Linux - Friheden til at programmere i C

Programmering med GNU/Linux

Version 2.0.20041104 - 2020-12-31

Donald J. Axel
SSLUG, Skåne Sjælland Linux User Group

Linux - Friheden til at programmere i CProgrammering med GNU/LinuxVersion 2.0.20041104 - 2020-12-31

af Donald J. Axel

Ophavsret © 2000-2005 Forfatteren har ophavsret til bogen, men udgiver den under "Åben dokumentlicens (ÅDL) - version 1.0".

Forskellige programmeringsteknikker illustreret med eksempler, mest for C programmører med lidt erfaring, men begyndere kan bladre om i Appendix A og se, om det er noget for dem.

Indholdsfortegnelse

Forord vi

1. Oversigt over bogen vi
2. Om forfatteren og bogens historie..vi
3. Linux-bøgerne.. vii
4. Ophavsret viii
5. Vi siger tak for hjælpen.. viii
6. Typografi ix

1. Repræsentation og modeller **1**

1.1. Et professionelt værksted..1
1.1.1. Hvad er en debugger?..3
1.1.2. Biblioteker - et eksempel..4
1.1.3. Hvordan kommer man igang? 4
1.2. Repræsentation af data..5
1.3. Repræsentation af tal..5
1.4. Simulering - Modeller..9
1.5. Planlægning af et program 10

2. Funktioner, modularitet og relevante teknikker..**13**

2.1. ANSI prototyper og modularitet 13
2.1.1. Et brohoved..14
2.1.2. Returværdi fra en funktion 14
2.2. Parameterlister og datatyper..19
2.2.1. Datatyper og beregninger 19
2.2.2. Prompt og input..21
2.3. ANSI prototyper og modularisering 24
2.3.1. Kursomregning..24
2.4. Filter programmer 30
2.4.1. Tegn og talværdier..31
2.4.2. Et typisk filter 33
2.4.3. Oversætter - teknik, forstadium..36
2.4.4. Summering af tal i en fil. 40
2.5. Analyse af input 41
2.5.1. En tag - parser..42
2.5.2. Tjek balancering af tags..46
2.5.3. Any-tag 48
2.5.4. Skelnen mellem tagtyper 51
2.5.5. Tilstandsvariabel..53
2.6. Fejl og håndteringen af dem..55
2.6.1. Hvilke slags fejl er interessante..55
2.7. Flere små programmer og øvelsesforslag..56
2.7.1. Tal formatering ved udskrift med printf. 56
2.7.2. Word count - med tak til Kernighan & Ritchie..57
2.8. Øvelser 59
2.8.1. Forslag til beregningsøvelser..59

3. Sammensatte datatyper og hvad man kan med dem i C **61**
3.1. Sammensætning af flere oplysninger til en enhed 61
3.1.1. En *struct* 61
3.1.2. Interface til datatype 63
3.1.3. Tabel, array, liste, sekvens, bunke 65
3.1.4. Operationer på datatypen 72
3.2. Konkrete og abstrakte datatyper 73
3.3. FILE: En abstrakt type 77
4. Parsning - hvordan oversættes et C program **80**
4.1. En declaration parser 80
4.2. En expression parser 81
4.3. En komplet compiler 88
4.3.1. uC - en mikroskopisk C compiler 92
4.4. Tilstandsmaskiner 96
A. Crash course i C sproget **99**
A.1. C i få ord 99
A.1.1. Data typer 101
A.1.2. Operatorer 104
A.1.3. Flow kontrol 109
A.1.4. Modularitet 111
A.2. Lager klassifikation, storage classes 113
A.2.1. Externe variable, funktioner og konstanter 114
A.2.2. Storage specifications 116
A.2.3. Konstanter og memory protection 118
A.2.4. Storage specifikation og scope 119
A.2.5. Kontrolvariable for løkker eller loops 122
A.2.6. Flygtige ukontrollerede variable 123
A.2.7. Oversigt over variabel - anvendelse 123
A.3. GNU programmerings værktøjer 124
A.3.1. GNU C Compiler options 125
A.3.2. objdump eksempler 125
B. Revisionshistorie for bogen **127**
C. Nyt i C99 ..... **129**
C.1. Uddrag af nyheder i ANSI revisionen fra 1999 129
C.2. Liste i tilfældig rækkefølge 130
C.3. Nyt i C93 (AM1) 131
Stikordsregister **132**

Figurliste

1. Donald Axel vii
2. ÅDL viii
1-1. glade, en GTK+ GUI generator. 1
1-2. glade, projekt vindue. 2
2-1. Eksempel på fejlhåndtering, informativ besked 56
3-1. Binært træ 73

Forord

1. Oversigt over bogen

- Kapitel 1: Repræsentation og modeller

  Beskriver, hvordan data bliver repræsenteret i en computer. Det er nøglen til forståelse af, hvordan man kan løse forskellige typer programmeringsopgaver, beregninger, simuleringer m.v.

- Kapitel 2: Funktioner, modularitet og relevante teknikker.

  ANSI prototyper og modularisering, datatyper og parameterlister, modularisering af en beregning.

  Rekursive funktioner.

- Kapitel 3: Sammensatte datatyper og hvad man kan med dem i C.
- Kapitel 4: Parsning - hvordan oversættes et C program.

  Hvordan man parser et beregningsudtryk. Hvordan man skriver en rekursiv descent parser. Som eksempler en kalkulator og en minimalistisk C-compiler. Komplet kildetekst medfølger.

- Appendiks A Kort introduktion af C sprogets elementer.

Bogen her har til hensigt at vise eksempler på forskellige grundlæggende teknikker.

Den henvender sig til programmører, som (måske) kender de første eksempler fra Kernighan & Ritchie's bog "The C Programming Language", men som gerne vil se flere små eksempler, der illustrerer de forskellige features i C sproget.

Den ihærdige begynder med en del teknisk flair og lidt erfaring kan dog også få glæde af bogen ved først at læse Appendix A, som er en kort og koncis introduktion til sprogets "mekanik".

Det er tanken at følge denne bog op med flere, som hver især behandler en bestemt type applikation.

- Donald Axel -

2. Om forfatteren og bogens historie

Figur 1. Donald Axel

Donald Axel donald_j_axel@get2net.dk (mailto:donald_j_axel@get2net.dk). Musiker, ex-radiojournalist, gymnasielærer, kursusinstruktør i C programmering, C++, Unix systemadministration m.v., edb-konsulent, har blandt andet været udstationeret som gæsteprogrammør hos Motorola med private GSM systemer for alarmtjenester etc.

Har et lille beskedent hus, der skal gøres noget ved. Men jeg spiller klaver midt i det hele og synes det er dejligt.

3. Linux-bøgerne

Bogen er en del af en serie, som kan findes på http://www.linuxbog.dk/

- *Linux – Friheden til at vælge installation* – Om at installere Linux.
- *Linux – Friheden til at lære Unix* – Om hvordan man bruger Linux' (og Unix') kommandolinjeværktøjer.
- *Linux – Friheden til at vælge grafisk brugergrænseflade* – Om alle de grafiske brugergrænseflader, der findes til Linux.
- *Linux – Friheden til at vælge programmer* – Om de programmer du kan få til Linux.
- *Linux – Friheden til systemadministration* – Om at administre sit eget linuxsystem.
- *Linux – Friheden til at programmere* – Programmering på Linux
- *Linux – Friheden til at programmere i C* – Om at programmere i sproget "C".
- *Linux – Friheden til at programmere i Java* – Om at programmere i sproget "Java".
- *Linux – Friheden til sikkerhed på internettet* – Om at sikre dit Linuxsystem mod indbrud fra internettet.

- *Linux – Friheden til egen webserver* – Om at sætte en webserver med databaser, CGI-programmer og andet godt op.
- *Linux – Friheden til at skrive dokumentation* – Om at skrive dokumentation (og andet) i SGML/DocBook, LaTeX eller andre formater.
- *Linux – Friheden til at vælge kontorprogrammer* – Kontorfunktioner på et Linux/KDE/OpenOffice.org-system.
- *Linux – Friheden til at vælge IT-løsning* – Om muligheder, fordele og ulemper ved at bruge Linux i sin IT-løsning.
- *Linux – Friheden til at vælge OpenOffice.org* – Om at bruge OpenOffice.org, både på Linux og på andre styresystemer.
- *Linux – Friheden til at vælge digital signatur* – Digital signatur på Linux.

4. Ophavsret

Denne bog er skrevet af Linux-brugere til Linux-brugere. Store dele af bogen er skrevet eller redigeret af enkelte forfattere, hvilket er nævnt i revisions-historien til bogen.

Bogen kan findes i opdateret form på http://www.linuxbog.dk/, mens prøve-udgaver kan findes på http://cvs.linuxbog.dk/.

Figur 2. ÅDL

Bogen er udgivet under "Åben dokumentlicens (ÅDL) – version 1.0" som kan læses på http://www.linuxbog.dk/licens.html. Du har bl.a. herved frit lov til at kopiere dette værk uændret på ethvert medium.

Kommentarer, ris og ros og specielt fejl og mangler bedes sendt til linuxbog@sslug.dk (mailto:linuxbog@sslug.dk), men er du medlem af SSLUG kan du i stedet for med fordel skrive til sslug-bog@sslug.dk (mailto:sslug-bog@sslug.dk).

5. Vi siger tak for hjælpen

Vi har haft stor glæde af folks støtte, rettelser og forslag til forbedringer - bliv ved med dette. Specielt vil vi nævne:

- Peter Toft har hjulpet med SGML koden og rettet et par fejl.
- Jacob Sparre Andersen har fundet nye stavemåder og rettet sprogfejl.
- Forfatteren takker Brian Eberhardt, direktør for SuperUsers a/s, for hans inspiration til bl.a. introduktionen til de grundlæggende elementer i C sproget i appendix A.

Du kan i Appendiks B finde en liste over alle de revisioner, som bogen har været igennem.

Hvis du har ord du ikke forstår, så kan http://www.whatis.com være interessant. Her kan du slå mange computerord op, dog kun på engelsk. I øvrigt kan bogens stikordsregister være interessant.

6. Typografi

Vi vil afslutte indledningen med at nævne den anvendte typografi.

- Navne på filer og kataloger skrevet som `foo.bar`
- Kommandoer, du udfører ved at taste, skrives som **help**
- Der er flere steder i bogen, hvor vi viser, hvad brugeren (som vi kalder "tyge") taster, og hvad Linux svarer. Det vil se ud som:

  ```
  [tyge@hven ~]$ Det her taster brugeren
  Det her svarer Linux
  ```
- Der er tilsvarende flere steder i bogen hvor vi viser hvad systemadministratoren (root) taster, og hvad Linux svarer. Det vil se ud som:

  ```
  hven# Dette taster systemadministratoren
  Dette svarer Linux.
  ```
  Det vigtige her er at kommandofortolkeren bruger nummertegnet (#) til at markere at man har systemadministratorrettigheder.

Kapitel 1. Repræsentation og modeller

1.1. Et professionelt værksted

Hvis du har en maskine med Linux i en af de større distributioner (fx. Red Hat eller Debian) har du samtidig en samling af de mest avancerede værktøjer til fremstilling af software.

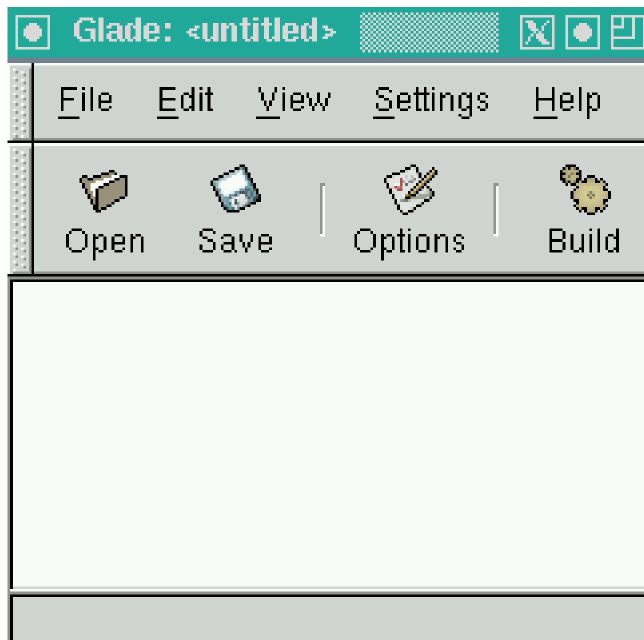
Det er ikke blot en "sandkasse", du sidder med, men et professionelt udviklingsmiljø. Der er optimerende oversætter, standardiserede biblioteker, supplerende biblioteker med et væld af funktionalitet, debugger, source management med alt, hvad dertil hører, og oven i det er Linux jo et professionelt afviklingsmiljø med automatisk jobstart, kommandofortolkere og andre fortolkede sprog i mange varianter, serverfaciliteter, samt ikke mindst dørvogterfaciliteter (Firewall værktøjer eller TCP-IP filtrering).

Det avancerede i denne her sammenhæng ligger ikke i et smart peg - og - klik interface til en programgenerator. Just for the record: En programgenerator er et udmærket værktøj i visse sammenhæng. Med sådan et værktøj kan du kan vælge mellem et begrænset antal på forhånd programmerede moduler ved hjælp af en mus, klik og vupti, så har du et resultat bestående af objekter, der er sat sammen (nogen kalder peg og klik - programmering for objektorienteret programmering, men det er en fordrejning af begreber).

Et sidespring: På Linux findes mindst én klik-og-peg generator, nemlig "glade", en GTK+ User Interface Builder, skrevet af Damon Chaplin, http://glade.pn.org. Kan anbefales.

Figur 1-1. glade, en GTK+ GUI generator.

Figur 1-2. glade, projekt vindue.



Nej, det avancerede består i, at du kan styre genereringen af programkode fuldstændig professionelt. Hvis du for eksempel ønsker at skrive en ny og bedre database server, så værsgo', gå i gang. Den nødvendige dokumentation er til rådighed, og der findes desuden lærebøger som forklarer om low-level ting og high-performance problematikker på Linux.

Oven i alt dette får du den fordel, at dine Linux programmer i de fleste tilfælde kan anvendes uden ændringer på andre Linux-systemer, inclusive 64-bit systemer, og med få ændringer på mange andre Unix systemer som for eksempel Solaris, AIX, HP-UX.

Mere overraskende er det måske, at man også kan køre Linux-udviklede programmer på Microsoft NT ved hjælp af Cygwin systemet fra Cygnus.

Det er en væsentlig del af filosofien i GNU og C - sproget, at man skal kunne genbruge sin kode, hvis hardwaren skal udskiftes; og det skal den før eller senere! Hardware forgår, brainware består. Sådan da.

Udover C-oversætteren har du med Linux adgang til C++ og mange andre sprog.

1.1.1. Hvad er en debugger?

Debuggeren er et værktøj, som kan vise, linje for linje, hvad der sker i et program. Selv om det hedder en debugger, så er det er ikke så godt at bruge den til at fjerne fejl [1].

Hvis et program er så fejlbehæftet, at man overvejer at anvende en debugger, så bør man begynde forfra og reimplementere sine programmer med de nyvundne erfaringer. Ok, der er undtagelser, hvor man kan lokalisere en vanskelig, periodisk fejl ved at anvende en debugger med omtanke, men det er ikke noget, som man skal gøre til en fast vane!

1.1.2. Biblioteker - et eksempel

Det vigtigste er imidlertid, at der med de almindelige Linux-distributioner er et stort arsenal af frie biblioteker til database, netværk, grafik etc.

Som et eksempel kan det fremhæves, at der findes frie biblioteker til netværksprogrammering. For eksempel *Remote Procedure Call*, både til klient- og tjenersiden. Det er en teknik, som kan benyttes til at udføre en routine på en anden computer (som selvfølgelig skal være forberedt til dette!) Dette er værd at understrege, idet den meget udbredte PC-software har indført nogle økonomisk betingede useriøse skel mellem server og klient programmer.

Desuden har du adgang til mange gode eksempler på, hvordan i hvert fald nogle af bibliotekerne anvendes. Kort sagt, hvis du går i gang, kan du satse på at nå et professionelt niveau på de områder, som du udvælger.

1.1.3. Hvordan kommer man igang?

Hvad gør man så for at komme igang?

Hvis der skulle være et mirakel-ord, som er nøgle til forståelse af computerens magi, så er det ordet DATA-REPRÆSENTATION.

Godt nok følger computerens elektriske signaler de fysiske love, men computeren er et instrument til at manipulere store mængder af impulser ud fra nogle få regler valgt af programmøren. Med andre ord, vi kan tillægge elektriske signaler en betydning. Et enkelt elektrisk signal kaldes oftest en bit, binary digit. Otte af dem kaldes en byte eller, mere officielt, en octet.

En enkelt bit kan være en besked om at åbne sluserne for at undgå en oversvømmelse. Eller en prik på en billedskærm. Eller forskellen mellem et punktum og et komma (i en dårlig skrifttype! Anstrengende for øjnene!)

Som regel bruger man flere bits til vigtige beskeder for at sikre, at man nu også har forstået hinanden rigtigt. Det kaldes redundans.

Bogstaver repræsenteres som regel ved en byte, men ved anvendelse af Unicode bruges 16 bit eller 2 bytes. Der er også systemer, som repræsenterer bogstaver ved hjælp af variabelt antal bytes. Kig for eksempel på en HTML-side.

1.2. Repræsentation af data

Hvis du ved en masse om computere, så vil du nok finde dette afsnit overflødigt. Spring straks videre!

En computer kan bruges til tekstbehandling, styring af produktionsmaskineri, til teleudstyr, og, selvfølgelig, til den "klassiske" lommeregner. Hvor kommer denne fleksibilitet fra? Fra programmøren, eller mere korrekt: De forskellige applikationer af computerteknikken beror på muligheden for at repræsentere information af mange forskellige typer ved hjælp af elektriske spændinger.

Konstruktøren af en harddisk bestemmer f.x., at når man sætter strøm på ledning 77, så er det en besked til harddisken om, at den skal begynde en read-operation.

De, der konstruerer computeren, bestemmer, hvad de forskellige elektriske signaler skal få de forskellige dele af maskineriet til at udføre. Når han én gang har bestemt, at ledning 77 er read-request, og maskinen er bygget efter hans anvisninger, så er vi bundet af denne mening med signal 77, men det kan altså laves om - næste gang, vi konstruerer et harddisk interface.

Den ene dag repræsenterer bit nummer 1000017 en kerne i en tomat, næste sekund måske et punktum på en skærm, og næste dag kunne den være en del af en ordre til en harddisk.

Hvis du vil i dybden med forståelse af computerens virkemåde og muligheder, kan du læse for eksempel Joseph Weizenbaum’s "Computer Power and Human Reason".

Datarepræsentation handler om teknikker til at repræsentere forskellige typer af objekter. For programmøren er det vigtigere at spørge, hvilke informationer, det egentlig er interessant at repræsentere. Det kan computeren ikke finde ud af uden kyndig vejledning fra programmører!

Hvordan vælger vi at repræsentere en tomat, hvad vælger vi at se? Er det prisen, der interesserer os? Vægten? Udseendet? Surhed? Arvelige egenskaber? Vi vælger ud fra en idé om, hvad vi vil med objektet.

1.3. Repræsentation af tal

Lad os se på, hvordan computeren repræsenterer tal. Det er vel det, den er bedst til? Jo, måske nok, men alligevel, der er faktisk en hage eller to ved computerens måde at repræsentere tal på.

Fx. taler vi om heltal, de naturlige tal, tallene fra 0 og opefter. Eller er det fra 1 og opefter ;-) Nå, i hvert fald så siger vi, at vi kan repræsentere heltallene med en computers kalkulator register eller en memory celle. Men det er ikke helt rigtigt! Det er nemlig kun et udsnit af de naturlige tal, som vi kan repræsentere. Så ikke engang denne enkle opgave kan en computer klare så godt som en almindelig dødelig! Forklaring følger.

Et register er en slags tælleværk, der i dag typisk består af 32 bits - binary digits. De enkelte digits er lavet ved hjælp af et transistor mønster, der kan være i to tilstande, enten i en tilstand, hvor strømmen kan passere, eller også i en tilstand, hvor strømmen blokeres. Når der er 32 bits, kan de danne ca. 4 mia. mønstre eller kombinationer af on/off, 0 eller 1. Det er maximum antal kombinationer. Hvis vi altså tæller fra 0, kan vi komme op på 4 mia med et 32-bit register.

Det er derfor, at der har været en grænse for filstørrelser på de almindeligst forekommende filsystemer. Siger 2 Gb grænsen dig noget? Det er den største fil på for eksempel Linux' ext2-filsystemer. Det kunne have været en 4 Gb grænse, men én af bit'ene er reserveret til et andet formål, nemlig til angivelse af, om systemet er i en fejltilstand efter en skrive- eller læseoperation. En bit, on/off, svarer altså til en fordobling af kombinationsmulighederne, se nærmere nedenfor.

Bits er altså repræsenteret ved hjælp af elektriske kredsløb, der kan aflæses af andre kredsløb.

Kalkulator registeret, det vigtigste register på den traditionelt opbyggede CPU, er en slags tælleværk bestående af bits (i dag som regel 32), som en kilometertæller, som kan udføre forskellige regnestykker på det tal, den indeholder.

En CPU kan have mange kalkulator-registre. De har navne, ofte noget med R1 og R2, i Intel-arkitekturen (x86 - CPU'er) hedder de imidlertid EAX, EBX, ECX etc (hvilket er copyrighted!) Mange af dem kan udføre regnestykker. På Intel 386 arkitekturen (og Pentium m.v.) kan man dog se en vis fortrinsstilling for EAX registeret. Men denne bog handler ikke om Intel. Hvis Intels CPU'er interesserer dig, så hent fra www.intel.com "Intel Architecture Software Developer's Manual" (det er 3 PDF-filer på ialt omkring 15 Mb), eller find en bog om assemblerprogrammering, eller prøv at læse den Assembly-HOWTO, som følger med bl.a. Red Hat Linux.

Hvis CPU-en har 32-bit registre, så kan man skrive tal fra 0 til 4 milliarder. Selv om det er mange, er det ikke ALLE de naturlige tal - de fortsætter jo opefter, mod uendelig, som man siger. Selv om det heller ikke er muligt at finde noget i universet, som er uendeligt, man bliver om jeg så må sige træt inden man når dertil, er det alligevel en mere begrænset repræsentation af talsystemet, som vi får med en 32-bit computer [2] Metoden til at tælle ved hjælp af bits, som du sikkert kender eller allerede har regnet ud, kan illustreres med følgende tabel (med kun 4 bit):

ottere	**firere**	**toere_**	**enere_**	**værdi med almindeligt 10-talsystem**
0	0	0	0	== 0
0	0	0	1	== 1
0	0	1	0	== 2
0	0	1	1	== 3
0	1	0	0	== 4
0	1	0	1	== 5
0	1	1	0	== 6
0	1	1	1	== 7

Som man kan se kan kolonnerne kaldes ENERE, TOERE og FIRERE i analogi med vores ti-tal system, hvor vi siger, at kolonnerne repræsenterer enere, tiere og hundreder etc.

Hvis vi har brug for mere, kan vi imidlertid godt sætte flere registre, eller flere memory celler, sammen, så vores tal kan blive større. Det er forholdsvis enkelt.

De negative tal kunne vi repræsentere ved at markere med en ekstra bit (eventuelt et helt andet sted) at vi skulle gange med -1. Nogle maskiner kan faktisk skelne mellem +0 og -0! Men det er ikke populært i dag. Man vælger som regel en anden metode.

Man indretter CPU registrene sådan, at når man tæller op, så bliver 1...1111 til 0...0000 (for ikke at skrive alle 32 bit, skriver jeg 1...1111.) Det ligner lidt en kilometertæller, som når den har nået det største tal, den kan vise, går tilbage til 0, altså for eksempel går fra 99999 til 00000.

Men altså, det skal forestille, at alle bit er tændt - det kalder man, at de er sat. Hvis det er et 32-bit register, svarer det til ca. 4 milliarder. Når man så lægger 1 til, bliver resultatet 0! Det ligner ikke helt den verden, vi ønsker at repræsentere med vores computer programmer. Derfor bør man under programmering altid være opmærksom på grænserne for de tal, man arbejder med. Det er imidlertid ikke så slemt, som det lyder.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Ovenstående forestiller en kilometertæller som tæller med nuller og enere. Den er løbet fuld! Næste gang, den skal tælle opad, skifter alle cifre, ligesom 99.999km ville skifte til 00.000km.

Altså, når alle bit er sat, og vi lægger en til, så slukker de alle sammen. Derfor kunne det være en god idé

at lade 1...1111 repræsentere det negative tal -1.

fortegn	________	________	________	**værdi med almindeligt 10-talsystem**
1	0	0	0	== -8
1	0	0	1	== -7
1	0	1	0	== -6
1	0	1	1	== -5
1	1	0	0	== -4
1	1	0	1	== -3
1	1	1	0	== -2
1	1	1	1	== -1

Som man kan se, nytter det ikke noget her at kalde kolonnerne for ENERE, TOERE og FIRERE. Hvis man skal give denne "konvertering af betydningen" et navn, så plejer man at kalde den 2's komplement. Men egentlig er det 2^{32}'s komplement, d.v.s. 2^{32}-tal.

Det er nok lettere at se det for sig med en illustration. Man repræsenterer f.x. 1 med en bit i den ene ende. Når man vender alle bits om, så 0'er bliver til 1 og omvent (inverterer), og lægger 1 til, så får man bitmønsteret til repræsentation af -1.

Eksempel 1-1. tallene fra 0 til 7 og 0 til -7.

```
0: 0000, Inverted: 1111, Complement(til 2^4): 0000
1: 0001, Inverted: 1110, Complement(til 2^4): 1111
2: 0010, Inverted: 1101, Complement(til 2^4): 1110
3: 0011, Inverted: 1100, Complement(til 2^4): 1101
4: 0100, Inverted: 1011, Complement(til 2^4): 1100
5: 0101, Inverted: 1010, Complement(til 2^4): 1011
6: 0110, Inverted: 1001, Complement(til 2^4): 1010
7: 0111, Inverted: 1000, Complement(til 2^4): 1001
```

Når der er tale om 32 bit heltals repræsentation, er det lettere at benytte hexadecimal notation. Tallene fra 0-9 skrives som sædvanlig 0-9, men 10-15 skrives a-f. Det betyder, at man kan repræsentere 4 bit med ét 'ciffer', hvor ciffer skal forstås som 0-9a-f.

Eksempel 1-2. tallene fra 0 til 7 og 0 til -7.

```
Word: 00000000, Inverted: ffffffff, Complement: 00000000
Word: 00000001, Inverted: fffffffe, Complement: ffffffff
Word: 00000002, Inverted: fffffffd, Complement: fffffffe
Word: 00000003, Inverted: fffffffc, Complement: fffffffd
Word: 00000004, Inverted: fffffffb, Complement: fffffffc
Word: 00000005, Inverted: fffffffa, Complement: fffffffb
Word: 00000006, Inverted: fffffff9, Complement: fffffffa
Word: 00000007, Inverted: fffffff8, Complement: fffffff9
```

Fortsætter man denne tankegang, så bliver det største tal med 32 - bit ca. 2 milliarder, og det er stadigvæk en pæn sjat.

Grunden til, at Linux' ext2-filsystem ikke kan have filer, som er større end 2 Gb er forresten, at grænsesnittene der bruges til at læse og skrive filer arbejder med 32 bit heltal (hvoraf en bit bruges til fortegnet).

Et program, som frembringer udskriften ovenfor (fig Eksempel 1-2) er vist i Afsnit 2.7.

Det er muligt at repræsentere tal på andre måder end den her viste. Men vigtigere, det er muligt at repræsentere andet end tal. Bogstaver kan repræsenteres ved hjælp af tal, den simpleste løsning på en computer med mulighed for at repræsentere tekst er jo den kendte metode med en byte pr. bogstav. Det er ikke nok, hvis der også skal være plads til at repræsentere græske bogstaver, matematiske symboler, og slet ikke, hvis også forskellige grafiske symboler, skrifttypebeskrivelser etc. skal repræsenteres. Men det er en helt anden historie.

Andre ting, som for eksempel grøntsager og frugt, må repræsenteres ved bits og bytes på samme måde som tal og bogstaver. Hvis du er ny i programmering, vil det klogeste være at afprøve nogle programmer nu, måske endda bruge 3 - 4 måneder på eksemplerne og øvelserne i Afsnit 2.1 kapitlet. Hvis du på den anden side kender lidt til programmering og gerne vil køre tanken om repræsentation til ende, så er der to begreber, som vi skal tygge lidt på: Simuleringer og modeller.

1.4. Simulering - Modeller

For lige at få lidt mere 'real-life' perspektiv på datarepræsentation, så forestil dig et program, som skal simulere noget, der foregår i det virkelige liv, matador-spil, adventure-spil (som på visse punkter forventes at svare til ting, vi kender fra dagligdagen), eller beregning af en vejrudsigt. Vind fra vest med lavtryk - hvor ligger grænsen (isobar linjen) for 990 millibar etc.etc. Man kan for eksempel opdele luftrummet over Skandinavien i firkanter med hver deres vejr-målinger og beregne interaktionen mellem firkanterne efter de fysiske love for tryk udligning, temperaturudvidelser etc. - pyha, godt det ikke skal udregnes i hånden!

Simuleringer er så forskellige. Man kan simulere reaktionerne på håndtag og visningen i instrumenter i et flycockpit for at give piloterne øvelse i at betjene styremekanismerne; eller man kan simulere eller efteligne lydene fra et spillende symfoniorkester. Simuleringerne svarer ikke på alle punkter til virkeligheden, men kan være nyttige og mere eller mindre tæt på virkeligheden.

Hvis vi skulle beregne det mest rentable dyrkningsforløb for tomater i drivhus, hvad skulle vi så vide om dem? Jeg behøver ikke spørge, om vi kunne spise resultatet af simuleringen! Men en ting er sikkert, der skulle mange oplysninger til, de fleste af dem skulle opsamles ved hjælp af grundige forsøg og målinger, som ville ende med nogle tal, der repræsenterede egenskaber ved forløbet.

Derimod så ligner et investeringsspil og en investerings service for bankkunder hinanden så meget, at man kan bruge det ene program som grundlag for den virkelige service. Her bliver der blot koblet betalingstransaktioner på spillet, med tilhørende retsansvar.

Kobler vi simuleringerne på virkelige objekter (med passende teknik, som i sig selv kan være komplicerede computersystemer) kan computeren anvendes som værktøj til at "styre virkeligheden", måske bedre, end vi kan gøre det, men til syvende og sidst blot som vores "forlængede arm".

Simulerings- og styringsteknikker hviler på evnen til at repræsentere begreber og objekter ved hjælp af computerens bits. Som nævnt er denne repræsentation altid mangelfuld. Den opfattelse af verden, som måtte ligge til grundlag for simuleringerne, kaldes en model. Modellen er altså ikke kun repræsentationen af statiske, ubevægelige, egenskaber, som tingene har, men også af deres måde at bevæge sig og indgå forbindelser med hinanden, sådan som for eksempel de enkelte styrehåndtag i en flysimulator må reagere på og med hinanden.

Men ligesom talrepræsentationen er mangelfuld, så vil modeller også være mangelfulde. Ikke kun fordi der mangler den syvende decimal, men fordi der er grænser for, hvor meget programmøren kan forudse. Havarisituationer, vejrsituationer, forskellige former for landskabsforhindringer etc. kan kun i et begrænset antal være repræsenteret i en computer. På et eller andet tidspunkt kommer man (efter meget arbejde) formentlig til en model, som tilfredsstiller brugerne af modellen så meget, at de opfatter modellen som en fuldt funktionel fremstilling af virkeligheden.

Den slags tanker virker specielt tiltrækkende på børn og reklamefolk. Måske også på fabrikanter af computerspil. Ord som cybernetics, cyberspace og cyborg spiller på dette element af "en verden inde i computeren".

Man kunne sige, at vi har "mappet", kortlagt, en del af den virkelige verden ved hjælp af vores computers bits og nogle programmer, som behandler dem i overensstemmelse med vores fortolkning.

Denne teknik, at "mappe" en mængde af elementer, således at de repræsenteres af en anden mængde, er den grundlæggende disciplin for applikationsprogrammører. Alan Turing har omkring 1940-1950 beskrevet det teoretiske grundlag for computerens virkemåde, og hans arbejde er stadig interessant for programmører, der vil forstå muligheder og grænser for computerens "indre univers".

1.5. Planlægning af et program

For at omsætte teori til praksis er det bedst at begynde med en opgavetype, som man kender lidt til i forvejen, og helst et område, som ikke involverer kompliceret teknik. I denne sammenhæng er spil og simuleringer er gode.

Det er vigtigt, at man ikke begynder for ambitiøst. Pas på ikke at gøre for meget ud af detaillerne i første omgang. Prøv for eksempel at skitsere, hvilke oplysninger, der bør være med i et CD kartotek. Når det

viser sig, at denne opgave er af næsten uoverskuelig kompleksitet og involverer ansættelse af 10 erfarne bibliotekarer (jvf. Knuth [2]) så er man gået for langt.

For at repræsentere et medlem i et medlemskartotek er det indlysende, at der skal være navn og medlemsnummer (eller er medlemsnummer nok?), måske skal navnet deles i for/efternavn, formentlig skal vi have adressen med. Hvordan vi vil nedfælde adressen kommer an på meningen med kartoteket. Normalt skal der være forskellige måder at kontakte personen. Det kunne måske klares ved et telefonnummer eller en e-postadresse. Skal der være felter for indbetalt kontingent eller vil vi henvise til et kontonummer og dermed bevæge os ind på posteringsteknikker?

Det er rart at prøve det nogle gange. Det er også ganske underholdende, fordi man (forhåbentlig) opdager en masse om, hvad man kan forvente af funktionalitet ud fra de data, man agter at registrere.

Opgave: Prøv at tegne/skitsere en repræsentation af noget, du kender godt, f.eks. et medlemskartotek, og beskriv de operationer, du ønsker at kunne foretage. Hvis det bliver nødvendigt at ændre i datarepræsentationen, så gør det, og tjek, at de gamle operationer stadig kan fungere. Prøv at holde skitsen så simpel som muligt, således at man stort set kunne gøre de samme ting ved hjælp af en editor (tekstbehandling).

Det er den samme fremgangsmåde som ved objektorienteret analyse og planlægning, med den lille undtagelse, at vi ikke hævder at afspejle den ydre verdens datastruktur i vores interne datarepræsentation.

Man kan selvfølgelig også begynde med de *operationer* , som man ønsker at have til rådighed. Derefter finder man ud af, hvilke data, som kræves for at kunne gøre det.

Med denne metode skal man være mere opmærksom på, hvordan de forskellige data skal grupperes. Det sjove ved den "objektorienterede" analyse er jo, at man anvender sit forudfattede billede af den eksisterende opgave til at gruppere sine data.

Uanset hvilken indfaldsvinkel man foretrækker, så er det vigtigt at tegne og tale med kolleger, i nødsfald med sig selv, om hvordan man kunne gribe sagen an. Brug meget gerne pseudo-programmering, d.v.s. almindelige sætninger, som beskriver, hvordan programmet skal fungere. Det kunne man kalde aktions-orienteret programmering ;-)

Den her foreslåede fremgangsmåde er jo den samme procedure som ved objektorienteret analyse til for eksempel programmering i C++, Det er ikke umuligt at skrive strukturerede, objektorienterede programmer ved hjælp af C sproget. Det er selvfølgelig nemmere at gøre det med C++ (eller bør være nemmere), men der er stadig nogle situationer, hvor C er mere effektivt.

Slutbemærkning:

1. Det kaldes aflusning eller debugging, bug == insekt, - der skulle engang være opstået en fejl i en af de første computere p.g.a. nogle insekter, som syntes, at der var varmt og rart at være oven på sådan

nogen elektronik-komponenter.

2. En lidt mere seriøs beskrivelse af tid, uendelighed og rum kan man finde i Stephen Hawking's "Brief History Of Time", nej, ikke Stephen King!

Kapitel 2. Funktioner, modularitet og relevante teknikker

Resume: Hello-world programmer isolerer og afprøver en speciel teknik. For at konstruere og lære nye teknikker er det en god ting at kunne isolere hver eneste lille funktionalitet i det nye program, man ønsker at bygge. Denne teknik kan man kalde terrasse-teknik.

Et C-program består af en serie definitioner af eksterne objekter. Et eksternt objekt kan være en funktion eller en datadefinition, d.v.s. en variabel. For at få overblik over en opgave fra begyndelsen af, er det godt at vide, hvordan man opdeler en opgave i mindre dele. Ikke blot hvilke funktionaliteter, som lader sig isolere, men også en idé om, hvordan man bærer sig ad med at opdele et program i mindre programdele. Det er hovedemnet for dette kapitel.

En opdeling forudsætter, at der er mulighed for at koble forskellige moduler sammen igen på et senere tidspunkt ved hjælp af libraries, header filer og make filer. Det vil nærværende kapitel give nogle eksempler på.

I C er funktioner grundlaget for opdelingen af programmer i mindre dele. Som Kernighan siger: funktioner gør det muligt at genbruge kode, gør det muligt, at en programmør bygger videre på en anden programmørs arbejde.

Det er vel at mærke sådan at forstå, at jeg, i mit nye program, kan includere al funktionalitet fra noget, som er skrevet i forvejen *uden at jeg skal ændre i eller kopiere fra den oprindelige kode.*

Grundlaget for genbrug er som sagt funktionen, men ved hjælp af separat kompilering, statiske variable og evt. dynamisk allokering af mere hukommelse kan denne teknik forfines, således at man reelt set råder over objektorienterede metoder. Det bedst kendte eksempel er ANSI C bibliotekets Input Output del, som bygger på begrebet en FILE. Man kan erklære en variabel af typen `FILE*` uden at ane, hvad den indeholder. Det varierer også fra system til system. Men fælles for alle `FILE*` er, at man kan åbne dem, `fopen()` , læse fra dem `fread()` og skrive til dem `fwrite()` finde en position, lukke igen og meget andet.

2.1. ANSI prototyper og modularitet

I dette kapitel ser vi på hvordan man opdeler en lille opgave i en beregningsdel og en hoveddel. Undervejs eksperimenteres med funktioner og prototyper.

Derefter ses på filter programmer, som er programmer, der læser noget input og transformerer dette til en bestemt slags output. For et givet input vil output altid være det samme.

Dette princip udstrækkes til et forholdsvis kompliceret program, som parser sgml tags og formaterer lidt på en sgml tekst. Endelig er der forslag til mange øvelser undervejs, og den interesserede læser vil sikkert ikke kunne lade være med at eksperimentere yderligere. God fornøjelse!

Men allerførst skal vi lige se på, hvilken betydning returværdien fra en funktion kan have på et helt andet program, nemlig en shell eller kommandofortolkeren. Vi bruger et minimal program til at isolere fænomenet.

2.1.1. Et brohoved.

Forhåbentlig kender du Kernighan & Ritchies bog The C Programming Language. Det første kapitel, den berømte "tour" gennem C sproget, starter med et program, der skriver "Hello, World!" på en uddataenhed (altså en skærm eller lignende).[1]

Det program kunne man jo så udnævne til stamfaderen for en hel kategori. "Hello-world" programmer isolerer en feature og afprøver, hvordan den virker. Et "hello-program" skal helst kunne køre, selv om vi nogen gange nøjes med at skrive en funktion for at se, om oversætteren accepterer den syntaks, vi anvender.

Et "hello-program" kan være et, som skriver noget på skærmen, eller det kan hente dato-information, så kan vi få bekræftet, om dato funktionerne opfører sig, som vi forventer, eller ej. Vi kunne kalde det for et minimalprogram. Vi kunne også sige, at vi isolerer de features i C sproget, som vi ønsker at lære/undersøge/afprøve. Metoden er vigtig at lære. Det er en ekstremt nyttig metode; med den kan man løse flere problemer, end man kan med en debugger.

2.1.2. Returværdi fra en funktion

Det, vi skal i gang med nu, er at undersøge, hvordan funktioner afleverer data til hinanden, og hvordan C sproget gør det lettere at lave sådan nogle "kasser", ofte kaldet black box, om hvilken man ved, at den kan dit og dat, og at den er helt uafhængig af resten af vores program.

Et helt grundlæggende "Hello-world" program er et, som simpelt hen afslutter med det samme! Sådan et kommer her:

Eksempel 2-1. HELLO - statuskode

```
/* frame.c Minimalt program til afproevning af statuskode. */

int main()
```

```
{
    return 0;
}

/* end of file frame.c */
```

Program-source, kildeteksten består af 8 linjer, hvis man tæller kommentarer og tomme linjer med. Aller øverst er der en kommentar, som fortæller kort hvad meningen med programmet er. En kommentar startes med "/*" og slutter med "*/".

Programmet består af en definition af ét eksternt objekt, nemlig en funktion, som har navnet "main". Parenteserne efter main fortæller, at main er et objekt af typen funktion. Parenteserne kaldes derfor "funktions-operator" [2] Selve koden i main er indrammet i krøllede parenteser, braces. Koden består af kun én sætning, eller *statement* , nemlig

```
return 0;
```

return er en specifikation af, at funktionen skal aflevere noget til den, som har bedt om at få udført funktionen (har *kaldt* den.)

return er et *reserveret ord* , d.v.s. et ord, som oversætteren er født med at kende. C sproget har 32 reserverede ord. (se Afsnit A.1 .)

Nullet er et "udtryk", (aritmetisk udtryk) med en talværdi. Vi kunne også have skrevet return 234 eller return 7000143. I dette tilfælde vil det dog være klogt at holde talværdien under 256.

Eksempler på andre expressions: kroner = timer * timeloen; hvor det forudsættes at kroner, timer og timeloen er variable, som indeholder fornuftige værdier. Et expression, som afsluttes med semikolon, kaldes et statement.

Hvis der er flere statements i en funktion, udføres de i rækkefølge, oppefra nedefter.

Et kald til en funktion, som f.eks. flg.: `abs(-5);` er også et expression, i dette tilfælde med værdien 5. Kald til en funktion vil ofte returnere en variabel af typen heltal, *integer* , og en integer i et expression kan erstattes af et kald til en funktion, som returnerer en integer.

Det er en konvention, at kørsel af et program, som benytter standardbiblioteket (med bl.a. læse- og skrivefunktioner) begynder med funktionen *main*. Når man kommer til slutningen af denne funktion, slutter programmet her med at returnere en statuskode til styresystemet. Denne statuskode bruger man til at markere om programmet blev afbrudt af en fejl (og lignende). Hvis et program slutter uden et "return <expression>;" er det sjusk.

Kernighan & Ritchie dropper return-sætningen en del gange i *bogen*, men det er faktisk sjusk alligevel! De gør det selvfølgelig fordi det er lettere at forklare et program, hvori der kun er de nødvendigste linjer.

Statuskoden bør fortælle, om programmet kunne køre uden fejl, (dvs. uden fejl, der er påført af ydre omstændigheder, som f.eks. at en datafil mangler). Det er altså programmørens mulighed for at sende et signal om at "alt er vel" eller "her opstod en fatal fejl".

Hvis programmøren vil fortælle systemet, at der var en fejl, skrives simpelthen:

```
return 255;
```

Tallet kan i Unix-kommandofortolker-sammenhæng læses i variabelen $?, som kan styre processtrømmen i et Unix-skalprogram *(en shell)* .[3] Øvelse: Ret, så programmet returnerer 117 og se, om du kan udskrive systemvariablen $? med kommandoen

```
dax@pluto$ echo $?
 0
dax@pluto$
```

Det lille program ovenfor kunne oversættes/compileres [4] med flg. kommando:

```
dax@pluto$ gcc frame.c -o frame
dax@pluto$
```

Derefter kan det køres fra current directory (i det aktuelle katalog, eller sagt på en tredie måde, fra det bibliotek, som vi står i [5]) med en kommando som:

```
dax@pluto$ ./frame
dax@pluto$
```

eller, hvis din PATH-systemvariabel ender på ':'

```
dax@pluto$ frame
dax@pluto$
```

Eksemplet lider imidlertid af en alvorlig skavank, vi kan jo næsten ikke se, om programmet rent faktisk kører. Det laver jo ikke noget! Derfor tilføjer vi en lille output kommando:

Eksempel 2-2. Skriv message på standard output.

```
/* frame2.c Skriv til stdout og afslut. */

#include <stdio.h>

int main()
{
```

```
    puts("Hello! Programmet frame2 kører nu...");
    return 0;
}

/* end of file frame2.c */
```

Her er flere ting, som er værd at lægge mærke til. Dels et include direktiv, d.v.s. en kommando, som fortæller oversætteren, at den skal læse en fil, der hedder stdio.h. Når filnavnet står i *vinkler* , så betyder det, at oversætteren skal lede der, hvor systemet normalt har sine filer med erklæringer, "include filerne". På Unix, Linux og andre systemer er det `/usr/include`, der gennemsøges først.

Oversætteren finder den pågældende file og læser den. Den indeholder *kun type erklæringer*.

Den erklæring, som vi skal bruge, ser ud som følger:

```
extern int puts (const char *__str);
```

Den kunne dog også have set enklere ud:

```
int puts (char *message);
```

Det kaldes en prototype. Denne gør det muligt for oversætteren at tjekke, at funktionskald vil fungere efter hensigten.

Prototypen "int puts(char*)" fortæller, at puts er en funktion, som returnerer en integer og forventer at få en character pointer som argument.

En prototype for vores main (som burde findes i en af glibc - systemets header filer) ville hjælpe oversætteren med at kontrollere, om vi overholdt interface mellem vores main og bibliotekets startup procedure. [6] Denne prototype ville se sådan ud:

```
int main(int argc, char *argv[], char *env[]);
```

Det er ikke nødvendigt at angive et navn, en identifier, på argumenterne, kun typen skal angives når vi skriver en prototype.

```
int main(int, char *[], char *[]);
```

Men det er oplagt at finde navne, som giver læseren en hjælp til at forstå meningen med funktionen. Argc står for argument count, argv for argument vector, en liste med alle parametre fra det program, som har startet vores program op; env står for systemvariable (eng. environment variables), her kan vi aflæse brugerpreferencer og ligenende. Når vi ikke bruger parametrene til main, kan vi nøjes med at skrive main(), altså en tom funktions-parentes.

Eksempel 2-3. En character pointer

```
/* frame3.c Demonstration af character pointer. */

#include <stdio.h>

int main()
{
    char *charpointer = "Hallo! Programmet frame3 kører nu...";
    puts(charpointer);
    return 0;
}

/* end of file frame3.c */
```

char *charpointer er erklæring af en variabel. Variabelen er en character pointer, det vil sige en adresse variabel. Den initialiseres på samme source linje, som den erklæres. Det er simpelthen en praktisk skrivemåde. Det svarer til:

```
main()
{
    char * charptr;             /* charptr er en variabel */
    charptr = "Hallo etc... ";
```

En literal string, "Hallo etc... "; er ikke en variabel, men er en besked til oversætteren om at initialisere et dataområde med den tekst, som vi nu ønsker os. For at kunne bruge teksten skal vi enten gemme adressen på den (altså cptr = "Hallo etc...") eller også give adressen på denne string til den funktion, som skal bruge den: `puts("Hallo etc..");`

En gengivelse af dette system af RAM-adresser og indhold kunne tegnes som en reol. Her er det en reol, hvor hver hylde har et indhold, der enten kan være 4 bytes eller 4 bogstaver, eller én adresse (32 bits).

```
Adresse
eller
hylde-nr.      Indhold

              ~             ~
              :             :
              +-------------+
              |             |
              |  o !   P    |
    800440    +-------------+
              |             |
              |  H a l l    |
    800436    +-------------+
              |             |
              |             |
              +-------------+
```

```
                   |             |
                   :             :
                   :             :
                   :             :
                   |             |
         120808    +-------------+  charptr er en adressevariabel,
                   |             |  som indeholder adressen
                   |  <800436>   |  på en string, "Hallo! Prog...etc."
charptr: 120804    +-------------+
                   |             |
                   |             |
         120800    +-------------+
                   |             |
                   |             |
         120796    +-------------+
                   :             :
                   :             :
                   |             |
                   :             :
                   :             :
                   |             |
```

Det, der skrives på skærmen, er tekst, som bogstav for bogstav ligger i programmets data-del.

Lad os nu gå over til beregningsprogrammer. Hvis de ikke har noget input, er de en slags "hello" programmer, fordi de isolerer de aritmetiske operationer. Selv om vi begynder med simple eksempler, kan det sandelig godt være nyttige programmer. Den oprindelige idé om computere var jo blot at de skulle beregne ting, som var trælse at beregne i hånden, planetbaner, skatteopgørelser og lignende ;-)

2.2. Parameterlister og datatyper

I foregående afsnit afleverede vi én oplysning til en funktion. I dette afsnit skal vi aflevere flere oplysninger og desuden se, hvordan en funktion kan udføre sit arbejde på en så robust og fornuftig måde, at vi ikke behøver at spekulere på, om programmet bliver ustabilt, hvis brugeren foretager sig noget tåbeligt.

Vi vil foreløbig nøjes med at bruge de indbyggede datatyper, men det er på ingen måde en begrænsning for C programmøren, at sproget "kun har" int, double, char etc, for vi kan jo opbygge andre datatyper ved hjælp af brugerdefinerede datatyper, struct, og vi kan manipulere de enkelte bits ved hjælp af bitvise operatorer.

På den måde kan man bygge udmærkede repræsentationer af de datatyper, som findes i andre sprog, men som savnes i C. For eksempel kan man opbygge en tegnbaseret repræsentation af tal, ligesom det findes i Lisp. Ved hjælp af denne repræsentation kan man opnå mange hundrede (tusinder) decimalers præcision i beregninger.

2.2.1. Datatyper og beregninger

Den grundlæggende datatype i C sproget er int, en forkortelse for integer, heltal. Størrelsen af denne datatype er ikke altid den samme, på en 8 bit computer er det 16 bit (hmm, fik du den?) og på en 64 bit computer er det 64 bit. Hvis man vil foretage 64-bit beregninger på en 8-bit computer, så kan det godt lade sig gøre, det vil blot være væsentligt langsommere end hvis man nøjes med at benytte maskinens "medfødte" datatype. Det kan implementeres som en oversætterfunktionalitet, eller det kan være, at det er implementeret som maskin-instruktioner, der benytter flere registre. Intel 386-arkitekturen benytter to registre til 64-bit aritmetik, men har dog ikke ægte 64-bit division. Det må man så lave på en anden måde.

Lad os prøve at foretage nogle beregninger med heltal. Det må gerne være nogle simple beregninger, sådan at vi nemt kan kontrollere, om programmet regner rigtigt. Sidenhen kan vi ændre det til noget mere imponerende.

2.2.1.1. En procentberegning

Eksempel 2-4. HELLO - beregning.

```
/* procent.c beregner en procentdel af et givet tal. */

#include <stdio.h>

int main()
{
    int procent = 17;
    int kroner = 100;
    int resultat;

    resultat = kroner * procent / 100;

    printf("resultat er: %d\n", resultat);

    return 0;
}

/* end of file procent.c */
```

Det er en stor fordel, at resultatet er nemt at kontrollere, brug simple tal indtil programmet er stabilt.

Vi har indført nogle variable - kroner, procent, resultat. Det er simpelt hen kasser med tal i. Der er straks fyldt værdier i kasserne.

Navnene er valgt sådan, at man kan regne ud, hvad meningen er med variabelen.

```
int kroner = 100;
```

betyder, at vi reserverer en plads til en heltalsvariabel og straks fylder tallet 100 i.

Selve beregningen kan skrives i programmet næsten på samme måde, som man ville skrive formelen på et stykke papir. Hvis man er vant til, at (x y) betyder x gange y, så skal man selvfølgelig passe på at man ikke glemmer multiplikations operatoren '*'.

```
resultat = kroner * procent / 100;
```

`resultat` er navnet på den "kasse", hvor resultatet skal gemmes, og den kaldes en "left value", venstre-værdi, fordi den kan stå på venstre side af et lighedstegn eller assignment (tildelings) operator. Venstre side af assignment operatoren skal være et udtryk, der kan evalueres som en adresse, ellers kan man jo ikke komme til at gemme resultatet af vejen. Hvad der måtte ligge af interessante værdier i sådan en "kasse", forsvinder efter et assignment.

`printf()` får at vide, at den skal skrive variablen "resultat" ud ved hjælp af procent-tegnet efterfulgt af 'd'. Nu får `printf()` ikke én, men *to* oplysninger, nemlig format string'en mellem double quotes, (gåseøjne) og talvariabelen *resultat*.

```
"resultat er: %d\n"   <== string-var
"resultat er: %d\n", resultat <== string-var, tal-var
```

Den anden linje kaldes en liste eller en *parameter liste* ; de to medlemmer er adskilt af et komma, komma er liste-operator.[7]

Det er nok lidt for besværligt at rette i programmet her, hver gang jeg vil udføre en procent beregning! Så i stedet laver vi i næste afsnit et program, som prompter for (beder om input af) det tal, der skal beregnes procent af. Dermed forlader vi "Hello-world" sfæren og bevæger os ind i fejlmulighedernes paradis, interaktive programmer.

2.2.2. Prompt og input

Input fortjener et kapitel for sig. Men lad os alligevel her skrive et program, som læser inddata fra tastaturet, og som godt nok forventer, at der bliver tastet tal (cifre) ind, men på den anden side ikke tager skade af, at brugeren indtaster noget helt andet - eventuelt skubber en bog henover tastaturet, så der kommer input i store mængder.

2.2.2.1. Input af tal og beregning

Eksempel 2-5. Input og beregning.

```
/* procent2.c prompt for tal og beregn procentdel. */

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

int main()
{
    int procent = 17;
    int kroner;
    int resultat;
    char inputlinje[800];

    printf("BEREGNING AF %d %%\n",procent);
    while (printf("Input tal:"), fgets(inputlinje,800,stdin)!=NULL) {
      kroner = atoi(inputlinje);
      resultat = kroner * procent / 100;
      printf("resultat er: %d\n", resultat);
    }
    return 0;
}

/* end of file procent2.c */
```

I nogle C-bibliotek implementationer skal man flushe printf-output hvis der ikke er en "newline" til sidst:

```
... printf("Input tal:"), fflush(stdout),
```

Denne måde at kæde expressions sammen med et komma er bestemt ikke den mest pædagogiske. Det er faktisk grimt! Men når jeg nu har gjort det alligevel, så er det for at understrege, at C sprogets liste-operator (komma'et) er anvendelig i mange sammenhæng. Men lad nu være med at skrive for meget komatøs kode!

Det er værdien af det sidste expression, som er afgørende for, om expression-listen evaluerer til sand eller falsk.

Input læses af fgets(3). Den skal have 3 oplysninger, adressen på den buffer, som den må lægge characters i, længden på bufferen (den læser maximalt længde - 1 og afslutter string med en null-byte) og den fil, som den skal læse input fra. Her anvendes "kodeordet" stdin, som er defineret i <stdio.h>.

fgets(3) returnerer NULL hvis der ikke er mere input. Dette kan ske hvis brugeren taster "symbolsk end-of-file", der som regel sættes til ^D med stty kommandoen. Men hvis input er omdirigeret, så sker det jo som forventet, når man rammer slutningen på filen.

Der er includeret en fil mere, stdlib.h, som indeholder prototype til atoi - Ascii TO Integer conversion.[8]

```
#define __P(args)  args
```

```
extern int atoi __P ((__const char *__nptr));
/* __P(args) er en kommando til præprocessoren, det kaldes
 * en macro, og den har til formål at gøre
 * bibliotek funktionerne brugbare sammen med mange
 * forskellige C-oversættere. Nogle varianter af C-oversættere kan
 * ikke forstå komplicerede prototypeerklæringer.
 */
/* derfor svarer denne prototype til:
extern int atoi(const char* string_som_skal_converteres);
 */
```

Det smukke i procent2.c er, at man ikke kan få programmet til at gå ned ved ondskabsful indtastning, når der promptes for et tal. Man kan godt få det til at regne forkert, hvis man indtaster et tal, som er større end 126 mio. Men dels er fgets() en robust funktion, som ikke laver buffer overflow, selv om brugeren indtaster 2 Gb data, og `atoi(char*)` er en robust konverteringsrutine, som ikke brokker sig, hvis input ikke er cifre. Hvis man indtaster bogstaver, ignoreres de, og der returneres 0.

88kr. vil blive konverteret til heltallet 88, hvilket svarer til, hvad man forventer. Senere vil vi lave en indtastningsrutine, som giver brugeren en warning, hvis han ikke indtaster tal.

Det er programmørens ansvar at sørge for den rigtige længdeangivelse til fgets' anden parameter. Det er lettere at holde styr på den slags, hvis man benytter preprocessor macro'er til at definere symboler for konstant-talværdier.

```
#define MAXLINJE 800
char inputlinje[MAXLINJE];

main()
{
    fgets(inputlinje, MAXLINJE, stdin);
    return 0;
}
```

Ikke semikolon i #define linjen, det er *ikke* et statement.

Øvelser for de lidt mere erfarne: Prøv at ændre programmet, så der anvendes double precision floating point variable. Prøv også at tilføje en kontrolberegning, som finder ud af, om der har været overflow. Hvis der er overflow, så skriv en fejlmeddelelse til brugeren i stedet for at skrive resultatet, men stop ikke programmet.

Prøv også at skriv en version, hvor man kan indtaste procentsatsen først, eller et, som udskriver værdien af kroner i både australske, canadiske og US dollar.

Lav et program, som udskriver flere valutaer pr. linje i en tabel, f.eks. svarende til kr. 100, 200, 300, 400, 500 ...)

2.3. ANSI prototyper og modularisering

Opdelingen i funktioner er grundlaget for strukturering af opgaverne, som man måtte være ved at løse.

Funktioner er den teknik, som gør det muligt at genbruge kode, sådan at man ikke behøver at begynde fra bunden hver gang, men kan bygge videre på andres arbejde.

Der er ikke noget problem, der er så stort, at det ikke kan deles op i mindre :-)

2.3.1. Kursomregning

For at illustrere, hvordan man trækker en beregning ud af et program, kunne vi bruge en beregning så simpel som x = a + b.

Skal eksemplet være en lille smule realistisk foreslår jeg imidlertid, at vi i stedet laver en kursomregningsfunktion. Man kan forestille sig, at der er skal så mange beregninger til (f.eks. en tabel i en udskrift eller priser på en faktura) at det er en stor lettelse at få udført beregningen i en funktion.

Det er forhåbentlig indlysende, at en "euro-funktion" ville kunne indgå i bankers udbetalingsautomater, i faktureringsprogrammer etc.etc.

Lad os derfor benytte kursomregning til at forske i den teknik, som kaldes modularisering. Det endelige mål er en omregningsfunktion i et bibliotek af forskellige financielle funktioner. Man må forestille sig valutaveksling med og uden gebyr og afrunding etc. Dette er så den rå omregning til brug for rapporter eller lignende. Det er simpelt hen en funktion, som får vores kroner og "afleverer" dollar (eller Euro). Udgangspunktet er en tilretning af Eksempel 2-5, så vi kan afprøve funktionen mens vi skriver på den. For at gøre det mere overskueligt, har jeg imidlertid valgt at sløjfe prompt-input delen; den kan man evt. selv tilføje efter mønsteret i Eksempel 2-5.

Jeg må indrømme, at jeg efter at have skrevet dette her afsnit syntes, at det var lidt rigeligt langt! Grunden til, at jeg lader det stå er, at jeg har set så mange programmører, der havde svært ved at forstå mekanikken i funktionskald. Først når man forstår hvad der sker i computeren under et funktionskald kan man udnytte C sprogets fulde styrke.

2.3.1.1. Beregning som del af main-koden

Eksempel 2-6. Dollar omregning, spaghetti [9] version.

```
/* dollar0.c Input Kroner, beregn Dollar. UDGANGSPUNKT. */

#include <stdio.h>
```

```
int main()
{
    int kurs = 865;
    int kroner = 100;
    int resultat;

    resultat = kroner * 100 / kurs;
    printf("Kroner %d giver Dollar %d\n", kroner, resultat);
    return 0;
}
/* end of file dollar0.c */
```

Først trækker vi beregningen ud af programmet og lægger den i en funktion, som vi kalder kr2dollar.

2.3.1.2. En ANSI prototype

Eksempel 2-7. Dollar omregning med beregning i funktion.

```
/* dollar1.c Input kroner, kald int kr2dollar(int) */

#include <stdio.h>

/* vi erklærer nu en prototype for vores funktion. En prototype kan
 * kendes på, at der efter funktionsparentesen er et semikolon - ikke
 * nogen braces, som ville signalere starten af en kodeblok.
 */

int kr2dollar(int);
int kurs = 865;

int main()
{
    int kroner = 100;
    int resultat;

    resultat = kr2dollar(kroner);
    printf("Kroner %d giver Dollar %d\n", kroner, resultat);
    return 0;
}

int kr2dollar(int kr)
{
    return kr * 100 / kurs;
}

/* end of file dollar1.c */
```

Bemærk, at main står øverst i programmet. C inviterer til top - down programmering. Vi kan kalde kr2dollar uden at have nogen som helst idé om, hvordan vi vil implementere den. Selvfølgelig er programmet ikke færdigt, før end vi har skrevet den sidste kode, men i nødsfald kan man somme tider

klare sig med en forsimplet udgave - eller en stub, en tom funktion - der, hvor man ikke har skrevet al koden.

Men funktionen kr2dollar er *erklæret* inden den anvendes, det er linjen lige neden under #include direktivet. Erklæringen er en slags forklaring til oversætteren af, hvad det er for en funktion. Den bevirker, at oversætteren opretter en entry i en symboltabel, så den kan slå op, hvad "kr2dollar" er for noget, næste gang den forekommer i kildeteksten.

Derfor ved oversætteren, hvad type der kommer ud af funktionen. Det kunne være, at det var en flydendetals dims i stedet for et heltal. (Ja forresten, det synes du nok, at det burde være! Det ville være rart med flydende tal for at få decimaler på, se Eksempel 2-10. Men strengt taget kunne vi få en mere præcis beregning ved at anvende 64-bits integers til at repræsentere 100-dele øre. For den avancerede: Prøv det! Og husk at indsætte et komma på det rigtige sted, når du skriver det ud.)

`kr2dollar()` består af KUN et return statement. Godt nok skal der regnes lidt, før end return værdien er klar, det er jo selve ideen i funktionen.

I almindelig stenalder C kunne man nøjes med at kalde funktionen uden at forklare oversætteren, at det var en funktion, der returnerede en integer. Det kaldes "implicit integer" regelen. [10]

2.3.1.3. Modulariseret udgave af beregningen

Nu skiller vi beregnings funktionen ud, så den ligger i en fil for sig selv - den er på vej til at blive en del af vores "financial library" (;-).

Desuden lader vi variabelen "resultat" udgå, for vi kan jo bare anbringe funktionskaldet der, hvor resultatet skal skrives.

Eksempel 2-8. Dollar omregning, modul version.

```
/* dollar2.c ask for Kroner and call int kr2dollar(int) */

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

int kr2dollar(int);

int main()
{
    int kroner = 100;

    printf("Kroner %d giver Dollar %d\n", kroner, kr2dollar(kroner));
    return 0;
}
/* end of file dollar2.c */
```

Som det kan ses, har vi klippet de nederste 4 linjer ud, hvor funktionen kr2dollar var defineret. Den står nu i en fil, som vi kalder kr2dollar.c:

Eksempel 2-9. kr2dollar modul.

```
/* kr2dollar.c - beregn dollar ud fra kroner */

int kr2dollar(int kr)
{
    int kurs = 865;
    return kr * 100 / kurs;
}
/* end of file kr2dollar.c */
```

Kursen er ikke mere tilgængelig i main, vi har isoleret den, så den kun kan ses i funktionen, som omregner. Det er en primitiv udgave af et udmærket princip.

Det ville være fint, hvis vi skrev en funktion, som hentede kursen fra en pålidelig kilde, f.eks. en eller anden nationalbank på internettet. Når vi så skulle bruge kursen, kunne vi kalde denne funktion.

De to filer kan oversættes på flere forskellige måder:

Enten:

```
gcc -Wall dollar2.c kr2dollar.c -o omregning
```

Eller:

```
gcc -Wall -c dollar2.c
gcc -Wall -c kr2dollar.c
gcc dollar1.o kr2dollar.o -o omregning
```

Eller:

```
gcc -Wall -c kr2dollar.c
ar -rv libfinans.a kr2dollar.o
gcc -Wall -c dollar2.c
gcc -Wall dollar1.o -L./ -lfinans
```

Læg lige mærke til, at vi har genereret en bibliotek file med en meget simpel kommando, ar -rv libfinans.a <objectfile> ...

Hvis vi skulle glemme prototypen for denne simple beregning, så vil der ikke opstå fejl i dette eksempel. Det skyldes, at vi stadig har regelen om implicit integer, når vi skriver standard C programmer. [11]

Med GNU C-oversætteren vil man dog få en advarsel: "implicit declaration of function ‘kr2dollar’". Det betyder simpelthen, at oversætteren har opdaget, at vi kalder kr2dollar, men ikke kan finde den i typetabellen. Oversætteren antager at funktionen returnerer en integer. Man får kun denne warning, hvis man anvender -Wall (Warning level, give us ALL warnings).

2.3.1.4. Modul med return type double

Lad os nu prøve at definere kr2dollar() som en funktion, der returnerer en double. Prøv nogle eksperimenter med programmet. Der er vist nogle forslag.

Eksempel 2-10. Dollar omregning, double version.

```
/* dollar3.c bed om Kroner og call double kr2dollar(double) */

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

double kr2dollar(double);  /* prøv at udelade denne her! */


int main()
{
    double kroner = 100;

    printf("Kroner %10.2f giver Dollar %10.2f\n", kroner, kr2dollar(kroner));
    return 0;
}
/* end of file dollar3.c */
```

I ovenstående eksempel er det nødvendigt, at der erklæres en prototype for kr2dollar. Hvis prototypen udelades, vil gcc, ligesom i forrige eksempel, stadig *kun* give en warning, og endda kun under forudsætning af, at -Wall anvendes!

```
$$\
gcc -Wall -c dollar3x.c # version uden prototype for kr2dollar():

dollar3x.c: In function ‘main’:
dollar3x.c:16: warning: implicit declaration of function ‘kr2dollar’
dollar3x.c:16: warning: double format, different type arg (arg 3)
```

Og her kommer så den anden fil med funktionen, som foretager omregning med double precision floating point parameteren kr.

Eksempel 2-11. kr2dollar, return type double, module.

```
/* kr2dollar.c - beregn dollar ud fra kroner, double */


double kr2dollar(double kr)
```

```
{
    return kr / 8.65;
}
/* end of file kr2dollar.c */
```

Oversættelse uden prototype vil som sagt alligevel resultere i en objektfil, som kan linkes med vores nye finans-bibliotek uden at man får en fejlmeddelelse. Men når man kører programmet, kan man se, at det ikke regner rigtigt, uha uha.

Når man oversætter et sådant program, vil oversætteren opfatte retur værdien fra en ikke erklæret funktion som en integer. Denne vil typisk være placeret i det primære register. På Intel x86 register EAX eller EBX. Oversætteren kan ikke kontrollere, om den kaldte funktion placerer sin retur værdi dér. Det er jo et helt andet modul, og måske endda et modul, som ikke er skrevet endnu.

Oversætteren "ser", at den retur værdien fra den funktion, som antages at være en integer, skal afleveres (her som argument til printf). Hvis returværdien er en integer, ligger den i EAX registeret. Derfor skriver compileren en instruktion, som skubber værdien af EAX op på applikations-stakken. Printf er, også i denne sammenhæng, lidt speciel, fordi den ikke aner, hvilke argumenter den får, før end den har læst format-specifikationen. Hvis vi skulle personificere printf, så ville den sige: "Jeg skal skrive et double precision floating point tal ud, ergo må der ligge sådan et på stakken. Det tager jeg!" Og det gør den så, den tager 8 bytes fra stakken uden at *kunne* kontrollere, om de rent faktisk repræsenterer en double eller en integer.

Det, som jeg prøver på at demonstrere, er konsekvensen af, at ANSI-C specifikationen ikke omfatter et krav om typekontrol under link-processen. Derfor er det nyttigt at tage notits af alle warnings.

Det er specielt vanskeligt med `printf(3)` , fordi det er tilladt at aflevere så mange parametre efter format-specifikationen, som man har lyst til, af de typer, som man har brug for. Det er vildt anarki, siger nogen, men det er uhyre praktisk. Med printf kan man formatere komplicerede rapporter med ét printf-statement, hvor det i C++ kan ende med mange linjers kompliceret kode, som skal styre forskellige skjulte interne variable i `cout` funktionens talkonvertering.

2.3.1.5. Header filer

For at automatisere processen med prototyper er det skik og brug at man laver en header fil til hvert projekt, som man har i gang. I vores minimal eksempel her:

Eksempel 2-12. Header fil for kr2dollar

```
/* File: dollar3.h, prototypes for finans-program ... */

double kr2dollar(double kroner);
```

Denne fil kan includeres i både der, hvor funktionen skal anvendes, og der, hvor den defineres (programmeres). Det giver jo kontrol med tingene.

Man får filen med ved at skrive: `#include "dollar3.h"` Bemærk, at der er anvendt double quotes om filnavnet fordi denne fil ligger i current directory.

NB! Der er en lignende regel, som tillader, at en (global) integer kan defineres 2 gange. Det er straks mere farligt - for tænk nu hvis det ikke var meningen - og det kan ikke tillades i C++.

2.4. Filter programmer

Filter programmer er nogen, som læser input og producerer noget output, som for samme input altid vil være det samme. Filter programmer er som skabt til batchkørsel, d.v.s. som jobs, der er automatiserede.

Her kommer det grundlæggende program:

Eksempel 2-13. Simpelt filter, input til output.

```
/* filter0.c */

#include <stdio.h>

int main()
{
    int c;
    while ( (c=getchar()) != EOF)
      putchar(c);
    return 0;
}
```

Dette program er så dejligt at eksperimentere med. Når vi har læst vores char med getchar, så kan vi gøre med den hvad vi vil, f.eks. konvertere den fra DOS-tegnsæt til UTF-8 tegnsæt. (Se nedenfor.) Men i første omgang, som her, skriver vi blot vores char ud nøjagtig som vi fik den ind. Hvis nu vi bruger en kommandolinje som nedenfor, kan vi kontrollere, at programmet faktisk kopierer nøjagtigt, d.v.s. at library funktionerne bag vores getchar() og putchar() er ok.

```
MITPROMPT$ filter0 < /usr/dict/words > words.cpy
MITPROMPT$ cmp /usr/dict/words words.cpy
MITPROMPT$ echo $?
           0
MITPROMPT$
# $? er statuskode eller exitcode, som indikerer om
# der er fejl eller ej. 0 betyder ingen fejl, ingen forskelle.
```

Man kunne tro, at det er uhyre ineffektivt at læse en stor fil et bogstav af gangen, men det er det ikke, hvis vores run-time library ellers er blot nogenlunde godt skrevet. For det første skal det nævnes at getchar og putchar er *macroer* , derved spares et funktionskald.

For det andet kan man med omdirigering på kommandolinjen bruge sådanne simple programmer til at behandle filer. Det er altså ikke nødvendigt at sidde og taste data ind til sine eksempler.

For at forstå den klassiske konstruktion ((c=getchar()) != EOF) er det en god idé at erstatte (c=getchar()) med c, variabelen, som indeholder det læste bogstav. Så står der:

```
    while (c != EOF) {
       ...
```

Et assignment (en tilskrivning, på lokalsproget) har en værdi, nemlig værdien af den sidst foretagne tildeling. Her er der kun en tildeling i udtrykket, så det er meget nemt. Værdien af (c=getchar()) er c.

Læg også mærke til, at det er en int vi bruger til at gemme vores indlæste bogstav. Integer er en datatype, som er større end char. Derfor kan den rumme en værdi, som garanteret ikke er et lovligt bogstav, og det er den, som systembiblioteket indsætter, når der ikke er mere input. Det er som regel -1, og EOF skal være defineret i stdin.

2.4.1. Tegn og talværdier

Lad os først bruge programmet til at skrive bogstavernes talværdi ud i både almindelige titals - notation, i hexadecimal notation, og, hvis bogstavet er udskrivbart, som glyf, eller tegn-repræsentation.

Eksempel 2-14. Filter, som skriver ascii ordinal value.

```
/* char2tal.c */

#include <stdio.h>

int main()
{
    int c;
    while ( (c=getchar()) != EOF) {
      printf("Decimal-værdi %3d, hexadecimal værdi 0x%2x ", c, c);
      if (c > 31 && c < 127)
          printf("%c\n", c);
      else
          printf(".\n");
    }
    return 0;
}
```

Dette program producerer noget output, som indeholde alle de samme informationer som den oprindelige fil. Den indeholder endda flere informationer, skønt ikke så mange som den kunne.

Vi har med dette program *mappet* input til output på en måde, så vi ikke har tabt data. Vi kan skrive et program, som ud fra outputfilen rekonstruerer den oprindelige fil.

Der er også en anden sjov ting ved programmet: Hvis vi skulle sende data over et ustabilt transmissionsmedium, så ville det være nemt at finde de linjer, hvor data var gået tabt. I mange tilfælde ville man ved mindre fejl endda kunne regne det rigtige input ud. Denne form for overskud af bits til at repræsentere data kalde redundans. "Redundancy" oversættes i ordbogen til overflødighed, men vi kunne også oversætte det til "rigelighed", "med ekstrareservebits som forsikring ..." (Bedre forslag modtages med glæde!)

Som en biting til C++ interesserede kan det tilføjes, at man i C++ nok også ville slippe nemmere afsted i dette tilfælde ved at bruge printf. C++ cout() er ikke så nem at bruge til formatering af kolonner og lignende.

Lad os dernæst bruge programmet til at tælle characters.

Eksempel 2-15. Simpelt filter som character counter.

```
/* filter1.c simpelt filter tæller chars istf at ouputte. */

#include <stdio.h>

int nc;

int main()
{
    int c;
    while ( (c=getchar()) != EOF)
        ++nc;
    printf("%d\n", nc);
    return 0;
}
```

Hvorfor initialiseres nc ikke? Jo, for det er en global variabel, og den er garanteret zeroed out. Al global hukommelse, som ikke er explicit initialiseret, er garanteret en nulstilling. Man kan selv gøre det, i øvrigt, hvis man har en snavset buffer: memset(buf,0,lengde);

Et eksempel på en kørsel af programmet:

```
MITPROMPT $ time charcount < /usr/dict/words
409048
    0.32s real     0.27s user     0.04s system
```

På min gamle maskine, 3/10 sekunder til at læse 400KB! Det er pænt (i forhold til maskinens formåen). Den "officielle" wordcount, wc < /usr/dict/words er dog 10 gange hurtigere!

Husk at denne charcount ikke tæller bytes, men bogstaver. På en linux maskine er dette normalt det samme. På et MicroSoft system vil længden af filen ikke svare til antal characters talt på *denne* metode, fordi MicroSoft operativsystemer (og andre) benytter carriage-return line-feed sekvenser til linjeskift. Når man kører "normal C" - eller simpel, POSIX-C - på en platform som MS-OS'erne, så filtreres alle cr - tegn fra.

For at se alle ascii koder kan du benytte man kommandoen:

```
PROMPT $ man ascii
[...]
PROMPT $ man groff-char
```

2.4.2. Et typisk filter

Et filterprogram, som svarer til den normale anvendelse af ordet filter, er et program som kan konverterer CodePage 865 til UTF-8. Det kan gøres på forskellige måder, men eksemplet egner sig godt til at demonstrere, hvordan man "mapper" en datamængde over i en anden.

Lad os for simplicitetens skyld gå ud fra, at hvert bogstav er en byte. (Desværre er jeg ikke klar over, om iso8859-1 altid er en byte, men det går vi altså ud fra, at de er.) Vi skal med andre ord kunne konvertere en byte til en anden byte. Antallet af mulige værdier i begge lejre er kun 256.

Den hurtigste teknik til en sådan opgave er derfor tabelopslag, som kan svare på, hvilken ny værdi vores byte skal have ved at bruge vores input som index i et array med 256 værdier.

For at prøve teknikken lader vi først en loop initialisere arrayet først, således at der ingen konvertering ville finde sted. Derefter prøver vi, om vores program kopierer input til output *uden* ændringer.

Eksempel 2-16. Codepage 865 til iso8859-1, forstadium.

```
/* ibm2iso0v1.c, forstadium til konverteringsprogram
 * Codepage 865 -> iso8859.1
 */

#include <stdio.h>

static char conv[256];

int main(int argc, char *argv[])
{
    int c;
    int jj;
```

```
    for (jj = 0; jj < 256 ; ++jj)
        conv[jj] = jj;

    while ( (c=getchar()) != EOF)
        putchar(conv[c]);

    return 0;
}
```

Somme tider klager folk over, at C programmet ikke har boundary tjek for arrays. Det er der selvfølgelig den gode grund til, at det ville være spild af tid i et godt program. Hvorfor vil ovenstående program ALDRIG gå ud over grænserne fra 0 til 256?

Hvordan tester man nu sådan et program? Her er Unix suverænt, man omdirigerer input fra en fil og samler test output op i en anden fil. På den måde opnår man, at selve test-proceduren kan automatiseres (evt. sættes ind som et target i en makefile.)

```
fri2c:  gcc -Wall ibm2iso0v1.c -o ibm2iso0v1
fri2c:  ibm2iso0v1 < textfile > textfile.kpi
fri2c:  cmp textfile textfile.kpi
```

Den næste terrasse, som vi vil nå op på, skal konvertere et enkelt tegn og lade resten være. Vi benytter derfor stadig loopen, men tilføjer nu en linje, som ændrer en enkelt af tabelværdierne, og ser, om vi får det ønskede resultat, hvis vi sender input af den pågældende værdi. (Det gør vi selvfølgelig).

Eksempel 2-17. Terrasse 2

```
/* ibm2iso0v2 - forstadium 2, konvertering af en enkelt værdi. */

#include <stdio.h>

static char conv[256];

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

    for (jj = 0; jj < 256 ; ++jj)
        conv[jj] = jj;

    conv['A'] = 'a';                  /* Ja, man må gerne! */
```

```
    while ( (c=getchar()) != EOF)
        putchar(conv[c]);

    return 0;
}
```

Når man kører ovenstående program (fra kommandolinje, med inddata fra tastaturet) skal man få lille a hvis man taster store a. Ellers skal alt output være det samme som input.

Den version af programmet, som vi slutter med her, behøver ikke at være den endelige version. De mange sjove grafiske tegn, som findes i codepage850 og 865 kan man jo forestille sig efterlignet på mange måder, afhængig af, om man skal ud på en printer eller en ascii skærm eller en grafisk tegneflade (stort arbejde!). Vi nøjes med en version, som konverterer de danske tegn i IBM-tegnsættet codepage 850 til ækvivalenterne i iso8859-1.

Eksempel 2-18. Terrasse 3

```
/* ibm2iso.c - terrase 3, konvertering af danske tegn. */

#include <stdio.h>

static char conv[256];

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

    for (jj = 0; jj < 256 ; ++jj)
        conv[jj] = jj;

    conv[134] = 'å';              /* 134 er codepage850 for å */
    conv[143] = 'Å';              /* 143 er codepage850 for Å */
    conv[145] = 'æ';
    conv[146] = 'Æ';
    conv[155] = 'ø';
    conv[157] = 'Ø';
    conv[130] = 'é';
    conv[144] = 'É';
                                  /* fortsæt listen efter behov */

    while ( (c=getchar()) != EOF)
        putchar(conv[c]);

    return 0;
}
```

Den teknik, som er vist her ovenfor, er selvfølgelig lidt for nem. Hvis man vil have en lynhurtig service ved hjælp af en opslagstabel, skal tabellen selvfølgelig være initialiseret i forvejen. Det må man med andre ord skrive i hånden:

```
char ctab[256] = 0,1,2,3, /* nej ikke stop her! */
```

Det er den teknik, som anvendes i library funktionerne isprint(3), isdigit(3), toupper(3) osv. fordi det simpelthen er den hurtigste måde. Hver konvertering koster kun en enkelt indexerings-operation - især hvis funktionerne er erklæret som inline funktioner (findes i gcc og er med i den nye standard C99).

2.4.3. Oversætter - teknik, forstadium

Ud over de grundlæggende typer tekstfiltre, som vi har set på her, ville det være nyttigt at se et par eksempler på programmer, som kan filtrere mime-kodede postfiler.

Denne gang nytter det ikke at lave byte til byte mapning. Vi har som input sekvenser, der begynder med et lighedstegn og fortsætter med en hexadecimal talværdi for det ønskede output tegn. Fx. vil i "blåbærgrød" i Mime-kodning blive skrevet således: bl=e5b=e6rgr=F8d. Det er klart en forbedring at få det oversat til noget, som ligner de danske tegn noget mere. Desuden kan man bruge Mime-kodning til at angive et linjeskift, der ikke skal med i den færdige tekst. Meget rart, hvis man har problemer med lange linjer i for eksempel en mail transport agent. Det gøres ved at afslutte linjen med et lighedstegn.

Her kommer først en simpel version, som kun kan operere i en kanal (på engelsk en pipe), d.v.s. den læser fra stdin og skriver til stdout. (Med programmer til kommando fortolkeren bash eller ksh, kaldet shell scripts, kan man klare fil-opsætning for alle andre situationer, så det er såmænd ikke så ringe endda!)

Eksempel 2-19. mime afkodning, std.input

```
/* mime2asc.c - konvertering af mime - koder til ascii. */

#include <stdio.h>

static char str[3];

int main(int argc, char *argv[])
{
    int c, c2;
    char *ptr;

    while ( (c=getchar()) != EOF) {
        if (c != '=')
            putchar(c);
        else {
            if ( (c2 = getchar()) == '\n')  /* aha! linjen ønskes ikke brudt */
                continue;                   /* print ikke ny-linje-tegn */
            else {
```

```
                if (feof(stdin)) exit(1);
                str[1] = getchar();
                if (feof(stdin)) exit(1);
                str[0] = c2;
                c = strtol(str, &ptr, 16);
                putchar(c);
            }
        }
    }

    return 0;
}
```

Programmet er uhyre simpelt i forhold til en produktionsversion, men indeholder det nødvendige for at fremhæve pointen. Håndteringen af End-Of-File er klodset, og fejl ved konvertering af de to tegn efter lighedstegn håndteres slet ikke. EOF håndteringen kunne håndteres ved at benytte en linjebuffer og fgets(), og strtol(3) giver os et errno, som vi kunne tjekke på.

Vi benytter os af det faktum, at ethvert '=' skal forstås på en speciel måde, det indleder en mime sekvens. Det kan man kalde for et meta tegn, et tegn, som ikke blot er et tegn. Det er en kommando, på samme måde som '\' er en speciel kommando i en C string, '\n' betyder newline og ikke bogstavet n.

Hver gang vi støder ind i et tegn, undersøges, om det er et lighedstegn, og hvis ikke går det ufiltreret til output.

Lighedstegnet smides væk og de to næste characters puttes i en streng, der bliver opfattet som en hexadecimal værdi. Denne konverteres ved kald til strtol(3) string to long. Denne fornemme konverteringsrutine får en pointer til stringen med hexadecimal tallet, adressen på en character pointer, som den så kan bruge til at rapportere, hvor meget den kunne konvertere, og endelig et tal, som er radix for konverteringen, d.v.s. at hvis vi vil konvertere et almindeligt tal, skal vi aflevere et 10-tal her.

strtol(3) kunne godt undvære parameter 2, eller rettere, vi kunne fortælle den, at der ikke er nogen pointer ved at aflevere en NULL pointer. `strtol(str,NULL,16);`

strtol ville returnere et 0, hvis de efterfølgende bogstaver ikke kan konverteres, og det ville selvfølgelig være en situation, vi burde undersøge nøjere, ligesom errno, en static variabel i library modulet, burde tjekkes. Prøv at gøre det, og giv programmet input med fejl i og kontroller, at programmet håndterer det på en fornuftig måde!

Lad os til sidst forbede programmet, så det selv kan finde ud af at åbne filer, konvertere indholdet og skrive resultatet ud i en fil med et andet navn.

Eksempel 2-20. Mime afkodning, fil input

```
/* mime2ascii.c - fil konvertering af mime - koder til ascii. */

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

void konverter(FILE *infil, FILE *outfil);

int verbose;            // global flag

void error(char *message, int exitcode)
{
    fprintf(stderr,"%s\n",message);
    exit(exitcode);
}

#define MAXNAME 512

int main(int argc, char *argv[])
{
    FILE *fp, *nyfilp;
    char nyfilnavn[MAXNAME];
    char errormsg[MAXNAME+80];
    int j=0;

    while (++j < argc) {
        if (argv[j][0] == '-')
            switch(argv[j][1]) {
                case 'v': verbose=1;
                          break;
                default: error("Option ikke forstået", 40);
            }
    }
    j=0;
    while (++j < argc) {
        if (argv[j][0] == '-')
            continue;
        if (strlen(argv[j]) > MAXNAME - 2)
            error("Filnavn er for langt", 100);
        fp = fopen(argv[j],"r");
        if (!fp) {
            sprintf(errormsg, "Kan ikke åbne fil ved navn %s",argv[j]);
            error(errormsg,2);
        }
        strcpy(nyfilnavn,argv[j]);
        strcat(nyfilnavn,".X");
        nyfilp = fopen(nyfilnavn,"w");
        if (!nyfilp)
            error("Kan ikke åbne fil for skrivning",103);
        if (verbose)
            fprintf(stderr,"Har åbnet %s, starter konvertering\n", argv[j]);
```

```
        konverter(fp,nyfilp);
    }
    return 0;
}

void konverter(FILE *fil, FILE *outfil)
{
    int c, c2;
    char *ptr;
    static char str[3];

    while ( (c=getc(fil)) != EOF) {
        if (c != '=')
            putc(c,outfil);
        else {
            if ( (c2 = getc(fil)) == '\n')  /* aha! linjen ønskes ikke brudt */
                continue;                   /* print ikke ny-linje-tegn */
            else {
                if (feof(stdin)) exit(1);
                str[1] = getc(fil);
                if (feof(stdin)) exit(1);
                str[0] = c2;
                c = strtol(str, &ptr, 16);
                putc(c,outfil);
            }
        }
    }
}
```

En kommandolinje, som består af 3 ord, for eksempel mime2ascii fil1 fil2 vil blive opdelt i de tre ord, altså "mime2ascii" er det første, "fil1" er det andet, "fil2" er det tredie ord.

argc er sat til 3 af den startup kode, som sætter vores program i gang. argv[0] er derfor ordet "mime2ascii", det vil sige vores eget programs navn. Da vores program *altid* har et navn, er argc altid mindst 1. Man gemmer sommetider programnavnet i en global character-pointer, fx. således:

```
char *thisprog;
main(int argc, char *argv[])
{
    thisprog = argv[0];
    /* mere program ... */
```

Hvis man skal have fat i sidste ord på kommandolinjen er det altså `argv[argc-1]` - *Husk,* at array elementer starter med 0, d.v.s. første element er argv[0].

For at vise, hvordan main får argumenterne fra kommandolinjen, kan man lave en "terrasse1" af dette program, hvor man nøjes med at løbe argumenterne igennem og skrive dem ud på skærmen.

Læg mærke til, at programmet gennemløber kommandolinjen 2 gange. Første gang er det blot for at få de eventuelle options. Vi har lavet en enkelt optionmulighed, nemlig -v for verbose. Det fungerer - men det er en meget rå form for kommandolinje analyse. Vi vil i andre eksempler vise flere måder at gøre dette.

Alt i alt er mime2ascii dog betydeligt mere praktisk end forgængeren. Det læser de filer, som man angiver på kommandolinjen, og skriver det konverterede indhold ud i en fil med samme navn med en tilføjelse af ".X".

Det kunne lige så godt være en anden endelse, og det står dig selvfølgelig frit for at ændre programmet til at lave filer, som ender på **.txt**. Nogle gange vil man måske ønske, at man sletter den oprindelige fil (eller gemmer den under et andet navn) og omdøber (renamer) den konverterede fil til det oprindelige filnavn. Dette er en god ide, hvis mime2ascii aldrig laver fejl! Det vil være en god opgave til programmeringsarbejde at polere dette filter, således at det fungerer perfekt og selv genererer fornuftige filnavne.

I afsnit Afsnit 2.5 udvidedes denne programtype, så den kan håndtere lidt længere sekvenser af tegn, således at man kan analysere input på et lidt højere abstraktionsniveau.

2.4.4. Summering af tal i en fil.

Et program, som ønsker at behandle linjer i st.f. characters, kan optimere IO ved at benytte fgets(2). Vi kan benytte denne funktion til at skrive et program, som læser en fil med tal og lægger dem sammen og skriver resultatet. Det er fascinerende at se et program, der kan behandle en megabyte datafil på brøkdele af et sekund.

Allerede nu kan det forudses, at hvis den kan "forstå" negative tal også vil kunne trække fra.

Der findes situationer fra det virkelige liv, hvor sådan et program kunne være nyttigt. Hvis vi f.eks. har foretaget et udtræk fra en stor database med alle telefon taksttelegrammer fra lørdag 24.00 til søndag 06.00, så kan vi beregne den samlede tid og hvor meget det ville koste at give natterabat. Men også datafiler fra alle mulige andre situationer ville kunne være input. Normalt vil man på Unix klare den slags med awk, men hvis man skulle optimere (f.eks. p.g.a. store datamængder fra en telefoni-data), så kunne det blive aktuelt at skrive det rå C program.

Pseudokode er måden at programmere på, hvis man ikke er interesseret i sprogets finurligheder, men blot ønsker at forklare mekanikken i et program. Vores fil-additionsmaskine vil i pseudokode se ud nogenlunde sådan her:

```
Så længe der er linjer,
  læse næste linje,
  konverter linjen til et tal, hvis muligt
     er det ikke et tal, så vis linjenr, linje og gå ud;
  læg tallet til totalen.
Print totalen.
```

Programmet er så simpelt, at vi skriver det i et hug.

Eksempel 2-21. Filter som konverterer linjer med tal til summen af tallene.

```
/* summer.c */

#include <stdio.h>
#include <stdlib.h>   /* for string to double, strtod() */
#define MAXL 80000
int main()
{
    char line[MAXL];   /* for input */
    char *ptr;         /* for strtod konverteringspointer */
    double tal;        /* for det laeste tal */
    double sum = 0;    /* for totalen */

    while ( (fgets(line,MAXL,stdin)) != NULL) {
        tal = strtod(line,&ptr);
        sum += tal;
    }
    printf("%18.2f\n",sum);
    return 0;
}
/* end of file summer.c */
```

`strtod(3)` (STRing-TO-Double) er en funktion, som konverterer en string til double. Denne funktion er mere avanceret end atoi, i det den kan sætte en fejl-variabel, hvis konverteringen ikke lykkes, og den kan flytte en pointer hen ad tekst strengen til det første bogstav, der ikke kunne konverteres. Man kan også bruge den på en mere simpel måde; man giver den blot NULL som anden parameter, og krydser fingre og siger: det skal nok gå alt sammen ...

2.5. Analyse af input

Er et oversætterprogram et filter? På en måde ja. En bestemt slags input giver altid et bestemt output. Output er afhængigt af (ét) input.

Tag nu som eksempel følgende to små programmer. De vil, hvis vi oversætter dem til statisk linkede, strippede filer, være helt ens. I de færdige programmer vil der ikke være noget ord "alfa" eller "beta" - det er blot et midlertidigt navn for nummeret på den memory - celle, hvor oversætteren anbringer vores tal. Prøv selv med kommandoen **strings prog1** .

```
/* prog1.c - lav om på variabelnavne og oversæt: gcc prog1.c -s -o prog1 */
int alfa 112233;
main(){ printf("min globale variabel har værdien %d\n", alfa); }

/* prog2.c ligner prog1 */
int beta 112233;
```

```
main()
{
    printf("min globale variabel har værdien %d\n", beta);
}
```

Selv om vi ændrer variables navne og laver om på linjedeling, kommentarer mv. får vi nøjagtigt samme maskininstruktioner ud af det, med andre ord, der er en entydig bestemmelse af ouput ud fra input. Men for samme output kan vi altså have flere forskellige input.

Kan et program analysere input? Det kommer an på, hvad man mener med analyse. Hvis man forventer en forståelse, så nej, men hvis analysen giver sig udslag i, at en struktur i input oversættes til en anden struktur, så jo, så kan et program analysere input.

Der sker en transformering af input, og den er bestemt af regler. I en parser er det reglerne, som er de mest interessante. Når vi ønsker at bygge en oversætter, så skifter vores fokus fra de laveste, små input enheder, d.v.s. bogstaver og tal, til større enheder, sætninger, blokke og funktioner.

Der er mange teknikker til at skifte fokus fra de laveste inputenheder, characters og words, til højere niveauer, blokke mv. En af mere taknemmelige metoder er den rekusivt nedstigende parser, som opbygges af et hierarki af funktioner, hvor de øverste tager sig af de store linjer, og de nederste i call-hierarkiet tager sig af de mindste, syntaktiske enheder.

Den rekursivt nedstigende parser (recursive descent parser) er en attraktiv metode for både begynderen og den viderekomne programmør. Bjarne Stroustrup fortæller i sin bog "The Design and Evolution af C++" at han oprindeligt ville bruge en recursive descent parser til Cfront, den første C++ compiler. " [...] it *is* possible to write an efficient and reasonably nice recursive descent parser for C++. Several modern C++ compilers use recursive descent." Stroustrup[1], p. 69.

For begynderen er det imidlertid en stor mundfuld at forstå en fuld-skala rekursiv nedstignings parser, men det klares, som bekendt, ved at dele opgaven op i små bidder. Intet problem er så stort, at det ikke kan deles i flere mindre. Hvem har sagt det?

I de næste eksempler vil vi derfor opbygge en tag-parser efter terrasse-princippet, d.v.s. at vi lægger ud med at løse en beskeden del af opgaven. Først når den virker, kommer vi yderligere features på. Man er nødt til at skrive nogle simple inputfiler for at teste programmet, men det kan nu være ganske fornøjeligt at se, hvad der sker, når man "leger" med input.

2.5.1. En tag - parser

Når man skriver tekst i sgml format, så vil det være rart, at man kan få hjælp til at kontrollere sine tags. Det program, jade [12], som fremstiller html, postscript, pdf-filer etc.etc. ud fra vores kildetekst, skal selvfølgelig også kunne tjekke for, at man har anvendt tags på den rigtige måde. Men inden man kommer dertil, kan det somme tider være rart med et mindre program, som tjekker, at man har husket alle

end-tags, og endnu bedre, et program, som kan formatere lidt på afsnit og indrykning, så kildeteksten er lettere at læse. Man kunne så fortsætte med at lave et program, som ville udskrive en formateret ascii udgave.

Et sådant program kunne for eksempel få flg. input:

```
<chapter><title>Dette er kapitel 1.</title> <para> Dette kapitel
handler om de teknikker, som en programmør kan bruge til at
analysere input.  </para> </chapter>
```

Når vi har kørt input gennem vores tag-kontrolprogram, forventer vi, at output skal se ud nogenlunde som flg.:

```
<chapter>
  <title> Dette er kapitel 1.
  </title>
  <para> Dette kapitel handler om de teknikker, som en programmør
         kan bruge til at analysere input.
  </para>
</chapter>
```

Hvis der mangler en end-tag, skal programmet komme med en fejlmeddelelse. Programmet kommer (forhåbentlig) til at ligne indent programmet, som forskønner C - kildetekst ved at sørge for systematisk indrykning og placering af kommentarer m.v. (se Afsnit A.3.)

Man kan diskutere detailler omkring formatering, men i første omgang vil vi blot have den simplest mulige løsning. Og ikke nok med det, den første version af programmet skal blot afprøve IO mekanismerne.

Hvis vi skal analysere tags, skal vi kunne se mindst et ord af gangen. Det er heldigvis meget nemt, idet vi kan nøjes med en linjebuffer. Et ord vil aldrig være delt hen over en linje. Orddelinger på grund af linjelængde forekommer *ikke* .

Vores linjebuffer skal være stor nok, selvfølgelig, og vores program skal reagere fornuftigt på buffer overflow, derfor bruger vi fgets(), som stopper, når bufferen er fuld.

For at se på det næste bogstav skriver vi en lille funktion, ch(), som blot returnerer det næste tegn i vores linjebuffer. Men når vi læser og fremrykker pointeren, så skal der selvfølgelig tjekkes på, om der er et gyldigt tegn forude, hvis der ikke er, kaldes fillbuf.

I stil med Eksempel 2-19 kan vi dog lige tilføje en lille feature til dette program, nemlig, at hver gang, der ses en kile, d.v.s. < ofte kaldet mindre-end tegnet, konverteres til en sgml-kode, "<". Det har vist sig at være et nyttigt program, når man vil inkludere for eksempel en stump af et C-program i en sgml kildetekst.

Eksempel 2-22. En tag - parser, forstadium

```
/* sgmlfmt_pre1.c Forstadium til mini program, som tjekker balancen i
 * sgml tags. I den nuværende skikkelse kan programmet erstatte
 * alle "mindre-end" tegn med sgml-koden for dette, altså &lt; -
 * så det kan såmænd bruges til at filtrere C-programmer, der
 * skal includeres i en sgml-tekst. */

#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <errno.h>
#include <error.h>

int status;
int eofile;                        /* global end of file */

#define MAXB 8000

char buf[MAXB];
int endbuf;
int bufindex;

int init();
int fillbuf();
int ch();
int gch();
int parse();

int main()
{
    if (init() == 0)
        return 1;
    while (!eofile) {
        if (ch() != '<')
            putchar(gch());
        else {
     printf("&lt;");
     gch(); /* discard character '<' */
        }
    }
    return 0;
}


int init()
{
    /* insert initialization of global vars here */
    return fillbuf();
}

int fillbuf()
{
```

```
    char *rv;

    if (!(rv = fgets(buf, MAXB, stdin)))
 strcpy(buf, "");
    endbuf = strlen(buf);
    bufindex = 0;
    return (int) rv;
}

/* ch() returnerer den næste character i input stream *men* læser
 * ikke, og ændrer ikke noget i den eksisterende buffer. Det
 * svarer til at spørge "Hvad er det næste input tegn, hvis vi nu
 * gad gå videre?!"
 */

int ch()
{
    return buf[bufindex];
}

int nch()
{
    return buf[bufindex + 1];
}



/* Hvis vi altid vil kunne se mere end én character fremefter, må
 * vi udskifte gch() og fillbuf() funktionerne, således at de
 * tjekker, hvor meget er der i bufferen, og hvis der er for få
 * (mindre end ønsket) skal de efterfylde bufferen og justere
 * pointere.
 * Det søde ved denne implementering er imidlertid, at vi altid
 * har én character lookahead (garanteret), men hvis vi ser på et
 * ord, kan vi se hele resten af ordet, fordi et ord ikke kan
 * krydse en linjedeling.
 */

/* gch() returnerer samme som ch() men flytter pointeren en plads
 * frem. Hvis vi ved fremadrykning rammer end of line (en
 * nul-byte) må vi fylde bufferen, så ch() næste gang har et tegn
 * at kigge på.
 */

int gch()
{
    int c;
    if (nch() == 0) {
 c = ch();
        if (!fillbuf()) {
            eofile = 1;  /* will take effect for the next char */
        }
 return c;
```

```
    }
    return buf[bufindex++];
}
```

Programmets main kunne såmænd udmærket nøjes med at benytte getchar og putchar. Der er ikke noget vundet i *dette* program ved at benytte en særlig input mekanisme. Men det, som er det egentlige formål med programmet, er jo også at bygge grundbestanddelen til et andet. Det er "terrasse 1", og hvis det virker og kopierer input til output (med undtagelse af < tegn), så er grunden lagt til næste udgave af programmet, som blot skal tjekke, om en tekst har balancerede tags.

Funktionerne ch() og gch() *administrerer* bufferen. Andre funktioner i programmet kigger ikke direkte i den globale variabel "buf[MAXB]". Det kunne være en privat variabel, som lå i et IO-modul. (Prøv at ændre programmet på den måde.)

En af de ting, der drillede ved konstruktionen af programmet, var, at gch() skal fylde bufferen - den er jo ansvarlig, klart nok, for at der er en næste byte at læse. gch() er den eneste, som kalder filbuf. Men hvornår skal den fylde bufferen? Hvis buffer indexet peger på newline eller en streng-slutning (null-byte), så kunne det være signal til at fylde bufferen op. Da det alligevel blot er en newline, så kunne man smide den væk.

Men det er ikke en god idé (her) - for at det skal kunne lade sig gøre at gengive linjeskift i for eksempel litteral tekst er det nødvendigt at lade gch() kigge en byte fremad.

Så man skal altså huske at "gemme" den sidste byte i bufferen i en variabel, og derefter fyldes bufferen igen.

Derved opstår der et andet problem. Hvornår rammer man end of file. Hvis caller af gch() tester på end of file INDEN han bruger den character, han har fået med gch(), så vil der - i denne version - gå kage i systemet.

I dette lille program er det imidlertid overskueligt at huske, at man skal anvende returværdien fra gch() inden man tester for end of file (eller teste for end of file inden man kalder gch()).

Input mekanismen, som den er præsenteret her, er en forsimplet udgave af input mekanismen i small-C compileren, se Afsnit 4.3.

Programmet anvender IKKE pointere. I stedet indexeres array'et med variabelen bufindex.

2.5.2. Tjek balancering af tags

I den næste version af tag-parseren, tagbal01.c, kan man se, hvordan input mekanismen er bevaret.

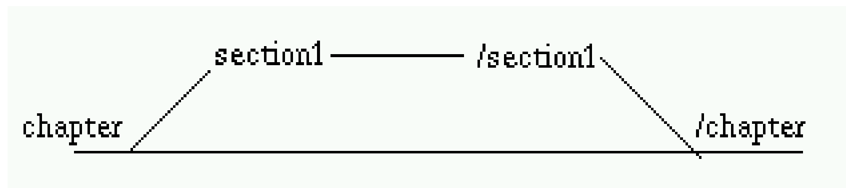
Programmet er ikke gengivet her i sin fulde udstrækning, men ligger som en fil i kataloget med eksempler.

Opgaven består denne gang i at skrive et program, som kan læse en tag og "huske den", indtil vi kommer frem til næste tag. Hvis den næste tag er en end-tag (med skråstreg, for eksempel </chapter>) så skal teksten *efter* skråstregen være samme ord, som vores start tag. Hvis det er en ny start tag, fx. <section1>, så skal den accepteres, og programmet leder nu efter en end-tag for section1.

Vi kan beskrive flow'et på flg. måde:

```
så længe der er en tag <navn>
   hvis den næste tag er en start tag
       kald rekursivt
   ellers
       hvis det ikke er en end-tag, som matcher <navn>
           meld fejl
       ellers returnér ok
```

Når vi har fundet en end-tag, som matcher start-taggen, skal vores funktion altså returnere true (eller ok). Denne simple version skriver m.a.o. ikke noget output.



Den funktion, som skal klare chapter-/chapter, kan selvfølgelig være lavet sådan, at den *kun* leder efter chapter. Men da mekanismen er den samme, når vi leder efter section eller para, så er det nok bedre at lade vores funktion acceptere et hvilket som helst tag navn, og så gemme det i en character buffer. Derved kan vi bruge den samme mekanisme for både chapter, section1 og alle mulige andre.

Programmet kunne i simplest mulige form simpelt hen se sådan ud:

```
scan_for_tags()
{
    char newbuffer[MAXT];
    int j = 0;

    if (ch() != '<')
        parse_error("Need tag here\n");
    // ok, der er en tag
    gch();
```

```
    if (ch == '/')
        parse_error("Cannot have end tag here\n");
    while ( ch() != '>' )
        newbuffer[j++] = gch();
    // Nu skipper vi al almindelig tekst indtil næste tag.
    while (ch() != '<')
        gch();
    // Nu stoler vi på, at vi kan se 2 characters frem
    if (!match("</")) {
        // ikke en endtag
        scan_for_tags();   // tag lige alle indskudte tags her!
    }
    // Ok, denne gang er det en end-tag:
    gch();gch(); // nu er vi fremme ved tag-navnet
    if (match(newbuffer)) {
       if (gch() != '>')
          parse_error("Need end-angle here\n");
       else
          return 1; // ok.
    }
    parse_error("tag-name non-match\n");
    // parse_error returnerer ikke, så vi kommer aldrig her, men
    // vil gerne gøre compileren glad...
    return 0;
}
```

Ovenstående funktion kalder sig selv, hvis den ser en ny start-tag efter at den er gået i gang med at lede efter en end-tag.

Det er en simpel version af rekursiv nedstigning. Der er flere forskellige ting, denne her parser ikke kan klare. Hvis man skriver sin tag hen over en newline (hvilket er lovligt) så vil parseren includere newline i tag-navnet. I det hele taget tager den alt, bogstavelig taget alt, inden slut vinkelen, og gemmer det som et tag-navn. Endvidere tjekker den ikke for buffer overflow, men det kunne man nemt indføje. Den er ikke desto mindre en god demonstration af, hvordan rekursionen kan gøre det muligt for os at håndtere mange forskellige slags input. Hvis der er 17 tags inden i hinanden, fint. Hvis vi først lægger 5 inden i hinanden og bagefter får 2 endtags og så igen en start-tag, fint, så klarer funktionen også at validere den slags input.

Det vil være en god øvelse at afprøve ovenstående. Det findes ikke som en fil i eksemplerne. Man kan klippe funktionen her og indsætte den i en kopi af sgmlfmt_pre1.c -- eller man kunne skrive hele programmet for at få lidt øvelse.

2.5.3. Any-tag

Lad os forfine ideen fra foregående eksempel. I stedet for at lede efter start-tag tegnet, så overlader vi alt det beskidte arbejde (inclusive håndtering af linjeskift og spaces) til en funktion, som vi vil kalde anytag(). Derved bliver programmet meget lettere at læse.

Hvis funktionen anytag() ser en tag, fint, så gemmer den navnet i den buffer, den har fået og returnerer true. Hvis den ikke finder en tag, så returnerer den blot false. Det må vores parser så benytte sig af. Her kommer pseudo-koden:

```
hvis der er en ny starttag <navn>
    så længe der er mere input
        lad tekst data passere
        hvis der er en end-tag
            begynd forfra (d.v.s. se efter en ny starttag)
        ellers skift niveau (d.v.s. rekursivt kald)
ellers
    returner false
```

Opmærksomheden kan nu koncentrere sig om, hvordan man scanner for en tag med tilhørende slut-tag. Ovenstående pseudo-kode svarer til funktionen parse_level1. Navnet giver en forudanelse om, at man senere kunne have forskellige levels i parseren, som hver især er udtryk for, at der gælder forskellige regler for forskellige tags. Men her bliver de alle behandlet ens. Der er ikke nogen regler for, hvilke tags der må være inde mellem andre tags. Programmet skal blot kunne tjekke, at der efter en <tag>, kommer en </tag> (altså en slut-tag af samme type) senere, evt, efter nogle andre, indskudte (nestede) tags og med "tekst data" både før og efter.

For at få tag'en analyseret kalder parse_level1 den dertil indrettede funktion anytag(), som afleverer navnet eller keywordet på taggen i den buffer, som den får adressen på. Parse_level1 gemmer navnet på taggen for senere at kontrollere, at den tilsvarende end-tag har samme navn.

parse_level1 accepterer nu ord ved at kalde getword. Typisk ville det være efter en para-tag. Når getword støder ind i et '<'-tegn stopper den. Når den returnerer 0 er det fordi den er stødt ind i en ny tag (eller end-tag) -- eller fordi der ikke er mere input.

Nu kommer det spændende! Hvad må der komme efter? På dette sted i input kan der komme en end-tag eller en ny tag. Hvis det er en end tag, har vi sluttet ringen og bør gå et niveau ned og se, om der er mere tekst efter end-taggen. Det gøres ved at breake og hvis der ikke kommer en ny tag returneres fra parse_level1.

Hvis den nu har kaldt sig selv, så er vi ikke på det yderste niveau. Der returneres til stedet lige efter det rekursive call, og programmet begynder forfra på while-loop(2). Herved vil det igen ser efter med getword, om der er "tekst-data". Men altså på et niveau lavere. (Det kan ses på indrykningerne i output).

Her kommer det centrale sted i programmet tagbal01.c:

Eksempel 2-23. Tjek om sgml-tags balancerer

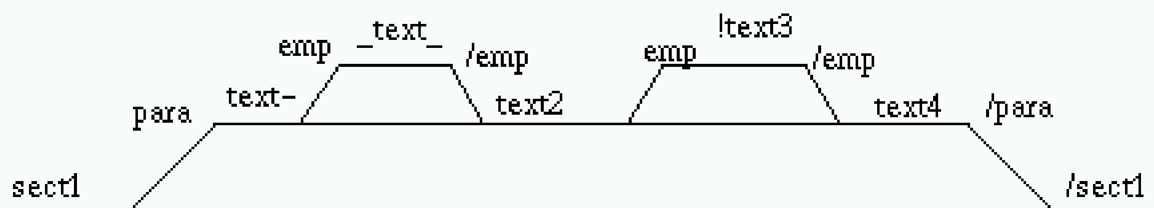
```
#define MAXT 80
```

```
int parse_level1()
{
        char tagname[MAXT];
        while (!eofile) {                               /*(1)*/
                if (!anytag(tagname)) {
                        return 0;
                }
                while (!eofile) {                       /*(2)*/
                        while (getword())               /*(3)*/
                                putword();
                        if (have_endtag(tagname))
                                break;
                        (void) parse_level1();  /* nested tags */
                }
                blanks();
        }
        return 1;
}
```

Efter en nested tag kan der komme mere tekst (som efter emphasis tag fx.) Man skal derfor spørge efter flere words når man kommer tilbage fra rekursivt kald til parse_level1(). Det er derfor, at der forrest i while-løkke (2) er et tjek for tekst-data, nemlig getword() (3).

Endvidere kan der jo komme en ny tag-konstruktion lige efter en komplet cyklus *på samme niveau*. Derfor må man *ikke* returnere, hvis have_endtag() returnerer true; altså hvis vi finder en end-tag, så skal vi se efter, om der er en start-tag mere på samme niveau. Hvis der *ikke* er det, *så skal der returneres* .

Nøglen til forståelse er at følge flowet og forestille sig input i stil med flg.: <sect1> <para> text-text <emphasis> _text_ </emphasis> text2 <emphasis> !text3! </emphasis> text4 </para></sect1>



Indsæt printf() statements forskellige steder i programmet og afprøv med små input filer. Sæt ord mellem para-tags. Prøv at undersøge, hvad der sker, hvis man ikke ignorerer return value i det rekursive kald til parse_level1().

Det vil være en god øvelse at tjekke, om der i programmet er mulighed for buffer overflow nogen steder.

2.5.4. Skelnen mellem tagtyper

tagbal02.c er bygget op nøjagtigt som tagbal01.c, men kan skelne mellem et par grundlæggende typer sgml-tags, nemlig comment, programlisting og literal. Den kan også anvende en speciel regel for programlisting og literal, tekst data imellem dem bliver leveret videre som den er, uden formatering eller ombrydning af linjer.

Eksempel 2-24. Udvidelse af parse-level1 med kendte tag-typer

```
int parse_level1()
{
    char tagname[MAXT];
    while (!eofile) {
        blanks();
        if (comment()) {
            continue;
        }
        if (programlisting(tagname) || litt(tagname))
            (void) do_litteral(tagname);
        else if (!anytag(tagname))
            return 0;
        while (!eofile) {
            while (getword())
                putword();
            if (have_endtag(tagname))
                break;
            (void) parse_level1();      /* nested tags */
        }
        blanks();
    }
    return 1;
}
```

Find funktionen parse_level1() og sammenlign med tagbal01.c. Nu er det ikke kun anytag, den kalder. Der er kommet special-funktioner til, som håndterer henholdsvis kommentarer, programlisting og litteral-tags. De er bygget op som anytag(), de returnerer false, hvis de ikke ser den tag type, som de er "sendt ud for at lede efter".

Klart nok må man så vente med at kalde anytag() til efter at de andre har fået chancen. Anytag() accepterer jo hvad som helst, der er en tag, så andre funktioner ville overhovedet ikke blive kaldt.

I en sgml parser kunne anytag() benytte en liste over tilladte tag navne; hvis en tag ikke stod i denne tabel, så var den ulovlig og programmet skulle stoppe med en fejlmeddelelse.

Et eksempel på kørsel af programmet, med en lidt forsimplet form for sgml-tags, kommer her.

```
dax@pluto: cat lidt.sgml

<chapter>
<sect1 id="tag_balance_tjekk">
<para>
Dette       er
en prøvetekst.
</para>
<para>
mere tekst i ny paragraf
<footnote><para>
fodnote text kommer her.
</para> </footnote>
</para>
<programlisting>
  while (++x < 10)
   printf("Hello!\n");
</programlisting>
</sect1>
</chapter>
<!-- her kommer en kommentar -->
<azerty> Programmet klager ikke over en ulovlig tag.
</azerty>

dax@pluto: tagbal02 < lidt.sgml
<chapter>
  <sect1 id="tag_balance_tjekk">
    <para> Dette er en prøvetekst.
    </para>
    <para> mere tekst i ny paragraf
      <footnote>
        <para> fodnote text kommer her.
        </para>
      </footnote>
    </para>
    <programlisting>
                 while (++x < 10)
                         printf("Hello!\n");
    </programlisting>
  </sect1>
</chapter>
<!-- her kommer en kommentar -->
<azerty> Programmet klager ikke over en ulovlig tag.
</azerty>

dax@pluto:
```

Som det kan ses, beklager programmet sig ikke over, at der forekommer en ulovlig tag sidst i input. Programmet formaterer teksten, ikke så avanceret, men nyttigt nok, og foretager indrykning, hver gang der forekommer en tag er inde i en anden tag.

Hele programmet ligger som en fil i eksempelsamlingen, tagbal02.c.

Forslag til øvelse: Dette program kan med få modifikationer benyttes til at trække teksten ud af en sgml fil, og endda i en læselig formatering. Overvej, hvilke forbedringer, som kunne gøre denne tekstudtrækning - ascii formatering - endnu mere nyttig. (Jeg har ikke nogen løsning på denne øvelse - endnu!)

2.5.5. Tilstandsvariabel

Det er lidt af en provokation at kalde det næste program for et filter. Man kunne lige så godt kalde det en parser eller en tilstandsmaskine. Det er også en lexical analyzer. Men strengt taget er det også et filter. Der kommer noget input ind fra (kun) én kilde, og afhængigt af dette spytter programmet noget andet ud. Der er en årsagssammenhæng fra input til output. Det samme input vil altid give det samme output. Men det går kun den ene vej. Et bestemt output kan produceres af mange forskellige slags input.

I modsætning til de tidligere eksempler er der i dette eksempel en variabel, som husker den tilstand, vi er i.

Og hvad er det så for et program? Et WORD count filter! Input er tekst, og output er blot antallet af ord. Provokation! Er det virkelig et filter? Ja, for det opfylder jo alle de ovenstående krav. Det sluger input fra en kilde og omformer det til output, som, ok, kan være det samme for flere forskellige slags input, men samme input -- altid samme output. Vi baserer vores program på en lille bemærkning i Kernighan & Ritchies version af word countEksempel 2-27: Nemlig at begrebet "et ord" kan forfines, så man kan skelne mellem rigtige ord og tal, tegnsætning og lignende.

Hvis du ikke kender det grundlæggende wordcount program, så er if sætningerne i dette program næsten uforståelige. Se derfor Eksempel 2-27 hvis du ikke kender det.

Programmet her starter med at være i en tilstand, som vi kalder HVID. Når vi ikke har læst noget, så må vi være på HVIDt papir.

Vi skal benytte en variabel til at huske denne tilstand, og vi kalder den status.

For at gøre det lettere at læse programmet, lader vi denne variabel være af enum - typen. Så kan en god oversætter holde styr på, om vi tilskriver den andet end symbolske navne som tilhører typen. Dog kun med warnings.

Eksempel 2-25. Ord - tælling

```
/*file ordtael.c */

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

typedef long long _int64;

/* taellere: char, words, num, andet, lines */
_int64 nc, nw, nn, na, nl;

enum status_t { HVID, ALFA, TAL, PUNC };

enum status_t status;

int main()
{
    int c;
    while ( (c=getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            +nl;
        if (ispunct(c))
            ++na;
        if (isspace(c)) {
            status = HVID;
        } else {                                    /*(1)*/
            if (status == HVID || status == PUNC) {
                if (isalpha(c)) {                   /*(2)*/
                    status = ALFA;
                    ++nw;
                } else if (isdigit(c)) {            /*(3)*/
                    status = TAL;
                    ++nn;
                }
            } else if (ispunct(c)) {                /*(4)*/
                    status = PUNC;
            }
        }
    }
    printf("Antal: chr %Ld, ord: %Ld, tal %Ld, andet %Ld, lin. %Ld\n",
            nc, nw, nn, na, nl);
    return 0;
}

/* OBS: Der er lidt flere braces i ovenstående eksempel end
 * nødvendigt, det er i håb om bedre læselighed.
 */

/* end of file ordtael.c */
```

Det er statusvariabelen, som "husker", om vi er inde i et ord eller ej. Og det er statusvariablen, som gør det muligt at skrive programmet på en *forholdsvis* læselig måde Hvis opgaven imidlertid blev forsøgt løst *uden* statusvariabel, så ville der komme endnu flere if-sætninger. Inde i dem måtte man så lave nogle loops med læsning af input, så længe vi er i et ord. Og så bliver programmet fuldstændigt barokt kompliceret, prøv selv!.

(1) Hvis vi lige er har fat i noget, der er "ikke-space" og tilstanden stadig siger HVID, så betyder det jo, at vores tilstand lige netop nu ændrer sig. Derfor bør vi tælle antal af ord (eller tal, tegnsætning, etc.) op.

(2) Som ord tæller vi alt, hvad der begynder med bogstaver.

(3) Som tal tæller vi alt, hvad der begynder med cifre.

(4) Læg mærke til, at ETHVERT skilletegn udløser en tilstandsændring til PUNC, fordi denne sætning er sideordnet med den if, der står lige efter den (1) mærkede else.

Hvis indrykningen i dette program bliver smadret, så er det komplet umuligt at forstå meningen med det.

Programmet er udmærket til at danne sig et skøn over forholdet mellem tal og ord i en artikel.

Ordtael.c er i stand til at taelle et funktionskald som f.eks. qwerty(6000) som et ord, et tal og 2 andre tegn. Til gengæld tæller det 123Mb som et tal (uden ord).

```
MITPROMPT$ ordtael <<STOP
hej(42);
STOP

Antal: chr 9, ord: 1, tal 1, andet 3, lin. 1
```

Hvis vi skal forfine programmet, så er det klogt at skifte taktik. Løsningen ovenfor skalerer ikke godt, når antallet af tilstande vokser, og input indeholder mange flere kategorier. En tilstandstabel kan bedre håndtere opgaven. Se Kapitel 4.

2.6. Fejl og håndteringen af dem.

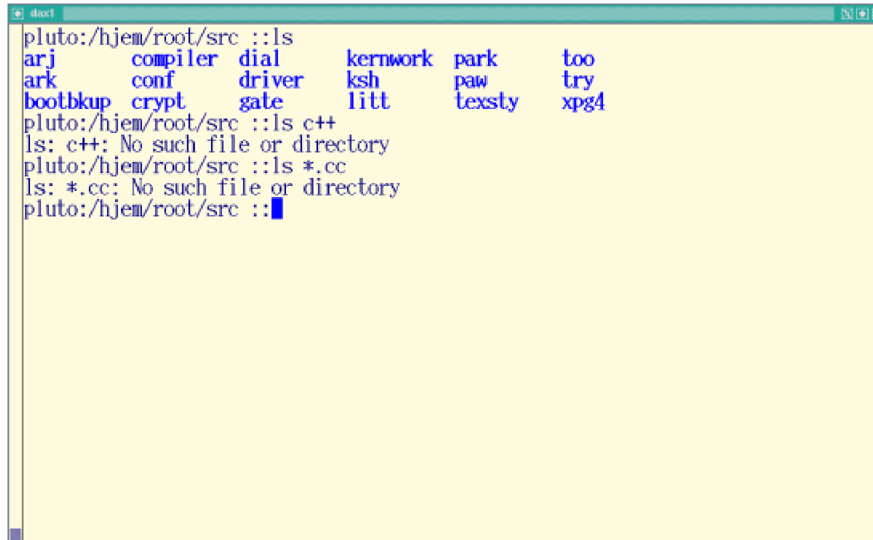
2.6.1. Hvilke slags fejl er interessante

Det er egentlig ikke så interessant, set fra en programmørs synspunkt, om en disk bryder sammen. Det kan man nemlig ikke rigtig gøre noget ved, når det er sket. Den, der skriver drivere til styresystemet, ville

måske nok kunne lave noget programmel, som forudsagde, at hardwaren trængte til service eller udskiftning. Det er indenfor mulighedernes grænser.

Hvis vi skriver et almindeligt filteringsprogram, f.eks. et, som konverterer fra MIME/html characters til extended ascii eller rettere UTF-8, så bør vi altså ikke begynde at lave tjek af CPU-temperatur, disktilstand etc. Det hører hjemme et andet sted og er ikke interessant for vores filterprogram.

Figur 2-1. Eksempel på fejlhåndtering, informativ besked



2.7. Flere små programmer og øvelsesforslag.

For at demonstrere styrken af små 10-24 linjers programmer ("Hallo-programmer") kommer her en serie sådanne små særlinge.

2.7.1. Tal formatering ved udskrift med printf.

Her kommer kildeteksten til et program, som kan skrive en listing af tallene fra 0 til 7, inverteret og som 2's complement. (Som lovet i Eksempel 1-2).

Eksempel 2-26. Tallene fra 0 til 7 i hexadecimal notation.

```
/* bitinvert.c viser hexadeximalt tallene fra 0 - 8  */
/* og desuden invertering, og såkaldt 2-s complement */
```

```
#include <stdio.h>


char *thisprg;

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

   thisprg = argv[0];

   for (jj=0;jj<8;++jj) {
     printf("Word: %08x, Inverted: %08x, Complement: %08x\n",jj,~jj,~jj+1);
   }
   return 0;
}

/* end bitinvert.c */
```

2.7.2. Word count - med tak til Kernighan & Ritchie

Her er en simpel variant af Kernighan & Ritchie's word count program. Det er en ultra simpel tilstandsmaskine. Den har kun to tilstande, enten er den i HVID tilstand, eller i SORT.

Hvis den er i HVID tilstand, har den læst et bogstav, som var whitespace, d.v.s. typisk space eller newline eller tab. Hvis den er i SORT tilstand, har den læst et tegn, som kræver tryksværte - eller rettere, et, som ikke er et whitespace tegn.

Programmet skelner altså ikke mellem ord og tal eller tegnsætning. Det anser bogstaver mellem to spaces for at være et ord! Så simpel en ord-tællingsmekanisme kan såmænd da være meget anvendelig i mange sammenhæng, f.eks. `echo * | ordtael` vil vise, hvor mange filer, der er i det aktuelle katalog. Det er grundlaget for det lidt mere avancerede program Eksempel 2-25 som kan skelne mellem ord og tal og skilletegn.

Eksempel 2-27. Simpel ordtælling

```
/*file wc.c */

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

int nc, nw, nl;  /* static vars garanteret 0 */

enum status_t { HVID, SORT };

enum status_t status;
```

```
int main()
{
    int c;
    while ( (c=getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (isspace(c))
            status = HVID;
        else                          /* <--- SE kommentar!!!*/
            if (status == HVID) {
                status = SORT;
                ++nw;
            }
    }
    printf("%d %d %d\n", nc, nw, nl);
    /* lav en version, der KUN taeller ord! */
    return 0;
}

/* end of file wc.c */
```

Læg mærke til, at programmet altid tæller characters, når der er læst en. Hvis det er en newline, så tæller vi linjer. Klart nok. Hvis det nu er en slags space (eller newline, tab, backspace, vertical tab, formfeed ...) så lader vi den gå i HVID-tilstand.

Den efterfølgende ELSE er kernepunktet for forståelse. Hvad ved vi, hvis vi havner i ELSE-sætningen? Jo, vi ved, at vores nys indlæste character IKKE var hvid.

Hvad ved vi så, hvis STATUS er lig med HVID, og vores nys indlæste bogstav var sort?

Joda, ... giv dig tid!

Så ved vi, at det foregående bogstav var hvidt (altså space) og at det nuværende er sort, altså, det første bogstav i et ord. Så derfor skifter vi tilstand til sort og tæller det nye ord med.

Så vores tilstandsvariabel `status` er meget praktisk, fordi den kan huske for os, hvad det foregående bogstav var, eller i bredere forstand, hvilken slags bogstav det var, og dermed hvilken tilstand vores tællemaskine var i.

Det er et program, som er sindssygt svært at forstå for en begynder, som heller ikke har erfaring med andre sprog. Sådan er det forresten altid når if-sætninger gror oven i hinanden. Hvis du har arbejdet dig igennem læsning af programmet og - bedre - har fået det til at køre, evt. med dine egne tilføjelser, så du kan se, hvad der sker undervejs, så har du taget et stort skridt fremad indenfor programmering (og logisk tænkning).

2.8. Øvelser

Skriv frame2 om til frame2a, hvor du selv angiver prototypen for puts, og se, om det oversætter og kører lige så godt som frame2.c, der jo bruger headerfilen <stdio.h>

2.8.1. Forslag til beregningsøvelser

Skriv procent.c om, så du også udskriver indholdet af variablene kroner og procent.

Skriv procent.c om, sådan at den beregner 99% af 100 millioner (jo jo, det kan man gøre i hovedet, men det er for at gøre det nemt at kontrollere resultatet!) Dette er ikke en helt triviel opgave.

Lav en version af programmet, som selv kan finde ud af, om der er overflow på beregningen. Hint: resultat / procent * kroner. For den viderekomne kan opgaven løses ved, at man benytter en assemblerinstruction, som tjekker for overflow flaget.

Slutbemærkning:

1. Hvis du ikke kender bogen "The C Programming Language" af Kernighan & Ritchie, og hvis du ikke er en øvet C programmør, så vil jeg anbefale, at du køber eller låner den og bruger nogle uger til at arbejde kapitel 1 igennem - lav så mange af de ekstra øvelser, som du kan nå. Nærværende kapitel er en ikke en erstatning for den oprindelige "tour", men et supplement. Man kalder det for learning by doing eller deduktiv spiralpædagogik; vi udleder, hvordan C sproget fungerer ved at prøve det mange gange og ved at gøre øvelser lidt sværere hver gang.

   Hvis Kernighan & Ritchie bogen også forekommer for vanskelig - den er nemlig heller ikke for helt grønne begyndere - så er der nogle andre introduktionsbøger, som kan guide dig igennem grundlæggende øvelser i programmering, f.eks. "Practical C" fra O'Reilly.

   Hvis du har lidt gåpå-mod, kan du måske alligevel klare dig ved at bladre/klikke dig om til appendiks A i denne bog; dér gives en oversigt over C sprogets fire forskellige bestanddele, datatyper, operatorerne, flow-konstruktionerne - og om opdeling af programmer i moduler.

   K&R bogen giver imidlertid flere eksempler end jeg har med i Appendiks A, og forklarer variationer på programmerne, variationer, som er rigtig gode til at få én igang med selv at forsøge. Det kan du selvfølgelig også gøre med eksemplerne i denne bog.

   Det specielle ved Kernighan & Ritchies programeksempler i Kapitel 1 af den berømte bog er, at programmerne er nyttige. På en kommandolinje kan de bruges med det samme til endda ret fornuftige og realistiske ting. Hvis du ikke er fortrolig med kommandolinjesyntaks, så kan du finde eksempler i bogen »Linux – Friheden til at lære Unix«. Det er en god idé at eksperimentere lidt med de simpleste Unix-programmer, inden du går videre. Prøv f.eks. date, cal, uptime, id, who, finger, "echo hej", strings /usr/bin/ls, file *, ls, pwd, cd, du, df, man gcc, man ld, man ld.so, ldd /usr/bin/ls osv. (klassiske unix kommandoer).

2. Tabellen Eksempel A-5, viser parenteserne *"()"* øverst, fordi bindingen mellem identifier og () er stærkere end bindinger mellem andre operatorer.

3. I Microsoft miljøer som "errorlevel", der kan bruges af if-sætninger i batch-filer.

4. Generer det, hvis jeg staver engelske computer-udtryk på engelsk?
5. Se "Friheden til at programmere" afsnittet om C sproget, hvis du har brug for lidt mere indføring i, hvordan man bruger kommandolinjen.
6. I glibc2x: se efter filerne `./sysdeps/elf/start.S` og `./sysdeps/generic/libc-start.c`
7. Parametre er oplysninger til en funktion.
8. Sørg for, at du virkelig ved, hvad ascii er for noget!
9. Spaghetti er en derogativ betegnelse for en lang, uoverskuelig liste med programmeringsinstruktioner. (Eller er det noget andet? ;-)
10. Det hænger sammen med, at der alle funktioner i de aller første C-oversættere returnerede en integer. Funktioner, som returnerer doubles er stadig i mindretal.
11. Reglen kan være meget praktisk for den erfarne programmør, som i visse situationer kan gøre et program lidt mere læseligt, fordi der er mindre "støj".
12. Jade - James' DSSSL Engine, James Clark har også skrevet groff programmet.

Kapitel 3. Sammensatte datatyper og hvad man kan med dem i C

Vis mig dine data, og jeg skal sige dig, hvad dit program vil kunne gøre (det ligner et citat :-) Når man arbejder med et problem, udvælger man mere eller mindre automatisk de oplysninger, som man mener er relevante for at løse opgaven. Omvendt er betingelsen for, at en operation eller beregning er mulig, den, at man har tilstrækkelig mange oplysninger til rådighed.

Som regel kan vi registrere flere oplysninger om vores problemstilling, end vi bryder os om at repræsentere i programmet. Ved at vælge fra og ved at anbringe oplysninger i en klar struktur kan vi opnå at programmerne bliver lettere at skrive og læse.

Sammenstilling og strukturering af operationelle data er langt den vigtigste disciplin for en programmør.

3.1. Sammensætning af flere oplysninger til en enhed

Når man sammensætter forskellige oplysninger i en klump, kalder man resultatet en struct (struktur, engelsk: structure). Gennem tiderne har denne gruppering af oplysninger fået mange betegnelser: node, record, entitet, genstand (engelsk: item), element. Den mest generelle engelske betegnelse er *aggregate data types.*

Da sådan en struct som regel forekommer som et element i en tabel, er der ifølge Knuth også forfattere, som har brugt betegnelsen "perler" om structs. De kan jo så trækkes på en snor.

Det er en af de vigtigste metoder til at opnå god programstruktur.

3.1.1. En *struct*

Sammensatte datatyper er nyttige, når vi i et program har brug for at samle informationer om et "objekt i den virkelige verden".

Eksempel 3-1. En struct

```
struct tomat_t {
   int typenummer;
   char artsnavn[80];
   int goedningsforbrug;
   int pladskrav;
   int temperaturkrav;
   int saesonpris[24];
};
```

Her har jeg forsøgt at beskrive en tomat, sådan som en handelsgartner ville gøre det. Han ville være interesseret i omkostningerne, som er sammensat af gødningsforbrug, udgifter til vanding og opvarmning, forrentning af drivhuse med videre, alt sammen ganget med varigheden af dyrkningsperioden.

Når man erklærer en struct efter ovenstående mønster, indsættes en oplysning i compilerens symboltabel om typens navn, størrelse samt offset og type på de enkelte elementer.

"tomat_t" kaldes en type tag, og den tjener to formål

- dels kan vi senere erklære flere variable af denne type;
- dels kan vi erklære en pointer til samme type inden i vores struct. Derved kan vi "trække dem som perler på en snor".

Ovenstående tomat_t kan altså fungere som typen på en variabel. Når man definerer en variabel, reserveres der noget hukommelse i det færdige program, en plads til denne variabel.

Eksempel 3-2. Definition af en tomat-variabel

```
struct tomat_t sunglow;
```

Det er ikke alle programmører, som kan lide denne notation. Der er der nogle, som benytter sig af *typedef* til at danne nye typebetegnelser, på den måde slipper man for hele tiden at skrive ordet *struct* .

Eksempel 3-3. Struct type ved hjælp af typedef

```
typedef struct tomat_tag {   /* taggen er ikke nødvendig i dette eks. */
    int ident;
    char name[80];
    int spacing;
    /* etc - etc. */
} tomat_type, *tomat_ptr;

tomat_type softball;
tomat_ptr current_tomat;
```

Det er meget rart, at man kan se på ordet tomat_type, at det ikke er en variabel, men er en type. Ellers må man huske, at et ord foran et andet *skal* være en typebetegnelse. Til gengæld kan det være sværere for compileren at finde ud af at diagnosticere fejl. I C++ er det altid tilladt at udelade nøgleordene struct og class, undtagen i erklæringen af typen.

```
tomat_type red_sun;
```

I parentes bemærket er der danske virksomheder, som har haft enorme ekstraudgifter på at bruge danske betegnelser i programmer, som skulle eksporteres, så det er nok klogt ved alle større projekter at erkende babelstårn problematikken og benytte engelsk, latin eller esperanto.

Eksempel 3-4. Erklæring og definition

```
struct tomat_ty {
   int identifikation;
   char art[80];
   int goedningsforbrug;
   int pladskrav;
   int temperaturkrav;
   int saesonpris[24];
} sungold;
```

I eksempel Eksempel 3-4 er der både erklæret en type, nemlig *tomat_ty*, og en variabel, *sungold*. Bør kun anvendes i ultrakorte programmer (stenografi-orienterede;-).

Her er et eksempel på et andet udvalg af informationer om tomater. Det er tænkt som registrering af tomater, der skal deltage i en udstilling, hvor der uddeles præmier for bedste sort, og hvor måske journalister skal have adgang til billedmateriale.

Eksempel 3-5. Udvælgelse af oplysninger

```
struct tomat {
   int loebenummer;
   int udstiller_nr;
   char navn[80];
   char beskrivelse[280];
   enum billedtype;
   char fotograf[80];
   enum karakter[MAXK];
};
```

Man må forestille sig, at udstillingsledelsen har et ekstra kartotek over udstillere, og heri kan de finde navn og adresse ud fra udstiller_nr. Desuden har jeg forestillet mig, at man registrerer tidligere karakterer (bedømmelser) for den pågældende tomat. Der er jo forskellige udstillinger, eller det kunne være, at samme tomat-sort deltog for anden gang.

Som regel er der langt flere oplysninger, end vi er interesseret i, sådan rent programmeringsmæssigt. Det er en disciplin for sig selv at udvælge og vurdere informationernes anvendelighed.

3.1.2. Interface til datatype

Sæt nu, at vi ikke rigtigt vidste, om vores tomat-beskrivelse ville kunne holde i hele programmets levetid. Der er flere måder at håndtere behov for ændringer.

Den mest anvendte metode går ud på at isolere repræsentationen af tomaten fra de dele af programmet, som anvender tomat-informationer. Det siger sig selv, at dette kræver ekstra kode. Det er imidlertid med de nye C compilere muligt at optimere den ekstra kode på samme måde som i C++ - nemlig ved at erklære en funktion for *inline*.

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

struct tomat_t {
   int typenummer;
   char artsnavn[80];
   int goedningsforbrug;
   int pladskrav;
   int temperaturkrav;
   int saesonpris[24];
};

struct tomat_t *create_tomat(int ident, char *name,
        int goedningsforbrug, int pladskrav,
        int temperaturkrav, int *saesonpris)
{
    struct tomat_t *temp;
    if (!(temp = malloc(sizeof(struct tomat_t)))) {
        fprintf(stderr,"Not enough memory\n");
 exit(255);
    }
    memset(*temp,0,sizeof(struct tomat_t));
    temp->typenummer = ident;
    strncpy(temp->artsnavn,name,79);
    temp->goedningsforbrug = goedningsforbrug;
    temp->pladskrav = pladskrav;
    temp->temperaturkrav = temperaturkrav;
    if (saesonpris)
        memcpy(temp->saesonpris,saesonpris,sizeof(temp->saesonpris));
    return temp;
}

main()
{
    struct tomat_t *tomat;
    int dyrkning_dage;
    tomat = create_tomat(1012, "sungold", 450, 30, 27, NULL);
    dyrkning_dage = beregn_periode("2001-08-29", tomat);
    return 0;
}
```

Læg især mærke til main funktionen. Der oprettes en pointer til tomat-typen. Denne pointer får noget at pege på ved at kalde create_tomat-funktionen med tilhørende initialiseringsparametre. Vi snyder med pris-tabellen, som skulle være initialiseret med fx. gennemsnitspriser for hver uge.

Hvis man nu ændrer struct tomat_t, sådan at pladskrav ændres fra rækkeafstand til arealforbrug, og hvis der yderligere tilføjes oplysninger om optimale dag/nat-temperaturer for den pågældene plantesort, så skulle vi ændre vores initialiserings-funktion. Men hvis der var gammel kode (fx. den main, som er med i eksemplet) der stadig skulle kunne fungere, så ville det være muligt at skrive initialiseringen sådan, at man stadig kunne bruge denne "gamle" main(). Det skulle blot være muligt at beregne (eller anslå) værdien af pladsforbrug ud fra den gamle type oplysninger.

Der er ikke noget i vejen for, at oplysningen om gødningsforbrug, der her afleveres som gram, i stedet kunne gemmes som kilo i en float-variabel. Et kendt eksempel på en lignende programmeringsmetode er FILE typen, som beskrives i Afsnit 3.3.

Ved at anvende interface operationer i stedet for at tilgå tomat-struct'ens data direkte, bliver det muligt at ændre detailler i tomat-struct'en uden at ændre programkode ret mange steder. Får man behov for at konvertere datatyper eller for at dele repræsentationen af informationer op på en anden måde, så vil man slippe godt afsted med det uden større problemer.

Hvis et nye behov *kun* består i at have flere data til denne struct, for eksempel af hensyn til nye beregningsprogrammer, så er det ikke så vanskeligt at udvide en struct selv om programkoden tilgår de enkelte felter direkte. Problemer med ændringer bliver først alvorlige, når programmernes størrelse og kompleksitet bevirker, at ingen længere har det fulde overblik over flow. Hvilket forekommer :-((

3.1.3. Tabel, array, liste, sekvens, bunke ...

Med et eksempel fra Donald Knuth kunne vi se på, hvordan man kan repræsentere kort og bunker af kort i et kortspil.

En struct, som illustrerer et enkelt kort, kunne for eksempel se således ud:

```
/* der benyttes bitfields for at vise denne
 * feature. I stærkt memory kritiske applikationer kan det
 * nogen gange betale sig at anvende bitfields. Kode til
 * manipulation af bitfelterne vil imidlertid gøre koden
 * langsommere og større.
 */

struct kort_t {
    unsigned tag:1;         /* bagsiden op = 1 */
    unsigned farve:2;       /* 0 klør, 1 ruder, 2 hjerter, 3 spar */
    unsigned rang:4;        /* 1 = es, 2 = toer, 13 = konge */
    struct kort_t *next;    /* kort nedenunder */
    char titel[11];         /* forkortet navnebetegnelse */
```

```
};
```

Indholdet af et felt i en struct kan være hvad som helst - dog med den lille begrænsning, at C-oversætteren skal kende alle typerne. Med et berømt citat: "Folk kan få bilen i den farve, de ønsker, bare de ønsker den sort." (Henry Ford om Ford-T).

Der er én undtagelse til den regel, at oversætteren skal kende alle typer. Man kan godt have en pointer til et objekt af ukendt størrelse! Det er meget rart, når man har behov for et felt som `struct kort_t *next` , der henviser til kortet nedenunder. Mere om det senere.

Vi kan have tal eller tekst i et felt - eller en anden struct, hvis den er erklæret tidligere i programmet.

Tag (engelsk: mærke, udtales "tagg") sat lig med 1 betyder, at kortet vender bagsiden opad, tag == 0 betyder, at det vender forsiden op. Feltet farve er selvfølgelig *kortspilsfarve* d.v.s. klør, ruder, hjerter eller spar. Ved at benytte 2 bits får vi værdierne 0, 1, 2 og 3 til rådighed.

```
enum farve_t { kloer, ruder, hjerter, spar };
```

For at knytte en betydning mellem 0 til 3 og kortspilsfarverne kan vi benytte en enumeration (tælleremse, nummer-system). Reglerne for en sådan ses Eksempel A-3. Klør (kloer) bliver en symbolsk betegnelse for 0, ruder for 1 etc. Man kan godt tildele et symbol i en enumeration en værdi, for eksempel `enum farve_t { kloer=1, ruder, hjerter, spar };` Imidlertid har jeg ikke gjort det her. Hvis vi anvender kulør-værdierne 0-3 kan vi have dem i 2 bit, ellers må vi bruge flere. Selv om det ikke er essentielt i dette eksempel at spare på bits, så er det en helt almindelig fremgangsmåde.

```
struct kort_t {
    unsigned tag:1;         /* bagsiden op = 1 */
    enum farve_t farve:2;   /* variabel af enumeration type */
    unsigned rang:4;        /* 1 = es, 2 = toer, 13 = konge */
    struct kort_t *next;    /* kort nedenunder */
    char titel[11];         /* forkortet navnebetegnelse */
};
```

Rang henviser til kortets værdi med es som ener, konge som tretten'er. Også her kunne vi benytte en enumeration for at få en forbindelse mellem "konge" og tallet 13.

Feltet `struct kort_t *next;` er et felt, som kan indeholde adressen på et andet kort, i dette tilfælde kunne det være kortet, som ligger nedenunder. I mange tilfælde kunne man nøjes med at angive afstanden fra en base adresse, et *offset* . Hvis der ikke er noget kort nedenunder, kan vi give feltet værdien 0 som i denne sammenhæng kaldes NULL.

Adressen på et objekt kaldes også et link eller en pointer (vejviser, pegepind) eller reference til dette objekt. I teksten her benyttes ordet pointer om en adressevariabel, ikke om adressen alene.

Der er en sjov ting i denne struct: Feltet *next* er af samme type, som den selv er, `struct kort_t *` .

I C sproget er der 4 oplysninger om en pointer, de findes i oversætterens symboltabel:

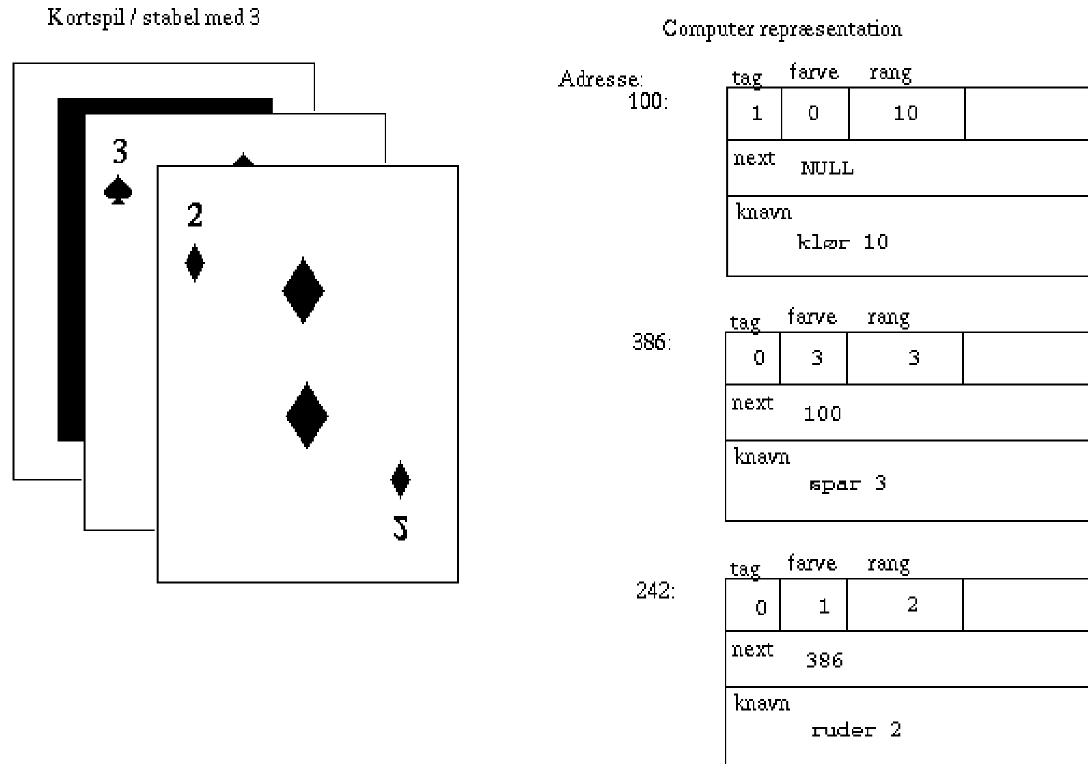
- selve pointerens adresse (som for enhver anden variabel),
- dens størrelse (antallet af bits eller bytes, som er nødvendige for at adressere en character variabel),
- størrelsen af det, den peger på; det kaldes også skalering,
- Graden af indirection (om det er en pointer til en pointer til en pointer til ... til en dims, antallet af stjerner i deklarationen af variabelen)

En pointer er en variabel. Størrelsen af en pointer er altid kendt. På en 32 bit maskine er den 32 bit (4 bytes). Det er muligt at afsætte plads til en pointer uden at vide, hvad den peger på. At afsætte plads er det samme som at definere variabelen.

Det siger sig selv, at når man afrefererer pointeren, d.v.s. henter eller kopierer det objekt, som den peger på, så skal objektets størrelse være kendt. Denne størrelse kaldes også en skaleringsfaktor. Når man lægger j til en pointer, bliver j ganget med skaleringsfaktoren og resultatet lagt til pointeren.

Når det er nødvendigt, kan man erklære en pointer til et objekt af ukendt størrelse. Det er faktisk det, som vi gør med feltet next. Mens oversætteren arbejder på opbygningen af struct kort_t, er størrelsen på selvsamme struct ikke kendt endnu. Derfor bliver next-pointeren først endeligt tildelt sin skalerings-faktor (størrelsen af det objekt, den peger på) i det øjeblik, at oversætteren er færdig med struct kort_t.

Når vi anvender next feltet til at fortælle, hvilket kort, der ligger nedenunder, så vil vi kunne repræsentere en lille bunke kort på denne måde:



Nu kan vi erklære et "helt kortspil" som et array med 53 kort, 13 i hver farve plus en joker. Definitionen ser ud som her:

```
struct kort_t kort[53];    /* husk: fra 0 til og med 52. */
```

Hvis vi erklærer et "helt kortspil" og initialiserer alle kortenes felter, vil de 4 farvers kort kunne initialiseres på følgende måde (hele kildeteksten i filen kort02.c); vi burde give andre navne til 11,12 og 13, altså knægt, dame og konge:

```
    for (kuloer = kloer; kuloer <= spar; ++kuloer) {
        for (j = 0; j < 13; ++j) {
            kort[kuloer*13+j].tag = 0;
            kort[kuloer*13+j].farve = kuloer;
            kort[kuloer*13+j].rang = j+1;
            sprintf(kort[kuloer*13+j].knavn,"%s %d", farve(kuloer), j+1);
            kort[kuloer*13+j].next = NULL;
        }
    }
```

Kortspils-farverne bruges som om de var heltal (det er de jo også) og kan endog bruges til at specificere stop og start i for-løkken, som styrer initialisering. Hvis man kompilerer med C++ compileren, altså g++, så vil den dog klage over operationen ++kuloer, men i øvrigt hjælpe med at tjekke, at vi bruger enumerationen uden at tildele den forkerte værdier (som fx. kuloer = 17). gcc vil end ikke nævne, at vi tildeler en variabel af typen farve_t en værdi, der ligger udenfor grænserne. Det gør vi i den funktion, som returnerer en string variabel for farvens navn.

Det er ikke den eneste måde at gøre tingene på. Det er heller ikke den mest effektive, som er et initialiseret statisk array. For at mindske skrivearbejdet kan man generere det meste af C-koden til initialisering af dette statiske array ved hjælp af et awk-program.

Men programmet her er jo ikke et, som kræver ultimativ optimering, så derfor er det nok bedre at lægge vægt på læsevenligheden.

For at gøre programmet endnu mere læsevenlig kan analogt med de fire farver lave en enumeration for rang og tilhørende betegnelse, hvilket vil gøre det muligt at håndtere det korte navnefelt for es, knægt, dame og konge på en bedre måde. (Prøv at gøre det - det er en god øvelse.)

Den anden ting, som kunne påkalde sig en kommentar er, at vi ikke anvender et multi-dimensionalt array og at vi i øvrigt bare forlænger i den anden ende for at få plads til en joker. Læg mærke til hvor vanskeligt det er at håndtere jokeren i kildeteksten kort02.c! Skal den have en rang? Skal den have en farve, og i så fald hvilken? Skal vi ændre vores farve/rang-system, således at man kan håndtere en ekstra farve for jokeren? Skal man have flere jokere og skelne mellem rød joker og sort joker? I kort02.c lader vi jokeren være af typen kloer (0), og giver den blot en højere rang. Hvis den skal bruges i et kortspil eller en kabale af en slags, så må vi sørge for at programkoden tager højde for, at der kan være et ekstra kort. Det er ikke så væsentligt for eksemplet her, så vi lader den ude af betragtning i de følgende eksempler.

Repræsentationen af vort kortspil i hukommelsen ser nu ud som følgende illustration antyder; vi går ud fra, at størrelsen på en struct kort_t er 20 bytes:

```
adresse (offset):
0            20          40          60          80
+-----------+-----------+-----------+-----------+-----------+...~
| 0 0 0 NULL| 0 0 1 NULL| 0 0 2 NULL| 0 0 3 NULL| 0 0 4 NULL|
|"kloer 1  "|"kloer 2  "|"kloer 3  "|"kloer 4  "|"kloer 5  "|
+-----------+-----------+-----------+-----------+-----------+...~
```

... og vores lille bunke fra før kan antydes med følgende skitse:

```
+------+           +------+           +------+
|next: |           |next: |           |next: |
|NULL  |           | 820  |           | 180  |
|      |           |      |           |      |
|      |           |      |           |      |
|      |           |      |           |      |
|      |           |      |           |      |
```

```
|klør  |           |ruder |           |spar  |
|10    |           | 2    |           | 3    |
|      |           |      |           |      |
+------+    ....   +------+    ...    +------+
adresse:
180                280                820
```

Ovenstående skitse er imidlertid ikke den mest læselige måde at beskrive forbindelsen mellem de tre kort, så man benytter som regel pile i stedet for som på næste illustration.

Eksempel 3-6. Kort-bunke

```
+----------------+     +----------------+     +----------------+
| 0 1 1       *--+---->| 0 3 2       *--+---->| 0 0 9       *--+----+
| ruder 2        |     | spar  3        |     | klør 10        |    |
+----------------+     +----------------+     +----------------+    |
                                                                    |
                                                                 ___|___
                                                                  -----
                                                                    ~
```

Her er adresserne på hver struct er ikke vist; de er jo også helt irrelevante.

Den anvendte struktur kaldes en linket liste (eller linked liste). Det betyder simpelt hen en sammenkædet liste.

Når en pointer ikke peger på en adresse, sætter man den pr. konvention til 0 eller NULL, en adresse, som intet normalt program nogensinde vil forsøge at røre ved. Det kan tegnes som en jordforbindelse (således som det er forsøgt på ascii-tegningen ovenfor.)

Vores kortspilsrepræsentation er lavet som et array af kort. Den sidste tegning Eksempel 3-6 viser, at arrayet ikke er essentielt for "bunke" teknikken med en next-pointer for hvert kort.

Man kan sætte hvad som helst sammen efter det princip, at et element peger på det næste. Det er en af de væsentligste programmeringsteknikker, og den har derfor fået mange navne: Linket liste eller bare liste, sekvens, tabel. Man kunne sagtens udelade arrayet og blot have to - tre variable af typen kort_t, som man satte sammen i en liste.

Imidlertid er arrayet en udmærket løsning for et kortspil. Kortene er jo til stede alle sammen hele tiden, med mindre vi skal tage højde for en haj, der har kort i ærmet. Man har mulighed for at håndtere kortene som et array, når man for eksempel vil "blande" dem ved at bruge en random - generator til at plukke kortene ud, et efter et, fra arrayet i tilfældig rækkefølge og sætte dem øverst i bunken. Prøv det som en programmeringsøvelse (eller se eksempel kort04.c).

Hver gang vi vil lave en bunke, skal vi have et startkort. Det vil sige, at man skal have en variabel, som kan huske, hvor vi starter henne. Et minimalt eksempel kan se ud som her:

Eksempel 3-7. Start punkt for kortbunke

```
/* kortspillet forudsættes initialiseret.
 * Hvis kildeteksten anvendes uden en initialiseringsloop som i
 * kort02.c, vil alle k->navn felterne være tomme.
 */


struct kort_t k[53];

struct kort_t *kort_bunke_demo()
{
    struct kort_t *start, *temp;
    temp = start = k;               /* starten af vores bunke
                                     * er første element i array k.
                                     */
    temp->next = k[1+rand()%52];    /* vi plukker et tilfældigt kort
                                     * værdien af 1+rand()%52 ligger
                                     * mellem minimum 1 og max 52
                                     */
    temp = temp->next;              /* ændring af temp,
                                     * men start er stadig
                                     * adressen på det første kort
                                     */
    temp->next = NULL;              /* temp peger på de plukkede kort,
                                     * vi nulstiller next feltet på det
                                     * nye kort */
    return start;                   /* returnerer adressen på det første */
}

void bunke-og-vis()
{
    struct kort_t * kp;             /* her får vi start-kortet */
    int nk = 0;
    kp = kort_bunke_demo();
    do {
        printf("Kort nr. %d er %s\n", nk++, kp->knavn);
    } while ( (kp = kp->next) != NULL);
}
```

Adressen på dette start - kort gemmes *uden for kort-arrayet.* (Alternativt kan man bruge dobbelt-linkede lister og finde startpunkter som de kort, der har en NULL-pointer. Derfra kan man følge en anden pointerkæde til man igen havner på en NULL pointer - men alt i alt kræver det meget mere programmering.)

I eksemplet kort04.c, som blander kortene og skriver resultatet ud, anvendes variabelen struct kort_t *oeverst som den, der husker starten af bunken.

```
/* uddrag af kort04.c */
{
    struct kort_t *oeverst;
    bland(kort, MAXK, &oeverst);
    show_kortbunke(bunke);
}
```

En anden morsom ting ved denne repræsentation er, at kortene kun kan indgå i én bunke. Meget realistisk!

3.1.3.1. Udvælgelse af oplysninger

Vi har i disse eksempler udvalgt nogle oplysninger, som vi syntes var relevante for at repræsentere et kortspil. Man skal imidlertid gøre sig klart, at man foretager et valg.

Vi kan med *next feltet* fortælle, hvilket kort, der ligger nedenunder, men vi har ikke et felt for kortet ovenover, hvis der er noget. Det kunne man jo godt ønske sig i nogle situationer.

Her er en liste over nogle andre ting, vi ikke har interesseret os for: Kortenes størrelse, vægt, placering på bordpladen (eller andre steder), slitage, ejermand, design af bagsiden, temperatur, molekylernes beskaffenhed ...

Prøv at tænke efter, om der er flere oplysninger om et kortspil, som man kunne repræsentere og find ud af, hvornår man ville bruge de oplysninger. For eksempel ville en kortspilsfabrikant måske være interesseret i "molekylernes beskaffenhed", fordi han gerne ville vide noget om kartonens kvalitet og holdbarhed. Find flere egenskaber, som kan omsættes til tal eller tekst.

3.1.4. Operationer på datatypen

Det er forhåbentlig blevet klart, at det kan betale sig at spørge, hvilke informationer, som er væsentlige, og hvilke operationer, som skal kunne udføres på disse informationer. Ud fra dette vælger man metoden til at strukturere sine data.

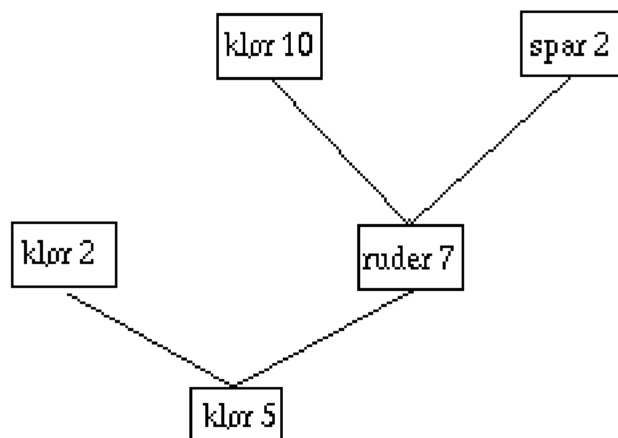
Datastrukturer som lister, array’er, kø og stak (som vi ikke har set eksempler på endnu) og lignende strukturer har hver især deres fordele.

Med den struktur, som vi har valgt i eksempel kort04.c, kunne vi let "se", hvilket kort, der lå øverst. Hvis vi skal lægge et kort ovenpå, så er det nemt nok. Operationerne til at blande dem og gennemløbe (og udskrive) den blandede bunke går helt fint, og kan pakkes ind i funktioner, som gør programmet let at læse.

Men hvis vi vil lægge kort ind midt i bunken, så er denne repræsentation måske ikke den bedste, men den kan dog bruges: Man *kan* bladre sig frem til den position, man ønsker at indsætte på.

Hvis man anvender et binært træ, kan man indsætte et element i en ordnet sekvens, uden at det koster meget ekstra tid for algoritmen.

Figur 3-1. Binært træ



Nu er kort-spils arrayet i forvejen sorteret på grund af måden, det er initialiseret, så vi har ikke brug for en sortering af hele kortspillet som sådan. Hvis vi derimod ønsker at programmere en omgang whist eller bridge, skal der først blandes kort og derefter deles ud til fire bunker (spillere). Man ville sikkert foretrække at udskrive spillernes bunker (hænder) ordnet efter kortenes rang. Dér burde vi overveje at repræsentere bunkerne som 4 binære træer.

Dette afsnit er - sammen med de næste afsnit om konkrete og abstrakte datatyper - under omarbejdelse. Hvis du har nogen spørgsmål eller ønsker at bidrage skal du være velkommen. Jeg regner med at få skrevet en hel del i løbet af perioden 24. juni - 31. juli 2001.

3.2. Konkrete og abstrakte datatyper

Bemærk at dette afsnit er under omredigering, og at der er gentagelser af oplysninger fra forrige afsnit.

Hvis du har kommentarer, ris eller ros eller forslag, så er du velkommen til at kontakte mig (eller andre SSLUG'er). Min adresse har tidligere været vist lidt forkert, derfor en gentagelse:
donald_j_axel@get2net.dk

Konkrete og abstrakte datatyper nævnes ofte i samme åndedrag som objekt-orienterede sprog, men man har også stor glæde af at vide noget om disse emner, når man programmerer i C.

Er C et objektorienteret sprog? Hvorfor er det ikke objektorienteret, når et C program er en række definitioner af eksterne objekter?

Det korte svar er, at almindelig C ikke har virtuelle funktioner. Desuden er der ikke support for generisk algoritme programmering. Det er imidlertid interessant at vide, at C sproget som sådan ikke forhindrer programmøren i at tænke objektorienteret.

Hvorfor så ikke bruge C++ og glemme alt om C? Der er flere svar på dette spørgsmål. C oversætteren er stadig mere effektiv end C++, selv om C++ teoretisk set burde generere kode af samme størrelse og effektivitet. Et andet svar er, at C er sjovere og mere læseligt end C++, men det er måske ikke alle, der er enige i den betragtning. Hvis man vil forstå, hvad der sker i C++, er det en fordel at forstå C sproget fuldt ud. [1]

Eksempel 3-8. Kompilering af samme program med C og C++ oversættere

Læg mærke til, at der benyttes flag -O for optimering, -s for at fjerne symboltabellerne, således at programmet kun kommer til at bestå af maskininstruktioner og dynamisk link - information. Programfilen bliver lidt mindre for C oversætteren, men ikke meget. Programmet er så lille, at resultatet kun må opfattes som en strømpil. Prøv med nogle af de større programmer - og se, om det er muligt at rette lidt i kildeteksten, så C++-oversætteren accepterer program kildeteksten!

Endelig bemærkes, at man får versionen af oversætter-systemet frem ved kommandoen gcc -v eller g++ -v . Man må ikke sammenligne programfiler genereret med for eksempel gcc 2.8.1 med programfiler, som er genereret med gcc 2.95.2 eftersom der kan være meget stor forskel på oversætterens håndtering af alignment, optimering m.v.

```
/fri $ls -lo cirkle1.c
-rw-r--r--   1 root          361 Apr 30 00:11 cirkle1.c
/fri $gcc -Wall cirkle1.c -O -s -o cgg1
/fri $ls -lo cgg1
-rwxr-xr-x   1 root         3028 Apr 30 00:11 cgg1
/fri $gcc -v
Reading specs from /sources/gcc/bin-2.95.2/lib/gcc-lib/i586-pc-linux-gnu/2.95.2/specs
gcc version 2.95.2 19991024 (release)
/fri $g++ -Wall cirkle1.c -O -s -o cxx1
/fri $ls -lo cxx1
-rwxr-xr-x   1 root         3176 Apr 30 00:12 cxx1
/fri $g++ -v
Reading specs from /sources/gcc/bin-2.95.2/lib/gcc-lib/i586-pc-linux-gnu/2.95.2/specs
gcc version 2.95.2 19991024 (release)
```

```
/fri $
```

Hvis man skriver store programmer som for eksempel et operativsystem eller et GUI library, så er det en fordel at kunne tænke og arbejde på højt niveau. Derfor er det nyttigt at skrive "objektorienteret" også når man arbejder med "almindelig-C" programmering. Senere i dette kapitel vil vi sammenligne to udgaver af en linked liste, den ene skrevet i C og den anden i C++. Så kan du selv dømme. Men aller først lidt introduktion om objekter, konkrete og abstrakte datatyper.

De fleste sprog har nogle mekanismer, som er rigtigt objektorienterede, nemlig håndteringen af forskellige numeriske typer.

Vi kan have en integer i en variabel og gange den med en float og lægge resultatet i en double uden at compileren gider fortælle, at der skal konverteres. Taber vi præcision ved at konvertere fra double til integer, vil de fleste compilere give en warning, men de konverterer dog.

Det er egentlig objektorientering i en nøddeskal. Definer din algoritme (fx. addition) og sørg for, at den kan håndtere forskellige data, d.v.s. objekter, på en passende måde. Det er lidt vanskeligt at skrive operatorfunktionerne på en sådan måde, at de kan klare alle situationer, for eksempel både fortegns-minus og subtraktions-minus, ofte kaldet unært og binært minus. Som en øvelse i objekt-orienteret tankegang kan man prøve at definere en struct, som skal repræsentere brøker, som for eksempel 2/3, der jo ikke er det samme som 0.6667. I "almindeligt" C vil man bruge funktionskald til at udføre aritmetiske konverteringer og operationer, og det kunne nemt komme til at se lidt klodset ud, som for eksempel her:

```
f()
{
    struct broek andel;
    struct broek afgift;
    struct broek *b_bogpris;

    andel.t = 1;
    andel.d = 3;
    b_bogpris = new_broek(360,1);
    broek_multiply(&afgift,b_bogpris,&andel); /* ikke kønt */
    free(b_bogpris);

    /* se filerne bogpris1.c bogpris2.cxx for hele source.  */
}
```

I C++ er det muligt at erklære en variabel af typen broek som for eksempel nedenstående eksempel, og derefter benytte de tilhørende operationer udtrykt ved de i forvejen kendte operatorer:

```
broek andel(1,3);
int bogpris = 360;
broek afgift = bogpris * andel;  /* ret nemt at læse */
```

Det forudsætter selvfølgelig, at man har defineret typen broek og tilhørende funktioner for operatorerne på en passende måde!

En sådan brugerdefineret datatype kaldes somme tider en konkret datatype, i modsætning til en abstrakt datatype. Den konkrete datatype har ingen "virtuelle funktioner" sådan som den abstrakte har det. Læs videre:

Betegnelsen abstrakt datatype (ADT) bruges somme tider om alt, hvad der kan indkapsles, men den mest rimelige anvendelse er nu den, som Stroustrup angiver i sine forskellige bøger om C++. Hvis vi skal skrive en hardware driver til en grafisk device (en som er i stand til at tegne prikker på en angivet position) så kan vi definere en klasse "figur" - men for at opnå bedste hastighed ønsker vi, at måden, den tegnes på, er optimeret for henholdsvis cirkel og rektangel. C++ supporterer den slags konstruktioner:

Vi kan i C++ definere en generel klasse, "figur", som omfatter de grundlæggende egenskaber ved figur og tillige de operationer, som hører sammen med den. Nogle af disse operationer kan vi endnu ikke sige nøjagtigt hvordan vi ønsker implementeret. Hvis vi gjorde, ville det ikke være optimalt.

```
struct figur {  // man kan skrive class i.st.f. struct
    public:
        figur(){;}
        virtual void tegn_figur() =0; // =0 er nødvendigt
};
```

Derefter definerer vi subklasser, det vil sige klasser, som overtager (arver) egenskaberne fra den generelle klasse. Sådan en klasse kaldes også specialiseret. Det kunne være cirkel, som jo er mere specialiseret end "figur". Denne subklasse *skal* definere en brugbar funktion til at udføre handlingen "tegn_figur".

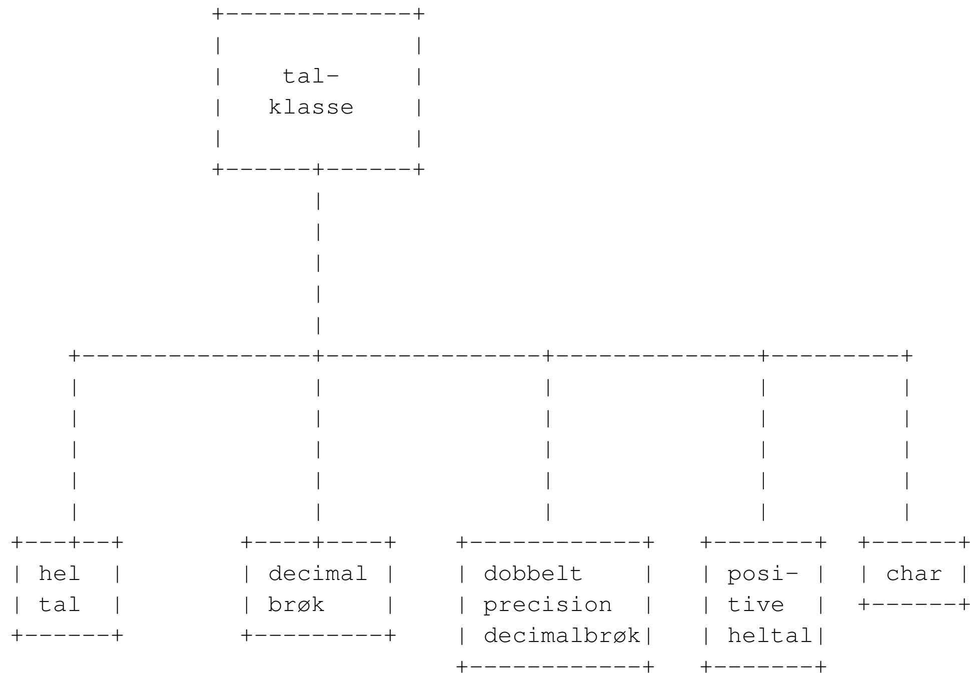
```
struct cirkel : figur {
    public:
 void tegn_figur(){
  printf("Tegner figur/cirkel...\n");
 }
};
```

En funktion i klassen "figur", som erklæres at være virtuel, kan ikke kaldes, den eksisterer jo i virkeligheden kun, mens programmet kompileres. Det vil give oversættelses-fejl, hvis de nedarvende klasser ikke definerer en (rigtig) funktion. [2]

I C kan man ikke få compileren til at hjælpe sig med at "huske", at man skal skrive en optimeret, håndgribelig funktion som tegner cirkel-figuren.

For at fortsætte parallellen med de numeriske typer, int, double, long double etc., kan man sige, at de alle nedstammer fra en abstrakt datatype "tal". I "tal-klassen" er der virtuelle funktioner for addition, subtraktion etc. De er virtuelle, fordi de ikke eksisterer for klassen "tal", men kun for de specialiserede klasser, heltal, decimalbrøk og så videre.

Eksempel 3-9. Talklasse - en abstraktion



For heltals-objekter skal oversætteren realisere operationen med heltal maskin-instruktioner, men hvis der derimod er tale om decimalbrøker skal oversætteren realisere operationen ved hjælp af flydende-komma maskin-instruktioner.

C sproget er ikke kun en "fattig" udgave af C++. C sproget bliver til stadighed påvirket af udviklingen indenfor bl.a. C++ og tager de bedste ting, som er fremkommet, med i standarden, se evt. appendix C, C99. C og C++ moduler kan sættes sammen. C sproget vil stadig blive valgt, hvor størrelse og hastighed er vigtigst, for eksempel i kerne-moduler og drivere. Derfor er det vigtigt at kunne håndtere højniveau programmering i C, og det er det, som det efterfølgende afsnit handler om.

I de efterfølgende afsnit vil vi bruge betegnelsen "abstrakte datatyper" mere løseligt om alt, hvad der kan indkapsles. Vi stiller os tilfreds med, at implementering af funktionaliteten kan udskiftes uden at brugere af datatypen skal ændre deres kode.

3.3. FILE: En abstrakt type

For at programmere på højt niveau i C sproget skabes abstraktioner, som kan bruges til noget nyttigt. Fx. er en fil en abstraktion, som operativsystemet leverer. En fil er en ordnet sekvens af bytes. En disk-fils bytes ligger ikke altid i en pæn, ordnet rækkefølge på disken. Operativ-systemet får det imidlertid til at se sådan ud.

Endvidere sørger Linux for, at for eksempel en seriel port ser ud som en fil, selv om den ikke er det. Man kan skrive til et serielt interface, som om det var en fil, og læse fra den på samme måde. Ved hjælp af fil abstraktionen kan vi hente data fra eksterne kilder, så som diske, netforbindelser og tastaturet.

Grundlaget for en sådan abstraktion er som regel en structure. For at bevare overblikket, for ikke at blande detaljerne ind i den abstraktion, som vi ønsker at skabe, indkapsles sådanne data ofte, således at al tilgang sker gennem funktionskald. Derved bliver det lettere at anvende den pågældende abstraktion, man skal blot udstede en "kommando": Læs, skriv, luk, flyt o.s.v.

Fil tilgang klares med funktionerne fopen(), fread(), fprintf() etc., som benytter datatypen FILE som parameter, for fopen() er det returværdien. FILE typen skal man ikke "dissekere", det er tilladt, men ikke fornuftigt at benytte de variable, som typen består af. FILE benyttes blot som et håndtag til den datastrøm, hvorpå man ønsker at udføre IO-operationer.

Eksempel 3-10. FILE typen

```
/* fileopen.c - demonstrerer fopen(). */
#include <stdio.h>

main()
{
 FILE *fp;
 fp = fopen("fileopen.c","r");
 if (!fp) {
  perror("Kan ikke aabne filen \"fileopen.c\"");
  exit(2);
 }
 printf("Har aabnet filen \"fileopen.c\" \n");
 return 0;
}
/* se Eksempel 2-20 for læsning fra en fil */
```

FILE er en abstrakt datatype. Hvis vi benytter vores program til at læse en "special-fil", for eksempel en driver-entry via /dev/ kataloget eller en fil i /proc/ kataloget, så har vi samme funktionalitet, som en C++ abstrakt datatype kan give os. For FILE typens vedkommende leveres funktionaliteten ikke af vores programkode, men af filsystemet i linux-kernen.

FILE "håndtaget" benyttes til at tilgå filer, og de indre dele af FILE (pointere, interface til kernen etc.) er ukendt for os, vi behøver ikke at vide noget om buffere og pointere, [3].

Det kan ikke understreges nok, at nøglen til succes er kendskab til funktions-interfacet og de andre teknikker til at indkapsle de data, der naturligt hører sammen.

Slutbemærkning:

1. Anvendelse af objektorienterede fremgangsmåder er somme tider ikke sagligt begrundet (ifølge Stroustrup; jeg kommer til at skylde et sidetal i "The C++ Programming Language". Det er fx. ikke

hensigtsmæssigt at opbygge et grafisk library som ét stort klassehierarki; det indskrænker mulighederne.

2. Det giver ikke en oversætter fejl, men en link fejl i c++ 2.95.2 hvis man glemmer =0; efter en virtuel funktion i baseklassen.
3. Vi må selvfølgelig gerne vide, hvad der gemmer sig i en FILE, prøv selv at se efter i /usr/include/stdio.h og ikke mindst i /usr/include/libio.h som rummer selve struct-definitionen, bl.a.

```
char* _IO_read_ptr;   /* Current read pointer */
char* _IO_read_end;   /* End of get area. */
char* _IO_read_base;  /* Start of putback+get area. */
```

   Men selv om vi ved noget om _IO_FILE ville vi ikke drømme om at bruge denne viden i en end-user applikation!

Kapitel 4. Parsning - hvordan oversættes et C program

4.1. En declaration parser

Erklæringer kan være vanskelige at læse, især når der indgår pointere til funktioner. Installation af en signal handler med funktionen signal(2) er kendt for sin vanskelige prototype.

Det ville være en god øvelse at skrive en komplet declaration parser (og en sådan er på ønskesedlen til en udvidet version af denne bog). Imidlertid findes der allerede en meget instruktiv parser til interaktiv / didaktiv anvendelse, cdecl.

Cdecl manual-page går ud fra, at man er bekendt med de væsentligste problemstillinger, men den forklarer ikke, hvordan man løser læselighedsproblemet, hvis man ikke lige har cdecl ved hånden!

For at gøre tingene vanskeligere, er Linux/Gnu signal.h fuld af defines og særlige syntaktiske konstruktioner, som skal lette læsningen for den erfarne multiplatform programmør - men som gør det fuldstændigt umuligt for den almindelige begynder at finde hoved og hale. I dette tilfælde er manual page for signal(2) en lettelse. Der er oven i købet en forklaring på, hvordan man kan opbygge deklarationen ved hjælp af "typedef"ning. Manual siderne for glibc er med i RedHat og andre distributioner, men er ikke en del af glibc systemet, der kun anvender info-pages.

Men i header filerne - kast et blik på /usr/include/signal.h - er der så mange hensyn til diverse platforme at det bliver næsten ulæseligt. Leder vi på "signal(" finder vi:

```
#define signal(sig, handler) __sysv_signal ((sig), (handler))
```

Ovenstående define kan man ikke fodre cdecl med. Heldigvis er den ikke så svær at forstå. Signal er en funktion som skal erstattes af __sysv_signal. De to parametre skal gives videre som de er. Det ene skal være et signal, (fx. INTR, svarende til control-C) og det andet skal være den funktion, som vi vil have kørt, når vores program modtager signalet. Men så må vi jo kigge efter, hvordan headerfilen definerer __sysv_signal().

```
extern __sighandler_t __sysv_signal __P ((int __sig, __sighandler_t
__handler));
```

Heri indgår der - desværre - også en #define macro, nemlig __sighandler_t. Så det er en større sag at finde rundt i.

Det, som cdecl er glimrende til, er at fodre den med en vanskelig prototype og så se, hvordan den vil forklare det.

Vi finder med **man signal** følgende prototype:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

Er det en void funktion? Jeg spørger bare ... Nej, det er ikke en void funktion, det er en funktion, som returnerer en pointer til en anden funktion, som er void. Nemlig den tidligere signal handler. Så kan man jo geninstallere den, hvis man på et tidspunkt skal tilbage til forrige niveau af signal handling.

```
ax@pluto:/udvik/$cdecl
cdecl>void (*signal(int signum, void (*handler)(int)))(int)
syntax error
```

Ja, desværre kan denne udmærkede lille applikation heller ikke klare denne iøvrigt korrekte prototype, så der er virkelig et problem her.

Løsningen er at lære sig "højre-venstre" teknikken.

Højre venstre - teknikken består i at læse indefra den identifier, som man ønsker at forstå. I ovenstående erklæring "signal".

Til HØJRE for signal er der en parentes start. Det betyder: "Signal er en funktion ..."

Efter parameter parentesen er der "lukket" ved hjælp af en slut-parentes ekstra. Derfor må vi gå til VENSTRE. Vi er nu nået til, at vi forventer angivelse af retur-type.

Til venstre står der '*' hvilket vi læser som "returnerer en pointer til ..." - til hvad?

Parentes start spærrer for yderligere adgang til venstre, så vi går mod højre, udenfor den matchende parentes og ser efter denne endnu en parentes, aha, en pointer TIL EN FUNKTION, der står jo igen parenteser, og i øvrigt med en int som parameter. Det mest vanskelige er, at den VOID, som står forrest på linjen, er retur type angivelse til denne funktionspointer.

Det er ikke nemt. Læs man - siden for signal, den forklarer (som nævnt ovenfor), at meningen med signal er, at den returnerer den tidligere handler, så man kan reinstallere den senere.

4.2. En expression parser

En kalkulator er et program, som kan fodres med beregningsudtryk og som derefter beregner resultatet. Som regel skrives resultatet på en computerskærm, men det kan jo lige så godt være på en papirstrimmel eller i et felt i et regneark. Et udtryk kaldes på engelsk et expression.

Hvis man vil forstå, hvordan en oversætter/compiler virker, så er en kalkulator et godt sted at begynde. En kalkulator består af analysator, som kaldes en expression parser, og en beregningsdel. Expression parseren har til opgave at analysere input og se, hvad der skal gøres.

En sådan expression parser findes også i en oversætter. I stedet for at foretage beregning, så udskriver oversætteren assembler kode, der vil kunne foretage beregningen.

En væsentlig del af kunsten at skrive en oversætter består i at skille tingene ad. Det kræver stor forståelse for opgaven, en stor abstraktionsevne, for kode-generator delen er jo flettet ind, så at sige, i analysen. Hver gang man kommer til et delresultat, skal det omsættes til kode. Endvidere kræves der også en forståelse for assembler koden. Derfor er det meget godt at begynde med en kalkulator. Den er simplere at skrive, nemmere at forstå, men rummer de samme problemstillinger.

I en compiler har man brug for en expression parser, hver gang der er et statement. Et statement eller en (programsprog-)sætning kan i simple tilfælde blot være et beregningsudtryk efterfulgt af et semikolon. I mere komplicerede tilfælde er det for eksempel keywordet *if* efterfulgt af en betingelse og så et eller andet stykke kode, som styres af *if'et* . Betingelsen er, i C sproget, et expression (altså et udtryk). Den efterfølgende sætning vil i mange tilfælde være et funktionskald (også et udtryk, strengt taget) eller en tildeling som i C sproget også er et udtryk.

Så en expression parser er en vigtig del af computer-software. Her kommer et eksempel på en kalkulator, som beregner ganske almindelige regnestykker som 2+3 eller mere komplicerede, hvori der indgår aritmetiske funktioner og en enkelt variabel, X, som er resultatet af foregående beregning. Ideen her er hentet fra gode gamle Compas Pascal, men det er altså C-kode, dette her.

En lignende kalkulator i C++ er en god opgave for viderekomne. (Jeg vil senere prøve at komme med nogle sammenligninger med Stroustrups eksempel, Stroustrup[1997] p. 108.)

Eksempel på input:

```
calcu '200 * sin(0.444)'
85.911

calcu <<SLUT
2 + 3
5 * X
SLUT
```

```
Calc:      5.0000
Calc:     25.0000
Calc:

calcu
Calc: 32/square(2)
         ^Error
```

Programmet benytter til den viste fejlmeddelelse en særlig slem variant af printf format specification, som skriver et antal spaces ud styret af en variabel: `printf("En padded string: %*s\n",lengde,string);` Meget smart - men første gang lidt vanskeligt at læse og forstå. Det styrer angivelsen af error positionen.

Programmet er i sin nuværende form ganske anvendeligt, fordi det kan fungere som erstatning for expr programmet, der stiller alt for mange krav til spaces og anden formatering til de expressions, som skal evalueres. Men programmet kan simpelt hen også anvendes til beregning af prislister (det har det faktisk været!)

Eksempel 4-1. Calculator, recursive descent expression parsing

```
/* file calcu.c */
/* (c) Donald Axel GPL - license */
/* ANSI - C program demonstration, command line calculator */
/* Recursive descent parser */
/* Improve: Make a HELP command. Add more variables.       */

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



#define MAXL 8196
char gs[MAXL];
char *cp;
char *errorp;
double oldval;

/* local prototypes: */
int calcu();
int evaluate(char *line, double *prev_result);
int stricmp(const char *s1, const char *s2);
int strnicmp(const char *s1, const char *s2, int len);


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

    jj = 0;
```

```
    while (++jj < argc) {
        strcat(gs, argv[jj]);
    }
    if (argc == 1)
        return calcu();
    strcat(gs, "\n");
    rv = evaluate(gs, &oldval);
    if (!rv)
        printf("%g\n", oldval);
    else
        printf("Calcu:%s\n%*s\n", gs, rv, "^Error");
    return rv;
}


/* Description: */
/* calcu() sets up a string which is then evaluated as an expression  */
/* If (argc>1) main sets up string for evaluate() and prints result.  */
/* stricmp does not stop at '\n' - so we have to compare with "xx\n"  */
/* gettok() could solve that problem. TRY to use gettok().            */



int nextchar()
{
    ++cp;
    while (*cp == ' ')
        ++cp;
    return *cp;
}


int eatspace()
{
    while (*cp == ' ')
        ++cp;
    return *cp;
}


int calcu()
{
    FILE *ifil;
    char line[MAXL];
    int rpos;
    double r;

    ifil = stdin;
    while (1) {
        errorp = NULL;
        printf("Calc:");
        if (!fgets(line, MAXL, ifil))
            break;
```

```
        if (strlen(line) && strnicmp(line,"QUIT",4)
&& stricmp(line,"Q\n"))
            rpos = evaluate(line, &r);
        else
            break;
        if (!rpos) {
            printf("%-18g\n", r);
            oldval = r;
        } else {                /* prints Error in field min. 12 wide */
            printf("%*s\n", rpos, "^Error");
        }
    }
    return rpos;                /* if interactive rpos should always be 0 */
}


/* More local prototypes. This could, of course, be a separate file. */
double expression();
double product();
double potens();
double signedfactor();
double factor();
double stdfunc();


int evaluate(char *s, double *r)
{
    cp = s;
    eatspace();
    *r = expression();
    eatspace();
    if (*cp == '\n' && !errorp)
        return (0);
    else
        return (cp - s) + 11;
}


double expression()
{
    double e;
    int opera2;

    /* printf("test arg:%s\n",cp); */

    e = product();
    while ((opera2 = *cp) == '+' || opera2 == '-') {
        nextchar();
        if (opera2 == '+')
            e += product();
        else
            e -= product();
    }
```

```
    eatspace();
    return e;
}


double product()
{
    double dp;
    int ope;

    dp = potens();
    while ((ope = *cp) == '*' || ope == '/') {
        nextchar();
        if (ope == '*')
            dp *= potens();
        else
            dp /= potens();
    }
    eatspace();
    return dp;
}


double potens()
{
    double dpo;

    dpo = signedfactor();
    while (*cp == '^') {
        nextchar();
        dpo = exp(log(dpo) * signedfactor());
    }
    eatspace();
    return dpo;
}


double signedfactor()
{
    double ds;
    if (*cp == '-') {
        nextchar();
        ds = -factor();
    } else
        ds = factor();
    eatspace();
    return ds;
}


double factor()
{
    double df;
```

```
    /* while (*cp!='\n') {
       putchar(*cp++);
       }
     */

    switch (*cp) {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        df = strtod(cp, &cp);
        break;
    case '(':
        nextchar();
        df = expression();
        if (*cp == ')')
            nextchar();
        else
            errorp = cp;
        break;
    case 'X':
        nextchar();
        df = oldval;
        break;

    default:
        df = stdfunc();
    }
    /* printf("ddt: df = %lf, *cp = %c\n",df,*cp); */

    eatspace();
    return df;
}


char *functionname[] =
{
    "abs", "sqrt", "sin", "cos", "atan", "log", "exp", "\0"
};

double stdfunc()
{
    double dsf;
    char **fnptr;
    int jj;
```

```
    eatspace();
    jj = 0;
    fnptr = functionname;
    while (**fnptr) {
        /* printf("%s\n",*fnptr); */
        if (strncmp(*fnptr, cp, strlen(*fnptr)) == 0)
            break;

        ++fnptr;
        ++jj;
    }
    if (!**fnptr) {
        errorp = cp;
        return 1;
    }
    cp += (strlen(*fnptr) - 1);
    nextchar();
    dsf = factor();
    switch (jj) {
    case 0: dsf = abs(dsf);  break;
    case 1: dsf = sqrt(dsf); break;
    case 2: dsf = sin(dsf);  break;
    case 3: dsf = cos(dsf);  break;
    case 4: dsf = atan(dsf); break;
    case 5: dsf = log(dsf);  break;
    case 6: dsf = exp(dsf);  break;
    default:{
            errorp = cp;
            return 4;
        }
    }
    eatspace();
    return dsf;
}


/* end calcu.c */
```

4.3. En komplet compiler

Nu har vi haft mulighed for at se nærmere på en tag-parser og en expression parser. Hvor langt er der så til at skrive en komplet compiler? Ikke så langt. Vi får brug for en kode-generator, d.v.s. den del, som ved, hvordan maskin instruktioner for addition, subtraktion, kopiering og funktionskald etc. ser ud. Desuden vil det være klogt at lave en præprocessor, som kan håndtere #include of #define.

Når man kan skrive en expression parser som Afsnit 4.2 og -- i stil med tag-parseren Eksempel 2-24 -- kan behandle forskellige typer input efter forskellige regler, så har man teknikken til det meste af en C compiler. Kodegeneratoren hæftes på parser-delen, således at hver gang man har analyseret sig til en basis-operation (tildeling, aritmetik, funktionskald osv) så skrives de tilsvarende assembler instruktioner til outputfilen.

Preprocessor er derimod det aller første behandlings-trin i C-compileren. Alle linjer med havelåge tegn (hash character, eller nummer-tegn *#*) bliver behandlet af præprocessoren. Det er #include, #define, #ifdef mv. De kaldes direktiver eller præprocessor kommandoer.

En compiler kan somme tider indgyde ærefrygt. Hvordan får man et stykke software til at forandre programsætninger til maskinkode, oven i købet lange, komplicerede programsætninger, med iffer og while, med komplicerede beregningsudtryk og så videre. Og maskin-instruktioner, er det noget for hvide mennesker? Er det et mirakel, at compileren kan finde ud af at oversætte `if (a > b) duut();` til maskininstruktioner, eller kender den måske maskin-instruktionerne i forvejen? Ja - selvfølgelig kender compileren maskinen og dens instruktionssæt i forvejen!

Det er efter min erfaring et vigtigt skridt, måske det vigtigste, for forståelse af computeren, at man forstår, hvordan CPU og adressebus fungerer. En CPU kan bedst sammenlignes med en regnemaskine, som har flere små resultat-display, eller celler til at gemme kalkulationsresultater. En sådan celle kaldes et register, og CPU'er kan anvende disse direkte i deres instruktioner. Til gengæld er der begrænsninger på, hvordan man kan anvende tal (data) fra RAM modulerne. Nogle CPU typer insisterer på kun at lave aritmetik med register-indhold. Andre, som Intel-x86 er mere alsidige og derfor også mere komplicerede.

Intel 386 familien har til almindelig afbenyttelse for alle slags programmer registrene %eax, %ebx, %ecx og %edx. Desuden er der nogle ekstra registre, %esi og %edi, som er specielt gode til index-operationer, samt et base-pointer register %ebx og et stack-pointer register %esp. Desuden er der selvfølgelig en instruktions pointer %eip og mange specielle registre, bl.a. til administration af hukommelsen. De er ikke interessante i denne sammenhæng.

Et typisk lille assembler program i GNU assembler til at lægge to tal sammen med kunne skrives sådan her:

Eksempel 4-2. Assembler programmering

```
# dette er en kommentar
        .comm tal1,4,4
        .comm tal2,4,4
        movl tal1, %eax
        addl tal2, %eax   # nu lægges tal2 til tal1,
                          # resultatet ligger i %eax
        ret
```

GNU assembleren benytter traditionelt sin egen syntaks. Man skal huske, at Intel assembler er forskellig på væsentlige punkter (og er copyrighted!).

Der findes en "nasm" netwide portable 80x86 assembler, som benytter en mere Intel-agtig syntaks, hvis man foretrækker det.

Eksempel 4-3. Output fra GNU-C oversætter

```
ax@pluto:fri$cat xyzzy.c
xyzzy()
{
    return 42;
}
ax@pluto:fri$cc -O2 -fomit-frame-pointer -S xyzzy.c
ax@pluto:fri$cat xyzzy.s
 .file "xyzzy.c"
 .version "01.01"
gcc2_compiled.:
.text
 .align 4
.globl xyzzy
 .type  xyzzy,@function
xyzzy:
 movl $42,%eax  # <<=== OBS! HER ER OVERSÆTTELSEN!!!
 ret
.Lfe1:
 .size  xyzzy,.Lfe1-xyzzy
 .ident "GCC: (GNU) 2.95.2 19991024 (release)"
ax@pluto:fri$
```

Den linje, som interesserer os mest, er `movl $42,%eax` . Det er oversættelsen af `return 42` . movl betyder move long, altså flyt en integer, som er lang. På intel x86 kaldes 32 bit integers for long. Andre assembler sprog ville kalde det for en load operation. $42 betyder en konstant med værdien 42 (Dollar = værdi). %eax er en snedig måde at betegne 386 multi purpose registrene, procent tegnet gør, at dette symbol ikke kan være en variabel, så programmøren kan altså godt have en variabel eax i programmet, uden at den støder sammen med %eax.

Når jeg i en funktion skriver `return 42;` -- så sker der åbenbart det, at værdien, som skal returneres, lægges i det CPU-register, som er det almindeligst anvendte (og - tjek Intel manualer - også det hurtigste). Når computerprogrammet vender tilbage til de instruktioner, som kaldte vores funktion, så har den altså resultatet i %eax. I gamle dage kaldte man %eax for kalkulator registeret. I dag kan flere registre bruges som kalkulatorregistre.

Ellers er det meste af ovenstående output er fileheader og functions type-erklæringer, som jo blot kan genereres som en ramme, hvori man indsætter funktionsnavnet.

Eksempel 4-4. uC oversættelse

```
ax@pluto:fri$uC -A xyzzy.c
ax@pluto:fri$cat xyzzy.y86
#* * * * uCC Small-c Compiler v3.01,  Feb. 2001 * * * *
#Linux version 0.9 beta; 32 bit 80386 code, Serial#0018
#
 .text
 .align 4
#xyzzy()
 .globl xyzzy
 .type xyzzy,@function
xyzzy:
 pushl %ebp
 movl %esp,%ebp
#{
#    return 42;
 movl $42,%eax     # <<=== OBS! HER ER OVERSÆTTELSEN!!!
 movl %ebp,%esp
 popl %ebp
 ret
#}
 .extern
# const strings:
.section  .rodata
.LC0:
# end of constant strings


 .ident "uC 0.9e. 2001-06-02"
ax@pluto:fri$
```

Det er meget tekst, der kommer ud af uC oversættelsen, men igen er det vigtigste linjen med `movl $42,%eax` .

I foregående gcc oversættelse var der færre instruktioner. Det skyldes, at vi anvendte en option -fomit-frame-pointer, som betyder, at compileren (om muligt) ikke skal udspy instruktionerne, som saver base-pointer registeret %ebx.

En compiler oversætter de enkelte statements til symbolsk assembler, det vil sige maskin instruktioner i tekst form. Oversætterens output er en fil med symbolske maskin-instruktioner, kaldet assembler source.

Man behøver ikke at kunne *så forfærdelig* meget assembler for at få noget ud af dette afsnit. De nødvendigste ting bliver forklaret undervejs.

Måske det bedste ved uC er, at der er masser af opgaver, som man kan gå i gang med. Man kan pynte lidt hist og her på hjælpetekster, banner tekster osv. Man kan også lave ganske små ændringer, som gør den nemmere at bruge. Men man kan også forbedre den *væsentligt* på kodekvaliteten ved at tilføje nyttige funktioner i kodegeneratoren.

Når man ser, hvordan et program bliver "nedbrudt" under oversættelsen, så kan man få den helt store aha-oplevelse. P.J. Plauger har på et tidspunkt skrevet, at han for at lære et nyt programmeringssprog skrev en (lille) compiler til det pågældende sprog. Det er derfor, at dette eksempel er taget med.

4.3.1. uC - en mikroskopisk C compiler

I dette afsnit præsenteres en lille C compiler, som man selv kan skrive - eller skrive videre på, nemlig Small-C, som jeg i denne version har kaldt Micro-C, eller uC.

Small-C compileren er legendarisk. I microcomputerens barndom skrev Ron Cain en artikel (eller en artikelserie?) om en lille C compiler, som kunne kompilere sig selv, og som faktisk var/er et nyttigt sub-set af C sproget. Small-C var skrevet til Intel 8080 CPUen, en 8/16 bit CPU og det udbredte styresystem CP/M (Control Program for Microcomputers) fra Digital Research inc.

Den oprindelige source til Small-C har jeg aldrig kunnet finde. Der findes imidlertid mange varianter. En af de bedste er James E. Hendrix's Small-C 2.2, men den er på den anden side også mere kompliceret. En anden version for MS-DOS hed Caprock System PC eller CPC, og det er den, som jeg har arbejdet videre på. Den vigtigste ændring er, at den kører 32 bit under linux og at den i øvrigt benytter samme call-sequence som GNU-C.

Kildetekst og nogle primitive make filer m.v. finder man i et gzippet tar arkiv i eksempel-kataloget, uC09e-3.tgz hedder det pr. 2. juni 2001.

Eksempel på udpakning og installation:

```
[root@linus /root]# mkdir -p /hjem/src/compiler/ucc
[root@linus /root]# cd /hjem/src/compiler/ucc
[root@linus ucc]# tar xzvf /tmp/uC09e-3.tgz # OBS! Nyeste version kan hedde noget andet
[root@linus ucc]# make
[root@linus ucc]# make install
```

Hvis man kører **make install**, vil programmerne lave et symbolsk link fra /usr/local/uC til det sted, hvor man installerer fra. Derfor skal man ikke flytte installations-dir. Man kan let selv lave om på disse ting. Hvis man i forvejen har et directory /usr/local/uC vil man blive bedt om at fjerne det manuelt, så installationen er altså ikke destruktiv.

Leder man på nettet kan man sagtens finde source til andre C compilere, og som bekendt er GNU C "fri software", der distribueres med kildetekst og dokumentation, så den kan man jo også begynde at læse på.

Inden jeg valgte at det skulle være uC, som skulle med som eksempel i denne bog, havde jeg Dennis Ritchies oprindelige "pre-struct" C, LCC, cc68 og cc386 i kikkerten, før end jeg valgte denne. Den er lille og kunne nemt tilpasses til GNU/Linux miljøet. Og -- indrømmet -- jeg kendte den godt i forvejen!

Den nuværende version nummerering indeholder to oplysninger. Linux-porteringen (32 bit versionen) er version 0.9, og den centrale del, parseren, har jeg ladet hedde version 3.01 fordi den jo er baseret på tidligere versioner.

Uddrag fra uCannoncering - filen:

Revision: Version 09 (næsten poleret ...) pr. 6-apr-2001.

Parser-delen er version 3.01

Med tilføjelse af options for extended stackframe og for saving %ebx. (brug -h for at se options).

Assembler output oversættes af GNU "as" og linkes med ld via en gcc kommando.

Den kan bootstrappes på en RedHat 5.2, 6.0 og RedHat 7.0 (benyt --bx option til kompilering af main, se Changelog.dk); jeg formoder, at der også er andre distributioner, som lader sig anvende.

Koden er "acceptabelt ineffektiv", idet den performer ca 1/3 langsommere end gcc genereret kode. Det er egentlig ret forbløffende, idet compileren er meget simpel og har nogle indlysende skavanker, når der skal anvendes array indexering og pointere. Forklaringen er, at de simplere instruktioner, der anvendes, er hurtigere end de komplicerede instruktioner til adressering, som gcc kan generere.

uC er instruktiv i 2 henseender:

1. Dels viser den hvordan man laver en minimal C - compiler med en SUBSET af C, som alligevel er så kompatibelt med (old-style) C at den kan kompileres med en "stor" C compiler

2. Den kan oversætte sig selv.

Hønen og ægget! Hvad nu, hvis man ikke lige har en C compiler ved hånden. Kan man så få denne compiler til at fungere? Ja, det kan man. Den er lille nok til, at den kan hånd-oversættes.

```
EN SPECIFIKATION AF uC
```

```
-----------------------------------------------------
Preprocessor direktiver:
-----------------------------------------------------

#include <stdio.h>  /* leder i /usr/local/uC/include */

#include "fil.h"    /* current dir */

#define YXI 1       /* ikke parameter-macroer */

#ifdef              /* Betinget compilering */

#ifndef
#else
#endif
#asm                /* for hånd - optimering! */
#endasm

-----------------------------------------------------
Datatyper:
----------------------------------------------------
int x;
char w;
int array[n]
char arr[n];
int *ip;
char *cp;

eller

char s[];

extern int y;
&function();

-----------------------------------------------------
Operatorer:
-----------------------------------------------------
Assignment: =  ++ --
Aritmetik:  + - * / %
Bit:        ~ ^ | &
+shift:     << >>
Logiske:    INGEN (sic)
Dereference *
Address     &

Pointer aritmetik er tilladt (er unsigned)
Struct klares som char[n] med #define offsets

-----------------------------------------------------
Flow
-----------------------------------------------------
```

```
if (x) {}
else   {}

while(x) {
break;
continue;
}

return x;

-----------------------------------------------------
Modularitet
-----------------------------------------------------
extern int x;   /* static mangler */

/* extern f(); automatisk hvis ukendt fcn-navn optræder. */

functions returnerer en int. Altid!
function type må ikke specificeres, men den er jo også
redundant, da funktioner altid returnere en int.
prototyper er forbudt! Hu!

separat kompilering og linkning med gnu as og ld.

-----------------------------------------------------
Slut på specifikation af uC
-----------------------------------------------------
```

Det er meningen her at give en ganske kort beskrivelse af denne compiler. En forsmag kan være den centrale funktion, som parser eksterne objekter:

Eksempel 4-5. C oversætter, parsning på øverste niveau

```
/*                                              */
/*      Behandling af alt input                 */
/*                                              */
/* På dette niveau er kun statiske erklæringer  */
/*      defines, includes, og function          */
/*      definitions legale                      */
/* Input kaldes fra (a)match                    */

parse()
{
    while (eof == 0) {
        if (amatch("extern", 6)) {
            cextern = 1;
            if (amatch("char", 4)) {
                declglb(cchar); /* declare global */
                ns();
            } else if (amatch("int", 3)) {
```

```
                declglb(cint);
                ns();
            } else {
                declglb(cint);
                ns();
            }
            cextern = 0;
        } else if (amatch("char", 4)) {
            declglb(cchar);
            ns();
        } else if (amatch("int", 3)) {  /* cannot handle function type! */
            declglb(cint);
            ns();
        } else if (match("#asm"))
            doasm();
        else if (match("#include"))
            doinclude();
        else if (match("#define"))
            addmac();
        else
            newfunc();
        blanks();               /* preprocess, force eof if pending */
    }
}
```

Mere i næste version af "Friheden til at programmere i C" ...

4.4. Tilstandsmaskiner

Tilstandsmaskiner er betegnelse for 2 ting: (1) generelt for computer programmer, idet alle programmer er tilstands-maskiner, og (2) en speciel teknik, som består i, at man bruger en tabel til at beskrive de tilstande, et program kan være i. Tilstandstabeller anvendes blandt andet i parsere.

Hvis man vil se en "rigtig" tilstandsmaskine, så kan man studere output fra yacc eller bison parser-generatorer. De egner sig imidlertid også til opgaver. Da ID-Software frigav koden til Quake og Quake-2 projekterne, kunne man se, at botternes bevægelser og strategi blev styret af simple tilstandsmaskiner.

Tilstandsmaskiner kaldes mere præcist Finite State Machines, (FSM) maskiner med et endeligt antal tilstande. Tilstandene kan noteres i en tabel, for der er jo kun et endeligt antal af dem. Hver tilstand er karakteriseret af, at der er et begrænset antal muligheder for, hvordan maskinen kan reagere. Maskinen reagerer på input, som skal kunne opdeles i et endeligt antal kategorier. Input kan være fra et tastatur, men i et spil kan det selvfølgelig også være fra en joy-stick.

For at få et billede af en FSM kan man forestille sig en adventure-lignende situation: Jeg står foran indgangen til en grotte, og jeg kan vælge at gå ind, kravle op af bjergvæggen, gå baglæns eller til højre eller venstre. Jeg vælger at gå ind. Nu har jeg andre muligheder, tilstanden er en anden. Det at gå fra én tilstand til en anden kaldes en tilstandsændring (state transition) og mulighederne for hver tilstand noteres i tilstandstabellen.

Hvis jeg havde valgt at gå tilbage, havde jeg haft andre muligheder bagefter, jeg var gået til en anden tilstand.

For at tabellen kan være på siden har jeg udeladt muligheden for at gå til venstre.

foran-grotte tilstand	**Inp.kategori 1**	**Inp.kategori 2**	**Inp.kategori 3**	**Inp.kategori 4**	**Inp.kategori 5**
--	Skift til ind i grotte-tilstand	Skift til væk-fra-indgang tilstand	Skift til op-over-grotte tilstand	Skift til højre-for-grotte tilstand	Skift til ubrugeligt-input tilstand

Det er væsentligt for teorien at der er en kategori af ikke-brugbart input, og det har yderligere den fordel, at tilstandstabellen bliver mere overskuelig. Næste tabel viser hvilke muligheder, vi har, når vi er gået ind i grotten, jeg er i "grotte tilstand". Der er samme antal input-kategorier, blot for at gøre det lettere at håntere implementeringen senere.

inde-i-grotte tilstand	**Inp.kategori 1**	**Inp.kategori 2**	**Inp.kategori 3**	**Inp.kategori 4**	**Inp.kategori 5**
--	Skift til støde-ind-i-væg tilstand	Skift til foran-grotte tilstand	Skift til ubrugeligt-input tilstand	Skift til gennem-dør tilstand	Skift til ubrugeligt-input tilstand

I et rigtigt spil skal der selvfølgelig være mange, mange tilstande, ligesom der i et godt adventure spil er mange huler, grotter og underlige steder, man kan komme hen. I hver situation udløser input-katetori X et skift til en tilstand (det kan godt være samme tilstand og evt. en action, for eksempel at man kommer i besiddelse af nye ting - og det er så en anden tilstand. Antallet at tilstande kan derfor være ret stort i sådan et spil, og man laver derfor heller ikke én FSM for hele spillet, men for eksempel som i Quake en tilstandsmaskien for en "bot". Her bliver input til "bot"en så ikke nødvendigvis input direkte fra tastaturet, men resultatet af en beregning fra en tilstandsmaskine et andet sted i programmet.

Antallet af tilstande, en CPU kan være i, fås ved at tage alle registre og flag og lægge i forlængelse af hinanden og så tælle antallet af bits, T, så er antallet 2^T

Det almindeligste eksempel på en tilstandsmaskine er parsning af en tekststring, som formodes at indeholde et floating point tal, for eksempel `123.45`. Vi benytter den internationale standard for floating point, altså punktum, i hele kapitlet. Det kan være en god øvelse at lave et program, som også kan acceptere flydendetals-komma.

Næste udgave af denne bog vil bringe nogle overskuelige eksempler på tilstandstabeller, genererede med bison og "håndlavede". Følg med på www.sslug.dk i de nye udgaver af "Friheden" bøgerne!

Appendiks A. Crash course i C sproget

C består som andre programmeringssprog af:

- Datatyper. Vi vil gerne have forskellige slags variable til at opbevare tal, tekst og billeder.
- Operatorer. Vi vil gerne kunne foretage beregninger med vores data.
- Løkker og betingede sætninger (*if*, *while* m.fl.). Vi vil gerne kunne gøre noget specielt, hvis vores tal er blevet negativt.
- Modularitet. Vi vil gerne kunne genbruge kode, som for eksempel beregning af sinus eller udskrift til terminalvindue.

A.1. C i få ord

Programmeringssproget C blev oprindeligt lavet for at kunne skrive styresystemet Unix i et højniveausprog. C er derfor et effektivt sprog, det vil sige, at koden bliver oversat til så få maskininstruktioner som muligt, og derfor kører programmerne hurtigt.

C er også et lille sprog, hvis man måler det i antallet af reserverede ord (kun 32). Det gør det nemt at lære.

```
Reserverede ord:

auto            double          int             struct
break           else            long            switch
case            enum            register        typedef
char            extern          return          union
const           float           short           unsigned
continue        for             signed          void
default         goto            sizeof          volatile
do              if              static          while
```

C99 standarden tilføjer flg. reserverede ord: `inline, restrict, _Bool, _Complex, _Imaginary.` Desuden bør man have in mente, at C++ nøgleord også bør undgås, d.v.s. class, private, public, virtual etc.

Anvendt på rette måde bliver koden meget læseligt, og derfor anvendes C sproget til mange forskellige slags opgaver, fra hardware-niveau til højniveauopgaver.

C++ er en udvidelse af C. C++ er tænkt som et bedre C. Det understøtter objektorienteret programmering, dataabstraktion og generiske algoritmer. Der er dog et par forskelle mellem ANSI-C og den grundlæggende C syntaks i C++.

Et minimalt C-program:

Eksempel A-1. Hello world! programmet.

```
/* Dette program skriver Hello, world! i et terminalvindue. */

main()
{
     printf("Hello, world!\n");
}
```

Vores program definerer en funktion, d.v.s. et stykke kode, som udfører en opgave. Navnet på vores funktion er main. main er det sted, hvor et C program begynder at køre. Parenteserne efter ordet main fortæller, at det er en funktion.

Koden, det, som bliver til maskininstruktioner, står mellem to krøllede parenteser (eng. braces). Koden består af et kald til en anden funktion, printf, som får til opgave at skrive vores tekst på skærmen. Vores tekst står mellem dobbelt-quotes (gåseøjne, citationstegn). "\n" er måden at skrive et linjeskift, som indgår i en tekst streng. '\' - tegnet, kaldet backslash, signalerer, at nu kommer et bogstav, der skal opfattes på en anden måde. '\n' kaldes derfor et meta-tegn, og backslash anvendes som *escape kode*.

I kildeteksten kan man skifte linje stort set hvor man har lyst.

Ovenstående kildetekst bør kunne oversættes (compileres) til et kørbart program.

```
don@pluto $ gcc hallo.c -o hallo
don@pluto $
```

-o optionen fortæller compileren, at det færdige program skal skrives til en fil, der skal hedde hallo. Ellers vil compileren lægge det oversatte program i en fil ved navn a.out. Nogle moderne oversættere vil klage over udeladelser. GNU-C vil kun klage, hvis vi anvender kommandolinje option -Wall, som betyder: "Giv mig alle advarsler mod uheldige eller mangelfulde konstruktioner!" En helt korrekt version (også C++) ser ud som følger:

Eksempel A-2. Hello, World! med type-specifikationer.

```
/* Dette program skriver Hello, world! i et terminalvindue og er
 * meget omhyggeligt med anvendelse af typer. */

#include <stdio.h>

int main()
{
     (void) printf("Hello, world!\n");
     return 0;
}
```

Eksemplet her erklærer, at main er en funktion, som returnerer en int (integer, heltal). Med sætningen `return 0;` returnerer den jo så altså netop også et heltal, nemlig 0. `return 0;` betyder aflever et 0 til den, som satte funktionen i gang. Man siger, at caller får returneret en værdi.

printf er ligeledes en funktion, som returnerer en integer til caller; det er antallet af bogstaver, den har skrevet ud. Da vi ikke har brug for denne integer, fortæller vi oversætteren, at det er "med vilje", at vi ignorer return value ved at skrive *(void)* . Det kaldes et cast, støbeform: vi omstøber funktionens type!

A.1.1. Data typer

C sprogets indbyggede datatyper er:

```
int h;         /* integer, heltal, maskinens hurtigste type, native type */

char b;        /* velegnet til bogstaver, men i virkeligheden bare en lille */
               /* integer, 8 bits (eller 9 ...) */

float f;       /* floating point med kun 32 bit, kun til grafikkort o.l. */

double df;     /* 8 eller 10 bytes floating point type */

int ia[10];    /* array med 10 elementer, mange af samme slags */

struct person_type {             /* gruppering af mange typer i en klump */
   int alder;
   char navn[10];
};

struct person_type mig;            /* udlaegning af lager, instantiering */
struct person_type medlem[10];     /* array af struct */

int *j;                            /* adressevariabel, pointer */
```

En variabel, for eksempel *int xyzzy*, kan erklæres udenfor en funktion, så er det et eksternt objekt, eller inde mellem krøllede parenteser, braces, så er det en lokal variabel.

Jeg har med vilje udeladt en del grumsede detaljer i ovenstående type-oversigt. Hvis du, kære læser, er sur over det, så er her et resume af det udeladte:

En long integer er lige så stor - måske større - end en int. I gcc i386 er de begge 32 bit. Nye standarder vil indeholde forskrift om at oversættere skal understøtte _int8, _int16, _int32, _int64 etc. hvilket vil være praktisk for mange opgaver. Man kan definere sådanne typer selv ved hjælp af `typedef`. For eksempel:

```
typedef long long int _int64;
```

Heltalstyperne (char, int, short int, long int, _int64 eller long long) kan foranstilles *signed* (det er default) eller *unsigned*. Erklærer man en unsigned integer vil man i mange oversættere alligevel kunne tilskrive den en negativ value. Unsigned vinder overfor signed i type coercion i expression evaluering. (Coercion: tvang).

```
/* ramme for eksperimenter med variable af indbyggede typer */

int x;
char b;

main()
{
    double temp = -17.8;

    b = 67;
    printf("x er %d, b er %d, temperatur er %f\n",x,b,temp);
    {
        int *px; /* definition af variabel i top af blok. */
        px = &x; /* tag adressen af x og læg over i p */

        printf("adressen på x er %p\n",px);
    }

    return 0;
}
```

px er en adressevariabel, d.v.s en variabel, som kan indeholde en adresse. Det kaldes også en pointer. Det ses af, at erklæringen har en stjerne foran px. Man må gerne stille stjernen lige bagefter int, men syntaktisk binder den til px. Derfor kan man fx. erklære en integer pointer og en integer indenfor samme semikolon, `int* ptr2tal, tal;` (kønt er det ikke! Det bør virkelig undgås, dette her!)

"Og" tegnet, ampersand på engelsk, er adresse-operatoren d.v.s. at den tager adressen på den variabel, den stilles foran. `px = &x;` betyder tag adressen på x og læg den over i variablen px.

Adresser på en 32-bit CPU ligger mellem 1 og 4 milliarder eller 4 Giga. Altså 2^{32}.

A.1.1.1. Struct - forskellige typer grupperet i en klump

Med struct konstruktionen kan vi selv definere de datatyper, vi har brug for, deraf betegnelse *brugerdefinerede typer.* Notationen kræver lidt øvelse. Først bygger man en type op, det kaldes, at man *erklærer en type* . Derefter kan man definere objekter af denne type. Når man definerer et objekt, reserveres der hukommelse i det færdige program til objektet.

Navnet efter ordet struct kaldes en tag (eller type tag; det betyder et mærke eller en mærkat, udtales med g som i væg). Det kan sammen med ordet struct bruges som type-specifikation. I eksemplerne anvendes vinkler (større/mindre tegn, eller på engelsk: angles) til meta ord - der skal indsættes et unikt navn på de pågældende positioner.

```
struct <type_tag> {
   data-declarations ...
};

struct <type_tag> new_var;
```

Konkret eksempel:

```
struct hus_t {
   int grundareal;
   int bebygget_areal;
   int pris;
};

struct hus_t madsensvej_20;
```

Alternativt kan man opbygge en struct type ved hjælp af typedef. Denne metode foretrækkes af mange, fordi man så ikke behøver at anvende ordet struct når man definerer nye objekter.

```
typedef struct <type_tag> {
   <data-definitions> ...
} <type-specifier>;           /* <--- her er den nye types navn */

<type-specifier> min_variabel;
```

Et konkret eksempel på anvendelse af typedef-metoden:

```
typedef struct {
   int medlems_nr;
   char navn[800];
} medlems_type;                /* <--- her er den nye types navn */

medlems_type medlem[200000];
```

Somme tider gør ordet struct programmerne lettere at læse. Og ordet struct, gør det lettere for oversætteren at give forståelige fejlmeddelelser.

```
struct hus_t {                  /* først erklæres ny type */
   char adr[280];
   int pris;
};

struct hus_t mithus;            /* typen anvendes, ram udlægning sker */

main() {
   mithus.pris = 780000;           /* nu kan vi bruge vores variabel */
   strcpy(mithus.adr,"Byvej 20";  /* initialisering af string kræver strcpy */
}
```

Desuden findes der en nummereringstype, enumeration, som typisk anvendes til en serie symbolske konstanter:

Eksempel A-3. Enumeration

```
enum farve_t { red, green, blue };
```

red vil her svare til værdien 0, *green* til 1 etc. etc... red, green og blue vil indgå som identifiers på linje med andre variable. Man kan nu erklære en variabel af denne type, og en C++ compiler vil kunne kontrollere, at vi ikke tildeler vores variabel ulovlige værdier. En C-compiler overlader dette ansvar til programmøren.

En variant af struct er "union", som svarer til en slags forudbestilt typecasting. Den er som skræddersyet til en program-generator, der skal generere en oversætter - altså en oversætter som oversætter en oversætter. Dette æske system har fra tidligt i 70'erne i unix sammenhæng fået det sjove navn YACC, Yet Another Compiler Compiler. I Gnu tool-settet er det blevet til BISON - fordi bison er en fætter til Yak-oksen. Ja-ja, læs mellem linjerne, at denne her union *er* kun til specielle lejligheder :-))

Eksempel A-4. Union

```
union yylval_type {
    long itype;
    tree ttype;
    enum tree_code code;
    const char *filename;
    int lineno;
};
union yylval_type yylval;
```

A.1.2. Operatorer

Aritmetiske operatorer.

```
+ plus, addition
- minus, subtraktion
* asterisk, multiplikation
/ stroke ell. slash, division
% percent, modulus, rest af heltalsdivision.
```

Assignment operatorer, tildeling.

```
Op.   ex.          Beskrivelse
-------------------------------------------------------------------------------
=     a = b     :  læg værdien af b over i a.

++    ++c       :  læg en til c, increment operator.

                :  ++ operatoren kan stå efter identifieren:
  if (c++ < 42) :  sammenlign med 42 og læg bagefter 1 til c.

--    --c       :  træk 1 fra c, decrement operator.
-------------------------------------------------------------------------------
```

Desuden kan lighedstegnet kombineres med de forskellige aritmetiske operatorer:

```
x += kaxi; /* tael x op med vaerdien af variablen kaxi */
```

Relationelle operatorer, sammenligning.

```
< mindre end
> større end
<= mindre end eller lig med
>= større end eller lig med

OBS!  ==    test for lighed
      !=    test for not lighed
            Husk det ved at tænke på at
            den logiske operator NOT er et udråbstegn.
      !     not (tager kun én operand, hvis logiske værdi inverteres)
```

Et expression er sandt, når det er forskelligt fra 0.

Et expression er falsk, når det er lig med 0.

Eksempler:

```
  if (a > b)
      printf("a er for stor\n");

  if (x > 1 || x != y)
      printf("x er større end 1 eller x er forskellig fra y!\n");
```

Logiske operatorer.

```
&& logisk AND
|| logisk OR

!  logisk NOT
```

Bitwise operators, bitvise operatorer, manipulerer de enkelte bit eller bitkolonner efter den almindelige logik.

```
&    bitvis AND              1 & 1 == 1; 17 & 1 == 1;
|    bitvis OR               1 | 0 == 1; 17 | 1 == 17;
^    bitvis XOR              1 ^ 0 == 1; 17 ^ 1 == 16;
~    bitvis NOT              ~1    == 0xfffffffe;
>>   shift right             1 >>1 == 0; 17 >>1 == 8;
<<   shift left              1 <<1 == 2; 17 <<1 == 34;
```

Pas på NOT operatoren, det er en tilde. I almindelig netscape opsætning er den meget svær se.

Bitvis NOT er det samme som bitvis invertering. I ovenstående eksempel er resultatet skrevet ud fra en antagelse af, at der er tale om en 32-bit størrelse. Hvis der er tale om 64 bit, så vil der være 15 f'er i stedet for "kun" 7. Hvis 32-bit størrelsen skulle skrives som bit-mønster, så er der selvfølgelig 32 "cifre" der enten er 0 eller 1. Det er en god ting at lave et mellemrum for hver fjerde. Så kan man bedre jævnføre med hexadecimal notation.

```
~1 == 1111 1111 1111 1111 1111 1111 1111 1110.

      f    f    f    f    f    f    f    e

      15   15   15   15   15   15   15   14

Så altså, det er lidt nemmere at læse hexadecimal notation:

~1 == 0xffffffe
```

Der er desværre ikke nogen standard funktion, som udskriver bit-mønster for en integer. Hvis man vil skrive resultatet som BIT-mønster, så må man programmere en funktion, der tester med bitvis and og derefter foretager et shift, fx.

```
    /* udskriv x's bitmønster: */
    {
        int i = 0;
        char numstr[35];                    /* 2 + 32 + end of string */
```

```
        strcpy(numstr,"b:");                /* "b:" for binaert format */
        numstr[34] = 0;                     /* End of string mærke */
        while (i++ < 32) {
            numstr[34-i] = (x & 1) + '0';
            x = x >> 1;
        }
        printf("%s\n",numstr);
    }
```

Du kan afprøve de ovenstående ved at sætte den ind i en main() funktion, hvor man allerøverst definerer en int x = 0xf0f0f0f0. (File: bitmonst.c).

De andre bitmanipulationer kan også afprøves med små programmer som nedenstående, god øvelse:

```
main()
{
    printf("17 >> 1 == %d \n",17>>1);
    return 0;
}
```

Spørgsmålstegns-operatoren

Specielt for C sproget er betingelses-operatoren (conditional operator) spørgsmålstegnet:

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

Svarer til:

```
  if (a > b)
     yxi = a;
  else
     yxi = b;
```

Andre operatorer

Ud over ovenstående findes der adskillige specielle operatorer, der kun må anvendes efter forudgående aftale med typetjekningssystemet. ;-)

```
 &   adresse operator, tager adressen af et objekt.

 *   asterisk operator, tag indholdet på den adresse, som specificeres
     efter stjernen,  dereferering af adresse.

 ->  kan bruges ved dereferering af en struct pointer.

 .   Bruges til adressering af et struct element.

 ()  Funktions operator.
```

```
[]  Array operator.

sizeof  (en pseudo funktion, kan bruges som operator, giver os
         størrelsen af det objekt, som den står foran.)

,   Komma, listeoperator.
```

Her er et simpelt eksempel - lav selv flere, bare for at afprøve de enkelte operationer!

```
int charcnt[256];

main()
{
    int yxi;
    double kaxi;
    long double kolme;

    printf("Size of charcnt: %d\n",sizeof charcnt);

    /* sizeof anvendes ofte som en pseudo funktion */
    printf("Size of yxi....: %d\n",sizeof(yxi));

    printf("Size of double.: %d\n",sizeof kaxi);
    printf("long double....: %d\n",sizeof kolme);

}
```

Eksempel A-5. Operator Præcedens

operatorer	niveau	associativitet	kommentarer
`() [] -> .`	1	v-h	Funktions-parenteser, array-index, struct-element. Dette niveau er det, der binder mest. Grupperings-parenteser er ikke en operator, men anvendes til at ændre rækkefølge af evaluering.

operatorer	niveau	associativitet	kommentarer
`! ~ ++ -- -` `(cast) * & sizeof`	2	h-v	not, bit not, increment, fortegns-minus, cast, pointer afreferering, adressen på, objekt-størrelse
`* / %`	3	v-h	multiplikation, division, modulus
`+ -`	4	v-h	addition, subtraktion
`<< >>`	5	v-h	shift
`< <= > >=`	6	v-h	sammenligning, relationsoperatorer
`== !=`	7	v-h	relationelle, test for lighed/forskel
`&`	8	v-h	bitvis AND
`^`	9	v-h	bitvis XOR
`|`	10	v-h	bitvis OR
`&&`	11	v-h	logisk AND
`||`	12	v-h	logisk OR
`?:`	13	v-h	betingelses-operator
`= += -= *= /= %=` `&= ^= |= <<= >>=`	14	h-v	tildeling (assignment)
`,`	15	v-h	komma, listeoperator; den mest adskillende operator; dette niveau skiller næsten lige så meget som semikolon

A.1.3. Flow kontrol

Flow kontrol er maskinens måde at reagere på data. Man kan klare sig glimrende med færre, for eksempel med if, while og goto! Men C sproget er kendt for sine gode flow-konstruktioner.

```
if (betingelse_opfyldt)
    do_dyt();
```

Test inden udførelse:

```
while (betingelse_opfyldt)
    do_looping();
```

Kør loop-body mindst en gang:

```
do {
    mindst_en_gang();
} while (betingelse_opfyldt);
```

Bemærk at man altid bør bruge braces her for at undgå forveksling med en ordinær while-løkke.

Behagelig kontrol med tællevariable:

```
for (initialisering; betingelse; optælling) {
   loop_body ...
}
```

for eksempel

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

Inde i loops kan man:

```
break:      Goto lige efter loop-end.

continue:   Begynd forfra med test af betingelse.
```

En switch realiseres om muligt som en jumptabel. En jumptabel er en *meget* effektiv måde at teste på størrelsen af en int:

```
switch (integer_variabel) {
case 17:
         do_beep();
         break;
case 42:
         do_hurra();
         break;
default:
         do_whine();
}
```

For at komme ud af en masse loop-kontrol statements:

```
   goto label;

   /* kode ... */
label:
```

Bemærk, at case linjerne i switch statementet ligner og opfører sig som labels, de er faktisk labels. Man fortsætter nedefter i næste case, hvis der ikke er et break statement.

Et par eksempler, meget simple, først et eksempel, som smager på værdien af en variabel:

```
main()
{
    int i,j,k;

    i = 27;
    j = 2;

    if (i > j)
        k = i;
    else
        k = 2;

    return k;
}
```

Et eksempel på en for-løkke:

```
main()
    int fahr, celsius;

    for (fahr = 0; fahr < 200; fahr = fahr + 20) {
        celsius = 5 * (fahr - 32) / 9;
        printf("Fahrenheit %3d svarer til celsius %3d\n", fahr, celsius);
    }
}
```

A.1.4. Modularitet

Funktionsbegrebet i C gør det muligt at genbruge kode; man kan bygge på andres arbejde i stedet for at begynde på bar bund hver gang.

Et C program består typisk af mange forskellige filer, der kan oversættes hver for sig. En fil - evt. med tilhørende header files - der kan oversættes alene til et objekt modul, kaldes en oversættelses-unit eller en translation-unit.

Allerede i vores første program benyttede vi os af, at der i et bibiotek *library* lå en funktion (printf) som kunne skrive tekst ud på terminalvinduet.

Det er muligt at have private variable i en translation unit.

```
/* modular programmering - fil nr. 1, kryptio.c */
```

```
/* main læser fra tastatur (eller omdirigeret fil)
 * og skriver det krypterede bogstav ud på skærmen.
 */

#include <stdio.h>

main()
{
    int c;
    while ( (c = getchar()) != EOF) {
        c = krypter(c);
        putchar(c);
    }
}
```

Til ovenstående main vil vi nu skrive et simpelt modul, som foretager kryptering:

```
int krypter(int inputchar)
{
    return inputchar + 1;   /* 'cæsar' kryptering */
}
```

Uha, det viser sig snart, at folk gennemskuer vores simple kryptering, så nu laver vi en rigtig kryptering. Det smarte er, at vi kan erstatte dette modul uden at lave om på det eller de programmer, som anvender vores funktion "krypter()".

Bliv nu ikke forskrækket over, at der er en del ting i næste eksempel, som ikke er fyldestgørende forklaret endnu. Prøv at læse det, evt. indtaste og oversætte det. Prøv så at rette i det for at se, hvad der sker undervejs. Når et problem er kompliceret, så skil det ad i mindre dele og indsæt printf statements, så du kan se, hvad der sker undervejs.

Eksempel A-6. Simpel kryptering

```
/* file krypter1.c - en brugbar (men forsimplet)
 * krypteringsalgoritme. Kan ikke håndtere linjeskift m.v.
 */

int krypter(int inpchar)
{
    static char *keystring = "Under traeerne var der stille og roligt.";
    static int inuse;
    static char *mv;
    static int keylen;

    if (!inuse) {
        inuse = 1;
        mv = keystring;
        keylen = strlen(keystring);
    }
    if (mv - keystring > keylen)
```

```
        mv = keystring;
    if (inpchar < ' ')
        inpchar = ' ';
    else if (inpchar > 126)
        inpchar = '~';
    return (inpchar + *mv++) % 93 + 33;
}
```

Virker kun for ren ASCII tekst. [1]

Ordet static, som står foran de 4 variable, som skal anvendes i modulet her, betyder, at de ikke må kunne bruges fra andre moduler eller funktioner i det færdige program. Det er såmænd ikke fordi de skal være hemmelige, men blot for at sikre, at vi har styr på, hvor der sker ændringer af variablen, som peger fremad i krypteringsnøglen.

Hvis du blev bidt af ovenstående, så lav en dekryptering til den. Eller læs eksempel-programmet afkrypt1.c. Desuden er der en forbedret version, som håndterer linjeskift, krypter2.c og afkrypt2.c

Det var små eksempler på anvendelse af datatyper, operatorer, flowkontrol statements og modularisering. Det næste afsnit af crash course i C programmering går lidt mere i dybden med disse emner.

Først et spørgsmål: Lagde du mærke til, at variablen 'keylen' havde fået ordet *static* stillet foran? Det er en storage specifikation. Læs videre!

A.2. Lager klassifikation, storage classes

Emner:

- externe variable, funktioner og konstanter
- funktioner og lokale variable (auto)
- programsektioner, stak, heap, bss, data, text
- kontrolvariable for løkker eller loops
- scope (synlighed) og varighed, lokale static variabel
- rekursion og statics
- funktioner som returnerer pointer til buffer
- volatile,
- register variable.

Et C program består af definitioner af eksterne objekter, som kan være funktioner eller variable. Eksterne objekter kan bruges overalt i det program, hvori de forekommer, også fra separat oversatte funktioner.

Modsætningen til eksterne variable er de variable, som defineres inde i en funktion og de parametre, som funktionen modtager når den bliver kaldt. De eksisterer, mens funktionen kører. De bliver automatisk oprettet når programmet kører funktionen (eller blokken) og nedlægges igen, når programmet forlader funktionen. De kaldes derfor også automatiske variable. Det reserverede ord *auto* kan anvendes om lokale variable, men er overflødigt - det stammer fra de tidligste versioner af C sproget.

Funktioner er altid eksterne objekter; man kan ikke definere en funktion inden i en anden funktion, sådan som man for eksempel kan i Pascal sproget.

Selv om eksterne objekter uden videre er "synlige" i hele programmet, er der en metode til gøre dem private for et modul. Det skal vi se på senere i dette afsnit.

A.2.1. Externe variable, funktioner og konstanter

For helt at være klar til diskussion af eksterne variable skal det lige understreges, at et C program næsten altid opbygges af separat oversatte moduler. De fleste af dem ligger måske i et library - måske er det standard-library, libc.a - Andre ligger i vores egne oversatte moduler, og vi kan give oversætteren besked om det på kommandolinjen ved simpelthen at nævne alle de moduler, som skal med. Oversætteren består af mange forskellige programmer, og det sidste af dem, nemlig linkeren, foretager sammenkædningen af modulerne (link = hægte eller led i en kæde).

```
gcc -o mitprogram modul_1.c modul_2.c modul_3.o
```

Oversætteren finder ud af, at modul_3.o er et oversat modul og nøjes med at medtage det i link-kommandoen, som bygger den kørbare fil, "program-image filen" eller bare "programmet".

En definition af en (ekstern) variabel består af en type angivelse og et navn, kaldet en variabel-identifikation, på engelsk: identifier.

```
typespecifikator variabel-navn [, variabel-navn] ... ;
```

Altså typisk for eksempel følgende:

```
int en_taeller;
```

En funktion defineres nogenlunde på samme måde, altså ved en type og et navn (lige som en variabel) men derefter har vi brug for nogle flere oplysninger, nemlig en parentes med argumentlisten, og de program-sætninger (statements), som udgør funktionens *krop*, omsluttet af braces.

```
typespecifikator funktion-navn(parameter-liste)
{
    funktion-sætning(er);
}
```

En minimum funktion ses nedenfor:

```
int placeholder() { }
```

Selv om en tom funktion ikke laver noget, kan den dog være ganske nyttig under udviklingen af et større program ved simpelthen at være til stede, evt. returnere en brugbar (default) værdi.

Der skelnes i C sproget mellem en definition og en deklaration. På dansk ser man ofte ordet erklæring i stedet for deklaration.

Hvis vi ønsker at anvende en variabel, som allerede er defineret i et andet modul, så bruger vi en deklaration. Det er en forklaring, så at sige, til oversætteren om, at der er en variabel til rådighed i et andet modul i det færdige program, som hedder sådan og sådan og er af den og den type, men som altså IKKE findes i nærværende overstættelses-enhed (translation unit).

```
extern int en_taeller;
```

Nu ved oversætteren, at der i et andet modul findes en variabel, som hedder en_taeller, og at det er ok, at vi bruger den.

Hvis vi derimod vil reservere lager til en variabel, så kaldes det definition af en variabel. Når programmet kører og vi opretter et objekt (enten ved hjælp af dynamisk lagerallokering eller ved hjælp af lokale variable i en kodeblok) kaldes det instantiering af et objekt.

I C foregår deklaration og definition ofte på samme tid. Når jeg skriver *int xyzzy* foretager jeg både deklaration og lagerudlægning.

Hvis jeg derimod skriver *extern int xyzzy* udlægges der ikke lager, men jeg indføjer et punkt i oversætterens symboltabel. I denne entry ligger oplysning om navn og type.

I det andet modul skal der være defineret en variabel på *det yderste niveau* - så er det en ekstern variabel, som kan eksistere uafhængigt af de funktioner, som det andet modul måtte indeholde (måske indeholder det kun variable). En variabel, som er erklæret i *det yderste niveau* er altid en variabel, som kan ses af andre moduler. Den er, pr default, eksporteret, ville man måske sige i sprog som ADA og Oberon.

En ekstern variabel er - i C sproget - også altid en *statisk* variabel. Sådanne statiske variable har lige så lang levetid som det program, der anvender dem. På den måde ligner de programmets instruktioner.

Programkode og statiske data ligger da også anbragt på samme måde i lageret, nemlig forneden, på de lave adresser, under det område, som anvendes til hhv. bunke (heap) og stak. Både heap og stak kan vokse, så det er meget passende, at de enes om at udnytte det store mellemrum fra "bunden" med faste data og programinstruktioner og "toppen" hvor operativsystemet forlanger at få lov til at være i fred.

toppen af lageret
Operativ system kode
Stak for bruger-applikationen
Mellemrum mellem stak og heap
Heap til dynamisk lageradministration
.bss, nulstillede externe variable
.data initialiserede, eksterne variable
.string konstanter mv.
.text, program instruktioner

Der er dog lidt forskel på, hvor data anbringes, er de initialiserede, således som for eksempel vores keystring i eksempel Eksempel A-6, anbringes de i en sektion, som kaldes .data. Hvis vi ikke selv specificerer, at data skal initialiseres, anbringes variablen i den sektion, som kaldes BSS (eller .bss), og alle bit-terne vil blive nulstillet.

Navnet *bss* har rødder i IBM mainframe assembler sproget, d.v.s. system 370, 390, 3900 og MVS. Det står for *Block Started by Symbol*.

Lad os i stedet huske det som *Base Static Storage*. Det er garanteret, at variable, som erklæres uden for en funktionsblok men som ikke er explicit initialiserede ved en "initializer", er nulstillede ved programmets start. Det sørger opstartsmodulet af programkoden for, traditionelt kaldet c0.c

Læg især mærke til, at lokale variable anbringes på "stak for applikationen". Det er de højest tilgængelige adresser for vores program. Herfra kan stakken vokse (hov - den vokser nedad!) indtil den møder heap. På en linux-maskine kan den vokse mere end to gigabyte, før der sker sammenstød. MS NT har "kun" afsat 2 Gb til applikationer.

A.2.2. Storage specifications

Det reserverede ord *static* betyder selvfølgelig statisk. Det kan, som vi har set i Eksempel A-6, bruges, hvis man *inde* i en funktion ønsker at have en variabel, som overlever funktionen, så at sige, husker fra gang til gang, hvad værdi, den havde sidste gang. For eksempel en sidetæller i en udskriftsfunktion.

```
void topmargin_udskrift(char *section_navn)
{
     static int sidenummer;
     printf("%s %d\n", section_navn, sidenummer);
}
```

Stakken gør det muligt at have lokale variable i funktioner, og for at have et sæt variable HVER gang funktionen kaldes, det er betingelsen for, at man kan benytte rekursion. Tilsvarende er heap'en det

område, som benyttes til dynamiske lager-strukturer som for eksempel en "liste" og et "træ". (se kap. 3, Afsnit 3.3).

De steder, hvor der i nedenstående opstilling af lager-anvendelse er anbragt nogle punktummer på sidelinjen er steder, hvortil der ikke nødvendigvis er knyttet noget RAM, fysisk memory. Programmets anbringelse i lageret ser normalt ud sådan her:

```
+----------------------------+
| Den øverste gigabyte       |
|   som operativsystemet     |
|     benytter og beskytter  |
|                            |
.                            .
.                            .
.                            .
|                            |
|                            |
+----------------------------+
|                            |
|  Stack område for          |
|          brugerprogram     |
|                            |
|  Her laves en ramme med    |
|  data for hver function    |
|  som kaldes                |
|                            |
.                            .
.                            .
.                            .
.                            .
|                            |
+----------------------------+
|                            |
|  Heap - der, hvor man kan  |
|    allokere mere lager     |
|                            |
|  Benyttes af funktionerne  |
|    malloc(3), calloc(3),   |
|     free(3) og realloc(3). |
|                            |
|                            |
+----------------------------+  -
|                            |   \
|  bss -                     |    \
|    blok started by symbol  |     \
|  ell. base static storage  |      NULSTILLES
|                            |      ved load af program
|  static variable som IKKE  |     /
|  er initialiserede         |    /
|                            |   /
+----------------------------+  -
|  .data                     |
```

```
|  initialiserede strings mv.|
+----------------------------+
|  .text                     |
|  Programmets instruktioner |
|  d.v.s. kommandoer til CPU |
+----------------------------+
| Lille stykke i bunden til  |
|   Operativsystemets        |
|   Hardwarehaandtering      |
+----------------------------+
```

Hvis vi ønsker at anbringe en variabel i .bss sektionen kan vi enten erklære den uden for en funktion eller benytte lagrings-specifikationen *static*. Sidstnævnte metode muliggør statiske variable med lokal synlighed (altså *kun* synlige inde i en funktion eller blok). Hvis vi benytter "static" på eksternt niveau, d.v.s. uden for en funktionsdefinition, har specifikationen den virkning, at variabelen ikke får ekstern synlighed (external linkage).

Hvis en variabel erklæret inde i en funktion skal kunne huske sin værdi fra gang til gang, når funktionen kaldes, skal den defineres som static, hvilket vil bevirke, at den bliver anbragt i .bss sektionen.

A.2.3. Konstanter og memory protection

Der er forskellige måder at organisere kørselen af et program, men stort set alle benytter ovenstående opstilling som udgangspunkt. Det egner sig til virtual memory management, hvilket jo fra starten var udgangspunktet for Linux, og det giver nogle fordele, som er ret store. Vi kan simpelt hen lade vores program benytte 3 Gb memory - også selv om vi ikke har installeret så meget. Afhængig af operativsystemets snedighed vil der kun blive tildelt fysisk memory, når det er strengt nødvendigt. Det vil sige, hvis vi vil skrive til memory. Når vi har loadet vores program og vil starte det op igen fra en anden brugers terminal, kan man stole på, at operativsystem kernen ikke bruger mere RAM til at loade endnu en kopi af program instruktionerne.

Så opdelingen af programmet i sektioner skyldes både vores egne behov for at strukturere programmet og så et hensyn til operativsystemet. Sektionerne skal gøre det lettere for operativsystemet at optimere anvendelsen af memory. Nogle sektioner skal kun læses, for eksempel program-instruktionerne, andre skal både læses og opdateres, og i hhv. heap og stack skal man kunne udvide området under kørselen.

Hvis nu vores program alligevel forsøger at skrive på fx. adressen for en funktion, så skal Linux kernens memory management modul reagere ved at stoppe programmet.

På samme måde er der dele af initialiserede data, som ikke må overskrives. Fx. streng-konstanten "Hello, world!\n" i eks. Eksempel A-1. Man kan heller ikke så nemt komme til at gøre det i netop det eksempel. Men se lige her:

Eksempel A-7. Konstant streng

```
/* constviol.c, kørselsfejl når man ændrer en konstant. */
#include <stdio.h>

char * ms1 = "Hello, ever changing world!\n";

int main()
{
     *ms1 = 'A'; /* segmentation violation, signal 11 */
     return 0;   /* abort forinden, exitkoden er 139 == 128+11 */
}
```

A.2.4. Storage specifikation og scope

Scope betyder synlighed (within scope = "inden for synsvidde"). En af de rigtig gode ting ved C sproget er, at man kan skrive programmerne således, at variable kun er tilgængelige i en funktion eller et modul. Derved kan man uden stort besvær anvende objektorienterede programmeringsprincipper.

Eksempel A-8. Scope eller synlighed

```
/* ... demonstration af scope regler, synligheds regler. */

char * ms1 = "Min nye Hello-world rapport";
char * ms2 = "(c) 2001 Donald Axel, 3210 Vejby";
static char *msp = "Denne char-ptr kan kun bruges i denne fil."

headerlinje(){
     static int sidenummer; /* kan kun bruges i denne funktion */
     printf("%s Side %d\n", ms1, ++sidenummer);
}
```

Hvis vi definerer en variabel, for eksempel ms1, på samme niveau som funktioner, så kaldes den et eksternt objekt. (Husk: Et C program består af en række definitioner af eksterne objekter.)

Variablen ms1 vil - uden særlig angivelse af, at den er "public", blive synlig i andre kildetekst filer. De skal blot erklære en extern char * ms1;

Eksempel A-9. Extern declaration

```
/* External declaration */

extern char * ms1;        /* synlig i hele denne source-fil.*/

bundlinje(){
     extern char * ms2;   /* kun synlig i denne funktion */
     printf("%s\n", ms1);
```

```
     printf("%s\n", ms2);
}
```

Hov! Hvad er nu det - kan man deklarere en "extern" *inde* i en funktion? Jada, og så er det meningen, at oversætteren skal redde os, hvis vi kommer til at bruge den *uden* for funktionen. Men deklarationen af extern char *ms2; inde i funktionen forudsætter, at variablen er defineret (og helst også initialiseret, i dette tilfælde!) et andet sted, d.v.s. i en anden kildetekst-fil (source-file).

Bemærk dog, at det stadig accepteres af mange oversættere, at man erklærer en ekstern variabel flere gange med samme navn. Det kan man vist roligt sige er en praksis, som bør undgås. Specielt kan det nævnes, at gcc accepterer at man definerer den samme variabel to eller flere gange, hvis den defineres i ydre niveau og dermed implicit er ekstern variabel!

```
/* fil 1. Demonstration af uheldig praksis. */
/* programmet oversættes uden advarsler - ikke engang option
 * -Wall, som ellers betyder "giv mig alle advarsler"
 * (warnings, derfor -W), får gcc til
 * at reagere på denne konstruktion! Men benyttes g++ får man
 * heldigvis en fejl på denne uheldige fremgangsmåde. */

int min_globale_variabel; /* ok, almindelig erklæring af en variabel. */

int min_globale_variabel; /* uha - burde udløse en warning! */
int min_globale_variabel; /* uha - burde udløse en warning! */

int main()
{
     printf("Værdien af min_globale_variabel er %d\n", min_globale_variabel);
     return 0;
}
```

Det er jo godt nok kedeligt, hvis man tror, at man har to forskellige variable, og oversætteren tror, at der kun er en. Heldigvis findes der et værktøj (ud over c++) som kan afsløre denne og andre sjuskefejl. Lint, en statisk syntakstjekker, har været på banen siden Unix version 7, altså den første offentlige udgave fra 1978. Der findes nu en fri lclint, som er fantastisk god til at gennemgå programmer, og som vel at mærke kan komme igennem include-filer uden brok. Lclint er så pedantisk, at man somme tider ønsker at slå en masse warnings fra, men det er der heldigvis også altid anvisninger på.

```
fri2c: lclint globviol.c
pluto:/qf3/attic/don/fri ::lclint globviol.c
LCLint 2.4b --- 18 Apr 98

globviol.c:14:5: Variable min_globale_variabel redefined
  A function or variable is redefined. One of the declarations should use
  extern. (-redef will suppress message)
   globviol.c:12:5: Previous definition of min_globale_variabel
```

```
globviol.c:15:5: Variable min_globale_variabel redefined
   globviol.c:12:5: Previous definition of min_globale_variabel
globviol.c:12:5: Variable exported but not used outside globviol:
                    min_globale_variabel
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (-exportlocal will suppress message)

Finished LCLint checking --- 4 code errors found
```

lclint er endog så omhyggelig, at den advarer mig om, at min_globale_variabel faktisk ikke bliver brugt i andre moduler (der er jo heller ikke andre lige for tiden ...) og at jeg derfor burde erklære denne variabel for static. Godt gået.[2]

Hvis man omvendt kun har erklæret alle sine eksterne variable med keywordet extern, så vil gcc beklagende meddele "undefined reference to min_extern". (Se øvelser i kap. 2 for at få mere føling med håndtering af variable i C programmer).

Tilbage til external variables!

Lige som extern kan også *static* bruges inde i og ude af funktioner.

Resumé: En variabel, som er erklæret i starten af en funktion, er lokal - den er kun synlig i funktionen. Hvis man angiver, at den skal være static, så anbringes den fysisk i en datasektion, som eksisterer i hele programmets køretid. Men den er stadig kun synlig i funktionen, hvori den er defineret.

Så hvad nu hvis man anvender det reserverede ord `static` *udenfor* funktionerne, altså lige som eksterne objekter? Jo, så bliver variablens synlighed indskrænket - ikke til en funktion, vi er jo udenfor funktionerne - men til den fil, hvori den er erklæret.

En af de lidt morsomme ting i C sprogets anvendelse af ordene er, at ordet *static* derved kommer til at betyde, at en variabel er "privat" for det modul, hvori den er erklæret!

Den oprindelige betydning af specifikationen "static" er selvfølgelig, at variablen skal være statisk. Den skal anbringes i sektionen for globale data, statiske data. De kalde statiske, fordi de eksisterer hele tiden, så længe programmet kører. Men for at give programmøren en hjælpende hånd med at kontrollere, hvor den pågældende variabel kan være blevet ændret (hvis der af en eller anden grund skulle opstå fejl i programmet?) så kan synligheden af statiske variable på denne måde begrænses.

Static variabel erklæret inde i en funktion = scope i funktionen, static inde i et modul = scope i modulet.

Et modul kaldes i syntaksbeskrivelser en oversættelsesenhed (eng. translation unit) og består af en eller flere filer (include filer) som oversættes til et objekt-modul.

Variablen i eksemplet ovenfor, sidenummer, kan kun bruges inde i funktionen headerlinje(), men den husker alligevel nummeret fra gang til gang, når funktionen bliver kaldt.

Hvis en funktion som for eksempel strftime(3) gerne vil aflevere en text string med datoen formateret efter callers behov, så er det også nødvendigt at returnere i en buffer, som ligger UDEN for funktionen.

Man kunne somme tider ønske sig, at ordet static blev erstattet af "modul-specifik" eller "privat". Det kan man jo bare gøre selv ved hjælp af #define. I så fald skal man lige huske, at private er et reserveret ord i C++. Man kunne måske i stedet anvende "privat" (uden ’e’ til sidst!) eller, bedre, et ord som "hidden" eller ... "local"? SÅ ville der rigtig være rod i semantikken! Problemet er nævnt i forbindelse med revision af C som basis for C++, men som sagt finder private anvendelse i C++ , og det er på en lidt anden måde.

Husk at et C program består af eksterne objekter, som kan være enten funktioner eller variable.

A.2.5. Kontrolvariable for løkker eller loops

Det fremhæves somme tider, at C++ er særlig smart fordi man kan erklære variable der, hvor man har brug for dem. Det er sandelig også en god ting. Man har dog for nylig indført nogle forbedringer (læs rettelser) således at en kontrol-loop variabel, som erklæres i for eksempel en for-løkke, går ud af scope ved afslutningen af den blok, som styres af for løkken. Ellers kunne der nemt opstå noget rod.

Denne fejlkilde er elimineret i C ved at man kan erklære variable i enhver blok (hver gang man skriver en krøllet parentes start, brace start), og denne variabels scope går så indtil blokkens slutning. Nemt og logisk. Denne facilitet anvendes alt for lidt!

Eksempel A-10. En lokal kontrol variabel

```
/* loopcount.c, demonstration af ad-hoc variabel. */

#include <stdio.h>

char *thisprog;

int main(int argc, char *argv[])
{
    thisprog = argv[0];
    printf("Program %s er startet ... \n", thisprog);
    {
        int jj = 0;              /* jj bliver oprettet */
        while(jj++ < 10)
            printf("For %d. gang: Ih hvor vi kører\n", jj);
```

```
    }
                                /* nu er jj nedlagt "automatisk" */
    printf("Program %s exiter graciøst!\n", thisprog);
    return 0;
}
```

For at tillade oversætteren at optimere kan det forekomme, at interne blokkes auto-variable faktisk oprettes samtidig med at funktionens stack-frame sættes. Således gør gcc.

A.2.6. Flygtige ukontrollerede variable

En speciel variabel klassifikation er nødvendig for programmer, som deler memory med andre processer eller threads. Her anvendes nøgleordet *volatile*. En variabel, som er volatile, vil ikke blive optimeret væk, d.v.s. at når programmet skal benytte variabelens værdi, vil der blive foretaget en læsning, og en eventuel gammel værdi i et CPU register vil blive kasseret.

Volatile specifikationen anvendes også på maskiner, som ikke har et separat IO space men benytter en del af RAM som IO-porte.

A.2.7. Oversigt over variabel - anvendelse

Oversigt over anvendelsen af static:

- En variabel erklæret uden for funktioner er en global variabel. Det er egentlig det samme som at sige, at den er statisk, den bliver stående som den er, gennem hele programmets levetid, på linje med funktionsobjekter.
- En variabel erklæret udenfor funktioner får, hvis den er initialiseret, plads i "data" sektionen, og i "bss" sektionen, hvis den ikke er initialiseret. I så fald kan man regne med, at den er nulstillet.
- static for external object: specifikation af, at variablen *kun* skal være synlig i det oversættelses modul (translation unit) hvori den forekommer.
- static for variabel i en funktion (kaldes lokal static) betyder, at den kun er synlig i den funktion, hvori den er erklæret.

Her er en præcisering af, hvordan man kan definere forskellige slags lokale variable.

- Uden storage spec: En lokal variabel allokeres på funktionsstakken. Skal initialiseres hver gang funktionen køres.
- reserveret ord auto: overflødigt, samme som ovenfor.
- reserveret ord register: Oversætteren vil, hvis det er muligt, anbringe denne variabel i et af maskinens hurtige registre. Man kan ikke tage adressen på en registervariabel.
- reserveret ord extern: Variablen er defineret et andet sted, normalt i en anden oversættelses enhed.

- reserveret ord static: Variablen anbringes i regionen for statiske variable. Den er kun synlig i den blok, gerne en funktionsblok, hvori den er deklareret. (Den er selvfølgelig også synlig i den oversættelses enhed, hvori den er defineret!)
- En lokal variabel behøver ikke at blive defineret øverst i funktionens yderste blok. Man kan definere nye lokale variable, hver gang man skriver en start-brace (klamme).

A.3. GNU programmerings værktøjer

GNU står for Gnu is Not Unix. Det skyldes, at man ikke må bruge navnet Unix uden at betale for det, det var et registreret varemærke i 80'erne, da Richard M. Stallman (http://www.gnu.org) begyndte arbejdet med open source tools.

Heldigvis er Unix i vore dage ikke mere et varemærke, men en betegnelse for et operativ system, som overholder standarder. Se www.opengroup.org (http://www.opengroup.org)

De vigtigste værktøjer er: Compileren, gcc, assembleren as eller gas, linkeren ld, og object librarian ar. Med disse programmer kan man bygge en programfil.

Til at inspicere indholdet af programfiler og objektfiler, også kaldet dot-o filer, kan man benytte objdump. Det kan også disassemblere maskininstruktioner i objektmoduler.

Desuden er der den Gnu debuggeren, gdb. Der er adskillige klikbare interfaces til gnu-værktøjerne. Begynd med xxgdb og fortsæt med gide, Gnome Integrated Development Interface, og slut med de store, forkromede værktøjer sourcenavigator og kdevelop, alle sammen meget udmærkede, men tidrøvende. Skal jeg fremhæve nogen på andres bekostning må jeg indrømme, at indpakning og configure, make og make install af source navigator har imponeret mig. Det fungerede for mig på 2 (forskellige) RedHat platforme uden problemer.

Ud over klik-interfaces har man specielt meget glæde af **indent,** et program, som kan foretage formatering af en C kildetekst. Det er desværre ikke i stand til at håndtere alle syntakskonstruktioner på lige god måde, men har man en større mængde filer, som skal gennemarbejdes, er indent meget nyttigt. Kan man ikke lide de tabulatortegn, der er i en fil, kan man fjerne dem (anbefales!) med **expand** .

Indent og expand og mange andre programmer kan med fordel køres som filtre på den tekst - eller dele af den tekst - som man har i editoren. Dette tillader vi og vist nok også Gnu Emacs. Se i øvrigt Friheden til at vælge applikationer. (http://www.linuxbog.dk/)

Nævnes skal også **gvim** , den forbedrede editor, vi-improved, som med sit system-interface, hjælpesystem, syntaksfremhævning, muse-interface og ortogonale kommandostruktur ender med at være mange programmørers favoriteditor. Der er dog flere, som benytter Gnu Emacs, Richard Stallmans bud

på, hvordan en editor skal konstrueres. Den har features, som gvim ikke har, først og fremmest sammenligning og opdatering af to næsten ens filer. Man kan nemt finde kommandoerne til Emacs i menusystemet, og farver m.v. kan sættes (ligesom for andre programmer) i $HOME/.Xdefaults. Jeg nævner det her, fordi den RedHat valgte farve kombination ser frygtelig dyster ud. Begge editorer har en tutorial, som guider begynderen igennem de vigtigste operationer.

Dette afsnit er lige nu meget kort og ikke færdigt! De vigtigste options til GNU-C compileren er:

A.3.1. GNU C Compiler options

```
-c              compile kun, skriver en <navn>.o fil (dot-o fil, .o)
-d              output list af define, forudsætter -E
-g              inkluder debug information i resultat fil
-pedantic       check code after ANSI C code check spec-s.
-save-temps     save intermediate files, ufn.i ufn.s (hvorfor ikke ufn.o?)
-o xxx          send output til fil ved navn xxx. Hvis xxx ender
                   på .o, vil compileren *kun* gå til det step,
                   der generer .o filer.

-s              strip efter linkage
-v              verbose, vis kommandoer fra gcc "administrator" program.

-I xxxx         path til ekstra include directory, vil blive søgt først.
-I-             Path(s) specificeret før denne option søges kun
                for "xxx" files i double quotes. Der er meget mere om
                -I option(s) in cpp.1 and cpp.doc

-E              send preproces output til ?
-MM             skriv MAKE regler for hvordan program er opbygget, bevirker,
                   at også -E switchen bliver sat.
-N? stack_check?

-O              Optimer. Skriver bedre kode, men pas på, kan ændre sekvens.
-P              Præproces uden linjer m/# info
-C              Lad kommentarer komme med
-trigraphs      Process ANSI standard trigraph sequences

-D"str"         Definér macro eller hvad du nu vil definere.
-u"str"         AF-definer en macro! (d.v.s. annuller virkningen
                   af en #define)
```

A.3.2. objdump eksempler

Her kommer eksempler på objdump anvendelse.

Slutbemærkning:

1. I parentes bemærket, hvis du vil gøre krypteringen ovenfor rigtig ubrydelig skal du forbedre den lidt. Anvend længere keystring (altid længere end meddelelsen), formatér output i linjer, som er lige lange. Bemærk, at hvis input består af lutter samme tegn, så kan denne version ikke rigtigt skjule sin keystring. Ikke at den kommer i klartekst, men det er muligt at regne baglæns og finde keystringen. Man skal behandle sekvenser af samme tegn på en speciel måde, hvor man angiver antal og tegn. Algoritmen er en tilpasning af Kejser Augustus' og Livia's kode, som beskrevet i Robert Graves' "I, Claudius".

   Endelig er der ikke support for såkaldte offentlige keys, der kan bruges til afkodning men ikke indkodning.

2. lint og den systemuafhængige C-oversætter har som oprindelig ophavsmand Stephen C. Johnson, som er en af de kendte oversætter-teoretikere der arbejder på Bell Labs.

Appendiks B. Revisionshistorie for bogen

- Version 2.0.20041104 - 4. november 2004: Jacob Sparre Andersen: Retter sprog, lidt opmærkning og et par programmeringsfejl (med god hjælp fra Peter Makholm). Donald Axel retter trykfejl.
- Version 2.0 - 25. januar 2004: Jacob Sparre Andersen: Retter sproglige fejl. Donald Axel får endelig skrevet noget om tilstandsmaskiner.
- Version 1.9 - 6. april 2003: Donald Axel retter to mindre fejl.
- Version 1.8 - 1. september 2002: Jacob Sparre Andersen - Harmoniseret brugen af **tar**. Rettet logiske fejl i SGML-koden.
- Version 1.7 - 14. juni 2002: Steinar Børmer retter en forkert formel.
- Version 1.6 - 10. marts 2002: Ny licens for bogen - Åben dokumentlicens. Forklaring af C99 restricted pointer. Lidt forklaring af recursive descent parser, med reference til Stroustrup[1997] p.108. Rettelse af kodeeksempel dollar2kr.c, tak til rasmus(på)glud.dk.
- Version 1.5 - 29. december 2001: Jacob Sparre Andersen: Sproglige smårettelser.
- Version 1.4 - 21. oktober 2001: Jacob Sparre Andersen: Rettet enheder til (MHz = megahertz, Mbit = megabit, Mb = megabyte).
- Version 1.3 - 11. august 2001: Jacob Sparre Andersen: Sproglige rettelser. Donald Axel: Stavefejl og småfejl rettet.
- Version 1.2 - 9. juli 2001: Gennemskrivning af kapitel 2. Struct gennemgang udvidet. Kortspil eksempler er opdateret. C++ version af kortspilsbunke (kræver gcc 3.0). Eksempel kort04.c, blanding af kort, tilføjet. Afsnit om sammensatte datatyper er forbedret. Ændringer overskrifter og indledninger for at få emner og pointer tydeligere understreget. Beskrivelse af den trinvise udvikling af sgml-formateringsprogrammet samt små illustrationer tegnet med xpaint. Eksempler bogpris1.c og bogpris2.cxx til afsnit om konkrete datatyper. Forbedrede forklaringer og omtale af uC. Ole Tange fangede en stribe format-fejl.
- Version 1.1 - 24. maj 2001: Google har scannet Friheden-bøgerne igen! derfor var det vigtigste i denne omgang at udfylde alle tomme huller, om ikke andet, så med en beskrivelse af, hvordan man kan gribe emnerne an. Det vil sige, at der er kommet indhold til afsnittet om abstrakte datatyper, eksempel og tegning til linket liste mv. Men det er en nødløsning. Linked liste illustreres bedst med eksempler fra linux-kernen, fra et adventure-spil, eller måske med repræsentation af et kortspil, sådan som Donald Knuth gør det i "The Art of Computer Programming" p. 228ff.
- Version 1.0 - 15. april 2001: uCc oversætteren er bearbejdet, således at den fungerer sammen med det nyeste glibc(2.2). Derfor nye switches, --bx og --save-all og --parm-ext. Har skrevet et instruktivt eksempel på en simpel "filter" parser: sgmlfmt_pre1.c, som konverterer af "<" til "<", og tagbal02.c, som tjekker om SGML-mærker er i balance. Simple instruktive versioner. tagbal02 introducerer rekursion. Afsnittet om abstrakte datatyper er ved at komme på plads.
- Version 0.9 - 12. marts 2001: Donald Axel: De mest trængende ændringer af demo C-oversætteren uC er foretaget. Nye eksempler på filtre og parsere; Appendix A udvidet en del. Flere eksempler. Synkronisering mellem de forskellige dele, men det vigtigste har i første omgang været at dække grundlæggende teknikker med eksempler og forklaringer. Programgenerator Glade nævnes. Crash course udbygget med et afsnit, som går i dybden med storage specification, (lager-specifikation?) af

datatyper. Henrik Christian Grove fandt en dum trykfejl i oversigt over bøger. Jacob Sparre Andersen retter diverse sproglige fejl.

- Version 0.3 - 4. februar 2001: Flere filter programmer. Fatale fejl i prog.exempler rettet. Tilføjet nyt kapitel og indledning. Jacob Sparre Andersen: Rettet diverse sproglige fejl.
- Version 0.2 - 29. december 2000: Første offentlige version.

Appendiks C. Nyt i C99

C.1. Uddrag af nyheder i ANSI revisionen fra 1999

Den følgende liste over nyheder i ANSI C (http://www.open-std.org/jtc1/sc22/wg14) omfatter både ændringer i selve C-grammatikken (syntaksregler, regler for typekonvertering, nye datatyper), af præprocessing, af krav til oversætter og krav til biblioteker (lib-filer). Desværre er jeg i skrivende stund ikke i stand til at kommentere alle punkter.

Hvad er for eksempel de nøjere regler for Variable Length Arrays, VLAs? VLA er noget, som skulle være nyttigt i forbindelse med de massive beregningsopgaver, som er en vigtig computeranvendelse i dag.

Der er dog mange ting, som kan læses ud af n897.pdf filen omtalt nedenfor. "Pointere med begrænsninger" er en pointer der peger på et objekt (som pointere jo ofte gør). Men alle referencer til dette objekt *skal* nu gå igennem denne pointer. Derved kan man opnå nogle optimeringer, som ellers ikke er mulige. Oversætteren kan stole på, at denne pointer ikke bliver kopieret eller "aliaset" til en eller anden variabel.

Man kan købe selve ANSI C 1999 standarden i Adobe-PDF format fra ANSI Standard Store (http://www.ansi.org). Næsten lige så interessant (måske endda mere interessant) er tankerne bag standarden. Dem kan man finde i dokumentet n897.pdf på http://www.open-std.org/jtc1/sc22/wg14"; teksten på linket er N897, draft rationale available (http://www.open-std.org/jtc1/sc22/wg14).

Charteret for revisionsgruppen ligger også at hente på ovenstående adresse. Det har (selvfølgelig) været en vigtig del af charteret, at C sproget skal være så kompatibelt med C++ som muligt.

Der er sikkert nogle, som sammen med mig vil begræde, at man ikke blot kan nævne en funktion uden at skrive en prototype for at informere compileren om return type og parametertyper. Men det er nu ikke så slemt endda; der vil i mange år endnu være en switch som tillader ANSI-89 reglerne.

De vigtigste ændringer er:

- Funktioner *skal* erklæres med prototype, inden de anvendes. En ukendt funktion antages ikke at have int som return type.
- Implicit integer er ligeledes afskaffet. Tidligere var det tilladt at skrive yxi = 1 øverst i en blok, hvis man ville erklære og initialisere en integer. Det må man ikke mere. En variabel skal erklæres med angivelse af type. Sammenhold dette med, at det nu skal være tilladt at erklære en variabel der, hvor man har brug for den, ligesom i C++.

- Inline funktioner skal supporteres af oversætteren (det kan GNU-C allerede).
- Så vidt jeg kan forstå vil der også være begrænset support af typetjek i præprocessor aritmetik. Det skal også kunne specificeres, om præprocessor aritmetik er signed eller unsigned.
- Kommentarer som i C++ og simula: // dette er en kommentar.

I bedste open doc stil efterlyser jeg bidrag til uddybelse af disse punkter.

Listen er hentet via http://www.ansi.org. Også her http://www.dkuug.dk/JTC1/SC22/WG14 kan man finde oplysninger. Men links har det med at blive gamle. Brug eventuelt www.google.com til at finde nye links til ANSI-C standard.

C.2. Liste i tilfældig rækkefølge

Ok, jeg har tilladt mig at gruppere emnerne på en lidt anden måde end det originale paper. (http://www.open-std.org/jtc1/sc22/wg14/www/newinc9x.htm)

- pointere med begrænsninger
- variabel længde arrays
- fleksible array medlemmer
- udvidede heltals typer og <inttypes.h> og nu <stdint.h>
- afskaffelse af regel om implicitte funktions erklæringer.
- afskaffelse af implicit integer reglen
- long long type og library funktioner for denne
- forøgelse af grænser for oversættelse
- længere variabelnavne
- pålidelig integer division
- universelle navne til det anvendte tegnsæt
- binære floating point literals og printf/scanf specifikation af konvertering
- sammensatte stringkonstanter
- designated initializers
- // slash-slash kommentarer
- blanding af erklæringer og kode (erklæring af variable, hvor man har brug for dem)
- heltal (integer) konstant type - regler
- ændringer af type konvertering (integer promotion)
- præprocessor aritmetik kan gøres i signed/unsigned integers.
- support af complex og imaginære tal (header fil <complex.h>)

- type generiske matematik macroer <tgmath.h>
- vararg makro'er
- tilføjelse af flere matematik library funktioner (<math.h>)
- decimalbrøk miljø tilgang (for overflow? <fenv.h>)
- IEC 559 (IEEE aritmetik) support
- udvidet time structure (struct tmx) og library funktioner
- efterstillet komma tilladt i enum erklæringer
- %lf konverterings specifikation er tilladt i printf
- inline funktioner
- boolean type (<stdbool.h>)
- idempotent type qualifiers
- tomme makro arguments
- ny struct type kompatibilitets regler (tag compatibility)
- _Pragma præprocessor operator
- standard pragma'er
- __func__ predefineret nøgle-ord
- vscanf funktionsfamilien
- snprintf funktionsfamilien
- VA_COPY makro
- tilføjelse af flere strftime konverterings specifikationer
- LIA compatibility annex
- depreciering (afskaffelse på længere sigt) af ungetc i starten af binær fil
- remove deprecation of aliased array parameters annullering af depreciering af alias-ede array parametre

C.3. Nyt i C93 (AM1)

- begrænset support af ISO-0646 <iso646.h> (bemærk at der arbejdes på højtryk på unicode support i Gnome projektet, oversætters anmærkning).
- wide character d.v.s. 16-bit character support og library (<wchar.h> and <wctype.h>)

Stikordsregister

Symboler

ÅDL, viii

A

abstrakte datatyper, 77
ANSI funktioner, 14

B

brugerdefinerede datatyper, 61
bss, 118

C

C99 standard, 99, 129
cdecl, eksempel pÃ¥ anvendelse, 81
copyright, viii
Cygwin, 3

D

deklaration, 115

E

enumeration
  nummereringsliste, 66
erklÃ¦ring, 115
extern (nÃ¸gleord), 115, 120

F

FILE typen, 77
filter, 31

H

heap, 116
hello-programmer, 14
heltal, 6

I

implicit integer regel, 129

K

Kernighan, Brian, 14
keywords, 99
kommentarer, 15
konkrete datatyper, 76
konstanter, 118

L

lagring, specifikation af, 116
lclint, 120
lint, 120

M

mapning, 10
memory
  layout, 117
minimal-programmer, 14

N

nÃ¸gleord, 99

O

objekt, eksternt, 13
ophavsret, viii

P

pointer
  oplysninger om, 67
  skalering, 67
privat, 121
prototype
  funktion, 17

R

read-only memory, 118
reserverede ord, 15
retur vÃ¦rdi, 14
return, 100
Revisionshistorie, 127
Ritchie, Dennis, 14

S

scope, 119
skrivebeskyttet lager, 118
stak, 116
static, 116
static, anvendelse af, 120
static, betydning privat, 121
Storage specifications, 116
synsvidde, synlighed, 119
sÃ¦tning, statement, 15

U

udtryk, expressions, 15
union, 104