippedut[0] = 17;
    dippedut[1] = 3;
    dippedut[2] = 12;

    for (i = 0; i < 3; i++)
        printf("%d\n", dippedut[i]);
    return 0;
}
```

```
$ cc array1.c
$ ./a.out
17
3
12
$
```

Figur 3.1.1

Programmet erklærer et array dimensioneret til at indeholde 3 elementer i linien:

```c
int dippedut[3];
```

Det betyder, at dippedut indeholder 3 heltal, det første er det nul'te element, (dippedut[0]) det sidste er dippedut[2]. Det kan altså snyde lidt fordi et array altid starter med nul. I linierne:

```c
dippedut[0] = 17;
```

```
dippedut[1] = 3;
dippedut[2] = 12;
```

Læser vi antallet af dippedut nr 1 ind i dippedut[0] o.s.v. Herefter kommer det smarte, hele array'et kan udskrives i en enkel løkke, på kun to linier. Det er måske ikke så smart endnu, men meget snart har du et lager på 2763 dippedutter, så er det smart du kan skrive en liste ud, på to programlinier.

Opgave 3.1.1 Tilpas programmet array1.c således, at der udskrives startende med det sidste element i arrayet. (Altså baglæns)

Opgave 3.1.2 Udvid programmet array1.c således at der kan være 5 duppeditter.

Der må ikke forsøges indlæses i mere end de dimensionerede elementer, da der så kan ske en overskrivning af andre programdele. (Mere om det det senere)

For at undgå fejltilskrivninger i array, og for at mindske forvirringen, bør man undgå at bruge magiske tal når man programmerer. Magiske tal er tal der dukker op inde midt i koden, som i linien:

```
for (i = 0; i < 3; i++)
```

Den kvikke iagtager kan hurtigt se der er en sammenhæng med linien:

```
int dippedut[3];
```

Hvis vi ændrer tretallet et af stederne har vi et problem. Hvis vi ændrer det i for løkken, vil vi enten ikke få udskrevet alle elementer, eller vi vil skrive noget ubestemt ud af arrayet. Prøv at ændre i < 3 til i < 4, og se hvad der sker. (Det er ikke sikkert der sker noget. Du vil ikke brænde noget af, men specielt hvis du bruger et Microsoft baseret operativsystem, er det tænkeligt at maskinen skal genstartes. Du bør lave denne lille øvelse fordi, før eller siden kommer du til at lave fejlen ubevidst, og så er det rart at kunne genkende symtomerne.)

Hvis vi ændrer tretallet i erklæringen af arrayet har vi også et problem. Dette er årsagen

til at dette begreb kaldes magiske tal, hvis man ændrer et sted, sker der noget et helt andet (eller mange andre) sted(er), det virker magisk, men bunder i almindeligt rod.

For at fjerne det magiske tal har vi indført en define i programmet array2.c.

```c
/* Filnavn = array2.c */

#include <stdio.h>

#define MAX 3

int main()
{
    int a;
    int i;
    int dippedut[MAX];

    dippedut[0] = 17;
    dippedut[1] = 3;
    dippedut[2] = 12;

    for (i = 0; i < MAX; i++)
        printf("%d\n", dippedut[i]);
    return 0;
}
```

```
$ cc array2.c
$ ./a.out
17
3
12
$
```

Figur 3.1.2

Vi har lavet det der kaldes en makro hvilket betyder, at når compileren møder MAX inde i kildeteksten, sættes det ind der står til højre får MAX i define linien, før den fortsætter compilering.

I dette tilfælde fungerer MAX som en global konstant, altså en variabel der ikke må ændres, når den først er tilskrevet, og som kan ses af hele programmet. Egentlig kan MAX kun ses fra hvor den er erklæret, og fremad i kildeteksten, derfor placeres #define's

normalt i starten af kildeteksten, lige efter evt. `#include`.

Det er kotyme i C programmering, at skrive konstanter med store bogstaver, for at adskille dem visuelt fra variabler.

En makro kan ikke tilskrives efter den er erklæret, hvorfor du vil få en kompilerfej hvis du skriver noget i retning af:

```
#define MAX 2

int main()
{
     MAX = 3; /* FEJL, kan ikke kompileres */

     return 0;
}
```

Arrays er praktiske hvis man har en større mængde tal der skal behandles på en ens måde. I programmet gsnit.c bergnes gennemsnittet af et talsæt nemt og enkelt ved at benytte array.

```
/* Filnavn = gsnit.c */

#include <stdio.h>

#define MAX 10

int main()
{
    int i;
    float talSaet[MAX];
    float sum = 0;

    talSaet[0] = 1.23;
    talSaet[1] = 8.31;
    talSaet[2] = 5.13;
    talSaet[3] = 3.42;
    talSaet[4] = 7.77;
    talSaet[5] = 4.24;
    talSaet[6] = 2.63;
    talSaet[7] = 9.91;
    talSaet[8] = 0.17;
    talSaet[9] = 6.27;
    for (i = 0; i < MAX; i++)
        sum += talSaet[i];
    printf("%2.4f\n", sum/MAX);
    return 0;
}
```

```
$ cc gsnit.c 
$ ./a.out 
4.9080
$
```

Figur 3.1.2

Bemærk at selve beregningen foretages i for løkken, altså på kun to programmeringslinier. Hvis vi udvidder talsættet til 1.534.267 elementer, er det eneste i programmet der skal ændres define linien. (Talene vil vi i så fald nok hente fra en fil, mere om det senere.)

`Opgave 3.1.3` Tilpas programmet gsnit.c, således at programmet udskriver alle de tal der ligger inden for +/- 10% af gennemsnittet.

3.2 Strenge

Strenge er ikke en selvstændig type i C, det er altid et array af char afsluttet med nultegnet ('\0'). Før man forsøger at tilskrive et array, skal det dimensioneres, det gøres på følgende måde:

```
char s[10];    /* Et array der kan indeholde 10 tegn,
                    kaldes også en streng. */
```

Alternativt kan man tilskrive array'et, samtidig med man erklærer det:

```
char s[] = ”abc”;
```

Ved at benytte denne teknik vil compileren dimensionere array'et, i dette tilfælde vil det få størrelsen 4, de tre bogsaver a, b og c, samt nultegnet der altid skal afslutte en streng.

Teknisk set vil arrayet se således ud:

```
s[0] = 'a'
s[1] = 'b'
s[2] = 'c'
s[3] = '\0'
```

Hvis man undlader at dimensionere arryaet får man ikke altid fejlmeddelelser under kompileringen (det afhænger af compileren, den bør give en warning, men gør det ikke altid) men der kan ske de mest besynderlige ting på de underligste steder i dit program, hvis du ikke gør det.

For at undgå fejlfortolkninger i forbindelse med input, er det en god ide at betragte alle input som tegn, og derefter konvertere det hvis der er behov for det. Programmet chartilint.c har en funktion der konverterer fra char til int.

Selve konverteringen foregår i charTilInt funktionen. Funktionen er ikke smart nok til at

finde ud af om input er lovligt, hvilket betyder, at hvis nogen kalder funktionen med en streng hvori der er et kommatal, så går konverteringen galt. Men med korrekt brug, er det en ganske praktisk funktion der benytter sig af, at 10 er grundtallet i talsystemet.

```
/* Filnavn = chartilint.c */


#include <stdio.h>

int charTilInt(char input[])
{
     int i, summa = 0;

     for (i = 0; input[i] != '\0'; i++)
     {
        summa = summa * 10 + input[i] - '0';
     }
     return summa;
}

int main()
{
     char testData[] = "3257";

     printf("%d\n", charTilInt(testData));
     return 0;
}
```

```
$ cc chartilint.c
$ ./a.out
[illegible]
$
```

Figur 3.2.1

For at konvertere fra en char til et decimaltal udføres funktionen:

```
input[i] - '0'
```

hvor det i'te element er en tegnrepræsentation for et tal. Taltegnet konverteres til et heltal ved at fratrække tegnværdien af tegnet nul ('0'). I stedet for nultegnet kunne vi blot have

fratrukket værdien for nultegnet, men det vil gøre programmet platformafhængigt. Ved at bruge tegnet for nul opnår vi at programmet kan kompileres på alle typer af computere der har en standard C kompiler.

Opgave 3.2.1 Udvid programmet chartilint.c med en funktionen int test(char s[]). Funktionen skal kontrollere om at der ikke er et komma i den streng der skal konverteres.

Opgave 3.2.2 Udvid funktionen int test(char s[]) i programmet chartilint.c, således at den kun godkender rene heltal. (a123 er ulovligt input)

Med kendskab til strenge kan vi nu begynde at hente hele linier med input fra tastaturet til vores program. Dette er gjort i programmet array3.c

```
/* Filnavn = array3.c */

#include <stdio.h>

#define MAX 255

int main()
{
    int i,h;
    int ind;
    char input[MAX];

    i = 0;
    while ( (input[i++] = (char) getchar() )
        != '\n' && (i < MAX) )
        ;
    input[i] = '\0';
    h = 0;
    while(input[h] != '\0')
        printf("%c", input[h++]);
    return 0;
}
```

```
$ cc array3.c
$ ./a.out
abc
abc
$
```

Figur 3.2.1

Der er et par nye ting i programmet. Først og fremmest sker der en del i while løkken. Vi starter med at "kaste" resultatet af getchar(), der er et heltal, over i det i'te element i input, som er af typen char. Derefter kommer der et linieskift, og betingelsen fortsætter. Det viser at vi må lave linieskift i kildeteksten, blot det ikke er midt i et udtryk.

```
while ( (input[i++] = (char) getchar() )
    != '\n' && (i < MAX) )
```

Tidligere stoppede vi input når vi mødte EOF, nu bruger vi i stedet '\n', som er tegnkonstanten for en ny linie. Det betyder vi henter fra standard input, til det bliver trykket på enter tasten. Hvis nu brugeren skulle få den ide, at indtaste mere end 255 tegn, så vil der komme overløb i array'et, og det kan under ingen omstændigheder accepteres, derfor er der indsat stopklodsen && (i < MAX).

Der er en ny ting mere i programmet, nemlig linien:

```
input[i] = '\0';
```

Det er nødvendigt før vi kan betragte arrayet som en streng.

Det betyder også, at i stedet for at skrive:

```
h = 0;
while(input[h] != '\0')
    printf("%c", input[h++]);
```

kan vi i stedet skrive:

```
    printf("%s", input);
```

I programmet array3.c kan main funktionen begynde at virke lidt rodet. Da det samtidig er praktisk at have en selvstændig funktion, der ikke gør andet end at hente en linie, har vi skrevet programmet hent_linie.c.

```
/* Filnavn = hent_linie.c */

#include <stdio.h>

#define MAX 255

void hentLinie(char linie[])
{
	int i = 0;
	char ind;

	while (i < MAX && ( ind = getchar() ) != '\n')
		linie[i++] = ind;
	linie[i] = '\0';
}

int main()
{
	char input[MAX];

	hentLinie(input);
	printf("%s%s\n", "Du indtastede : ", input);
	return 0;
}
```

```
$ cc hent_linie.c ↵
$ ./a.out ↵
abc↵
Du indtastede : abc
$
```

Figur 3.2.3

Programmet indeholder ikke de store overraskelser, vi har bare forfinet programmet array3.c. Ind imellem er det en god ide man kigger lidt på sine programmer, og ser om ikke der er noget der er praktisk at tage ud af en helhed, og placere i en anden helhed. Efterhånden som du får erfaring, kan du se hvor det kan være praktisk.

Programmet hent_linie.c er blevet udvidet med funktionen compare i programmet compare1.c

```c
/* Filnavn = compare1.c */

#include <stdio.h>
#define MAX 255

void hentLinie(char linie[]) {
    int i = 0;
    char ind;

    while (i < MAX && ( ind = getchar() ) != '\n')
       linie[i++] = ind;
    linie[i] = '\0';
}

int compare(char streng1[], char streng2[]) {
    int i = 0;

    while (streng1[i++] != '\0')
       if (streng1[i] != streng2[i])
          return streng1[i] - streng2[i];
    return 0;
}

int main() {
    char input[MAX];
    char testStreng[] = "abc";

    hentLinie(input);
    if (compare(input, testStreng))
       printf("%s\n", "De to strenge er ikke ens.");
    else
       printf("%s\n", "De to strenge er ens.");
    return 0;
}
```

```
$ cc compare1.c 
$ ./a.out
abc
De to strenge er ens.
$
```

Figur 3.2.4

Programmet benytter hentLinie funktionen fra før, helt uden ændringer, der er jo ingen

grund til at genopfinde den dybe tallerken. Main er lavet lidt om for at kunne teste compare funktionen. Selve comparefunktionen er værd at bruge lidt tid på.

Compare modtager to strenge som argumenter, og afleverer et heltal. Hvis returværdien er nul, betyder det at strengene er ens, hvis der er forskel på de to strenge, returneres forskellen i tegnværdi det sted i strengen hvor de er forskellige. Det sidste er praktisk hvis vi en dag vil til at lave en sortering.

Kernen i compare funktionen er while løkken.

```
while (streng1[i++] != '\0')
  if (streng1[i] != streng2[i])
    return streng1[i] - streng2[i];
```

Løkken stepper gennem streng1 indtil der står et nultegn på den plads i peger på. Hvis streng2 er kortere end streng1 kan det se ud som om vi læser ud over enden af streng2, men det er ikke tilfældet. Det skyldes at streng2 vil være afsluttet med '\0', hvilket vil medføre at betingelsen:

```
  if (streng1[i] != streng2[i])
```

Er sand, hvorefter funktionen afsluttes og returnerer forskellen i tegnværdien på det sted hvor strengene første gang er forskellige. I main tester if direkte på returværdien, det kan lade sig gøre fordi nul per defenition er usand, og alt andet end nul er sand.

Funktionen compare findes i C's bibliotek string.h, med samme argumentliste og returværdi som beskrevet, dog med navnet strcmp. Prøv at sammenligne med din kompilers dokumentation.

Ofte har man behov for en funktion der kan kopiere fra en streng til en anden. Det er vist i programmet kopier.c. Funktionen kopierer en char ad gangen, og stopper når indekset peger på nultegnet. Der er valgt en do ... whileløkke, fordi der altid skal kopieres mindst

et tegn. Hvis strengen er tom, vil det første tegn være nultegnet, der skal kopieres over i til.
Funktionen er så ofte benyttet, at den findes i C's standardbibliotek string.h med navnet strcpy.

```
/* Filnavn = kopier.c */

#include <stdio.h>

void kopier(char fra[], char til[])
{
    int i = 0;

    do
    {
       til[i] = fra[i];
    }
    while (fra[i++] != '\0');
}

int main()
{
    char testStreng[] = "Halløj!";
    char modtager[10];

    kopier(testStreng, modtager);
    printf("%s\n", modtager);
    return 0;
}
```

```
$ cc compare1.c
$ ./a.out
Halløj!
$
```

Figur 3.2.5

Bemærk den lille specialitet i linien:

```
while (fra[i++] != '\0');
```

hvor variablen i tilskrives, efter den har været brugt som pegepind ind i arrayet fra.

Inden du går videre med næste afsnit vil jeg varmt anbefale, at du gør en indsats for at løse de følgende opgaver, de er udformet for at give dig indblik i array, heltal, float og char.

Opgave 3.2.3 Tilpas programmet array3.c således, at output udføres af en programlinie. Altså ikke tegn for tegn, men som en streng.

Opgave 3.2.4 Prøv at fjerne nultegnet fra strengen inden du udskriver i opgave 3.2.3, hvad sker der? (Husk at gemme alt først, dit system kan gå ned)

Opgave 3.2.5 Tilpas programmet array3.c således, at output bliver udskrevet indkodet efter Cæsarkoden. Hint: Se opgave 2.3.2.

Opgave 3.2.6 Skriv et program der beder brugeren om at indtaste to operander, adskilt af et mellemrum. Disse skal konvergeres til int, og lægges sammen i en summationsfunktion. Resultatet skal udskrives på skærmen.

Opgave 3.2.7 Skriv et program der beder om to operander og derefter en operator for en af de fire regnearter (+. -, *, /). Resultatet af udregningen skal skrives på skærmen. Hint: Start med at lave en funktion for hver af de fire regnearter, derefter kan du lade en switch tjekke på operatoren, og vælge en af funktionerne.

Opgave 3.2.8 Skriv en funktion der konverterer fra en streng til et kommatal.

Opgave 3.2.9 Skriv en funktion der konverterer fra heltal til char array.

Opgave 3.2.10 Skriv et program der skal indeholde funktionen cat(char s1[], char s2[]). Funktionen skal addere strengen s2 til afslutningen af strengen s1.

Opgave 3.2.11 Skriv et program der skal indeholde funktionen int laengde(char s). Funktionen skal bestemme antallet af tegn i strengen s, og returnere dette antal.

3.3 Pointere

I C kan man sige at pointere er en udvidelse af array's, men det er nok mere rigtigt at sige, at array er et specialtilfælde af pointere, derfor er det naturligt på dette sted at glide over i begrebet pointere.

En pointervariabel er en variabel der indeholder en hukommelsesadresse, og intet andet. Når et program benytter den adresse der står inde i en pointervariabel til at pege på nogle data, så kaldes det en pointer. En pointer kan kun peger på data af same type som den selv, med mindre den er erklæret som en voidpointer.

Hvis du tænker på din computers hukommelse som et stort array, så er pointeren det indeks der peger ind i array'et. I programmet pointer1.c benyttes denne viden, idet pointeren `*pegepind`, erklæres til at pege på data af typen char.

```
/* Filnavn = pointer1.c */

#include <stdio.h>


int main()
{
    char alfabet[] = "abcdefg";
    char *pegepind;

    pegepind = &alfabet[0];

    while (*pegepind != '\0')
    {
       printf("%s\n", pegepind);
       pegepind++;
    }
    return 0;
}
```

```
$ cc pointer1.c
$ ./a.out
abcdefg
bcdefg
cdefg
defg
efg
fg
g
$
```

Figur 3.3.1

I programmet vil du bemærke, at vi bruger pegepind på to måder, med og uden en stjerne foran. Når der er en stjerne foran kaldes det pointeren, når der ikke er en stjerne foran er det pointervariablen, altså den variabel der indeholder en adresse i hukommelsen. I linien:

```
pegepind = &alfabet[0];
```

tilskrives pointervariablen pegepind med adressen på det nulte tegn i alfabet. Det sker fordi og (&) tegnet benyttet umiddelbart foran variablen, & afleverer variablens absolutte hukommelsesadresse. På grund af de nære bånd melem array og pointere, kunne vi også have skrevet:

```
pegepind = alfabet;
```

Det skyldes, at adressen på et array's nulte element er arraynavnet uden klammer.

Efter at have kopieret arrayets startadresse over i pointervariablen, går programmet ind i en whileløkke der varer ved indtil pointeren (ikke pointervariablen) peger på nulstrengen, der jo som bekendt skal være i enden af en streng. Inde i løkken udskriver vi den streng som *pegepind peger på, og tæller pointervariablen pegepind en op.

Resultatet er, at der linie for linie udskrives et tegn mindre af strengen, i øvrigt uden at programmet på nogen måde manipulerer med indholdet af strengen.

Hvis du gerne vil se lidt mere detaljeret hvad der sker, kan du ændre programmet, så det bliver som i pointer1a.c, der udskrives den adresse som pegepind indeholder hele tiden.

```
/* Filnavn = pointer1.c */

#include <stdio.h>


int main()
{
    char alfabet[] = "abcdefg";
    char *pegepind;

    pegepind = alfabet;

    while (*pegepind != '\0')
    {
       printf("%s\t%u\n", pegepind, pegepind);
       pegepind++;
    }
    return 0;
}
```

```
$ cc pointer1a.c
$ ./a.out
abcdefg     xxxxxx0
bcdefg      xxxxxx1
cdefg       xxxxxx2
defg        xxxxxx3
efg         xxxxxx4
fg          xxxxxx5
g           xxxxxx6
$
```

Figur 3.3.2

Slægtskabet mellem pointere og array er så nært i C, at vi ikke havde behøves en pointer for at udføre funktionen. Vi kunne i stedet have ændret whileløkken til:

```
while (alfabet++ != '\0')
  printf("%s\n", alfabet);
```

Men det ville være destruktivt, da vi nu ikke kan få fat i starten af variablen, vi jo har flyttet den til enden.

Opgave 3.3.1 Omskriv programmet pointer1.c således at pegepind ikke benyttes, i stedet skal alfabet behandles som en pointer, og løbende tælles en op inde i løkken.

Funktioner kan tage enhver datatype som argument, og den kan returnere enhver, men kun en datatype. I programmet funktion5.c returneres ingen data, hvorfor returtypen er void. Alligevel kan vi manipulere med data fra den kaldte funktion, det skyldes vi benytter pointere.

Funktionen smaaTilStore konverterer pudsigt nok alt indholdet i en streng fra små til store bogstaver. I main kalder vi funktionen på følgende måde:

```
smaaTilStore(minStreng);
```

Og det har vi jo lige lært betyder, at vi sender adressen på minStreng til funktionen. Vi kunne i stedet have skrevet:

```
smaaTilStore(&minStreng[0]);
```

men det er ikke kutyme.

Det er ikke minStreng der kopieres til funktionen, det er kun dens startadresse i hukommelsen der overføres til funktionen.

<pre>/* Filnavn = funktion5.c */
#include <stdio.h>
 	
void smaaTilStore(char input[])
{
    int i = 0;
 	
    while (input[i] != '\0')
    {
        if (input[i] >= 'a' && input[i] <= 'z')
            input[i] = input[i] - ('a' - 'A');
        i++;
    }
}
 	
int main()
{
    char minStreng[] = "Konverter dette.";
 	
    smaaTilStore(minStreng);
    printf("%s\n", minStreng);
    return 0;
}</pre>
	$ cc funktion5.c
	$ ./a.out
	KONVERTER DETTE.
	$

Figur 3.3.3

I selve funktionen modtages startadressen, og bliver den adresse hvor input får startadresse, vi har altså to variabler, (minStreng og input) der begge peger på den samme startadresse på en streng. Når funktionen herefter manipulerer med indholdet af input, er det det samme som at manipulere med det `minStreng` peger på, fordi de to pointere, der egentligt er erklæret som array's, peger på det samme.

Funktionaliteten med at konvertere fra små til store bogstaver ligger i argumentet:

```
('a' - 'A')
```

Der udregner forskellen i afstand mellem bogstavet store A og bogstavet lille a, derefter trækkes denne forskel fra den værdi tegnet har nu, hvorefter alle tegn vil være konverteret til store bogstaver.

While løkken stopper når variablen i peger på nultegnet ('\0'), der jo betyder slut på linien. Årsagen til while stopper når den når nultegnet er, at nultegnet har talværdien nul, og nul i en betingelse er jo som bekendt det samme som en forkastelse.

Ved at bruge pointere kan man optimere kopieringsfunktionen fra kopier.c programmet således:

```
void kopier(char *fra, char *til)
{
	do
	{
		*til = *fra;
		fra++; til++;
	}
	while (*fra != '\0');
	*til = *fra;
}
```

De to pointere *fra og *til modtager en adresse hver fra den kaldende funktion. Inde i løkken sættes den adresse som *til peger på, til at indeholde det samme som det *fra peger på. Dette fortsætter til *fra peger på nultegnet, hvorved løkken forlades, men vi er ikke færdige, fo vi er jo gået ud af løkken før vi fik overført nultegnet til *til.

Slægtskabet mellem array og pointere er faktisk så tæt, at man kan ændre kopieringsfunktionen til:

```
void kopier(char fra[], char til[])
{
     do
     {
          *til = *fra;
          fra++; til++;
     }
     while (*fra != '\0');
     *til = *fra;
}
```

Den skrappe programmør vil dog nok skrive den således:

```
void kopier(char *fra, char *til)
{
     while ((*til++ = *fra++ )!= '\0')
          ;
}
```

Her benyttes kun pointere, hvilket nok kan være lidt forvirrende for den utrænede. Men det hele går faktisk bare ud på, først at tilskrive det pointeren *til peger på, med det som pointeren *fra peger på. Hvis det er nultegnet stopper løkken, hvis ikke lægges der en til pointervariablerne fra og til med ++ operatoren.

Her bør du igen stoppe lidt op og øve dig i det gennemgåede, ved at løse følgende opgaver.

Opgave 3.3.2 Omskriv programmet compare.c således at det udelukkende benytter pointer i stedet for arrays.

Opgave 3.3.3 Omskriv programmet fra opgave 3.2.10 således at det udelukkende benytter pointer i stedet for arrays.

Opgave 3.3.4 Omskriv programmet fra opgave 3.2.11 således at det udelukkende benytter pointer i stedet for arrays.

Opgave 3.3.5 Omskriv programmet chartilint.c således at funktionen charTilInt udelukkende benytter pointer i stedet for arrays.

Opgave 3.3.6 Omskriv programmet smaatilstore.c således at funktionen smaaTilStore udelukkende benytter pointer i stedet for arrays.

3.4 Enumeration

Enumaration benyttes i forbindelse med at initiere en samling af data. En enumeration er en liste af konstante heltal. Det kan bruges til at skaffe overblik over størrelser som ellers kan virke uhåndterlige, prøv at se programmet enum1.c

Programmet starter med at skabe et enum kaldet ugedag, der indeholder samtlige ugedage i rækkefølge. Det medfører at MANDAG bliver en konstant der har værdien nul, TIRSDAG en konstant med værdien 1 o.s.v. Når en konstant er oprettet på denne måde, er det ikke legalt at ændre den. Hvis man senere i koden skriver:

```
MANDAG = 2;
```

vil man få en fejlmeddelelse fra compileren.

```
/* Filnavn = enum1.c */
#include <stdio.h>
enum ugedag {MANDAG, TIRSDAG, ONSDAG, TORSDAG,
             FREDAG, LORDAG, SONDAG};

void test(int dagnr)
{
    if (dagnr == ONSDAG)
       printf("%s\n", "Det er Onsdag");
}

int main()
{
    int i;

    for (i = MANDAG; i < LORDAG; i++)
    {
       printf("%d\n", i);
       test(i);
    }
}
```

```
$ cc enum1.c
$ ./a.out
0
1
2
Det er Onsdag
3
4
$
```

Figur 3.3.1

Ved at definere ugedag forrest i vores kildetekst, opnår vi at den kan ses af alle funktioner i denne fil. Hvis vi flyttede enum erklæringen ned efter test funktionen, før main, så vil compileren melde fejl når den kommer til linien:

```
if (dagnr == ONSDAG)
```

Det skyldes at ONSDAG ikke er synlig for compileren på det tidspunkt hvor den linie bliver forsøgt kompileret. Variabler kan erklæres på samme måde, så de blive synlige fra mere end en funktion.

Det første element i en enum vil altid få værdien 0, med mindre man skriver:

```
enum aendring {ET = 1, TO, TRE };
```

derefter vil nummereringen være fortløbende. Det kan dog også ændres ved at skrive:

```
enum aendring {ET = 1, TO, TI = 10, ELLEVE };
```

Opgave 3.4.1 Skriv et program der tæller fire gange fra 1 til og med tolv. Hver gang der er talt til otte, skal der udskrives "Dette er august måned" på skærmen. Hver gang der er talt til tolv skal der udskrives "Nu er det jul" på skærmen.

3.5 Arrays med flere dimensioner.

Vi mennesker er vandt til at se alt i to eller tre dimensioner. Et array kan også have flere dimensioner, men i denne bog vil vi stoppe ved de to dimensionelle arrays, som er kendt fra almindelige regneark. (der er søjler og rækker)

Det kan vise sig praktisk at have en todimesnionel array der indeholder antallet af dage, per måned, i skudår og ikke skudår. En sådan array kan se således ud:

```
int dageiMaaned[2][13] =
{
     {0, 31, 28 ,31 ,30 ,31 ,30 ,31 ,31 ,30 ,31 ,30 ,31}
     {0, 31, 29 ,31 ,30 ,31 ,30 ,31 ,31 ,30 ,31 ,30 ,31}
};
```

Ved hjælp af arrayet kan vi bestemme hvor mange dage der er gået siden nytår, dette kaldes også den Julianske dato. Hvis datoen er den 3/8, så er der gået:

```
int i;
int sum = 0;

for (i = 1; i <= 8; i++)
     sum += dageiMaaned{0][i];
sum += 3;
```

Under forudsætning at der ikke er tale om et skudår. Vi vil bruge denne teknik til et programet dato1.c, der kan beregne afstanden mellem to datoer.

Skudårsfunktionen kender vi fra tidligere, så den genbruger vi bare. Det smarte ligger i den måde vi gør brug af den to dimensionelle array. Vi benytter returværdien fra skudårsfunktion til at udpege den dimension i arrayet vi vil have fat i.

```
/* Filnavn = dato1.c */
#include <stdio.h>

int dageiMaaned[2][13] =
{
    {0, 31, 28 ,31 ,30 ,31 ,30 ,31 ,31 ,30 ,31 ,30 ,31},
    {0, 31, 29 ,31 ,30 ,31 ,30 ,31 ,31 ,30 ,31 ,30 ,31}
};

int skudaar(int aar)
{
    if ( (aar % 400) == 0 || (aar % 4) == 0 &&
         (aar % 100) != 0)
       return 1;
    else
       return 0;
}

int jullianskDato(int aar, int mdr, int dag)
{
    int i;

    for (i = 1; i <mdr; i++)
       dag += dageiMaaned[skudaar(aar)][i];
    return dag;
}

int main()
{
    int a = jullianskDato(2000, 7, 3);
    int b = jullianskDato(2000, 3, 1);

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

```
$ cc dato1.c
$ ./a.out
124
$
```

Figur 3.5.1

Todimensionelle arrays er en nødvendighed hvis man ønsker at have en array af strenge, da

strenge i sig selv er et array. I programmet maaned1.c har vi lavet en funktion der kan hente en måneds navn ud af et array bestående af pointere

```
/* Filnavn = maaned1.c */

#include <stdio.h>

char *maaned[] =
{
    "***",
    "Jan", "Feb", "Mar", "Apr", "Maj", "Jun",
    "Jul", "Aug", "Sep", "Okt", "Nov", "Dec"
};

int main()
{
    printf("%s\n", maaned[3]);
    return 0;
}
```

```
$ cc maaned1.c
$ ./a.out
Mar
$
```

Figur 3.5.2

Arrayet ved navn `maaned` indeholder 13 pointere der peger på data af typen char. Hver af disse pointere peger på hver sin måned. Derfor kan vi udskrive en streng ved at skrive:

```
printf("%s\n", maaned[3]);
```

maaned[3] indeholder jo netop en pointer der peger på strengen Mar.

En anden måde at bruge en pointerarray er, hvis man ønsker at hente data ind fra kommandolinien. I det fleste styresystemer er det første argument et program modtager navnet på sig selv, derefter følger de evt. indtastede argumenter.

For at kunne modtage kommandolinie argumenter, skal main have to parametre, den første skal være af typen int, og vil komme til at indeholde antallet af argumenter der

bliver overført. Det andet argument skal være et pointerarray, hvor hver pointer kommer til at pege på de enkelte argumenter der er overført, startende med det første.

```c
/* Filnavn = kmdlin1.c */

#include <stdio.h>

int main(int a, char *s[])
{
    int i = 0;

    do
    {
       printf("%d\t%s\n", i, s[i]);
       i++;
    }
    while (i < a);
    return 0;
}
```

```
$ cc kmdlin1.c -o kmdlin1.c
$ kmdlin1 Hans Peter Søren
0    kmdlin1
1    Hans
2    Peter
3    Søren
$
```

Figur 3.5.3

Opgave 3.5.1 Tilpas programmet dato1.c således, at det også kan bestemme afstanden med datoer i forskellige år.

Opgave 3.5.2 Tilpas programmet dato1.c således, at det kan håndtere datoer i tilfældig rækkefølge. D.v.s programmet skal sørge for at kalde funktionen jullianskDato i den rigtige rækkefølge.

Opgave 3.5.3 Tilpas programmet kmdlin1.c således, at det som første argumet modtager et heltal. Det tal skal konverteres til int og udskrives.

Opgave 3.5.4 Skriv et program der kan modtage en formel baseret på en af de fire regnearter, fra kommandolinien. Programmet skal herefter adskille operatorer og operander, konvertere operanderne til tal som computeren kan regne med, og udføre operationen, og udskrive resultatet. Hint: Det kan lyde besværligt, men vi har gennemgået alle delteknikerne tidligere. Du bør derfor genbruge tidlige brugt kode.

3.6 Filer

Det er ikke rigtigt sjovt at programmere før man har tjek på at gemme og hente data fra en fil, det er heldigvis rigtigt nemt i C. Programmet fil1.c udskriver indholdet af en teksfil på skærmen, i dette tilfælde er det kildeteksten selv der bliver udskrevet, den er jo altid ved hånden.

```
/* Filnavn = fil1.c */

#include <stdio.h>

int main(int a, char *s[])
{
    char ch;
    char fil[] = "fil1.c";
    FILE *fp;

    if ((fp = fopen(fil, "r")) != NULL)
    {
       while ((ch = getc(fp)) != EOF)
           printf("%c", ch);
       close (fil);
    }

    return 0;
}
```

```
$ cc fil1.c
$ ./a.out
/* Filnavn = fil1.c */

#include <stdio.h>

int main(int a, char *s[])
{
    char ch;
    char fil[] = "fil1.c";
    FILE *fp;

    if ((fp = fopen(fil, "r")) != NULL)
    {
       while ((ch = getc(fp)) != EOF)
           printf("%c", ch);
       close (fil);
    }

    return 0;
}
$
```

Figur 3.6.1

Det nye i dette program er vi har indført en pointer af typen FILE, det er denne pointer der peger direkte ud i filen. Funktionen `fopen` tager to argumenter, det første er navnet på den fil vi vil skrive til, det andet er en information om hvad vi vil gøre ved filen. I

dette tilfælde skriver vi blot ”r” fordi vi kun vil læse i filen. (r = read).
Hvis `fopen` returnerer værdien NULL er det fordi der er sket en fejl i forsøget på at åbne filen, det vil typisk skyldes, at du har indtastet et forkert navn, eller stavet filnavnet forkert i linien:

```
char fil[] = "fil1.c";
```

Hvis det lykkes at åbne filen, så læser `getsc(fp)` funktionen det første tegn fra filen, og returner der til `ch.` Inden `getc` returnerer tæller den `fp` en op, således at filpointeren nu peger på det næste element i filen, hele filen kan altså betragtes som en stor array.

Når der ikke er mere i filen vil styresystemet returnere en EOF (End Of File) og løkken forlades. Umiddelbart herefter møder vi linien:

```
close (fil);
```

Her lukkes filen hvilket betyder, at tidligst efter denne linie gemmes dataene rent faktisk på harddisk eller memorychip. Hvis du undlader at lukke filen er der stor risiko for at du mister data. Det er ikke slemt når du kun læser fra en fil, men det går nemt galt hvis du skriver til en fil, og glemmer at lukke den efter dig. En fil er øm over for skader når den er åben, derfor er det en god ide at betragte fopen og close på samme måde som start og slut tuborg, de skal altid følges ad.

```c
/* Filnavn = fil2.c */

#include <stdio.h>

int main(int a, char *s[])
{
    int i = 0;
    char fil[] = "tst.txt";
    FILE *fp;

    if ((fp = fopen(fil, "w")) != NULL)
    {
       while (fil[i])
           putc(fil[i++], fp);
       close (fil);
    }

    return 0;
}
```

```
$ cc fil2.c
$ ./a.out
$
```

Figur 3.6.2

Hvis vi i stedet ønsker at skrive til en fil, så skal fopen kaldes med ”w” som argument. (w er en forkortelse for write)

Du skal være opmærksom på, at når du åbner en fil med argumentet w, så bliver en evt. fil med samme navn slettet uden varsel. Hvis du vil varsles skal du selv programmere varslingen ind i dit program.

Efter at have åbnet filen løbes array'et fil gennem trin for trin, indtil alle elementer i det er gemt i filen. Derefter lukkes filen, og du kan prøve at læse den, enten med programmet fil1.c, eller med en ganske almindelig editor.

Hvis du ønsker at åbne en fil der allerede eksisterer, uden at slette den, skal du åbne den med a parametren.

Opgave 3.6.1 Skriv et program der kan bestemme størrelsen på en fil. Filnavnet skal være et argument ved start af programmet, og resultatet skal udskrives på skærmen.

Opgave 3.6.2 Skriv et program der kopierer en fil til en anden. Programmet skal startes med to argumenter, det første skal være navnet på den fil der skal kopieres fra, det andet skal være navnet på den fil der skal kopieres til.

Opgave 3.6.3 Skriv et program der sammenligner to filer. Hvis filerne er ens skal det skrives på skærmen, hvis de ikke er ens skal der skrives på skærmen fra hvilket tegn filerne divergerer.

Opgaver 3.6.4 Skriv et program der krypterer indholdet af en fil efter Cæsarkoden. De krypterede data skal gemmes i en fil. Begge filnavne skal indtastes under programkørslen.

Opgave 3.6.5 Skriv et program der dekrypterer den fil du krypterede i opgave 3.6.4. Programmet skal bede brugeren om filnavnet under eksekveringen.

Opgave 3.6.6 Skriv et program der analyserer tegnfordelingen i en testfil. Resultatet skal opgives i procentvis fordel for hvert tegn.

3.7 Et eksempel

Inden vi helt forlader disse viderunderlige pointere, vil vi se på et praktisk eksempel. Vi vil skrive det klassiske program monitor. Program benyttes til at fejlfinde på filer, og er er et meget nyttigt værktøj. En monitor fungerer på den måde, at dataene fra en fil udskrives 16 tegn ad gangen, først som hexadecimale tal, derefter et mellemrum, og så de samme 16 data som tegn, i det omfang de kan vises på skærmen som et tegn. Hvis det ikke kan vises som et tegn, placeres et punktum, for at vise her er et ikke visbart tegn. Efter 16 data skiftes der linie, og de næste 16 vises, indtil der ikke er mere i filen.

Kildeteksten til monitor.c er vist på figur 3.7.1. Programmet starter med at definere 16 som konstanten X. Hvis vi ikke gjorde det på den måde i dette program, vil risikoen for ”magiske” effekter være overhængende, da vi bruger 16 som grænse flere steder i programmet. Inde i selve programmet åbner vi det filnavn der blev overført fra kommandolinien, og vi begynder at læse i filen.

Hvis der er læst 16 data fra filen, skal vi til at skrive ud, derfor testes der således:

```
if ( !(i % 16) ) {
```

Da resten ved modolus operationen er nul hvis `i` er 16, skal vi bruge not operatoren, fordi nul betyder falsk i en betingelse.

Kildeterksten kan findes her: http://synkro.dk/bog/c/	$ cc monitor.c -o monitor ↲
	$ monitor monitor.c
	Der er ikke plads til at vise output her.
	$

Figur 3.7.1

Vel inde i udprintningen ser vi linien:

```
printf("%2x ", ch[j]);
```

Det betyder at de j'de element i ch skal udprintes som hexadecimalt tal, med to pladser. Når de 16 elementer er udskrevet, tabuleres der. Derefter skrives de samme 16 elementer som før ud, denne gang dog som tegn. Hvis et element ikke ligger i intervallet ! <= z, så vil der blive skrevet et punktum på skærmen, da kontroltegn kan give de mærkeligste effekter hvis de skrives til skærmen. Forestil dig hvad der ville ske hvis vi udskrev et linieskift som et linieskift, billedet ville blive skubbet, fordi der kom et linieskift midt i en linie.

Når 16 elementer er udskrevet som både hex og tegnværdier, skiftes der linie, og hele historien gentages, indtil filen ikke er længere.

Når filen ikke er længere, er der en chance ud af 16 for at vi har skrevet alle linier ud. For altid at få alle linier skrevet ud, gentages hele udprintningsfunktionen hvis ikke i lige præcis er 16.

Hvis programmet monitor.c køres med sig selv som argument, vil de sidste fire linier i output se således ud:

```
 9 70 72 69 6e 74 66 28 22 25 63 22 2c 20 27 2e   .printf("%c",.'.
27 29 3b  a  9  9  9 70 72 69 6e 74 66 28 22 5c   ');....printf("\
6e 22 29 3b  a  9  9  9 69 20 3d 20 30 3b  a  9   n");....i.=.0;..
 9 7d  a  9 7d  a  9 72 65 74 75 72 6e 20 30 3b   .......return.0;
 a 7d  a  a       ....
```

Prøv monitor af med forskelige filnavne, herunder bør du prøve at kige på den eksekberbare udgave af monitor, mens også andre eksekberbare filer kan du kigge på med monitor. Programmet er første skridt hvis du vil reverse enginere et færdigt program.

3.8 Opgaver

Opgave 3.8.1 Udvid programmet monitor.c, således at der kommer en fejlmelding, hvis det opgivne filnavn ikke findes.

Opgave 3.8.2 Tilpas programmet monitor.c således, at hver gang der er skrevet 20 linier på skærmen standser programmet, og beder brugeren om at taste return for at fortsætte.

Opgave 3.8.3 Tilpas programmet monitor.c således, at det filnavn der skal bruges skal indtastes under programafviklingen.

4 Strukturer og lister

En struktur i C er en samling af en eller flere variabler, der gerne må være af forskellige typer. Variablerne er samlet under et tag med det samme navn. Strukturer kaldes også for brugerdefinerede typer, i denne sammenhæng er det programmøren der er brugeren.

En liste er en samling af strukturer, man kan også kalde en liste for en database.

4.1 Begrebet struktur

Hvis vi skal håndtere nogle personinformationer kan vi gøre det på denne måde:

```
char person1[] = "Jens Petersen 23 år";
```

Det er bare ikke nogen god måde at gøre det på. Det skyldes at nogle gange vil vi søge på et efternavn, andre gange en alder, og helt andre gange er vi måske interesseret i et fornavn. Dette er en typisk situation for en programmør, hvorfor stort set alle programmeringssprog understøtter definerbare datatyper. En person struct kan se således ud:

```
struct personType
{
     int alder;
     char forNavn[20];
     char efterNavn[20];
}hans;
```

Her har vi lavet en `personType`, den kan indeholde data for en person. Der er skabt et tilfælde af typen, det kalder vi `hans`. At skabe et tilfælde betyder at vi opretter en variabel af typen `personType`, og vi kalder den variabel (det tilfælde af `personType`) for `hans`.

Det kunne også være gjort på denne måde:

```
struct personType
{
     int alder;
     char forNavn[20];
     char efterNavn[20];
};

personType hans;
```

I begge tilfælde kan vi derefter tilskrive variablen `alder` inde i `hans` på følgende måde:

```
hans.alder = 23;
```

Man kan initiere en struct når der skabes et tilfælde af den således:

```
struct personType
{
     int alder;
     char forNavn[20];
     char efterNavn[20];
}hans = {17, ”Hans” ”Hansen”};
```

Eller således:

```
struct personType
{
     int alder;
     char forNavn[20];
     char efterNavn[20];
};

personType hans = {17, ”Hans” ”Hansen”};
```

Herefter vil `alder`, `forNavn` og `efterNavn` være tilskrævet med værdierne 17, Hans og Hansen.

For at tilgå en variabel inde i en struct skal man bruge struct's navn, et punktum efterfulgt af navnet på den variabel man ønske at tilgå. Bemærk at struct's navn er `hans`, dens type er `personType`. Det er ofte forvirrende for begynderen, men hvis du konsekvent kalder struct'ens type noget med `Type`, så er det til at adskille typen og tilfældet. Tilfældet kan ofte kaldes et elektronisk arkivkort.

```c
/* Filnavn = struct1.c */

#include <stdio.h>

struct personType
{
    int alder;
    char forNavn[20];
    char efterNavn[20];
};


int main()
{
    struct personType hans;

    hans.alder = 23;
    strcpy(hans.forNavn, "Jens");
    strcpy(hans.efterNavn, "Petersen");
    printf("%s\n", hans.forNavn);
    return 0;
}
```

```
$ struct1.c
Jens
$
```

Figur 4.1.1

Bemærk envidere, at `personType` struct'en, bliver erklæret helt uden for funktioner, og kan derfor ses globalt. Det er med fuldt overlæg, da det jo er en type som alle funktioner skal kunne oprette et tilfælde af.

Det er normalt at erklære struct's i en selvstændig header fil, men mere om det i næste afsnit.

Opgave 4.1.1 Tilpas programmet struct1.c således, at efternavnet udskrives i stedet for

fornavnet.

Opgave 4.1.2 Tilpas programmet struct1.c således, at der udskrives både efternavn og fornavn, i den rækkefølge.

Opgave 4.1.3 Tilpas programmet struct1.c således, at der udskrives både efternavn og fornavn, i den omvendte rækkefølge

Opgave 4.1.4 Skriv et program der kan teste struct'en `bilType`. Typen skal indeholde variabler for bilens navn, dens årgang, og antal hestekræfter.

4.2 Array af struct's

I sidste afsnit brugte vi en struct til at oprette et arkivkort der kunne rumme en person. Det bliver først anvendeligt når vi kan have mange personer på den måde, derfor er det hyppigt brugt at skabe et liste af struct's, dette kan laves med et array. Erklæringen er ganske som vi er vandt til at oprette et array. Følgende linie opretter 100 struct's, der hver kan indeholde en persons data:

```
personType personer[100];
```

Vi har skabt en persondatabase. Programmet struct2.c er et databaseprogram, der demonstrerer brugen af et struct array. For at undgå for meget indtastningsarbejde, har vi genbrugt `hentLinie` funktionen fra kapitel 3.

```c
/* Filnavn = struct2.c */

#include <stdio.h>

struct personType
{
	int alder;
	char forNavn[20];
	char efterNavn[20];
};

int hentLinie(char linie[], int MAX)
{
	int i = 0;
	char ind;

	while (i < MAX-1 && ( ind = getchar() ) != '\n')
		linie[i++] = ind;
	linie[i] = '\0';
	return i;
}

int main()
{
	int i;
	int antal = 3;
	char tmp[4];
	struct personType personer[10];

	for (i = 0; i < antal; i++)
	{
		printf("%s", "Indtast fornavn : ");
		hentLinie(personer[i].forNavn, 20);
		printf("%s", "Indtast efternavn : ");
		hentLinie(personer[i].efterNavn, 20);
		printf("%s", "Indtast alder : ");
		hentLinie(tmp, 4);
		personer[i].alder = atoi(tmp);
	}

	printf("%s\n", "Udskriver");
	for (i = 0; i < antal; i++)
	{
		printf("%s\t", personer[i].forNavn);
		printf("%s\t", personer[i].efterNavn);
		printf("%4d\n", personer[i].alder);
	}
	return 0;
}
```

```
$ struct1.c 
Indtast fornavn : Jens
Indtast efternavn : Jensen
Indtast alder : 23
Indtast fornavn : Peter
Indtast efternavn : Petersen
Indtast alder : 32
Indtast fornavn : Ib
Indtast efternavn : Ibsen
Indtast alder : 89
Udskriver
Jens	Jensen	23
Peter	Petersen 32
Ib	Ibsen	89
$
```

Figur 4.2.1

I linien:

```c
struct personType personer[10];
```

oprettes 10 tilfælde af `personType`.

De 10 tilfælde har navnet `personer[0]` ... `personer[9]`. I indtastningsløkken er vi nød til at håndtere input som char til at starte med, for at kunne bruge `hentLinie` funktionen. Derfor er vi bagefter nød til at konvertere det indtastede til et heltal, hvis vi en dag skulle ønske at udføre beregninger baseret på folks alder, det sker i linien:

```
personer[i].alder = atoi(tmp);
```

Funktionen `atoi()` kommer fra stdio.h biblioteket og fungerer på samme måde som den `cahrTilInt` funktion vi skrev i kapitel 3. Alternativt kunne man beholde alderen på strengform, og konvertere til et int hvis man skal bruge tallet til en beregning.

Ud over det ovenstående bør der ikke være de store overraskelser. Det er værd at lægge mærke til, at vi hurtigt kan sætte større programmer sammen blot ved at genbruge tidligere skrevet kode. I denne sammenhæng er dovenskab faktisk lidt af en dyd.

Opgave 4.2.1 Tilpas programmet struct2.c således, at der indtastes og udskrives 5 personer.

Opgave 4.2.2 Tilpas programmet fra opgave 4.2.1 således, at personnavnene udskrives baglæns.

Opgave 4.2.3 Tilpas programmet struct2.c således, at der oprettes en indtastningsfunktion ved navn indtast, test programmet.

Opgave 4.2.4 Tilpas programmet fra opgave 4.2.3 således, at der oprettes en udskrivningsfunktion ved navn udskriv, test programmet.

4.3 Headerfiler

Efterhånden som vores programmer kan mere og mere, bliver de større og større, og dermed også mere uoverskuelige for programmøren. For at skabe overskuelighed er der skabt headerfiler, som er en eller flere filer som man kan bygge sit program op af. Vi vil demonstrere det ved at rydde lidt op i programmet struct2.c. Det nye program vil vi kalde struct3.c, og det skal have præcis samme funktionalitet som programmet struct2.c. Vi starter med at skrive headerfilen.

```
/* Filnavn = support1.h */


struct personType
{
	int alder;
	char forNavn[20];
	char efterNavn[20];
};

int hentLinie(char linie[], int MAX)
{
	int i = 0;
	char ind;

	while (i < MAX-1 && ( ind = getchar() ) != '\n')
		linie[i++] = ind;
	linie[i] = '\0';
	return i;
}
```

Headerfil har ikke noget output.

Figur 4.3.1

Headerfilen ved navn support1.h gemmes i samme directory som struct3.c filen, som vi kigger på om lidt. Vi giver headerfilen filextendet .h for at vise det er en headerfil, der er ikke noget i vejen for at give den extendet .c i stedet, men lige nu kaldet vi filen support.h. Headerfilen skal ikke kompileres endnu, det vil ske senere, når vi kompilerer det program der bruger support.h.

Hovedprogrammet er placeret i filen struct3.c. Der er et par ændringer i filen i forhold til struct2.c, for det første mangler erklæringen af `personType`, da den er placeret i headerfilen, `hentLinie` er også placeret i headerfilen, herved er der skabt mere overskuelighed i hovedfilen.

<table>
<tr>
<td>

```
/* Filnavn = struct3.c */

#include <stdio.h>
#include "support1.h"

int main()
{
	int i;
	int antal = 3;
	char tmp[4];
	struct personType personer[10];

	for (i = 0; i < antal; i++)
	{
		printf("%s", "Indtast fornavn : ");
		hentLinie(personer[i].forNavn, 20);
		printf("%s", "Indtast efternavn : ");
		hentLinie(personer[i].efterNavn, 20);
		printf("%s", "Indtast alder : ");
		hentLinie(tmp, 4);
		personer[i].alder = atoi(tmp);
	}

	printf("%s\n", "Udskriver");
	for (i = 0; i < antal; i++)
	{
		printf("%s\t", personer[i].forNavn);
		printf("%s\t", personer[i].efterNavn);
		printf("%4d\n", personer[i].alder);
	}
	return 0;
}
```

</td>
<td>Samme input og output som struct2.c</td>
</tr>
</table>

Figur 4.3.2

Det mest markante er linien:

```
#include "support1.h"
```

Når kompileren når til den linie, stopper kompileringen i struct3.c, og kompilering i support1.h påbegyndes. Når headerfilen er færdigcompileret, vil kompileringen fortsætte i struct3.c filen, derfra hvor den slap.Bemærk at der skal bruges gåseøjne ” omkring filnavnet, ikke klammer som vi hidtil har benyttet. Det skyldes, at filen er placeret i samme direktory som den der inkluderer den.

Vores program er stadig lidt rodet, så vi vil prøve at skabe en bedre struktur, og dermed et bedre overblik for programøren. Ved at starte med en lille pseodokode, kan vi designe den overordnede struktur, inden vi skriver den første linie kode, det kan eksempelvis se således ud.

```
while (der er tastet mindre end 3 personer ind)
     hentPerson
while (der er personer i listen)
     udskrivPerson
```

Og det passer jo meget godt med udseendet af main i programmet struct4.c. Ved at holde main nede i størrelse, er der skabt mere overblik, og senere ændringer i programmet er meget overkommeligt.

```
/* Filnavn = struct4.c */

#include <stdio.h>
#include "support1.h"

struct personType hentPerson(struct personType person)
{
	char tmp[4];

	printf("%s", "Indtast fornavn : ");
	hentLinie(person.forNavn, 20);
	printf("%s", "Indtast efternavn : ");
	hentLinie(person.efterNavn, 20);
	printf("%s", "Indtast alder : ");
	hentLinie(tmp, 4);
	person.alder = atoi(tmp);
	return person;
}

void udskrivPerson(struct personType person)
{
	printf("%s\t", person.forNavn);
	printf("%s\t", person.efterNavn);
	printf("%4d\n", person.alder);
}

int main()
{
	int i;
	int antal = 3;
	struct personType personer[10];

	for (i = 0; i < antal; i++)
		personer[i] = hentPerson(personer[i]);
	printf("%s\n", "Udskriver");
	for (i = 0; i < antal; i++)
		udskrivPerson(personer[i]);
	return 0;
}
```

Samme input og output som struct2.c

Figur 4.3.4

Det er værd at bemærke, at vi overfører en kopi af det `i'`de element til `hentPerson` funktionen, når kopien er udfyldt returneres værdien på same måde som med int, og andre variabler. De to funktioner `hentPerson` og `udskrivPerson` arbejder begge med en,

og kun en persons arkivkort, det er med til at skabe overblik når vi programmerer.

Med den nye struktur er det nemt at tilføje nye programdele, uden at vi skal til at ændre i det vi allerede lavede. Man kan sige en god programør er en doven programør, han er for doven til at lave det samme arbejde to gange, gentagelser har vi computere til.

Opgave 4.3.1 Skriv en menufunktion til programmet struct4.c. Det skal fungere ved, at programmet ved start tilbyder brugeren en menu. I menuen skal der være to valgmuligheder, indtast ny person og udskriv samlet personliste. Hint pas på du ikke indtaster flere end der er dimensioneret, og pas på du ikke kommer til at udskrive en tom liste.

Opgave 4.3.2 Udvid programmet fra 4.3.1 med en funktion der kan slette en person i arkivet. Menuen skal selvfølgelig tilpasses, således, at der er et menupunkt der hedder slet.

Opgave 4.3.3 Udvid programmet fra 4.3.1 med en funktion der kan ændre en person i arkivet. Menuen skal naturligvis tilpasses således, at der er et menupunkt der hedder ændre.

Opgave 4.3.4 Udvid programmet fra 4.3.1 med en funktion der kan gemme arrayet i en fil. Menuen skal naturligvis tilpasses således, at der er et menupunkt der hedder gemme. Hint: Dette er en krævende opgave på nuværende tidspunkt, så du skal ikke skamme dig over at vente med at løse opgaven.

Opgave 4.3.5 Udvid programmet fra 4.3.4 med en funktion der kan hente arrayet fra den fil du gemte til i opgave 4.3.4. Menuen skal naturligvis tilpasses således, at der er et menupunkt der hedder gemme. Hint: Dette er også en krævende opgave på nuværende tidspunkt, så du skal ikke skamme dig over at vente med at løse opgaven. Men når du har løst opgave 4.3.4 bør du også kunne løse denne opgave.

Opgave 4.3.6 Skriv et program der kan generere et spil kort. Hint: Et kortspil består af 52 kort. Hvert kort har en farve (Spar, Klør, Hjerter og Ruder) og en værdi (1 = es ... 13 = konge).

4.4 Kædede lister

Rigtig praktisk bliver struct's først når vi begynder at sætte dem ind i en kædet liste. En kædet liste er et variabelt antal struct's, der holdes sammen af pointere. Med kædede lister kan man løbende ændre på antallet af struct som man bruger, og det er ikke nødvendigt at dimensionere noget. Hvor array er en statisk datastruktur er kædede lister en dynamisk datastruktur, fordi vi hele tiden kan ændre størrelse.

Der er mange ting der bliver nemmere når vi bruger pointere i stedet for array, i dette afsnit vil vi konstruere struct4.c om, fra at bruge array til at bruge pointere, hjælpeprogrammet i headerfilen (support1.h) vil blive omskrevet så den passer den nye opgave og bliver nu kaldt support2.h.

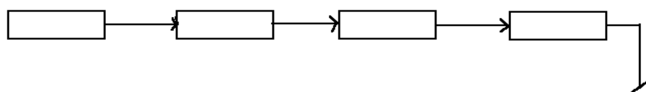
Strukturen `personType` har fået tilføjet en linie, med en pointer der kan pege på en `personType`. Det er kutyme at kalde denne pointer for `next`, da den skal pege på det næste element (arkivkort) i listen.

```
/* Filnavn = support2.h */


struct personType
{
	int alder;
	char forNavn[20];
	char efterNavn[20];
	struct personType *next;
};

int hentLinie(char linie[], int MAX)
{
	int i = 0;
	char ind;

	while (i < MAX-1 && ( ind = getchar() ) != '\n')
		linie[i++] = ind;
	linie[i] = '\0';
	return i;
}


void hentPerson(struct personType *person)
{
	char tmp[4];

	printf("%s", "Indtast fornavn : ");
	hentLinie(person->forNavn, 20);
	printf("%s", "Indtast efternavn : ");
	hentLinie(person->efterNavn, 20);
	printf("%s", "Indtast alder : ");
	hentLinie(tmp, 4);
	person->alder = atoi(tmp);
}

void udskrivPerson(struct personType *person)
{
	printf("%s\t", person->forNavn);
	printf("%s\t", person->efterNavn);
	printf("%4d\n", person->alder);
}
```

Figur 4.4.1

`hentLinie` funktionen laver vi ikke om på, der er ikke noget at optimere i den. Til gengæld er der en del ændringer i `hentPerson` og `udskrivPerson` funktionerne. For det første er funktionens argumet ændret fra at være en `personType`, til at være en pegepind til en `personType`. Det betyder at det ikke er nødvendigt med en returtype, da en pointer jo peger direkte ind i det dataareal man vil skrive i.

For det andet er alle punktummerne inde i funktionerne udskiftet med -> (bindestreg, større end; en pegepind). Det er valgt for at vise vi nu arbejde med pointere. Herudover er der ingen forskel fra tidligere.



Figur 4.4.2

En kædet liste kan vises grafisk som et antal kasser med en pil til at symbolisere nextpointere. Den sidste skrå streg kommer fra symbolet for en jordforbindelse i elektronikens verden, det er nulpointeren.

```
/* Filnavn = struct5.c */

#include <stdio.h>
#include "support2.h"

int main()
{
	int i;
	int antal = 3;
	struct personType *pNy, *pGammel, *pStart;

	pStart = pGammel = pNy =
		(struct personType *) malloc( sizeof(struct personType) );
	for (i = 0; i < antal; i++)
	{
		hentPerson(pNy);
		pNy =   (struct personType *)
			malloc( sizeof(struct personType) );
		pGammel->next = pNy;
		pGammel = pNy;
		pNy->next = NULL;
	}
	printf("%s\n", "Udskriver");
	pNy = pStart;
	while (pNy->next != NULL)
	{
		udskrivPerson(pNy);
		pNy = pNy->next;
	}
	return 0;
}
```

Samme input og output som struct2.c

Figur 4.4.2

Der er en del ændringer i hovedprogrammet, til at starte med opretter vi tre pointerer der kan pege på strukturer af typen `personType`. Derefter tilskriver vi de samme tre pointere med returværdien fra `malloc` funktionen. Malloc er en forkortelse for Memory ALLOCation, funktionens opgave er at frigive et stykke af computerens hukommelse, og aflevere en pegepind til starten af det frigivne område. Størrelsen på det reserverede dataareal er givet ved argumentet der skal være et heltal. I dette tilfælde finder vi størrelsen på vores struct ved at bruge `sizeof` operatoren, der returner operandens størrelse målt i hukommelsesenheder. Malloc returnerer en pointer til void, hvorfor vi er nød til at lave en typekonverteriong til en pointer der kan pege på en `personType`, ved hjælp af en cast.

Når denne linie er udført under eksekveringen, har vi tre pointere der peger på det samme sted i hukommelsen, som er tildelt af computerens operativsystem. Startpointeren skal blive ved med at pege på dette sted, fordi her vil vores liste starte, de to andre pointere er hjælpepointere.

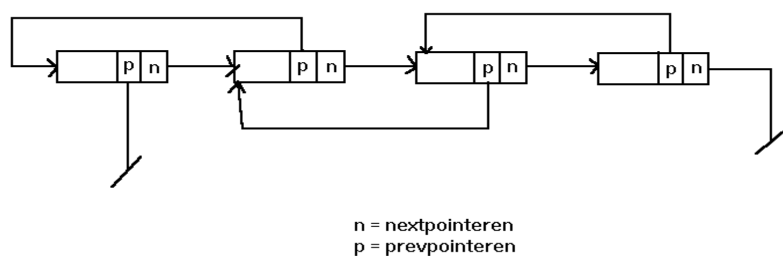
Når vi har oprettet det første dataareal, kalder vi `hentPerson` funktionen, med en pointer til arealet som argument. Når funktionen er udført reserverer vi et nyt stykke af hukommmelsen til det næste arkivkort, som kun `*pNy` får lov til at pege på. Derefter lader vi nextpointeren på det førstoprettede arkivkort, til at pege på det nyligt oprettede arkivkort. Det betyder at vi nu har en liste med to arkivkort, hvor det andet er tomt.

Derefter sættes `*pGammel` til at pege på den nyoprettede pointer, så den er klar til næste gang vi går rundt i løkken. I praksis kan løkken nu køre lige så mange gange du har løst til. Før eller siden vil du dog løbe tør for hukommmelse, det tager programmet ikke højde for. Hvis malloc ikke kan få lov til at frigive mere hukommelse, returner den værdien nul, hvis det er tilfældet brude programmet stoppe med en fejlmeddelelse.

Funktionen `udskrivPerson` fremhæver fordelen ved at bruge pointere, uden at ane hvor lang databasen er kan vi løbe alle data igennem blot ved at skrive:

```
while (pNy->next != NULL)
{
     udskrivPerson(pNy);
     pNy = pNy->next;
}
```

Når vi bruger pointere er det nemt at indsætte et nyt arkivkort inde midt i en liste. Det er vidst med programmet struct6.c hvor funktionen `indsaetArkivkort` modtager en pointer til arkivkortet lige før det sted hvor vi ønsker at indsætte vores nye arkivkort.



Figur 4.4.2

En dobbeltkædet liste kan vises grafisk med en pil for både next og prevpointerne. Det er nødvendigt med en nulpointer i begge ender af listen, da programmet i sagens natur kan køre begge veje i listen.

```
/* Filnavn = struct6.c */

#include <stdio.h>
#include "support2.h"

void indsaetArkivkort(struct personType *person)
{
	struct personType *tmp;

	tmp = (struct personType *) malloc( sizeof(struct personType) );
	strcpy(tmp->forNavn, "Hans");
	strcpy(tmp->efterNavn, "Hansen");
	tmp->alder = 23;
	tmp->next = person->next;
	person->next = tmp;
}

int main()
{
	int i;
	int antal = 3;
	struct personType *pNy, *pGammel, *pStart;

	pStart = pGammel = pNy =
		(struct personType *) malloc( sizeof(struct personType) );
	for (i = 0; i < antal; i++)
	{
		hentPerson(pNy);
		pNy =   (struct personType *)
			malloc( sizeof(struct personType) );
		pGammel->next = pNy;
		pGammel = pNy;
		pNy->next = NULL;
	}
	printf("%s\n", "Udskriver");
	pNy = pStart;
	for (i = 0; i < 1; i++)
		pNy = pNy->next;
	indsaetArkivkort(pNy);
	pNy = pStart;
	while (pNy->next != NULL)
	{
		udskrivPerson(pNy);
		pNy = pNy->next;
	}
	return 0;
}
```

```
$ struct6.c
Indtast fornavn : Jens
Indtast efternavn : Jensen
Indtast alder : 23
Indtast fornavn : Peter
Indtast efternavn : Petersen
Indtast alder : 32
Indtast fornavn : Ib
Indtast efternavn : Ibsen
Indtast alder : 89
Udskriver
Jens   Jensen  23
Peter  Petersen 32
Hans Hansen 23
Ib     Ibsen    89
$
```

Figur 4.4.3

For at have en pointer der peger ind midt i listen, har vi indført linierne:

```
pNy = pStart;
for (i = 0; i < 1; i++)
	pNy = pNy->next;
indsaetArkivkort(pNy);
```

i main. Funktionen `indsaetArkivkort` starter med at oprette en temporær pointer, og allokerer plads til vores data. Derefter indlæser vi Hans Hansens data. Hans Hansens arkivkort er derefter klar til at blive indsat i listen. Vi starter med at lade `tmp->next` pointeren pege på det samme som `person->next` peger på, det er jo det næste arkivkort i listen. Derefter sætter vi det forrige arkivkort i listen til at pege på vores nyoprettede arkivkort, og så er vi faktisk færdig. Da vi arbejder med pointere skal der ikke returneres nogen værdier.

Opgave 4.4.1 Tilføj funktionen `antalArkivkort` til programmet struct5.c. Funktionen skal tælle hvor mange udfyldte arkivkort der er i listen, og udskrive antallet på skærmen.

Opgave 4.4.2 Omskriv programmet fra opgave 4.3.1, således at det bruger pointere.

Opgave 4.4.3 Omskriv programmet fra opgave 4.3.2, således at det bruger pointere.

Opgave 4.4.4 Omskriv programmet fra opgave 4.3.3, således at det bruger pointere.

Opgave 4.4.5 Omskriv programmet fra opgave 4.3.6, således at det bruger pointere.

Opgave 4.4.6 Udvid programmet fra 4.4.5 med en funktion der kan dele kort ud. Funktionen skal have et heltals argument, der angiver antallet af kort der skal uddeles. Hint: Når man deler kort ud, tager man kortene ud af kortspillet, sætter dem ind i en ny kædet liste, og afleverer en pointer til den nye liste.

4.5 Dobbeltkædede lister

Ofte kan det være en fordel at en liste er dobbeltkædet, det gør det nemmere at manipulere med de enkelte elementer i listen. En dobbeltkædet liste er næsten magen til den var har set, for hvert element er der blot to pointere, `next` som vi kender, og `prev` (forkortelse for det engelske previos = forrige) som skal pege på det forrige element i listen.

For at benytte en dobbeltkædet liste skal personType udvides med en prevpointer, således:

```
struct personType
{
     int alder;
     char forNavn[20];
     char efterNavn[20];
     struct personType *prev, *next;
};
```

Der skal ikke ændres på `hentPerson` og `udskrivPerson` funktionerne, da vi har været så forudseende, at lade dem manipulere med et enkelt arkivkort, og ikke hele listen. Hovedprgrammet skal til gengæld udvuddes lidt, så vi har styr på prev pointerne.

```
/* Filnavn = struct7.c */

#include <stdio.h>
#include "support3.h"

int main()
{
	int i;
	int antal = 3;
	struct personType *pNy, *pGammel, *pStart;

	pStart = pGammel = pNy =
		(struct personType *) malloc( sizeof(struct personType) );
	pNy->next = NULL;
	pNy->prev = NULL;
	for (i = 0; i < antal; i++)
	{
		hentPerson(pNy);
		pNy =   (struct personType *)
			malloc( sizeof(struct personType) );
		pNy->next = pGammel->next;
		pGammel->next = pNy;
		pNy->prev = pGammel;
		pGammel = pNy;
	}
	printf("%s\n", "Udskriver");
	pNy = pStart;
	while (pNy->next != NULL)
	{
		udskrivPerson(pNy);
		pNy = pNy->next;
	}
	printf("%s\n", "Udskriver baglæns");
	pNy = pNy->prev;
	while (pNy != NULL)
	{
		udskrivPerson(pNy);
		pNy = pNy->prev;
	}
	return 0;
}
```

```
$ struct7.c
Indtast fornavn : Jens
Indtast efternavn : Jensen
Indtast alder : 23
Indtast fornavn : Peter
Indtast efternavn : Petersen
Indtast alder : 32
Indtast fornavn : Ib
Indtast efternavn : Ibsen
Indtast alder : 89
Udskriver
Jens   Jensen  23
Peter  Petersen 32
Ib     Ibsen    89
Udskriver baglæns
Ib     Ibsen    89
Peter  Petersen 32
Jens   Jensen  23
$
```

Figur 4.5.1

Ved at indføre linierne:

```
pNy->next = NULL;
pNy->prev = NULL;
```

Kan vi lave lidt mere praktisk kodning lidt længere nede i programmet. Det sker i linien:

```
pNy->next = pGammel->next;
```

hvor `pNy->next` bliver sat til at pege på det som `pGammel->next` peger på. Det er vigtigt dette sker før der bliver rørt ved `pGammel->next`, da vi ellers vil ”miste

forbindelsen" med nulpointeren. Ellers er det eneste nye i denne del af programmet linien:

```
pNy->prev = pGammel;
```

Her sættes `pNy->prev` pointeren til at pege på det forrige element, og herved etableres den dobbeltkædede liste.

For at vise den dobbeltkædede liste virker, løber vi baglæns genem listen, efter først at have udskrevet den forlæns. Årsagen til vi først klikker `pNy` en tand tilbage inden vi begynder er, at når vi opretter laver vi altid en for meget, så der er en klar til næste gang vi skal sætte data ind.

Nu kan vi laver det tankeeksperiment, at vi forbinder startpointerens prev med det sidste element i listen, og det sidste element i listens next pointer med startpointeren, så har vi en ringkonstruktion. Dette benyttes bl.a. til at skabe ringbuffere, der bl.a. benyttes til at hente data ud af et netværkskort.

Hvis vi ønsker at frigive hukommelse der er allokeret med `maloc()` skal vi kalde `free(void *p)` funktionen, med en pointer til det sted vi ønsker at frigive.

Opgave 4.5.1 Tilpas `indsætArkivkort` funktionen, så den kan bruges i struct7.c programmet. Husk at styre prevpointeren, ellers går det galt når listen skal skrives baglæns ud.

Opgave 4.5.2 Udvid programmet struct7.c med en funktion der kan bytte om på to elemener der ligger ved siden af hindanden.

Opgave 4.5.3 Udvid programmet struct7.c med en funktion der kan bytte om på to elemener der ikke ligger ved siden af hindanden.

4.6 Fra en kædet liste til en fil

Det kan være lidt omstændigt hvis vi altid skal bruge `putc` og `getc` til at gemme på en fil, da de kun tager et tegn af gangen. Derfor er `printf` viderudviklet så den også kan arbejde på filer, den kaldes da `fprintf`, og fungerer på næsten samme måde som den `printf` vi kender. Den eneste forskel er faktisk, at første argument skal være en filpointer, og vi kan teste returværdien så vi kan spørge om vi har nået enden af filen. Programmet struct8.c viser hvordan man kan benytte `fprintf` til at gemme strcturer direkte i filer, og indlæse dem igen med `fscanf`. Af hensyn til pladsen vises funktionen **`fraFil`** for sig selv.

```
/* Filnavn = struct8.c */

#include <stdio.h>
#include "support3.h"

void tilFil(struct personType *p) {
	char filnavn[] = "test.db";
	FILE *fp;

	if ( (fp = fopen(filnavn, "wb")) != NULL)
	{
		while (p->next != NULL)
		{
			fprintf(fp, "%4d%20s%20s",  p->alder, p->forNavn, p->efterNavn);
			p = p->next;
		}
		fclose(fp);
	}
	else
		printf("%s\n", "I/O Fejl");
}

int main() {
	int i;
	int antal = 3;
	struct personType *pNy, *pGammel, *pStart;

	pStart = pGammel = pNy =
		(struct personType *) malloc( sizeof(struct personType) );
	pNy->next = NULL;
	pNy->prev = NULL;
	for (i = 0; i < antal; i++)
	{
		hentPerson(pNy);
		pNy =   (struct personType *)
			malloc( sizeof(struct personType) );
		pNy->next = pGammel->next;
		pGammel->next = pNy;
		pNy->prev = pGammel;
		pGammel = pNy;
	}
	printf("%s\n", "Gemmer på fil");
	tilFil(pStart);
	free(pStart);
	pStart = fraFil();
	printf("%s\n", "Udskriver");
	pNy = pStart;
	while (pNy->next != NULL)
	{
		udskrivPerson(pNy);
		pNy = pNy->next;
	}
	return 0;
}
```

Figur 4.6.1

Programmet bygger i vid udstrækning videre på struct7.c men der er et par nye tekniker

som vi vil se på. Efter der er tastet 3 arkivkort ind, så kaldes funktionen `tilFil()` med adressen på starten af den kædede liste som argument, dette er indholdet af startpointervariablen, der kopieres ind i *p i funktionen. Funktionen søger at åbne filen med "wb" som argument, hvilket betyder, at hvis filen findes i forvejen, så bliver den slettet, og en ny oprettes, hvis den ikke findes, bliver den bare oprettet. Bogstavet b betyder at filen åbnes som en binær fil, hvilket kan være nødvendigt for at gemme andet end tegn.

Hvis det lykkes at åbne filen, gennemløbes den kædede liste, og for hver runde gemmes det samlede indhold af hver struct med linien:

```
fprintf(fp, "%4d%20s%20s",  p->alder, p->forNavn, p->efterNavn);
```

Den eneste forskel fra det vi kender, er brugen af filpointeren `fp`, til at gemme med. I denne ene linie skrives tre variebler til en fil, i den rækkefølge de er skrevet. Når linein er udført klikkes pointeren til liste en frem, og vi tager en omgang mere. Det fortsættes indtil vi møder nulpointeren. Funktionen er altså helt uafhængig af hvor mange arkivkort vi tidligere har indtastet.

Når der ikke er mere at gemme, lukkes filen med `fclose()`. Når denne linie er udført, vil filen blive gemt af styresystemet.

I hovedprogrammet er den næste linie vi møder:

```
free(pStart);
```

Den linie har til formål at frigive den lagerplads der blev optaget af vores liste. Efter kaldet til free er listen ikke længere i vores hukommelse. I den næste linie:

```
pStart = fraFil();
```

Hentes filen ind igen, ved hjælp af funktionen `fraFil()`.

```
struct personType *fraFil()
{
	int i = 0;
	char filnavn[] = "test.db";
	struct personType *p;
	FILE *fp;
	struct personType *pNy, *pGammel, *pStart;

	p = (struct personType *) malloc( sizeof(struct personType) );
	pStart = pGammel = p;
	if ( (fp = fopen(filnavn, "rb")) != NULL)
	{
		while(fscanf(fp, "%4d%20s%20s", &p->alder, p->forNavn, p->efterNavn) !=
EOF)
		{
			if (p = (struct personType *) malloc( sizeof(struct personType) ))
			{
				pGammel->next = p;
				p->prev = pGammel;
				p->next = NULL;
				pGammel = p;
			}
			else
				printf("%s", "Der er ikke mere RAM");
		}
		fclose(fp);
	}
	else
		printf("%s\n", "I/O Fejl");
	return pStart;
}
```

Figur 4.6.2

Når vi skal indlæse til en kædet liste, fra en fil, er vi nød til at oprette lagerplads med malloc, efterhånden som vi henter nye arkivkort ind fra filen. Vi har også brug for en startpointer, som jo skal afleveres til den der kalder funktionen. Til sidst er der også brug for en gammelpointer, til at holde styre på det forrige arkivkort, så vi kan få pointerne til at pege rigtigt.

Når vi læser ind fra filen skal vi benytte `fscanf` funktionen, der fungerer omvendt af fprintf funktionen, der indlæses indtil vi møder EOF (End Of File). Undervejs skal der oprettes nye dataarealer, på samme måde som da vi foretog indtastning, ellers er der jo ikke plads til vores data.

For at undgå en katastrofe hvis vi læser så mange data ind, at der ikke er mere RAM, tester vi for om `malloc` returnerer nul, hvilket betyder der ikke kunne reserveres mere

plads. Egentlig burde den første linie hvor vi bruger malloc også testes, men det er undladt for at skabe overskuelighed i eksemplet. Du bør altid teste om du fik den plads du rekvirerer. Hvis du en dag opbruger din RAM, og ikke kontrollerer det i programmet, vil du normalt få en programafbrydelse.

Til slut lukkes filen, og vi udskriver på skærmen for et teste vores lille program.

Specielt ved indlæsning er det en stor fordel at bruge kædede lister. Alternativt kunne man have valgt at bruge et array af struct's, men hvor stort skulle det være? Det er meget nemmere at bruge pointere, så kan man bare læse ind.

Inden du går videre til opgaverne, bør du prøve at kigge på test.db filen ved hjæp af monitorprogrammet fra kapitel 3.

Opgave 4.6.1 Tilpas programmet struct8.c således, at filnavnet overgives til tilFil og fraFil som argument.

Opgave 4.6.2 Tilpas programmet struct8.c med en funktion der skal gøre det muligt at søge efter en bestemt person i filen på fornavn

Opgave 4.6.3 Tilpas programmet struct8.c, med en funktion der kan hente det andet element fra filen, og skrive det ud på skærmen. Der må ikke benyttes en kædet liste. Hint: Læs et element ind ad gangen, overskriv det når du læser næste element ind.

Opgave 4.6.4 Udvid og tilpas programmet struct8.c med en menufunktion, der skal gøre det muligt at vælge mellem opret arkivkort, slet arkivkort, udskriv alle arkivkort, og forlad programmet.

4.7 Opgaver

Opgave 4.7.1 Skriv et databaseprogram til frimærker. Programmet skal kunne registrere og opbevare følgende data på en fil:

- Land
- Porto
- Møntfod
- Antal takker

Opgave 4.7.2 Udvid programmet fra opgave 4.7.1 med en funktion der kan tælle hvor mange frimærker der er fra et bestemt land i databasen.

Opgave 4.7.3 Udvid programmet fra opgave 4.7.1 med en funktion der kan beregne det gennemsnitlige antal takker for frimærker i databasen.

Opgave 4.7.4 Udvid programmet fra opgave 4.7.1 med en funktion der kan udskrive alle frimærker over en bestemt porto.

Opgave 4.7.5 Skriv et databaseprogram til en musiksamling. For hver CD skal der være registreret:

- Musiker
- Titel
- Indspilningsår
- Antal numre

Opgave 4.7.6 Udvid programmet fra 4.7.5 med en funktion der kan finde en bestemt CD.

Opgave 4.7.7 Udvid programmet fra 4.7.5 med en funktion der kan bestemme hvor mange forskellige kunstnere der er i databasen.

5. Praktiske ting.

I dette kapitel vil vi afrunde med lidt løst og fast. Erfaringsmæssigt er det ting og begreber begynderen finder irriterende, og derfor springer over. Men for at blive en god programmør skal disse begreber også beherskes, så hvis du springer dem over nu, kan du komme tilbage til dem senere, når energien er til stede. Det første vi vil se på er rekursion.

5.1 Rekursion

Rekursion er når en funktion kalder sig selv, det er faktisk en anden måde at lave en løkke på. Programmet rekurs1.c demonstrerer rekursion.

<pre>/* Filnavnb = rekurs1.c */
 	
#include <stdio.h>
 	
void rekurs(char input[], int antal)
{
    if (input[antal] != '\0')
    {
       input[antal] = input[antal] - ('a' - 'A');
       printf("%s", "Har nu konverteret");
       printf("%s%d\n", " bogstav ", antal);
       rekurs(input, antal + 1);
    }
    printf("%d\n", antal);
}
 	
int main()
{
    char data[] = "abc";
 	
    printf("%s", "Før kald til rekurs = ");
    printf("%s\n", data);
    rekurs(data, 0);
    printf("%s", "Efter kald til rekurs = ");
    printf("%s\n", data);
}</pre>
	$ rekurs1.c
	Før kald til rekurs = abc
	Har nu konverteret bogstav 0
	Har nu konverteret bogstav 1
	Har nu konverteret bogstav 2
	3
	2
	1
	0
	Efter kald til rekurs = ABC
	$

Figur 5.1.1

For at demonstrere effekten i et rekursivt kald, har vi ladet funktionen konvertere fra små til store bogstaver.

Til at starte med kaldes rekurs fra main med to argumenter, en streng der skal konvergeres til store bogstaver, og et heltal der oplyser på hvilket sted i strengen konverteringen skal begynde. Første gang rekurs køres er det element i strengen som antal peger på ikke nultegnet, hvorfor konvertering skal finde sted. Det betyder at det tegn i

strengen som antal peger på bliver konverteret. Selve konverteringen foregår i linien:

```
input[antal] = input[antal] - ('a' - 'A');
```

Da afstanden mellem store er og små bogstaver er konstant, bortset fra de danske tegn, kan vi bestemme denne konstante værdi ved at trække tegnværdien af bogstavet store A, fra bogstavet lille a. Ved at bruge denne teknik kan kildeteksten bruges på alle platforme. Hvis vi i stedet havde indtastet konstantens talværdi, ville programmet fungere forskelligt på forskellige platforme.

Derefter skrives der lidt ud på skærmen så vi kan følge med i hvad der sker, og så kommer linien:

```
rekurs(input, antal + 1);
```

Det medfører at funktionen rekurs kalder sig selv, men med en ny værdi for antal. Når der foretages et rekursivt kald som her, så vil der blive skabt en helt ny udgave af funktionen i computerens hukommelse, denne helt nye funktion er magen til, men ikke den samme som den vi lige kom fra. Den nye udgave af rekurs bliver kaldt med argumentet antal + 1, hvilket medfører, at den nye version af rekurs konverterer det andet element i char arrayet. Sådan fortsætter det frem til antal peger på nultegnet, hvorefter if betingelsen falder, der sker derfor ingen konvertering. I stedet udføres linien:

```
printf("%d\n", antal);
```

Hvilket medfører at tallet tre bliver udskrevet. Derefter afsluttes funktionen, og programmet vender tilbage til hvor det kom fra, hvilket er en tidligere udgave af rekurs, umiddelbart efter kaldet til rekurs. Det næste programet møder er en sluttuborg, hvorefter linien:

```
printf("%d\n", antal);
```

mødes, og der udskrives nu tallet 2. Dette fortsætter til programmet har viklet sig baglæns ud af alle kald til rekurs.

Rekursive kald kan være meget sjove at arbejde med, men da der afsættes ny hukommelse hver gang en funktion kalder sig selv under afviklingen, risikerer man at løbe tør for hukommelse. Dette vil altid ske, hvis slut betingelsen ikke kan opfyldes, og det svarer til at et program hænger, fordi det er gået ind i en endeløs løkke.

Forskellen på en løkke og et rekursivt kald der går i evigt kredsløb er altså, en løkke kører og kan stoppes ved at slå programmet ihjæl, et rekursivt kald stopper sig selv med en programafbrydelse når det løber tør for hukommelse.

Nogle operativsystemer håndterer ikke et rekursivt kald der løber tør for hukommelse særligt godt, og bryder sammen. I den forbindelse skal man være klar over, at et rekursivt kald også kan opbruge hukommelsen, blot ved at foretage et meget dybt rekursivt kald.

Personligt undgår jeg at bruge rekursive kald, for helt sikkert at undgå katastrofen med et crashet styresystem.

Opgave 5.1.1 Skriv et program der har en rekursiv funtion der indkoder efter Cæsarkoden. Hint: Se opgave 2.3.2.

Opgave 5.1.2 Skriv et program med en rekursiv funktion der konverterer fra et chararray til et heltal.

5.2 Include filer

C er udstyret med en række standard biblioteker der ofte, men ikke altid er placeret i et directory ved navn lib. Vi har brugt stdio.h ofte, men der er også andre praktiske biblioteker, her er nogle relevandte listet:

```
ctype.h
String.h
math.h
stdlib.h
```

Hvor disse filer ligger i dit filsystem, afhænger af din kompiler. Hvis du vil vide mere om dem, skal du slå op i din compilermanual.

Hvis du bygger dine egne headerfiler kan det være praktisk at placere dem samme sted som det projekt de skal bruges til. Der er to måder at includere en headerfil på:

```
#include <xxxx.h>
#include "yyyy.h"
```

Den første måde har vi set mange gange, den includerer en headerfil, som er placeret på et af compileren defineret sted. Den anden betyder, at filen er placeret i samme directory som den fil der inkluderer den.

Ofte har man brug for at bygge sit eget bibliotek op, så man kan genbruge den kode man allerede har skrevet en gang. Vi har jo netop sådan en kode i hentLinie funktionen, som det kan være praktisk at undgå at taste ind igen og igen. Derfor kan vi lægge den i en fil ved navn biblo.c, og gemme den i samme directory som de programmer der skal bruge den.

```c
/* Filnavn = biblo.c */

#include <stdio.h>

int hentLinie(char linie[], int MAX)
{
    int i = 0;
    char ind;

    while (i < MAX && ( ind = getchar() ) != '\n')
        linie[i++] = ind;
    linie[i] = '\0';
    return i;
}
```

Denne fil skal ikke compiles, den skal blot gemmes i samme directory som de programmer der ønsker at bruge den.

Figur 5.2.1

For at gøre funktionen mere anvendelig, overfører vi MAX i funktionskaldet. Når vi har gemt biblo.c kan vi prøve at skrive et lille program for at teste det, det kan jo eksempelvis hedde biblotest.c.

```c
/* Filnavn = biblotest.c */

#include <stdio.h>
#include "biblo.c"

#define MAX 255

int main()
{
    char streng[MAX];

    hentLinie(streng, MAX);
    printf("%s\n", streng);
}
```

```
$ cc biblotest.c
$ biblotest
abc
abc
$
```

Figur 5.2.2

I filen biblotest.c er der ingen funktion ved navn hentLinie, så vi burde få en compilerfejl. Men da vi har inkluderet filen biblo.c i linien:

```
#include "biblo.c"
```

Kan main "se" funktionen, uden at den fylder op i vores kildetekst. Det betyder, at hver gang vi compilerer biblotest.c, så vil vi også compilere biblo.c.

Ved at bruge inkludefiler og placere dele af programmet der hører naturligt sammen i hver sin fil, kan vi gøre vores kode mere overskuelig. Compileren er ligeglad med om vi gør det, men når først et program vokser forbi de 50.000 linier, hvilket er ganske normalt, så er det rart at kunne overskue kildeteksterne.

Headerfiler bygges normalt op således, at filer der indeholder C kildekode skal have filextendet .c, filer der indeholder datadefenitioner skal navngives .h. Et databaseprogram kan udfra denne tekik opbygges som tre filer:

data.h
data.c
main.c

Main vil inkludere data.c, der vil inkludere data.h for at ”få fat i” erklæringsdelen til data programdelen. I korte træk vil filerne se således ud:

```
/* Filnavn = data.h              */
/* Dette er fil for erklæringer */

struct personType
{
     int alder;
     char forNavn[20];
     char efterNavn[20];
     struct personType *prev, *next;
};
```

Figur 5.2.3

```
/* Filnavn = data.c                          */
/* Dette er fil for databasenprogrammet */

#include ”data.h”

void opretArkivkort(.....)
{
     ..........
}

void sletArkivkort(.....)
{
     ..........
}

void skrivArkivkort(.....)
{
     ..........
}
```

Figur 5.2.4

```
/* Filnavn = data.c                    */
/* Dette er fi for hovedprogrammet */

#include ”data.c”

int main()
{
     .........
}
```

Figur 5.2.5

Når du har fået en del programmeringserfaring vil du opdage at programmering består af

to elementer, data og kode. Kode manipulerer med data, og er altså aktiv, data er altid passiv, og kan være placeret i variabler eller på filer.

For at undgå den samme fil bliver kompileret flere gange, i tilfælde af at flere dele af et program inkluderer den samme fil, kan man bruge kompilerdirektiver. Der er vist et eksempel på et kompilerdirektiv i figur 5.2.6.

```
/* Filnavn = data.h */

#if !defined(DATA)
#define DATA

struct personType
{
        int alder;
        char forNavn[20];
        char efterNavn[20];
        struct personType *prev, *next;
};
#endif
```

Figur 5.2.6

Til at begynde med skal vi opfatte #if som både en if erklæring og et startblokmærke, og vi skal betragte `#endif` som et tilhørende blokslut mærke. Direktivet fungerer på følgende måde. Første gang kompileren forsøger at kompilere filen, undersøger den om der er defineret en konstant ved navn DATA, det er der ikke, derfor udføres næste linie hvor konstanten DATA oprettes. Derefter kompileres koden på normal vis. Hvis nu kompileren igen bliver bedt om at kompilere data.h, så vil if linien konstatere, at DATA er defineret, hvorfor betingelsen falder, og kompilering fortsættes efter blokslut mærket, der er `#endif`.

Opgave 5.2.1 Skriv en inkludefil der indeholder en funktion for hver af de fire regnearter. Funktionerne skal modtage to argumenter af typen heltal, og returnere resultatet. Test alle

fire funktioner fra et main program der inkluderer filen.

Opgave 5.2.2 Tilpas svarende fra opgave 4.7 således at programmet benytter headerfiler og kompilerdirektiver.

5.3 Mere om variabler og rækkevidde

Når et program blive bygget op af flere, ofte mange filer, er det vigtigt at kende variablernes rækkevidde, da man nemt kan komme til at erklære to variabler med samme navn i to forskellige filer.

Det er en regel i C, at variabler gælder derfra hvor den er erklæret, og i resten af den fil hvori den er erklæret. En funktions rækkevidde er fra hvor den er erklæret, til sidst i det samlede program. Dette kan bedst vises med et eksempel, kig på programmet global1.c.

```
/* Filnavn = global1.c */

int minVar = 17;

void manipuler()
{
	minVar = 23;
}

int main()
{
	printf("%d\n", minVar);
	manipuler();
	printf("%d\n", minVar);
	return 0;
}
```

Figur 5.3.1

Variablen `minVar` erklæres og tilskrives før nogen funktion, derfor kan denne variabel ses af alle funktioner i denne kildetekst, det kaldes også en global variabel. Derfor er det ikke så mærkeligt, at når vi ændrer på variablens indhold i manipuler funktionen, så kan dette ses i main funktionen. Hvis vi flytter variablen `minVar` over i en headerfil, og inkluderer filen i hovedprogrammet, så er der ingen forskel, som vist i programmet global2.c

/* Filnavn = global2.h */ int minVar = 17;	Headerfil.

Figur 5.3.2

```
/* Filnavn = global2.c */

#include "global2.h"

void manipuler()
{
      minVar = 23;
}

int main()
{
      printf("%d\n", minVar);
      manipuler();
      printf("%d\n", minVar);
      return 0;
}
```

```
$ global2.c 
17
23
$
```

Figur 5.3.3

Men hvis vi erklærer en lokal `minVar` i mainfunktionen kan vi skabe en helt lokal variabel der ikke har noget med den globale udgave af `minVar` at gøre. Dette er vist i programmet global3.c.

<table>
<tr>
<td>

```
/* Filnavn = global3.c */

#include "global2.h"

void manipuler()
{
      minVar = 23;
}

int main()
{
      int minVar = 1;

      printf("%d\n", minVar);
      manipuler();
      printf("%d\n", minVar);
      return 0;
}
```

</td>
<td>

$ global2.c ↵
1
1
$

</td>
</tr>
</table>

Figur 5.3.4

Hvis vi ønsker den lokalt erklærede udgave af `minVar` skal være den samme som den globale, så kan `minVar` erklæres som *extern* på følgende måde:

```
extern int minVar;
```

Variablen må ikke tilskrives når den erklæres som extern.

Man kan spørge sig selv om hvorfor have begrebet extern, man kan da bare lade være med at erklære variablen i funktionen, så har man umiddelbart fat i den globale. Årsagen er, at når programmer vokser i størrelse, og dermed bliver sværere at overskue, så er det praktisk at man kan angive i den kode man arbejder med, at dette er en global variabel, som er erklæret et helt andet sted.

Globale variabler er i det hele taget et tveægget sværd, de er ikke til at undvære, men hvis brugen af dem overdrives, taber man hurtigt overblikket over sit program.

Et godt råd er, undgå systematisk at benytte globale variabler i din programmering, det hævner sig i form af stigende uoverskuelighed efterhånden som du får flere ideer til dit program og derfor udvider det. Lad være med at lade din dovenskab lokke dig til at lave kommunikation mellem to funktioner med en global variabel, alt for ofte får man ikke optimeret sit program når man er blevet færdig med at teste det, og så hænger de globale variabler som tidsindstillede bomber, som går af på de mest ubelejlige tidspunkter, altså når du har allermest travlt.

Variabler kan også erkæres som *static*, hvilket betyder de kun er synlige i den fil hvor de er defineret. Static har andre og mere spændende egenskaber, men det vil vi komme tilbage til i C++ bogen.

Til sidst skal det også nævnes, at en variabel kan erklæres som en registervariabel, det betyder, at compileren opfordres til at gemme den erklærede variabel direkte i et af CPU'ens registre. Det har specielt betydning i forbindelse med programmeringen af mikrokontrollere, hvor pladsen normalt er særdeles trang.

Man kan erklære en int som registervariabel således:

```
register int i;
```

Også andre variabeltyper kan erklæres som register.

Opgave 5.3.1 Omskriv programmet 5.2.1 således at ingen af funktionerne har argumenter i deres funktionskald, men i stedet bruger globale variabler.

Opgave 5.3.2 Omskriv programmet fra opgave 5.3.1 således at hver funktion placeres i

en selvstændig fil.

5.4 Private typedefenitioner

I C har vi ikke bare struct der giver og typer vi selv definerer, vi kan også give typerne et navn vi bedre kan lide, det kan også kaldes et alias.

<table>
<tr>
<td>

```
// Filnavn = typedef.c
# include <stdio.h>

typedef int Heletal;

int main ()
{
        Heletal beholder = 99;

        printf("%d\n", beholder);
        return 0;
}
```

</td>
<td>

$ cc typedef.c -o ttest
$ ./ttest
99
$

</td>
</tr>
</table>

Figur 5.4.1

Programmet typedef.c viser vi skaber et alias for typen int, og kalder den for Heltal. Alt andet fungerer som vi er vand til.

5.5 Unions

Den sidste ting i C er unions, de ligner struct, men har ikke noget med en struct at gøre.

Her er et eksempel på en union:

```
union demo {
        char a;
        char b;
        char ab[2];
} navn;
```

Ovenstående union vil have en størrelse i computerens hukommelse på 2 char, hvilket typisk er to byte. Årsagen er størrelsen på en union er lige præcis størrelsen på den største variabel inde i unionen.

Det betyder at variablerne IKKE er adskilt, tværtimod. I ovenstående eksempel vil der gælde at:

```
a = ab[0]   og   b = ab[1]
```

Hvis vi skal tilgå en variabel inde i en union sker det på samme måde som med en struct, eksempel:

```
navn.a = 'Q';
```

Tilskriver variblen a med tegnet Q, og så vil navn.ab[0] samtidig have værdien 'Q', fordi den er på den samme hukkommelsesplads.

En union kan allokeres i hukommelsen med en pointer på samme måde som en struct.

I denne bog vil vi ikke dykke dybere ned i begrebet union, men hvis du skal arbejde med

programmering tæt på hardwaren, eksempel vis du skal sende noget ud eller ind af computeren, (kaldet I/O) bør du bruge lidt tid på at forstå dem.

5.6 Kodebrydning

Nu hvor vi har været rundt om C sprogets faciliteter, er det på passende at runde det hele af, og kigge på et større eksempel.

Hvert et sprog har forskel på hyppigheden af de forskellige bogstaver, for en kodebryder er det nyttigt at kende denne hyppighed, da den benyttes i forbindelse med kodebrydningsteknik. Det kan derfor være rart at have en funktion der bestemmer den relative hyppighed af bogstaver i en tekst.

Det kan også være praktisk at vide hvor mange byte en fil fylder, og hvor mange linier der er i den. Vi vil derfor skrive et program der kan beskrives med følgende pseodekode.

```
while (der er byte i input)
{
     analyser for forekomst af bogstaver
     analyser forekomst af linieskiftet
     tæl antal byte
}
Udskriv resultatet
```

Ud fra psoedokoden kan det ses at det er praktisk at have tre funktionskald for hvert gennemløb af while løkken Vi kan altså identificere tre funktionskald:

```
analyserBogstaver
analyserLinieskift
udskrivResultat
```

Der er ingen grund til at foretage et funktionskald for at tælle antallet af byte, det kan jo bare gøres løbende i en variabel.

Ud over de nævnte funktioner, skal vi have en datastruktur til at opsummere data i, det vil være praktisk at bruge et char array på størrelse med alfabetet. For at gøre det nemt, bruger vi kun det engelske alfabet, fra a til z.

Med den viden kan vi starte med at skrive vores main kode. Den kan se ud som vist på Figur 5.4.1.

Kode	Kommentar

`/* Filnavn = analyser.c */`
 	
`#include <stdio.h>`
`#include "program.c"`
 	
`#define MAX 27`
 	
`int main()`
`{`
`    int i;`
`    char ind;`
`    long nyLinie = 0;`
`    long antalByte = 0;`
`    int hyppighed[MAX];`
 	
`    for (i = 0; i < MAX; i++)`
`        hyppighed[i] = 0;`
`    while ( (ind = getchar()) != EOF)`
`    {`
`        analyserBogstaver(ind, hyppighed);`
`        nyLinie += analyserLinieskift(ind);`
`        antalByte++;`
`    }`
`    udskrivResultat(nyLinie, antalByte, hyppighed);`
`}`
	Kan ikke compileres endnu, så læs videre.

Figur 5.6.1

Koden kan ikke compileres endnu, da vi ikke har skrevet de funktioner vi kalder. Men programmet starter med at inkludere program.c filen, der skal ligge i samme directory som mainprogrammet. I denne fil vil vi placere de tre funktioner der skal bruges for at få programmet til at virke. Det betyder, at denne fil ikke vil ændre sig mere.

Inde i main erklærer vi arrayet:

```
int hyppighed[27];
```

Det vil vi bruge til at gemme summen af bogstavet a i hyppighed[0], sumen af b'er vil vi gemme i hyppighed[1] o.s.v. Derefter går vi ind i en for løkke, der har til opgave at initiere hyppighedsarrayat, så der står nul på alle pladser. Hvis vi ikke gør det, vil der stå noget ubestemt i arrayet, og det kan vi ikke risikere.

Derefter går vi ind i en while løkke, som vi har set før, vi kalder analyserBogstaver funktionen, som ikke er skrevet endnu.

Der sker ikke mere ukendt i hovedprogrammet, nu skal vi bare lige have lavet de tre hjælpefunktioner, og gemme dem i filen program.c. Ved at arbejde sig frem på denne måde kan man bevare overblikket hele tiden. Gør en funktion færdig ad gangen, test den, og gå videre til den næste. Figur 5.6.2 viser den færdige program.c fil, med de tre hjælpefunktioner.

```
/* Filnavn = program.c */

void analyserBogstaver(char ind, int hyppighed[])
{
	if (ind >= 'a' && ind <= 'z')
		ind = ind - ('a' - 'A');
	if ((ind >= 'A' && ind <= 'Z') )
		hyppighed[ind - 'A']++;
}

long analyserLinieskift(char ind)
{
	if (ind == '\n')
		return 1;
	return 0;
}

void udskrivResultat(long nLinie, long aByte, int hyppighed[])
{
	int i;

	printf("%s\n", "Analysen viser at der blev fundet:");
	printf("%d%s\n", nLinie, " Linieskift");
	printf("%d%s\n", aByte, " Byte");
	printf("%s\n", "Absolut tegnfordeling");
	for (i = 'A'; i <= 'Z'; i++)
		printf("%3c", i);
	printf("\n");
	for (i = 'A'; i <= 'Z'; i++)
		printf("%3d", hyppighed[i - 'A']);
	printf("\n");
}
```

Figur 5.6.2

Funktionen `analyserBogstaver` starter med at konvertere indkommende tegn til

store bogstaver, det gør den efterfølgende if sætningen nemmere at konstruere. Derefter lægges der en til i det arrayelement der passer med bogstavet, og så er den analyse afsluttet for det tegn.

`analyserLinieSkift` funktionen burde ikke kræve kommentarer på nuværende tidspunkt. Det kan måske undre, at vi overhovedet opretter en funktion for så lidt. Men det at vi gør det, gør main programmet mere overskueligt.

Selve udskrivningsfunktionen er der heller ikke noget specielt nyt i. Vi har altså skrevet et lille analyse prorgram, ved at bygge det frem i små step ud fra en pseodokode.

Output fra programmet, hvis det bruges sig selv som input er:

```
$cat analyser.c | ./a.out
Analysen viser at der blev fundet:
25 Linieskift
408 Byte
Absolut tegnfordeling
```

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
22	4	6	13	23	5	9	12	33	0	2	17	5	28	7	9	0	11	8	15	4	3	1	3	13	0

5.6.1 Afrunding

Hvis du har arbejdet dig gennem bogen og udført en stor del af opgaverne, er du nu kommet rundt om alle væsentlige dele i programmeringssproget C. Det kan du bruge til at begynde at skrive en masse programmer, eller du kan udvidde din horisont ved at lære et af de 2 andre meget udbrdte programmeringssprog, C++ og/eller Java.

Måske vil du hellere more dig med sciptsprog som Javascript eller PHP, under alle omstændigheder vil du opdage, den grundviden du nu har fået om programmering, kan du bruge i mange sammenhænge.

Opgave 5.6.1 Udvid analyser programmet således at bogstaverne relative fordeling også udskrives. Hint: Relativ fordeling er den procentuelle fordeling af bogstaver.

Opgave 5.6.2 Udvid analyser programmet så det også kan tælle de danske bogstaver med.

Opgave 5.6.3 Udvid analyser programmet så det også kan tælle antal kommaer og punktummer.

Opgave 5.6.4 Udvid analyser programmet så det også kan tælle, og udskrive hypigheden at tallene 0 ... 9.

5.7 Opgaver

Til sidst i denne C del er her nogle opgaver der bruger elementer fra alle sprogets faciliteter.

Opgave 5.7.1 Udvid analyser programmet således at det der skal analyseres kommer fra en fil, hvis navn og sti angives i kommandolinien.

Opgave 5.7.2 Udvid analyser programmet således at brugeren bliver mødt med en menu når programmet starter. Fra menuen skal brugeren kunne vælge, indlæs en fil, alalyser for tegnhyppighed, analyser antal linieskift og forlad programmet.

Opgave 5.7.3 Udvid menuen og programmet fra opgave 5.5.2 med et en funktion der hvis den vælges, medfører at output skrives til en fil og ikke på skærmen.

Opgave 5.7.4 Skriv et program der kan indkode efter cæsarkoden, men med variabelt offset. Hint: Offset = 1 betyder a = b, b = c ... z = a. Offset = 2 betyder a = c, b = d ... z = b.

Opgave 5.7.5 Skriv et decifreringsprogram der kan desifrere en cæserkode med et hvilket som helst offset.

Opgave 5.7.6 Skriv et decifreringsporgram der automatisk kan finde en cæsarkodes offset, og derefter bryde koden. Hint: Ethvert sprog har sin egen tegnfordeling. Start med at indlæse en fil med tekst på det sprog der skal brydes, analyser tegnfordeingen, og sammenlign den med alle kombinationer af tegnfordeling der opstår ved alle forskellige offset. Der hvor offset fra prøvefilen ligger tættest på den kodede fil, er decifreringen udført.

Opgave 5.7.7 Skriv et databaseprogram, der skal indeholde personinformationer. For hver person skal følgende data registreres:

Køn
Fødselsdato
Fornavn
Efternavn
Adresse
Telefonnummer

Data skal kunne gemmes og hentes fra en fil, ved hjælp af en menu.

Opgave 5.7.8 Udvid databaseprogrammet med en funktion der beregner gennemsnitsalderes for alle i databasen.

Opgave 5.7.9 Udvid databasen fra opgave 5.5.8 med et datafelt for årsløn.

Opgave 5.7.10 Skriv et program der kan konvertere en database fra programmet 5.5.8 til en der kan læses af programmet 5.5.9

Appendix A Operatorer

A.1 Aretmetiske operatorer

Operator	Anvendelse	Beskrivelse
+	op1 + op2	Adder op1 og op2
-	op1 - op2	Subtraher op2 fra op1
*	op1 * op2	Multipliser op1 og op2
/	op1 / op2	Divider op1 med op2
%	op1 % op2	Beregn restværdien af divisionen op1/op2
+	+operand	Indikerer positiv værdi (Fortegn)
-	-operand	Indikerer negativ værdi (Fortegn)
++	op++	Adder 1 til op: Evaluer før addering
++	++op	Adder 1 til op: Evaluer efter addering
--	op--	Træk 1 fra op: Evaluer før addering
--	--op	Træk 1 fra op: Evaluer efter addering

A.2 Relations- og betingelsesoperatorer

<table>
<tr><th>Operator</th><th>Anvendelse</th><th>Returnerer sand hvis:</th></tr>
<tr><td>></td><td>op1 > op2</td><td>op1 er større end op2</td></tr>
<tr><td>>=</td><td>op1 >=op2</td><td>op1 er større end eller lig med op2</td></tr>
<tr><td><</td><td>op1 < op2</td><td>op1 er mindre end op2</td></tr>
<tr><td><=</td><td>op1 <= op2</td><td>op1 er mindre end eller lig med op2</td></tr>
<tr><td>==</td><td>op1 == op2</td><td>op1 er lig med op2</td></tr>
<tr><td>!=</td><td>op1 != op2</td><td>op1 er forskellig fra op2</td></tr>
<tr><td>&&</td><td>op1 && op2</td><td>op1 og op2 er sande</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>op1 eller op2 er sand</td></tr>
<tr><td>!</td><td>!op</td><td>op1 er false</td></tr>
</table>

A.3 Bitvise operatorer

Operator	Anvendelse	Operation
>>	op1 >> op2	Skift bits i op1 til højre op2 gange
<<	op1 << op2	Skift bits i op1 til venstre op2 gange
&	op1 & op2	AND binært
|	op1 | op2	OR binært
^	op1 ^ op2	XOR binært
~	~op1	Komplementært binært

A.4 Præcendens

Når et udtryk skal evalueres sker det altid fra venstre mod højre, operator for operator. Hvilken operation der skal udføres næste gang afhænger af operatorenes præsedens. Et klassisk eksempel er:

2 + 3 * 4

Er det lig med (2 + 3) * 4 eller 2 + (3 * 4)? Det afhænger af operatorenes præcedens. En operator med højere præcedens udføres før en operator med lavere præcedens. Vi ved fra vores skolelærdom, at gange går forud for plus og minus. Computeren har brug for også at have nogle regler for præcedens, disse er givet i tabel A.4.1. Operatorer på samme linie har samme præcedens, og udtryk bestående af dem udføres fra venstre mod højre. Operatorer har faldende præcedens, des længere de kommer ned i tabellen.

<table>
<tr><th>Operator præcedens</th></tr>
<tr><td>() [] -> .</td></tr>
<tr><td>! ~ ++ -- + - * & (type) sizeof</td></tr>
<tr><td>* / %</td></tr>
<tr><td>+ - (binære)</td></tr>
<tr><td><<   >></td></tr>
<tr><td>< <= > >=</td></tr>
<tr><td>!=   ==</td></tr>
<tr><td>& (Binær AND)</td></tr>
<tr><td>^   (Binær XOR)</td></tr>
<tr><td>| (Binær OR)</td></tr>
<tr><td>&& (Logisk AND)</td></tr>
<tr><td>|| (Logisk OR)</td></tr>
<tr><td>?:</td></tr>
<tr><td>+= -= /= *= %= &= ^= |= <<= =>> =</td></tr>
<tr><td>,</td></tr>
</table>

Tabel A.4.1

Appendix B Standardbiblioteker for C

B.1 <assert.h>

Kan bruges til at tilføje diagnostik.

Makroer	assert(expression)	Afslutter programmet straks med en kommentaer for diagnose

B.2 <ctype.h>

Indeholder funktioner til håndtering af tegn(ctype -> c typer -> char typer -> tegn typer) De følgende funktioner returnere en værdi forskellig fra nul hvis argumentet er sandt, ellers returneres nul. Eksempelvis vil: `islower('a')` returnere en værdi forskellig fra nul, mens `islower('A')` vil returnere nul.

```
int isalnum(int c)        isalpha(c) eller isdigit(c) er sand.
int isalpa(int c)         isupper(c) eller islower(c) er sand.
int iscntrl(int c)        Kontrotegn
int isdigit(int c)        Heltal
int isgraph(int c)        Tegn dog ikke mellemrum
int islower(int c)        Små bogstaver
int isprint(int c)        Tegn der kan skrives på skærmen
int ispunct(int c)        isprint dog undtaget bogstaver, tal, og mellemrum
int isspace(int c)        mellemrum, formfeed, newline, carrige return, tab
int isupper(int c)        Store bogstaver
int isxdigit(int c)       Hexadecimelt tal
```

B.3 <errno.h>

Indeholder makroer til administration af fejlkoder

Makroer	EDOM	Værdien af errno efter en funktionerne i math.h har fejlet.
	ERANGE	Værdien af errno efter en funktionerne i

		math.h har fejlet.
	errno	Indeholder fejlkode hvis en funktion i math.h fejler

B.4 <float.h>

Indeholder funktioner til manipulation af flydende (komma) tal.

Macros	FLT_ROUNDS	Afrundingsform ved float -1 Ubestemt 0 Mod nul 1 Til den nærmeste 2 Mod positiv uendelighed 3 Mod negativ uendelighed
	FLT_RADIX	Exponent representations radix.
	FLT_MANT_DIG	Antallet af base-FLT_RADIX tal i mantissen. Af en float
	DBL_MANT_DIG	Antallet af base-FLT_RADIX tal i mantissa af en double
	LDBL_MANT_DIG	Antallet af base-FLT_RADIX tal i mantissa af en long double
	FLT_DIG	Antallet af betydende cifre i type float (mindst 6).
	DBL_DIG	Antallet af betydende cifre i et tal af typen double (mindst10).
	LDBL_DIG	Antallet af betydende cifre i et tal af typen long double (mindst10).
	FLT_MIN_EXP	
	DBL_MIN_EXP	
	LDBL_MIN_EXP	
	FLT_MIN_10_EXP	
	DBL_MIN_10_EXP	
	LDBL_MIN_10_EXP	
	FLT_MAX_EXP	
	DBL_MAX_EXP	
	LDBL_MAX_EXP	
	FLT_MAX_10_EXP	
	DBL_MAX_10_EXP	
	LDBL_MAX_10_EXP	
	FLT_MAX	Max repræsentation af float (at least 1E+37).
	DBL_MAX	Max repræsentation af double (at least 1E+37).

Macros	FLT_ROUNDS	Afrundingsform ved float -1 Ubestemt 0 Mod nul 1 Til den nærmeste 2 Mod positiv uendelighed 3 Mod negativ uendelighed
	LDBL_MAX	Max repræsentation af long double (at least 1E+37).
	FLT_EPSILON	Forskellen mellem 1 og den nærmeste værdi større end en som kan repræsenteres med float.
	DBL_EPSILON	Forskellen mellem 1 og den nærmeste værdi større end en som kan repræsenteres med double.
	LDBL_EPSILON	Forskellen mellem 1 og den nærmeste værdi større end en som kan repræsenteres med long double.
	FLT_MIN	Mindste normaliserede tal af typen float
	DBL_MIN	Mindste normaliserede tal af typen double
	LDBL_MIN	Mindste normaliserede tal af typen long double

B.5 <limits.h>

Indeholder definitioner af grænseværdier.

Makroer	CHAR_BIT	Number of bits for smallest object that is not a bit-field (at least 8).
	SCHAR_MIN	Minimum value for an object of type signed char (-127 or a more negative number).
	SCHAR_MAX	Højeste værdi for objekter af typen signed char (mindst +127)
	UCHAR_MAX	Højeste værdi for objekter af typen unsigned char (mindst 255)
	CHAR_MIN	Mindste værdi af en char
	CHAR_MAX	Højeste værdi for objekter af typen char (mindst +127
	MB_LEN_MAX	Det højeste antal byte i en char, mindst 1.
	SHRT_MAX	Den højeste værdi for et objekt af typen short int, mindst +32767.
	SHRT_MIN	Mindsteværdien af short int.

	USHRT_MAX	Den højeste værdi for et objekt af typen unsigned short int, mindst 65535.
	INT_MAX	Den højeste værdi for et objekt af typen int, mindst +32767.
	INT_MIN	Mindste værdi for et opbjekt af typen int. (-32767 eller mindre)
	UINT_MAX	Den højeste værdi for et objekt af typen unsigned int, mindst 65535.
	LONG_MIN	Mindste værdi for et opbjekt af typen long int. (-2147483647 eller mindre)
	LONG_MAX	Den højeste værdi for et objekt af typen long int, mindst +2147483647)
	ULONG_MAX	Den højeste værdi for et objekt af typen long int, mindst 4294967295.

B.6 <math.h>

Indeholder matematiske funktioner.

double sin(double x)	Returnerer sin(x) (radianer)
double cos(double x)	Returnerer cos(x) (radianer)
double tan(double x)	Returnerer tan(x) (radianer)
double asin(double x)	Returnerer asin(x) (radianer)
double scos(double x)	Returnerer acos(x) (radianer)
double atan(double x)	Returnerer atan(x) (radianer)
double atan2(double y, double x)	Returnerer atan(x/y) (radianer)
double sinh(double x)	Returnerer sinh(x) (radianer)
double cosh(double x)	Returnerer cosh(x) (radianer)
double tanh(double x)	Returnerer tanh(x) (radianer)
double exp(doublex)	Returnerer e^x
double log(double x)	Returnerer den naturlige logeritmen for x > 0
double log10(double x)	Returnerer base10 logeritmen for x > 0
double pow(double x, double y)	Returnerer x^y
double sqrt(double x)	$\sqrt{x}$ for x > 0
double ceil(double x)	Mindste heltal der ikke er mindre end x
double floor(double x)	Største heltal der ikke er større end x
double fabs(double x)	Returnerer den absolutte værdi af x
double ldexp(double x, int n)	$x*2^n$
double frexp(double x, int *exp)	Deler x i en normaliseret fraktion.
double modf(double x, double *ip)	Deler x i en brøk
double fmod(double x, double y)	Returnerer resten af x/y

B.7 <stddef.h>

Indeholder definition af meget brugte specialtyper.

Typer	ptrdiff_t	Den signed type der fremkommer ved at trække to pointere fra hindanden
	size_t	Den unsigned type der er resultat af en sizeof operation.
	wchar_t	
Makroer	NULL	Normalt (void*)0 eller (char*)0.
	size_t offsetof(type, member)	Afsrtanden i byte fra structurdefenitionen, til det første datamedlem.

B.8 <stdio.h>

Indeholder funktioner, makroer og typer til håndtering af input og output.

Typer	size_t	Heltals type der returneres af sizeof.
	FILE	Type for stream
	fpos_t	Typisk unsigned long.
Makroer	NULL	NULL pointeren
	_IOFBF	Et argument til setvbuf().
	_IOLBF	Et argument til setvbuf().
	_IONBF	Et argument til setvbuf().
	BUFSIZ	Bufferstørrelsen målt i byte.
	EOF	En negativ heltalskonstant for at vise End Of File.
	FOPEN_MAX	Det højeste tilladelge antal filer

		der kan være åben på en gang.
	FILENAME_MAX	Den største længde af strengen der skal indeholde filnavn.
	L_tmpnam	Den største længde af strengen der skal indeholde filnavn, genereres af tmpnam.
	SEEK_CUR	Offset fra nuværende filposition.
	SEEK_END	Offset fra nuværende EOF
	SEEK_SET	Offset fra nuværende filposition.
	TMP_MAX	Antallet af unikke filnavne genereret af tmpnam funktionen.
	stderr	Standard fejl (error) stream.
	stdin	Standard input stream.
	stdout	Standard output stream

Funktioner	
void clearerr(FILE *stream)	Sletter EOF og evt. fejlmarkeringer i den strø, strem peger på.
`int getc(FILE *stream)`	Henter det næste tegn fra stream
`int getchar(void)`	Fungerer som getc(stdin)
`char *gets(char *s)`	Henter det næste tegn fra stdin og placerer det i den buffer s peger på. Stopper ved newline eller EOF. '\0' indsættes sidst i det s peger på.
`int fclose(FILE *stream)`	Returnerer EOF ved fejl, ellers nul.
`int ferror(FILE *stream)`	Tester om fejlindikatoren er sat i stream,. Returnerer forskellig fra nul hvis den er sat, ellers nul.
int feof(FILE *stream)	Tester EOF mærket for det stream peger på. Returnerer forskellig fra nul hvis den er der, ellers nul.
`int fflush(FILE *stream)`	Returnerer EOF hvis der opstår en skrive fejl, ellers returneres nul.
`int fgetc(FILE *stream)`	Henter det næste tegn fra stream
`int fgetpos(FILE *stream, fpos *ptr)`	Sætter filpointerens placering i filen.
`char *fgets(char *s, int n, FILE *stream)`	Læser højst size - 1 tegn fra den strøm stream peger på. \0 indsættes til sidst.
`FILE *fopen(const char *filnavn, const char *mode)`	Åbner en fil angivet ved filnavn, i den tilstand der er angivet ved mode. Mode kan antage værdierne: "r" Åbn tekstfil som read only (kun læsning tilladt)

Funktioner	
	”w” Åbn tekstfil for skrivning og læsning. Overskriv evt. eksisterende fil. ”a” Åbn eller opret fil for skrivning (append) sidst i filen ”r+" Åbn tekstfil for opdatering (læs og skriv) ”w+” Åbn tekstfil for skrivning og læsning. Overskriv evt. eksisterende fil. ”a+” Åbn eller opret fil for skrivning (append) sidst i filen ”b” Binær fil Flere værdier kan kombineres, eks. open(”minfil”, ”wb”) Åbner en binær fil for skrivning. Returnerer en fil pointer hvis det lykkedes at åbne filen, elleres returneres NULL.
`int fprint(FILE *stream, const char *format, ...)`	Skriver til stream med den angivne formattering. Returnerer antal skrevne tegn hvis det går godt, negativt tal hvis det går skidt.
`int fputc(int c, FILE *stream)`	Tager tegnet repræsenteret ved c, kaster det til en char og skriver det til det stream peger på.
`int fputs(const char *s, FILE *stream)`	Skriver det s peger på til det stream peger på, uden \0.
`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`	Læser nobj gange af størrelsen size fra det stream peger på og placerer det i det ptr peger på. Returnerer antallet af læste tegn, eller nul hvis der opstod en fejl.
`FILE *freopen(const char *filnavn, const char *mode, FILE *stream)`	Returnerer stream hvis det lykkedes at åbne, ellers returneres NULL.
`int fscanf(FILE *stream, const char *format, ...)`	Indlæser formateret
`int fseek(FILE *stream, long offset, int origon)`	Placerer filpointeret til origon + offset.
`int fsetpos(FILE *stream, const fpos_t *ptr)`	Sæt filpointerens placering
`long ftell(FILE *stream)`	Returnerer nuværende filpointerposition
`sizeT fwrite(const void *ptr, size_t size, size_T nobj)`	Skriver nobj dataelementer size gange til det stream peger på.
void perror(const char *s)	Udskriver til stderror. Først udskrives det s peger på, derefter et kolon og et mellmerum . Derefter følger fejlmeddelelse.
`int putc(int c, FILE *stream)`	Skriver det s peger på til det stream peger på, uden \0.
`int putchar(int c)`	Det samme som putc(c, stdout)
`int puts(const char *s)`	Skriver det s peger på til stdout.
`int remove(const char *filename)`	Returnerer nul hvis det lykkes, ellers en værdii forskellig fra nul.
`Int rename (const char *gammeltNavn, const char *nytNavn)`	Omdøber gammeltNavn til nytNavn. Returnerer nul hvis det lykkes, ellers en værdi forskellig fra nul.
`void rewind(FILE *stream)`	Sætter filpointeren til start fil.
`int scanf(const char *format, ...)`	Indlæser formateret
`void setbuf(FILE *stream, char *buf)`	Sætter buffertype til stream. Buf kan pege på en af 3 typer:

Funktioner	
	_IONBF Ingen buffer _IOLBF Liniebuffer _IOFBF Komplet buffer
`Int sprintf(char *s, const char *format, ...)`	Samme funktion som printf, men output skrives til s og afsluttes med '\0'.
`int sscanf(s,...)`	Indlæser formateret
`Char *tmpnam(char s[L_tmpnam])`	`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`
`FILE *tmpfile(void)`	Returnerer en pointer til stream hvis det lykkes ellers NULL.
`int ungetc(int c, FILE *stream)`	
`vprintf(const *format, va_list arg)`	Indlæser formateret
`vprintf(FILE *stream, const *format, va_list arg)`	Indlæser formateret
`vprintf(char s, , const *format, va_list arg)`	Indlæser formateret

B.9 <stdlib.h>

Indeholder typedefinitioner, makroer og funktioner til udbredt brug

Typer	size_t	
	wchar_t	
	div_t Result type for the div() function. typedef struct { int quot; int rem; } div_t;	
	ldiv_t Result type for the ldiv() function. typedef struct { long quot; long rem; } ldiv_t;	
Makroer	NULL	En generel null pointer.
	EXIT_FAILURE	Argument for exit() eller returværdi fra unormal eksekvering.
	EXIT_SUCCESS	Argument for exit() eller returværdi fra normal eksekvering.
	RAND_MAX	Den højeste værdi der returneres af rand(), mindst 32767.
	MB_CUR_MAX	Positivt heltals udtryk der returnerer den højeste værdi for extended tegnsæt..

Funktioner	
`void abort(void)`	Stopper al programeksekvering straks
`int abs(int n)`	Returnerer den absolutte værdi af n
`int atexit(void (*fcn) (void))`	Registrerer funktion der skal kaldes nor programmet terminerer normalt.

Funktioner	
double atof(const char *S)	Konverterer s til double
int atoi(const *s)	Konverterer s til int
long atol(const char *s)	Konverterer s til long
void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp) (const void *keyval, const void *datum))	Binær søgning
void *calloc(size_t nobj, size_t size)	Returnerer en pointer til nobj
div_t div(int num, int denom)	
void exit(int status)	Stopper programeksekvering, og returnerer status til operativsystemet.
void free(void *p)	Frigiver plads i hukommelsen
char *getenv(const char *navn)	Henter den streng fra operativsystemet der er associeret til strengen navn
long labs(long n)	Returnerer den absolutte værdi af n
ldiv_t ldiv(long num, long denom)	
void *malloc(size_t size)	Returnerer en pointer til size byte i hukommelsen, NULL hvis der ikke er mere plads.
void qsort(void **base, size_t size, int (**cmp) (const void *, const void *))	Sorterer
int rand(void)	Returnere et pseudotilfældigt tal i området 0 til RAND_MAX
void realloc(void *p, size_t size)	Ændrer størrelsen på det som *p peger på.
void srand(unsigned int seed)	Benytter seed til at generere nyt pseudotilfældigt talrække
long strtol(const char *S, char **endp, int base)	
double strtod(const char *S, char **endp)	
unsigned long strtoul(const char *s, char **endp, int base)	
int system(const char *s)	Overfører s til operativsystemet.

B.10 <string.h>

Indeholder funktioner til manipulationer af strenge (tegnarrays)

`void *memchr(const char *cs, char c, size_t n)`	Returnerer en pointer til det første sted i cs hvor tegnet c befinder sig. Returnerer NULL hvis c ikke findes inden for de første n tegn
`int memcmp(const char *cs, char  c, size_t n)`	Sammenligner de første n tegn i cd med ct. Returnerer <0 hvis cs < ct, 0 hvis cs = ct og > 0 hvis cs > ct
`void *memcpy(char *S, const char *ct, size_t n)`	Kopierer n tegn fra ct til s og returnerer s.
`void *memmove(char *s, const char *ct, size_t  n)`	Som memcpy, men fungerer også hvis der er sammenfald i dataarealerne.
`void *memset(char *s, char c, size_t n)`	Indsætter c ind i de første n tegn af s, returnerer s.
`char *strcat(char *s, const char *ct)`	Tilføjer strengen ct til enden af s, tilføjer '\0' og returnerer s.
`int *strcmp(const char *cs, const char *ct)`	Sammenligner de to strenge. Returnerer <0 hvis cs < ct, 0 hvis cs = ct og > 0 hvis cs > ct
`size_t strcspn(const char *cs, const char *ct)`	
`char  *strerror(isize_t n)`	
`size_t strspn(const char *cs, const char *ct)`	
`char *strncat(char *s, const char *ct, size_t n)`	Tilføjer højst n tegn fra strengen ct til enden sf s, tilføjer om nødvendigt '\0' og returnerer s.
`int *strncmp(const char *cs, const char *ct, size_t n)`	Sammenligner højst n tegn i de to strenge. Returnerer <0 hvis cs < ct, 0 hvis cs = ct og > 0 hvis cs > ct
`char *strncpy(char *s, const char *ct, size_t n)`	Kopierer højst n tegn fra ct til streng s, returnerer s. Tilføjer '\0' hvis ct har færre end n tegn.
`char  *strpbrk(const char *cs, const char *ct)`	Returnerer en pointer til det første sted i cs hvor der er et tegn som også er i cs. Ellers returneres NULL

size_t strlen(const char *cs)	Returnerer længden af cs
char strstr(const char *cs, const char ct)	Returnerer en pointer til det første sted i cs hvor ct findes. Hvis ct ikke findes i cs returneres NULL
char *strrchr(const char *cs, char c)	Returnerer en pointer til det sidste sted i cs hvor c findesd. NULL hvis c ikke findes i cs.
char *strchr(const char *cs, char c)	Returnerer en pointer til det første sted i cs hvor c findesd. NULL hvis c ikke findes i cs.
char *strtok(char *s, const char *ct)	Søger efter efter token i ct.

B.11 <time.h>

I time.h er der funktioner til at administrere dato og tid. Til hjælp er disse variabler defineret:

int tm_sec sekunder
int tm_min minutter
int tm_hour timer
int tmnday dag i måneden
int tm_mon måned siden januar
int tm_year år siden 1900
int tm_wday dage siden søndag
int tm_yday dage siden 1/1
int tm_isdst sommertidsflag

char *asctime(const struct tm *tp)	Konverterer tiden fra struct'en *tp til en streng på formen: Man Jul 3 08:23:07 2000\n\0
char *ctime(const time_t *tp)	Konverterer kalendertiden til lokal tid
clock_t clock(void)	Returnerer proccestiden siden programmet startede, eller -1 hvis den ikke findes. Clock()/CLK_TCK er tiden i sekunder
double difftime(time_t tid2, time_t tid1)	Returnere differensen i tid.
struct tm *gmtime(const time_t *tp)	Konverterer kalendertiden til UTC. Returnerer NULL hvis den ikke er tilgængelig
time_t time(time_t *tp)	Returnerer den aktuelle tid, eller -1 hvis den ikke findes.

`struct tm *localtime(const time_t *tp)`	Konverterer kalendertiden til lokal tid
`size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)`	Formaterer dato og tids informationer fra *tp til s som beskrevet i *fm. *fmt er magen til formattering for printf(). %a forkertet ugedag navn %A fuldt ugedag navn %b forkortet månedsnavn %B fuld månedsnavn %c lokal tidformat %d månedsdag %H time (00 .. 23) %I time (00 .. 12) %J Juliansk dag %m måned %M minut &pAM/PM %S sekund %U ugenr %w ugedag %W ugenr %x lokal datoformat %X lokal tidsformat %y år (0 .. 99) %Y år inkl århundrede %Z Tidszone navn %% %

Stikordsregister

0.....54
apostrof.....36
argumenter.....75
argumentliste.....9
arkivkort.....124, 143
array......83, 89, 98f., 109, 111, 124, 131, 163
array af cha.....89
ASCII tabel.....51
binære.....57
black box.....71, 75
blok.....42, 47
Blokke.....42, 45
blokke.....18, 62
blokmærkene.....44
break.....70
case sensitive.....6
cast.....51, 72
cat.....32
char.....14, 47f.
data.....114
datatype.....13, 38
datatypen.....35
Datatyper.....47
datatyper.....47
dataareal.....132
default.....70
define.....85
directory.....152
dobbeltkædede liste.....140
dobbeltkædet.....136, 138
double.....47
End Of File.....33
enum.....106f.
Enumaration.....106
EOF.....33
erklære.....35
erklærer.....18, 48
error.....7
falsk.....62
fil.....114, 116
filen.....115, 143
filer.....118
float.....38, 47, 49
formateringstegn.....56
formattering.....53
formatteringsstreng.....13
funktion.....71, 73
global variabel.....158
Globale variabler.....162
gåseøjne.....10
headerfilen.....127, 131
headerfiler.....127, 152
hexadecimale.....118
hukommelsesadresse.....98
hyppighed.....163
håndværk.....77
if.....62, 68
if ... else.....17
Include filer.....152
indrynkning.....43
indtastning.....69
initiere.....18, 35
inkludefiler.....154
input.....89
int.....47f.
kastet.....52
kildeteksten.....8
KIS.....67
kodebryder.....163

kommentar...8
kompilerdirektiver..............................156
kompilere...6
kædede liste..131
kædet liste..133
liste...134
listen...137
long..47f.
løkke..22f.
magiske..85
magiske tal...85
main()..9
makro...86
malloc..134
modolus.......................................59, 118
neste...64
nesting..47
nestning..67
next..136
nextpointere..133
nulpointer...136
nultegnet......................................89, 103
ooo...54
operander...57
operatorer...57
output...53
oversætte...7
parametre...71
pege på..131
pegepind...............................99, 132, 134
pegepinde...83
piping...32
pointer......................................131, 136
pointerarray..112
pointere.....83, 98f., 104, 118, 132, 134f., 138
pointervariabel.....................................98
pointervariablen...................................99
pointervariablerne..............................104
postfix..57
prefix...57
prevpointerne......................................136
prototype...73f.
præcedens..60
pseudokode....................................71, 80
psoedokoden.......................................163
rekursion..147
returtypen...101
rækkevidde..................................45, 158
sand..95
sandt...62
scope..45
short...47f.
signed...48
sizeof..134
slutblok...45
sluttuborg..42
sortering..95
sourcecode..7
standard biblioteker............................152
standard input.......................................32
startblokmærke..................................43f.
startpointer...144
starttuborg..42f.
static...162
streng..10
strenge...111
struct........................121f., 124, 131, 143
structurer..3
struktur...121
switch..68
t..54
tekstfil...6
tekststreng...14
tuborg..9
type..49
Typekonvertering..................................49
unsigned..48

usand..95
v...54
variabel...121
variablen.............................35, 48, 52, 99
Variabler..12
variabler..................................49, 83, 158
void...72
voidpointer..98
warning...7, 89
xhh..54
?...67
*/..8
/*..8
\..54
\?..54
\"..54
\\...54
\´..54
\a..54
\b..54
\f...54
\n..54
\r...54
%...55, 59
+=..59