Programmeringsspråket C

Sverre Hendseth

January 16, 2015

Contents

1 Introduksjon

Her kommer pensum i C-delen — programmeringsdelen — av faget tilpassede datasystemer. Se gjerne på dette stoffet som en tutorial —- noe du bør sette deg ned og eksperimentere med.

På linux er bør det være trivielt å få skallet og c-kompilatoren som brukes i gjennomgangen her på plass (her brukes **gcc**, men **clang** er et alternativ). For windows har vi Visual Studio, og jeg kan godt anbefale "cygwin" som et unix-lignende utviklingsmiljø. Kan noen mac-brukere peke meg i retning av den gode måten på mac? (clang er der allerede, og kan kalles fra "the Terminal"?)

Jeg setter opp "vedlikeholdbarhet" som det kriteriet vi måler programvare kvalitet ut ifra. Hvis programvare er vedlikeholdbart, kan vi lett rette opp feil og legge til nye features og slik programvare er av god kvalitet. Teknikker, språk, biblioteker, moduler osv. er gode hvis de lar oss skrive vedlikeholdbar kode.

Metode nummer en for å oppnå vedlikeholdbarhet i et stort programsystem er å dele det i deler, enten dette er grupper av linjer, funksjoner, objekter, filer, moduler eller enda mer abstrakte ting.

For at dette skal ha noe for seg tilstreber vi at:

- En modul må kunne vedlikeholdes uten å kjenne resten av systemet.

- En modul må kunne brukes uten å kjenne indre virkemåte.

Kyb Intro-delen om sanntidssystemer kan settes i dette perspektivet. Hvordan kommer en tilstandsmaskin ut etter disse to kriteriene? Hvordan kommer delt variabel-synkronisering imellom tråder ut? (Svar: Bra og Dårlig :-)

2 Hello World

Formålet med å gå så grundig gjennom dette er *ikke* at du trenger denne kunnskapen for å kompilere C-programmer på en PC. Vi er ute etter den grunnforståelsen som skal til for å kunne forstå hva som foregår, finne feil på en effektiv måte, og for å kunne forholde seg til utvikling mot mikrokontrollere.

```
#include <stdio.h>

void main(void){
  printf("Hello World\n");
}
```

Vi introduserer kommandolinjen som en måte å jobbe på... Titt på Sverres bash tutorial for eksempel.

Vi kan kompilere - oversette fra kildekoden som du ser til et kjørbart program - dette programmet ved å kalle gcc:

```
sverre@eee:~/tilpdat$ ls
hello.c
sverre@eee:~/tilpdat$ gcc hello.c
sverre@eee:~/tilpdat$ ls
a.out  hello.c
sverre@eee:~/tilpdat$ ./a.out
Hello World
```

gcc er en paraply-kommando som titter på de filene og opsjonene vi gir den og så kaller vidre til forskjellige andre kommandoer. For å se nøyaktig hva som skjer kaller vi gcc med "-v" opsjonen.

```
sverre@eee:~/tilpdat$ gcc -v hello.c
```

Utskriften herifra er stor og stygg, så la oss ta litt om gangen:

2.1 Trinn 1 og 2 - Preprosessering og kompilering.

Etter å ha skrevet ut *i detalj* hvilken versjon av kompilatoren vi kjører kommer vi til følgende kommando (Jeg har ryddet litt og delt linjene):

```
/usr/lib/gcc/i486-linux-gnu/4.7/cc1 -quiet -v -imultiarch
  i386-linux-gnu hello.c -quiet -dumpbase hello.c
  -mtune=generic -march=i586 -auxbase hello -version
  -o /tmp/ccLo9Dgq.s
```

(Den tilsvarende linjen vil nok se litt forskjellig ut for deg.)

For denne og de neste kommandoene; slå opp i dokumentasjonen på kompilatoren ("man gcc" eller google) for å finne ut hva alle flaggene gjør. Grovt sett kalles kompilatoren som heter cc1 på filen hello.c og det genereres en fil i temp-folderen min som heter ccLo9Dgq.s.

2.1.1 Preprosessoren

Det er egentlig to trinn som gjøres her; Først kjøres hello.c gjennom **C preprosessoren**. Dette er egentlig et eget program som gjenkjenner de linjene i programmet vårt som starter med "#" - i vårt tilfelle "#include <stdio.h>"-linjen.

Du kan se resultatet av preprosessoren ved å gi gcc "-E" opsjonen eller ved å kalle preprosessoren **cpp** direkte.

```
sverre@eee:~/tilpdat$ gcc -E hello.c
...
sverre@eee:~/tilpdat$ cpp hello.c
...
```

Vi ser side etter side med innholdet av stdio.h - og alle andre filer som blir inkludert av den igjen. Vi legger også merke til at alle linjer som starter på # er borte og at "makroer" definert av #define er byttet ut med det de skal være. Det er et unntak her; vi har fortsatt "# 2 "hello.c" ..." linjer igjen. Dette er informasjon som kompilatoren trenger for å kunne peke på riktig fil og linje f.eks. når den skal rapportere en feil.

2.1.2 Kompilatoren

Trinn to er selve kompileringen; Med unntak av linjenummerinfoen så er det vi nå ser ren C-kode som kompilatoren oversetter til assembly-kode. Titt på assemblykoden ved å gi "-S" opsjonen til gcc. Dette vil genere hello.s i folderen vi står i istedet for denne temp-filen:

```
sverre@eee:~/tilpdat$ gcc -S hello.c
```

En rimelig strategi hvis du må programmere assembler en gang er å skrive et skjelett av programmet ditt i C, generere assembly og så optimalisere eller legge inn de instruksjonene du trenger manuelt.

Det er bare for ytelsen at preprosessoren og kompilatoren er bygd sammen.

2.2 Trinn 3 - Assembleren og objektfiler

Etter å ha fått utskriften fra cc1 finner vi kommandoen:

```
as -v --32 -o /tmp/ccgWZHys.o /tmp/ccLo9Dgq.s
```

as er assembleren og vi ser at assemblyfilen fra i sted blir transformert til en **objekt-fil** ccgWZHys.o. Dette er en binær fil og den inneholder en ganske rett frem speiling av assembly-koden. Dette er "nesten kjørbar" maskinkode - de bytene som representerer op-kodene mikroprosessorene kan utføre er på plass. Imidlertid er ikke faktiske adresser i minnet endelig bestemt enda, og en objektfil representerer bare en del av et program; objektfiler må **lenkes** sammen med andre objektfiler for å danne et kjørbart program.

En objektfil inneholder også slikt som tabeller over eksporterte "symboler" (navn på funksjoner og variable som kan aksesseres fra andre deler av programmet) og symboler som den er avhengig av, og som andre deler av programmet må stille med.

Gi gcc "-c"-opsjonen for å generere objektfilen. Dette vil generere hello.o.

```
sverre@eee:~/tilpdat$ gcc -c hello.c
```

Det å skille imellom å kompilerings- og lenke-prosessene - å først generere opbjektfiler og så lenke dem sammen etterpå - er en veldig vanlig måte å spare kompileringstid på - alle moduler kompileres til objektfiler og når en modul endrer seg trenger en bare å rekompilere den endrede modulen før en gjentar lenkingen.

2.2.1 Å bruke nm for å se innholdet av en objektfil

Kommandoen **nm** lar oss se hva som er inne i en objektfil (jeg hører **objdump** er alternativ):

```
sverre@eee:~/tilpdat$ nm hello.o
00000000 T main
         U puts
```

Fra rapporten til nm ser vi at "main" er et symbol som ligger fra adresse 00000000 i Tekst-segmentet i denne objektfilen. (Lenkeren vil kunne flytte rundt på disse tingene når den slår sammen de forskjellige segment-typene under lenkingen).

Vi ser også et Undfined symbol "puts", som må finnes i en annen objektfil hvis vi skal lage et komplett kjørbart program.

2.2.2 Minne-segmenter

Disse segmentene forteller hvilken type minne de forskjellige tingene skal ligge i; programkode skal som regel være skrivebeskyttet og i et embedded system ligge i minne som ikke mister dataene sine selv om strømmen går av. Likedan med konstanter, og verdier som brukes til å initialisere variabler. Globale variable som skal initialiseres fra start av programmet samles opp i et segment, og de tilsvarende verdiene legges gjerne ut likt slik at all initialisering kan gjøres i en kopieringsoperasjon. I et større system - som PC'en din - kan forskjellige programmer til og med dele read-only segmentene av de forskjellige bibliotekene slik at vi sparer minne og lastetid.

Legg til disse linjene som definerer globale variable til hello-programmet vårt:

```
#include <stdio.h>

int i;
int j = 0;
const int k = 0;
const int l;

void main(void){
  printf("Hello World\n");
}
```

... og sjekk hva som er i objektfilen nå:

```
sverre@kybpc435 ~/tilpdat $ gcc -c hello.c
sverre@kybpc435 ~/tilpdat $ nm hello.o
0000000000000004 C i
0000000000000000 B j
0000000000000000 R k
0000000000000004 C l
0000000000000000 T main
```

```
                 U puts
```

Sjekk hvordan i, j og k ender i forskjellige segmenter. (Igjen: Dette oppfører seg ikke nødvendigvis likt på andre systemer.)

Spørsmål: Hvorfor legges l i samme segment som i?

Svar: Vi har definert en konstant l, men uten å initialisere den til noe. Dette gir ikke mening, og kompilatoren tør/kan ikke behandle den som en konstant og "gjør det beste ut av det".

Spørsmål: Hvorfor er det puts og ikke printf som er det udefinerte symbolet?

Svar: Kompilatoren driver her med ting som ikke er dens business i utgangspunktet: printf er en så vanlig funksjon i C at kompilatoren har tatt på seg å vite noe om den, og når den ser av vi bare skriver ut en konstant streng, så trenger vi ikke printf-funksjonaliteten - vi kan greie oss med puts. Det hele er en optimalisering. Sjekk skall-kommandoen "man puts". Dette skal gi dokumentasjonen - manualsidene - til puts-funksjonen.

```
sverre@eee:~/tilpdat$ man puts
```

Endre printf-linjen til

```
printf("Hello World %d\n",1);
```

. . . og kompiler igjen

```
sverre@kybpc435 ~/tilpdat $ gcc -c hello.c
sverre@kybpc435 ~/tilpdat $ nm hello.o
0000000000000004 C i
0000000000000000 B j
0000000000000000 R k
0000000000000004 C l
0000000000000000 T main
                 U printf
```

Nå er ikke denne optimaliseringen mulig lengre og vi legger merke til at nå er det printf som blir det udefinerte symbolet.

En oppgave for den interesserte: Prøv å legge til et par static-variable i main-funksjonen

```
void main(void){
  static int m;
```

```
  static int n = 0;
  printf("Hello World % d\n",12);
}
```

Hvordan legges m og n ut i objektfilen?

2.2.3 Name-mangling og sammenlenking av C og C++

Ofte (om enn ikke her) ser vi også at det har skjer en oversetting av symbolnavnene: "printf" kan fort bli til "_printf" for eksempel. Dette er en konvensjon som kalles "name mangling" og kan være forskjellig fra kompilator til kompilator. Spesielt må en ta hensyn til dette når en skal lenke sammen C og C++ kode som har forskjellige konvensjoner for mangling.

Jeg kopierer hello.c til hello.cc for å oversette det samme programmet med c++-kompilatoren:

```
sverre@eee:~/tilpdat$ gcc -c hello.cc
hello.cc:6:11: error: uninitialized const 'l' [-fpermissive]
```

Ok - c++ aksepterer ikke uinitialiserte konstanter, det er jo rimelig... Jeg fjerner l-linjen og prøver igjen.

```
sverre@eee:~/tilpdat$ gcc -c hello.cc
sverre@eee:~/tilpdat$ nm hello.o
00000014 r _ZL1k
00000000 B i
00000004 B j
00000000 T main
         U printf
```

Hva skjedde med navnet på k'en vår?

En modul som nå var kompilert med C-kompilatoren ville ha vanskeligheter med å bruke k'en definert i c++ modulen.

(Det blir for drøyt å gå inn på hvorfor hvorfor i og j ikke ble manglet, og hvorfor i ble flyttet fra "Common" segmentet til "B-uinitialiserte data"-segmentet. Google "C linker common symbols", eller les mer om de forskjellige segmentene hvis du er nyskjerrig)

En extern "C" deklarasjon kan tvinge C++ til a mangle navn på C-måten. Da kan vi selvfølgelig ikke bruke objekter, funksjoner i klasser, overloadede funksjoner osv. men det lar oss lett sette opp koden vår slik av vi kan lenke C med C++.

Sjekk følgende program (hello2.cc):

```
#include <stdio.h>

int f1(int a){
  return a;
}

extern "C" {
  int f2(int a){
    return a;
  }
}

void main(void){
  printf("Hello World % d\n",12);
}
```

... og hvordan f1 og f2 blir manglet forskjellig.

```
sverre@kybpc435 ~/tilpdat $ gcc -c hello2.cc
sverre@kybpc435 ~/tilpdat $ nm hello2.o
000000000000000c T f2
0000000000000018 T main
                 U printf
0000000000000000 T _Z2f1i
```

extern "C" - deklarasjoner fungerer begge veier: Vi kan også kalle C-funksjoner fra C++ på denne måten

```
extern "C" int f1(int a, int b);
extern "C" {
#include <comedi.h> // Inkluder en C headerfil i et C++ program
}
```

For mange av C's header filer er dette allerede tatt hånd om i filene selv slik at de kan inkluderes både fra C++ og C uten å styre med dette.

2.3 Trinn 4 - Lenking

Neste kommando som blir kalt av gcc når vi kompilerer det opprinnelige hello world-programmet vårt er:

```
/usr/lib/gcc/x86_64-linux-gnu/4.8/collect2 --sysroot=/
  --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu
  --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2
  -z relro
  /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu/crt1.o
  /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu/crti.o
  /usr/lib/gcc/x86_64-linux-gnu/4.8/crtbegin.o
  -L/usr/lib/gcc/x86_64-linux-gnu/4.8
  -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu
  -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../lib
  -L/lib/x86_64-linux-gnu -L/lib/../lib
  -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib
  -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../..
  /tmp/ccgWZHys.o
  -lgcc
  --as-needed -lgcc_s --no-as-needed -lc -lgcc
  --as-needed -lgcc_s --no-as-needed
  /usr/lib/gcc/x86_64-linux-gnu/4.8/crtend.o
  /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu/crtn.o
```

Dette er lenkingen - her lages den eksekverbare filen, som av historiske årsaker blir hetende a.out siden vi ikke anga hva den skulle hete.

Lenkeren tar altså grovt sett et antall objektfiler inn, kombinerer alle segmentene av de forskjellige typene, flytter rundt på alle elementene innenfor hvert segment og tilpasser alle referansene til dem, og genererer et utlegg i minnet for både program og de forskjellige typene data.

I vårt tilfelle, når vi lager en eksekverbar fil for et operativsystem som Linux, kjøres nok ikke denne prosessen helt ferdig; Den kjørbare filen inneholder nok de siste instruksjonene til operativsystemet om hvordan programmet skal lastes og legges i minnet når det startes.

Noen forklaringer:

- Vi kjenner igjen objektfilen vår /tmp/ccgWZHys.o
- Vi ser det blir lenket noen andre objektfiler også: “crtxxx.o”. Dette er “C Runtime System”, som bl.a. setter opp stack, minne, initialiserer de ikke-konstante globale variablene våre, setter opp parametrene som ble gitt på kommandolinjen når vi startet programmet, kaller vår main-funksjon og sørger for at operativsystemet får tilgang på den verdien som vi returnerer fra main...

- -lxxx vil "lenke biblioteket" som heter "libxxx.a". Et bibliotek er en samling av objektfiler, hvor bare de som trenges vil bli tatt med.
- -L - opsjonen angir en folder hvor lenkeren skal lete etter bibliotekene.

Du kan bruke nm på den kjørbare filen også; Det er kommet til en god del symboler (funksjoner og variable) fra kjøretidssystemet og for å støtte opp printf-kallet vårt.

Men la oss finne ut hvor printf kom fra: Vi leter altså etter det biblioteket som definerer printf.

```
sverre@kybpc435 ~/tilpdat $ nm -o /usr/lib/*/lib*.a | grep " T printf"
...
/usr/lib/x86_64-linux-gnu/libc.a:printf.o:0000000000000000 T printf
...
```

Vi ser at printf blir definert i objektfilen printf.o som er en del av libc.a.

Vær oppmerksom på at rekkefølgen av biblioteker og objektfiler kan være vesentlig - hvis lenkeren ikke finner et symbol som du vet at du lenker med, så kan det være at det ikke ble lenket "sent nok" - bytt om på rekkefølgen.

Også: Når du får feilmeldinger fra gcc så vær bevisst på om meldingen kommer fra lenkeren eller kompilatoren (...eller preprosessoren - jeg tror aldri jeg har fått feil fra assembleren noen gang.) Det er liten vits i å lete etter det manglende semikolonet (som ville gitt opphav til en feil fra kompilatoren) hvis det er lenkeren som klager.

2.4 Trinn 5 - Lasting

Som nevnt kan det skje ting i det du ber om å få kjørt et program også. I det enkleste tilfellet kan vi tenke oss at den eksekverbare filen blir kopiert rett inn på et fast sted i RAM'en og utførelsen starter på et fast bestemt sted der inne. Imidlertid er det ikke usannsynlig at den eksekverbare filen heller er en oppskrift på hvordan minnet skal settes opp og initialiseres før kjøringen starter. Det kan brukes mikroprosessor eller maskinvare mekanismer for å la oss adressere internt i segmentene uavhengig av hvor i fysisk minne segmentene er. Det kan også skje en transformasjon av maskinkoden ved lasting - for å endre alle adressene til å stemme med de delene av fysisk minne programmet ble lastet i.

Dynamisk lenking er også et begrep: Her skjer lenkingen i det programmet lastes (eller enda senere - ved behov) noe som resulterer i mindre eksekverbare programmer. En fordel ved dynamisk lenking er også at flere

programmer som bruker samme bibliotek samtidig kan forholde seg til de samme Text og Read-Only segmentene i RAM.

2.5 Biblioteker & Make

Et bibliotek er bare en samling - et arkiv - av objektfiler, muligens med en egen index-tabell for å optimalisere lenkeprosessen. Vi setter sammen objektfiler til biblioteker med kommandoen "ar". Navnekonvensjonen for biblioteker er "libxxx.a" hvor vi kan lenke med gcc opsjonen "-lxxx". La oss lage et bibliotek av filene f1 og f2:

```
sverre@kybpc435 ~/tilpdat $ cat f1.c

int
f1(int a, int b){
  return a+b;
}

sverre@kybpc435 ~/tilpdat $ cat f2.c

int
f2(int a, int b){
  return a-b;
}

sverre@kybpc435 ~/tilpdat $ cat hello3.c

#include <stdio.h>

void main(void){
  printf("Summen av 1 og 2 er %d\n",f1(1.0,2.0));
}
```

Nå begynner skrivearbeidet å bli for stort for meg, så la oss bruke følgende make-fil for å automatisere kompilerongsprosessen:

```
sverre@kybpc435 ~/tilpdat $ cat Makefile

# en make "variabel"
OBJS=f1.o f2.o
```

```
# En regel som lærer make å lage objektfiler fra C-filer:
# '%' er wildcard, $< er en stadard-variabel som er input-filen.
%.o:%.c
        gcc -c $<

# En regel for å lenke sammen
# Denne er den forste konkrete regelen og er dermed default mål
# hvis du skriver bare "make"
# $@ er en standard-variabel som er output-filen i regelen
hello3: hello3.o libsverre.a
        gcc hello3.o -L. -lsverre -o $@

# Hvordan vi lager biblioteket
libsverre.a: $(OBJS)
        ar r $@ $(OBJS)
```

Lær deg make godt nok til at du kan forstå og lage make-filer på dette nivået! (Vær oppmerksom på at de blanke feltene foran kommandoene *må* være en TAB. Det *kan ikke* være mellomrom! og du får ikke noen god feilmelding om du gjør feil her!)

Rent bortsett fra å huske hvilke kommandoer vi må bruke for å kompilere, så er de sentrale egenskapene til make å finne ut i hvilken rekkefølge disse kommandoene må utføres i, og i hvilken grad en kommando faktisk trenges å utføres eller ikke. Uansett:

```
sverre@kybpc435 ~/tilpdat $ make
gcc -c hello3.c
gcc -c f1.c
gcc -c f2.c
ar r libsverre.a f1.o f2.o
ar: creating libsverre.a
gcc hello3.o -L. -lsverre -o hello3
sverre@kybpc435 ~/tilpdat $ ./hello3
Summen av 1 og 2 er -1852629543
```

Storslagent - kompileringen gikk jo godt - og bare med 4-5 tastetrykk. Når vi lenker med biblioteker vil bare de objektfilene som er nødvendige bli lenket med. Kjør nm for deg selv for å bekrefte at f2 ikke er en del av hello3.

Men verre er det at programmet ikke virker. Og det uten en antydning til problemer ved kompileringen... Skru på warnings "-Wall" og/eller kjør lint(/splint) for å få hint om hva som er feil.

Problemet er at lenkeren vil lenke et kall av funksjonen f1 mot en definisjon av funksjonen f1 uavhengig av hvilke parametre den tar. På kall-stedet kalles imidlertid en funksjon som tar to float-parametre så det vi ser som svar er integer-summen av de bitmønstrene som representerer flyttallene 1.0 og 2.0

Den gode måten å sikre seg mot slikt vil være å la en header-fil deklarere de to funksjonene i biblioteket og så la hello3.c inkludere denne.

```
#ifndef SVERRE_H
#define SVERRE_H

int f1(int a, int b);
int f2(int a, int b);

#endif
```

Hvis vi gjør dette og kompilerer igjen får vi riktig svar - nå vet kompilatoren at f1 tar to int'er og vil konvertere float'ene til int'er før kallet.

Legg merke til denne "#ifndef" dekorasjonen i headerfilen: Dette er en måte å hindre at headerfiler blir lest mer enn en gang (noe som ofte funker, men er unødvendig og kan gi opphav til rare feil og warnings om deklarasjonene ikke er konsistente), det kan også løse problemer med sykliske avhengigheter imellom headerfiler.

3 C - Språket

3.1 Versjoner av C

Det er flere versjoner av C-språket og jeg nevner de forskjellene her som har størst sannsynlighet for å være forvirrende hvis du skulle bruke en eldre kompilator eller se kode som er skrevet for en eldre versjon.

K&R C (boken kom i 1978)	Her defineres typen av funksjonsparametrene ikke i parameterlisten, men imellom denne og funksjonskroppen
Ansi C (1989)	Deklarasjoner må fortsatt være øverst i blokker av statements. "//" er ikke kommentartegn. Vi har ikke const-variable.
C99	
C11	Den dere bruker nå, temmelig lik C99.

3.2 Litt om C++

Du kan skrive C i C++. Grunnene til ikke å bruke C++-kompilatoren hvis du har den tilgjengelig er få såvidt jeg kan se.

gcc er smart nok til å bruke c++ kompilatoren når filen har .cc extension.

Til Slutt: C++ er et stort språk med mange avanserte features... Start i det små med å legge enkle klasser og objekter til den rene C-programmeringen og du har fått en måte å dele systemet i moduler på som du ikke hadde fra før. En del av de avanserte tingene i C++ bør ikke brukes i det hele tatt :-)

3.3 Programflyt

- for, if, while, do, switch
- break, continue, return, goto

Dette kan dere - slå opp eller kjør google hvis ikke.

Funksjoner er call-by-value: En funksjon får *kopier* av parametrene sine overlevert og returverdien blir *kopiert*. Dette betyr at selv om parameter-variabelen endrer seg i funksjonen vil ikke endringene være synlige på kallerens side. For å få call-by-reference slik at en funksjon kan endre på parametrene de får må vi bruke pekere...

3.4 Typer, Grunnleggende

- int (long int, long long int, short int - signed/unsigned). Antall bits i en integer svarer vanligvis til ordbredden i mikroprosessoren (unntaket er gjerne på 64-bits prosessorer hvor vi alikevel har 32 bits integere).
- char - signed/unsigned - dette er alltid en byte.
- float, double & long double
- pekere og funksjonspekere? (Vi kommer til disse)
- void? (Kan brukes som returverdi for en funksjon som ikke returnerer noe eller som typen til en generisk, typeløs peker.)
- Hva med bool? Nei ingen bool! Konvensjon: Alt forskjellig fra 0 er true i tester; Sammenlign aldri sanne verdier! Obs: konvensjonen er at ved status-retur fra funksjoner så er 0 (vanligvis) suksess.

3.4.1 Specifiers:

Disse kan stå foran en definisjon:

- **static** - i en funksjon: denne variabelen vil beholde sin verdi imellom kallene til funksjonen.
- **static** - utenfor en funksjon: denne variabelen eller funksjonen skal ikke eksporteres fra objektfilen - kan ikke kalles eller brukes utenifra modulen.
- **volatile**: Ikke optimaliser bort aksess til denne variabelen.
- **register**: Antyd ovenfor kompilatoren at denne variabelen bør få fast plass i et mikroprosessor register.
- **extern**: Denne variabelen finnes og er definert en annen objekt-fil. (Mao. dette er en deklarasjon)
- **inline**: (foran funksjoner): Kompilatoren vil *kanskje* erstatte kallene til denne funksjonen med instruksjonene til funksjonskroppen der og da.
- **const**: Denne variabelen kan ikke endres etter at den er initialisert.

3.4.2 Nye Typer:

Vi kan gi nye navn til typer ved typedef deklarasjonen:

```
typedef <type> <nytt typenavn>;
typedef unsigned long long TAntall;
```

3.4.3 Type Casting:

Ved å sette en type i parantes foran en variabel kan du få kompilatoren til å konvertere variabelen til den typen, - eller regne med variabelen som om den var av denne typen (dette var to forskjellige ting; av og til skjer det ene, av og til skjer det andre...).

3.5 Struct

Dette er en samling med variable satt sammen til en (som en metodeløs C++ klasse med bare public medlemmer). Medlemmene i en struct aksesseres med '.' som i *myStruct.structMember*, eller hvis du har en peker til en struct, med '->' som i *myStructPointer->structMember*.

For både struct, union og enum gjelder at de kan gis et navn som en del av typen som i det følgende eksempelet:

```
struct SverresStruct {
  int v1;
  int v2;
};
```

Denne kan senere refereres ved "struct SverresStruct" uten at innholdet trenges å gjentas. Imidlertid brukes ofte typedef til dette også og da står gjerne struc'en uten navn:

```
typedef struct {
  int v1;
  char v2;
  int v3;
} SverresNyeTypeSomErEnStruct;
```

NB: Utlegg i minne av structer: Kompilatoren står temmelig fritt her, egentlig, men alikevel er utlegget temmelig deterministisk og det er ganske vanlig å gjøre forutsetninger mhp. dette: I eksemplet over kan vi gå ut ifra at v1 ligger på den laveste adressen i minnet av de tre medlemmene, og at v1's adresse er den samme som struct-variabelens adresse. Vidre kan vi anta at v2 ligger på byte 5 i forhold til starten på structen. For v3 kan vi ofte regne med at den starter på byte 9 selv om v2 bare er på en byte - for at v3 skal komme på en adresse som er delelig på 4. Slike antagelser er ødeleggende for portabiliteten av programmer.

Til slutt: Vær oppmerksom på at C lar oss gi egne navn til enkeltbit eller bitgrupper innenfor en byte eller integer. Dette er en del av struct (og union) syntaksen og kan brukes for å unngå komplierte logiske uttrykk f.eks. i forbindelse med binær io.

3.6 Union

En union er en samling av flere måter å aksessere det samme minneområdet på. Hva skriver dette programmet ut (endian.c)?

```
#include <stdio.h>

typedef union {
  int v;
  char bytes[sizeof(int)];
} Tall;

void main(void){
  Tall t;
  int i;
  t.v = 258;
  for(i=0;i<sizeof(int);i++){
    printf("Byte %d er %d\n",i,t.bytes[i]);
  }
}
```

Forklaring: sizeof(int) er størrelsen av en int - i bytes, dvs. hvor mange byte'er som er i en int. Dette er for vårt system 4 byte'er. Dvs. at det minneområdet som en variabel av typen Tall opptar og som er max(sizeof(int), sizeof(char[4])) = 4 bytes. Disse fire byte'ene kan altså aksesseres enten som en int, via 'v'-medlemmet eller som en array med 4 elementer av typen char (som hver tar 1 byte) via array-medlemmet 'bytes'.

t.bytes[i] er en verdi av typen char (altså en unsigned byte - 0-255) som ligger i posisjon 'i' i byte-arrayen. Og siden byte-arrayen altså bare er en måte å aksessere det samme minnet som integeren t.v opptar, så får vi altså skrevet ut den 'i'ende byte'en av de 4 byte'ene som utgjør denne integeren.

Du kan se at 258 == 2 + 1*256 + 0*256^2 + 0*256^3

Her har vi også et portabilitets-problem: Utskriften fra dette programmet er avhengig av hvordan mikroprosessoren setter sammen byte'er til integere. Er den byte'en med lavest adresse mest eller minst signifikant? Det er forskjell på prosessorer her og problemstillingen kalles "endianness".

I mitt tilfelle skriver programmet ut

```
sverre@eee:~/tilpdat$ ./endian
Byte 0 er 2
Byte 1 er 1
Byte 2 er 0
Byte 3 er 0
```

Men på en annen prosessor kunne vi tenke oss (f.eks.)

```
Byte 0 er 0
Byte 1 er 0
Byte 2 er 1
Byte 3 er 2
```

3.7 Enum

Her kan vi definere et sett med symbolske navn som en variabel av denne typen kan ha som verdi.

```
enum {
  bla,rod,gronn
} farge;
```

Når alt kommer til alt er det ikke så stor forskjell på denne typen og en integer... Verdien bla her vil bli representert ved 0, rod med 1 osv. og både variabelen farge og symbolene kan bli brukt omtrent som integere i koden. - Dette er først og fremst en mulighet til å øke *leselighetен* av kode.

Vi *kan* styre hvilken verdi de forskjellige symbolene får:

```
enum {
  bla = 0,
  rod = 100,
  gronn = 1000;
} farge;
```

Dette brukes innimellom som en litt sær måte å definere konstanter i C-kode på for å kompensere for at C ikke hadde konstanter før.

3.8 Pekere

En peker er altså en egen datatype i C og representerer en posisjon i minnet - rett og slett indexen til en byte i den store arrayen av bytes som minnet ditt er. For moderne systemer kan det vel skje en oversetting imellom de adressene som programmet opplever og det faktiske fysiske minnet, men dette er ikke sentralt her.

Antall bits i en peker tilsvarer da i utgangspunktet antall linjer i prosessorens adressebuss, mens antall bits i en int tilvarer (som oftest) linjer i databussen. Alikevel håndteres pekere i stor grad bare som tall/integere i koden.

Pekere har i tillegg en type etter hva de peker på:

```
{
  int i;      /* En int */
  int * pi;   /* En peker til int */
  pi = &i;    /* pi settes til å peke på i: &i er altså
                 adressen til i: Den kan også tolkes som indeksen
                 til den første av de bytene som
                 i er lagret i i den store minne-tabellen */
  int ** ppi; /* En peker til en peker til en int: Pekere er
                 jo også lagret et sted i minnet - og har
                 dermed en adresse/index/peker til seg.
                 NB: ppi er fortsatt bare en peker. */
  ppi = &pi;  /* ppi settes til å peke på pi */

  int j;
  j = *pi;    /* *pi er verdien av det som pi peker på */
  j = **ppi;  /* **ppi er verdien av det som den pekeren som
                 ppi peker på peker på... */
}
```

Vær oppmerksom på at følgende definerer en peker og en int (ikke to pekere!):

```
int * pi,i;
```

Pekere og arrayer er samme sak i C. Kompilatoren prøver å hjelpe oss litt med det, men saken er at en tabell av integere er representert ved en peker til int. Vi får altså muligheten til å indeksere pekere (!): a[0] er det samme som *a, og a[1] som er det samme som *(a+1) er (verdien av) *det neste* elementet i den tabellen som a "peker på". Hvis a mot formodning ikke peker til et array-minneområde så er a[1] en referanse til ugyldig/ulovlig minne.

```
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ cat pekere2.c
#include <stdio.h>

int main(void){
  int i = 1234;
  int j = 1;
  int * pi;  /* En peker til int - uinitialisert */
```

```
  int a[10]; /* Definerer ikke bare en peker a til int, men
                initialiserer denne også til å peke på et
                minneområde (på stacken) med plass til 10
                integere.
                Denne plassen er imidlertid uinitialisert.*/
  pi = &j;
  j = pi[0]; /* Det samme som "j = *pi;": mao vi setter j=j her; */

  for(j=0;j<10;j++) a[j] = j; /* Initialiser tabellen */

  printf("a[%d] = %d\n",5,a[5]);
  printf("a[%d] = %d\n",-1,a[-1]);   /* Ugyldig minne */
  printf("a[%d] = %d\n",100,a[100]); /* Ugyldig minne */

  pi = &j;
  printf("pi[%d] = %d\n",0,pi[0]); /* En peker aksessert
                                      som array */
  return 0;
}
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ gcc -g -Wall pekere2.c -o pekere2
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ ./pekere2.exe
a[5] = 5
a[-1] = 16
a[100] = 2290740
pi[0] = 10
```

C genererer ingen sjekking av lovlige indekser for array'er: Det er fullstendig mulig å adressere seg utenfor lovlig, avsatt minne. Bytt ut pi[0] med pi[1] i siste printf i koden over. Ingen flere warnings...

Vær også oppmerksom på følgende effekter ved beregninger på pekere:

```
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ cat pekere3.c
#include <stdio.h>

int main(void){
  char * pc = (char *) 100;
  int * pi =  (int *) 100;
```

```
  pc = pc + 1;
  pi = pi + 1;

  printf("pc = %d\n",pc);
  printf("pi = %d\n",pi);
  return 0;
}
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ gcc -g pekere3.c -o pekere3
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ ./pekere3.exe
pc = 101
pi = 104
```

Dvs. at aritmetiske operasjoner på pekere tar denne array-egenskapen til pekere med i betraktningen, og størrelsen av datatypen kommer med i bildet: "pi[1]" skal tilsvare "*(pi + 1)".

Strenger: C har strengt (hø hø) tatt ingen streng-type, men konvensjonen er å representere strenger som arrayer av char. Siden strenger ofte er av forskjellig lengde sier konvensjonen også at for å unngå å hele tiden skulle holde rede på hvor lang en streng er, så legger vi til en ekstra byte på slutten av strengen som settes til 0.

```
{
  char str[10]; /* En array av chars med plass til en
                   streng på 9 tegn + 0-terminering */
  char *pc;     /* Er det en peker til en char? eller er det en streng? */
}
```

Husk at det minområdet du bruker for å lagre strenger må ha plass til termineringen!

Hva er "char ** p;"? En peker til en peker til en char? En Array av strenger? En peker til en streng? En array av char-pekere? Hmm. Vi vet ikke.

Typecasting: Vi kan endre typen til en variabel eller uttrykk midlertidig ved å sette en ny type i parantes foran. Typisk kan verdier koverteres imellom float og int på denne måten. Dette *er* "tungen rett i munnen" saker uansett hva du gjør, og ved å caste typer fratar du kompilatoren noe av muligheten til å sjekke det du driver med...

Ved typecasting av pekere vil pekeren ikke endres og det ikke gjøres noe forsøk på å kovertere det som pekeren peker på - og det sjekkes som sagt ikke at det du forsøkte å oppnå faktisk var det du fikk til.

Dette gir oss en måte å oppnå det samme på som en union gir oss; Heller enn å aksessere en union på forskjellige måter, vil vi kunne aksessere variabelen via en peker til den, og typecaste denne til den typen vi trenger. Vurder om dette ville hjulpet oss (med tanke på leselighet/vedlikeholdbarhet) i det store peker-eksemplet under.

3.9 Funksjonspekere

Vit at det finnes funksjonspekere i C. Typen til en slik peker er gitt både av returverdien og parametrene til funksjonen, og et funksjonsnavn uten parameter-parentesen bak (som "main") er en slik funksjonspeker.

Det store pekereksemplet under opererer med en peker til compare-funksjonen der.

3.10 Det store peker-eksemplet.

Vi lager en sorteringsfunksjon (insert_sorted) for å legge til elementer i en lenket liste. Ved å gjøre oss avhengig av at første medlem i element-struct'en er next-pekeren (pekeren til neste element i listen) kan vi lage denne funksjonen uten å kjenne mer til element-typen. - Vi kan altså bruke den på lenkede lister av alle typer elementer.

Det store hintet for å forstå denne koden er den typekonverteringen som skjer fra å være en peker til et element til å være en peker til next-pekeren. Sjekk:

```
TElement * pe;
/* Vi ønsker å tilordne NULL til next-pekeren */

pe->next = NULL; /* Det kan gjøres slik... */

/* Men vi skulle jo greie oss uten å kjenne typen til structen:
   (Dvs. kompilatoren vet ikke at det finnes et 'next' element)
   Vi benytter oss av at next ligger først - en peker
   til (begynnelsen av) elementet peker til
   next-pekeren også - det er jo samme sted */

void * pe = makeNewElement(); // Vi kjenner ikke til TElement denne gangen
```

```
  * ((void **) pe) = NULL; // Men vi vet at pe peker til en peker,
                           // Og denne pekeren er det vi tilordner
}
```

Så: det vi ønsker er å kunne bruke "modulen" vår slik: Vi definerer en element type og en sammenligningsfunksjon, deretter kan vi kjøre insertSorted for å få bygd opp listen.

Her er programmet som bruker listemodulen vår.

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

typedef struct SElement {
  struct SElement * next;
  int value;
} TElement;

int
mycmp(void * e1, void * e2){
  if(((TElement *) e1)->value > ((TElement *) e2)->value)
    return 1;
  if(((TElement *) e1)->value < ((TElement *) e2)->value)
    return -1;
  return 0;
}

void
dumpList(void * list){
  TElement * l = (TElement *)list;
  while(l != NULL){
    printf("List element is %d\n",l->value);
    l = l->next;
  }
  printf("\n");
}

void main(void){
  TElement * e;
  void * list = NULL;
```

```
  e = (TElement *) malloc(sizeof(TElement));
  e->value = 4;
  list = insertSorted(list,e,mycmp);

  e = (TElement *) malloc(sizeof(TElement));
  e->value = 5;
  list = insertSorted(list,e,mycmp);

  e = (TElement *) malloc(sizeof(TElement));
  e->value = 7;
  list = insertSorted(list,e,mycmp);

  e = (TElement *) malloc(sizeof(TElement));
  e->value = 2;
  list = insertSorted(list,e,mycmp);
  dumpList(list);
}
```

Modul-headerfilen: clist.h Denne trenger ikke å vite om typen av pekerene.

```
#ifndef LIST_H
#define LIST_H

typedef int (*CmpFunction)(void *,void *);

void *
insertSorted(void * list,void * element,CmpFunction f);

#endif
```

Og til slutt: selve sorteringsfunksjonen: clist.c Som heller ikke trenger å kjenne til pekertypene.

```
#include "clist.h"
#define NULL 0

void *
insertSorted(void * list,void * element,CmpFunction f){
  void * p;
```

```
  if(list == NULL || f(element,list)<0){
    /* Insert the element as the first element */
    * ((void **)element) = list;
    return element;
  }

  /* Find the element in the list to insert  after. */
  p = list; /* We know that list != NULL */
  while(*((void **)p) != NULL &&
        f(element,*((void **)p))>=0){
    p = *((void **)p);
  }

  /* Insert after p. */
  *((void **)element) = *((void **)p);
  *((void **)p) = element;
  return list;
}
```

Lykke til :-)

3.11 Deklarasjoner og definisjoner, headerfiler og moduler i C

Det er et skarpt skille i C imellom hva som er en *deklarasjon* og hva som er en *definisjon*. Tenk slik på det: Definisjoner er de delene av programkoden som gir opphav til noe i objektfilen, mens deklarasjoner er ikke det; hjelp til kompilatoren, nye typer,...

Den programkoden som gir opphav til en objekt-fil kalles gjerne en 'compilation unit'.

```
// Dette er deklarasjoner:

// Det finnes en variabel i et sted, kanskje i en annen objektfil
extern int i;

// Dette er en 'prototyp'; Det finnes en slik funksjon f1
int f1(int a,int b);
```

```
// Deklarasjon av en struct
struct Element {
  struct Element * next;
  int value;
}

// Typedeklarasjon
typedef long long int64_t;

// Her kommer noen definisjoner:

// Denne j-variabelen skal legges ut i objektfilen og være
// tilgjengelig for bruk fra andre compilation units
int j;

// Denne k-variabelen skal legges ut i objektfilen men *ikke* være
// tilgjengelig fra andre compilation units
static int k;

// En funksjons-definisjon, tilgjengelig for andre.
int f1(int a,int b){
  return a+b;
}

// For den interesserte og nyskjerrige:
// Hva skal vi si om denne: Er det en deklarasjon eller en definisjon?
inline int f(int a, int b){
  return a+b;
}
```

Det er vanlig å ha C programkode i både .c og .h filer. Det er ingen prinsippiell forskjell på disse filtypene i utgangspunktet, og for den saks skyld kunne vi godt ha hatt programkode i filer med andre extensions også. Men siden vi i C ikke har noe modulbegrep innbakt i språket (ingen objekter, ingen pakker, ingen grensesnitt,...) så blir det naturlig å betrakte en kompilation unit som en modul. Vi støtter opp om dette ved konvensjoner:

- .c-filer inkluderer ikke hverandre, men kan inkludere .h filer når det trenges.
- .h filene inneholder bare *deklarasjoner*. Tenk på hva som vil skje hvis du

har definisjoner i en headerfil... Hvis denne headerfilen blir inkludert av flere .c-filer så vil det genereres flere versjoner av den samme variabelen i hver av de forskjellige objektfilene.

- De deklarasjonene som er i en headerfil er de som trenges for å beskrive modulens grensesnitt. Mao. alt det som andre moduler trenger å vite om for å kalle funksjoner og aksessere variable i modulen vår — men ikke mer.

- Så lenge en modul ikke er for stor, danner "samhørende .c og .h filer" en god modul i C. Det er da to filer med samme fornavn xxx.c med implementasjonen og xxx.h med grensesnittet.

- Det er vanlig, selv om det ikke alltid er nødvendig, at implementasjonen av en modul inkluderer sin egen headerfil. Dette hjelper kompilatoren til å detektere inkonsistenser (... men den blir alikevel ikke særlig flink til det...)

- Hvis modulen er større enn at hele implementasjonen går i en fil, eller hvis det er viktig at ikke mer enn det som trenges blir lenket med, så kan en headerfil representere grensesnittet til flere .c-filer. Her bruker vi gjerne foldere og/eller biblioteker for å holde greie på de samhørende .c og .o-filene.

Moduler med samme funksjonalitet kan ha vidt forskjellige grensenitt. Det å lage moduler med gode grensesnitt er en viktig del av programvare design.

Gode, intuitive grensesnitt i headerfilen kan forsvare mye griseri i implementasjonen :-)

3.12 Implementasjon av tilstandsmaskiner

Jeg repeterer fra Kyb Intro:

Det finnes tusen måter å implementere tilstandsmaskiner på. Her er en:

Tilstandsmaskinen er en modul, selvfølgelig, med en funksjon for hvert event i grensesnittet. I implementasjonsfilen er det en static variabel, av enum-type, som holder rede på hvilken tilstand vi er i, og alle event-funksjonene har samme struktur, med en switch på tilstandsvariabelen:

```
void play(){
  switch(g_state){
    case State_Play:
```

```
      break;
    case State_Stop:
      startMotor(); // En aksjon
      ...
      g_state = State_Play; // En transisjon
      break;
    case State_Pause:
      ...
      break;
    ...osv.
  }
}
```

Headerfilen vil altså se slik ut:

```
#ifndef KASETT_H
#define KASETT_H

void play();
void stop();
void ff();
void rw();
void record();
void endTape();
void pause();

#endif
```

4 Minne-håndtering

4.1 Stack og Heap

Det minnet som et C-program disponerer er typisk (i en enkel, klassisk modell) delt i flere deler med forskjellige bruksområder:

- Vanligvis legges **stacken** i øverste del av minne (og gror nedover mot lavere adresser). Her legges parametrene til funksjoner som kalles, lokale variable i funksjonene (og, på assembly nivå, returadressen til hvor programmet skal fortsette å kjøre etter at funksjonen er ferdig). Når en funksjon returnerer så blir den tilsvarende delen av stacken frigitt.

- In nederste del av minnet legges selve programkoden, initialiserte og uinitialiserte globale variable, verdiene til konstanter (osv.) og til slutt i øverste del av det nedre minnet, **heapen** som hvor evt. dynamisk allokert minne blir allokert (av malloc()-funksjonen).

Dermed ligger stacken i den ene enden av minnet og heapen i den andre på den måten at stacken gror mot midten, og at når du allokerer fra heapen vil minne så langt som mulig unna stacken bli foretrukket.

"Tradisjonelt" er det ingen sjekk på om stack og heap overskriver hverandre. Det er en del av det å programmere for et embedded system å forsikre seg om at stack og heap har nok plass. Lenkeren tar opsjoner som lar oss styre dette.

I dette landskapet er det mange feil som kan gjøres, og C kompilatoren kan skjelden gi gode advarsler på forhånd for å unngå dem. Slike minne-feil har som regel også odde symptomer som ikke lett kan spores tilbake til feilen i koden. Vær forsiktig! Jeg lister noen typiske slike feil:

- Du har endt opp med en ugyldig peker, som peker et tilfeldig sted. I beste fall, hvis du kjører på et moderne system kan programmet kræsje der og da, når du de-refererer pekeren, men ellers kan du lese eller endre tilfeldige data i programmet ditt, andre programmer eller tråder, eller OS'et; inkludert programkode hvis dette ligger i skrivbart minne.

- Du har noe som *var* en gyldig peker, men som peker til minne som er blitt frigitt igjen. Slikt minne kan oppføre seg greit så lenge det varer, men på et senere tidspunkt kan de bli allokert igjen, og overskrevet.

- Stack og Heap kan overskrive hverandre.

- En funksjon kan returnere en peker til en av sine lokale variable - som blir en peker til ugyldig minne når funksjonen returnerer.

- **Minne-lekasje:** Hvis du glemmer å frigi allokert minne som du ikke bruker lengre - (og gjenbruker pekervariablene du har dit) - har du mistet kontakten med dette minnet for alltid, og hvis programmet gjør dette innimellom som en del av normal drift vil programmet/systemet før eller siden kræsje på en vanskelig-å-finne-feilen måte. Dette er i sterk kontrast til f.eks. java hvor kjøretidssystemet tar hånd om å gjenbruke minne som ikke er i bruk lengre. Det er ingen garbage-collection i C.

- **Minne-fragmentering:** Du kan være uheldig med bruksmønsteret

– rekkefølgen av malloc og free-kallene slik at selv om det er mye ledig minne så er det ledige minnet fordelt over mange små områder slik at en stor allokering kan mislykkes selv om det skulle være nok totalt ledig minne.

Finger-ormen (fra 1988) virket slik: Ved å overfylle strengvariable på stacken kunne (kan!) en få tilgang til å sette programutførelse til egen kode (som også er en del av strengen hvis du har planlagt det riktig :-) ved å overskrive retur-adressen på stacken. Sjekk at du ikke overfyller strenger!

Det finnes flere flotte verktøy som kan avsløre om vi har slikke minne-problemer i koden vår. Sjekk valgrind, ...

Følgende program kan skrive ut hva du har på stacken:

```
static void
f(char a,char b,char c,char d,char e,char f,char g){
  char fString[5];
  int i;
  char *p;
  strcpy(fString,"ABCD");
  p = &g;
  for(i=0;i<100;i++){
    int j = i;
    int v = (int) p[j];
    printf("At i=%3d we find\t %d\t - %c\n",j,v,p[j]);
  }
}
int main(void){
  char mainString[5];
  strcpy(mainString,"abcd");
  f('1','2','3','4','5','6','7');
  return 0;
}
```

4.2 malloc og free

- **void * malloc(size)** Returnerer en peker til (ny)allokert minne på size bytes. (0 hvis kallet feiler.)
- **free(p)** tar en peker p som tidligere har vært returnert fra malloc og frigir minnet igjen.

Typisk:

```
{
  struct myStruct * ps;
  /* ps er uinitialisert og peker et tilfeldig sted i minnet */

  ps = (struct myStruct *) malloc(sizeof(struct myStruct));
  /* ps peker nå til gyldig minne - til en riktignok uinitialisert myStruct */

  free(ps);
  /* ps peker ikke til gyldig minne lengre. (men om du fortsetter å
     bruke pekeren går det kanskje bra en stund :-) */
}
```

5 Diverse

5.1 Litt mer om preprosessoren

Det er vel lite substans her som vi ikke allerede har vært innom:

Dere har sett #include statementene hvor #include <file> leter etter filene i en liste av foldere hvor systemet har header-filene sine. Typisk "/usr/include". #include "file" starter letingen i current directory.

Dere har sett omkransningen

```
#ifndef SVERRE_H
#define SVERRE_H

int f1(int a, int b);
int f2(int a, int b);

#endif
```

i headerfilen. Preprosessoren kan brukes til "condisional compilation" ved å teste på om slike symboler er definert eller ikke. I tillegg til å definere symboler ved #define kan de også defineres på kommandolinjen til kompilatoren ved -D opsjonen.

define kan også definere makroer på den måten at definerte symboler vil bli byttet (en tekstlig utbytting, før kompilatoren får fingrene i koden din!) ut med "verdien" eller "kroppen" sin i resten av filen:

```
#define SUM(a,b) a+b

void main(void){
  printf("SUM(2,3)*5 = %d\n",SUM(2,3)*5);
}
```

Hvorfor skriver programmet over ut 17 og ikke 25?

5.2 Gdb

gdb er debuggeren. Den ble demonstrert såvidt i forelesningen. For at debuggeren skal kunne hjelpe oss så godt som mulig må informasjon om hvilke linjenummer i kildekoden de forskjellige instruksjonene tilhører embeddes i objektfiler og i den kjørbare filen. Dette gjør vi ved -g opsjonen til gcc.

En sesjon med gdb:

```
Sverre@home-sverre3b:~/tool/html/c/cdemo
$ gdb hello.exe
GNU gdb 6.3.50_2004-12-28-cvs (cygwin-special)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU
General Public License, and you are
welcome to change it and/or distribute copies
of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type
"show warranty" for details.
This GDB was configured as "i686-pc-cygwin"...
(gdb) list
1       #include <stdio.h>
2       #include "sverre.h"
3       #define SUM(a,b) a+b
4
5       int
6       main(){
7         int i;
8         i = f1(3.0,4.0);
9         // Går dette?
10        int j;
(gdb) break 7
```

```
Breakpoint 1 at 0x40107a: file hello.c, line 7.
(gdb) run
Starting program:
/cygdrive/e/sverreshome/tool/html/c/cdemo/hello.exe

Breakpoint 1, main () at hello.c:8
8         i = f1(3.0,4.0);
(gdb) help step
Step program until it reaches a different
source line.
Argument N means do this N times (or till
program stops for another reason).
(gdb) step
f1 (a=3, b=4) at f1.c:5
5         return a+b;
(gdb) help where
Print backtrace of all stack frames, or
innermost COUNT frames.
With a negative argument, print outermost
-COUNT frames.
Use of the 'full' qualifier also prints
the values of the local variables.

(gdb) where
#0  f1 (a=3, b=4) at f1.c:5
#1  0x0040108e in main () at hello.c:8
(gdb) print a
$1 = 3
(gdb) step
6       }
(gdb) step
main () at hello.c:11
11        printf("3+4=%d\n",i);
(gdb) step
3+4=7
12        printf("SUM(2,3)*5 = %d\n",SUM(2,3)*5);
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Lykke til.

5.3 Versjonskontroll

Jeg anbefaler sterkt å sette seg inn i et versjonskontrollsystem. Det er nyttige verktøy: ("CVS" og "subversion "er vel innarbeid og kurant, ikke for vanskelig å sette opp. "git" holder vel på å ta over...)

- Tar vare på (alle) gamle versjoner av filene dine.
- Lar flere arbeide samtidig på de samme filene.
- arkivet/repository funker som en ekstra backup
- Lar deg flytte arbeidsplass uten å ende med et "levende" prosjekt for hvert sted.
- kan vise deg forskjeller imellom versjoner, når bugs ble innført, osv.
- Kan holde rede på samhørende varianter av flere filer, inkludert å organisere forskjellige versjoner av total-systemet og flytte endringer imellom variantene.
- ...

Dere trenger aldri mer å ha flere foldere med forskjellige versjoner av prosjektet deres hvor dere ikke er helt sikker på hva som var den vesjonen som var demonstrerbar...

Hvis noen gjør jobben med å sette opp et felles repository for en gruppe studenter under universitetets infrastruktur - send meg en kokeoppskrift så kan jeg publisere den her?

5.4 Objektorientering i C

Spørsmålet kommer opp omkring hva en skal gjøre med et objektorientert design, hvis en skal implementere i C...

Det store svaret her er at objektorientering er en måte å dele systemet i deler på og når du først har gjort dette koster ikke en overhead i kompleksitet ved at du implementer i et ikke-objektorientert språk *så* mye for vedlikeholdbarheten.

Det finnes ingenting som *ikke* kan implementeres i C - det er bare et spørsmål om hvor enkelt :-) Du kommer langt med et C-type modulbegrep - en .c og en .h fil per objekt, som opererer med structene for dataene og hvor alle "medlemsfunksjoner" tar en slik struct som første parameter.

Polymorfi kan en få ved å lagre funksjonspekere til de virtuelle funksjonene i egne tabeller som det pekes til fra disse structene.

Problemet med dette er at kompilatoren ikke er forberedt på å detektere feil i det som i C må bli "konvensjoner" i stedet for "språk" som i java eller C++

5.5 Sammendrag av nevnte gccopsjoner

-v	verbose: Fortell nøyaktig hva du driver med.
-E	Kjør bare preprosessoren
-S	Stopp etter kompilering - generer assembly-fil
-c	Stopp etter assembly - generer objektfil
-I dir	Let også i denne folderen etter include-filer
-L dir	Let også i denne folderen etter biblioteker
-g	Legg inn linjenummerinfo også i objekt- og kjørbare filer.
-lxxx	Lenk med de objektfilene som trenges fra libxxx.a
-Wxxx	Skru av eller på warnings fra kompilatoren. "-Wall" skrur på alle.
-Ox	Optimaliser.
-Dxxx	Definer xxx som symbol til preprosessoren (som #define)

5.6 Sammendrag av nevnte kommandoer

gcc	paraply-program som hjelper oss med kompilering.
cpp	C preprosessorene
nm	Lar oss se hvilke symboler som er nevnt i en objekt- eller kjørbar fil
make	Leser makefilen "Makefile" og gjøre (bare) det som trenges å gjrøes
(sp)lint	Detekterer uggen C-kode bedre enn kompilatoren
ar	lager et biobliotek av flere objektfiler
ranlib	Legger til en kjekk tabell over hvilke symboler som defineres av hvilken objektfil i et bibliotek. Ofte gjør 'ar' dette automatisk
strip	fjerner debug-informasjon (Linjenumre, navn) fra en objekt- eller kjørbar fil
valgrind	La valgrind kjøre programmet ditt for å avsløre minne-problemer
gdb	Debuggeren