Tilpassede Datasystemer
Heisprosjekt

© 2014-2019
Institutt for teknisk kybernetikk
NTNU

Revisjonshistorie

2014	Øyvind Stavdahl
2014	Anders Rønning Petersen
2016	Øyvind Stavdahl
2016	Konstanze Kölle
2017	Ragnar Ranøyen Homb
2017	Bjørn-Olav Holtung Eriksen
2018	Filippo Sanfilippo
2018	Kolbjørn Austreng
2019	Kolbjørn Austreng

1 Introduksjon

1.1 Motivasjon

Programmeringsspråket C benyttes regelmessig i et bredt spekter industrisammenhenger, særlig i sanntidsapplikasjoner og maskinnær programvare. Denne oppgaven vil gå ut på å bruke C til å implementere et styresystem for en heis. I tillegg til selve implementasjonen, skal systemet beskrives- og dokumenteres i UML.

Målet for prosjektet er å gi praktisk erfaring med utvikling av et system med definerte krav til oppførsel, ved å benytte UML og C som verktøy. Det er også anbefalt å benytte seg av den pragmatiske V-modellen til å strukturere arbeidet under prosjektet, og for å sikre verifikasjon av akseptkriterier.

1.2 Vurdering

Heisprosjektet vil telle på sluttkarakteren i faget. Prosjektet er delt i tre tellende deler, som i utgangspunktet (før eventuelle sluttjusteringer) teller like mye:

- Dokumentasjon av systemet; UML og beskrivelse av APIene til hver modul.
- Kodekvalitet; vedlikeholdbarhet av produsert kode.
- Dekningsgrad av kravspesifikasjonen; FAT.

I tillegg til dette innbefatter heisprosjektet godkjenning av "del 2" av øvingene i faget:

- Versjonskontroll med Git.
- Automatisk bygging med GNU Make.
- Debugging med GDB og Valgrind.
- API-dokumentasjon med Doxygen.

Øvingene har ingen direkte innvirkning på sluttkarakteren i emnet, men må være godkjent for å få tilgang til eksamen. Utover dette vil en god innsats på øvingene hjelpe dere med de tellende delene av prosjektet.

1.3 Utstyrsbeskrivelse

Vi skal bruke en fysisk modell av en heis i løpet av prosjektet. Denne modellen består av tre hoveddeler; selve heismodellen, et betjeningspanel, og en motorstyringsboks. Det finnes en heismodell på hver av Sanntidssalens arbeidsplasser.

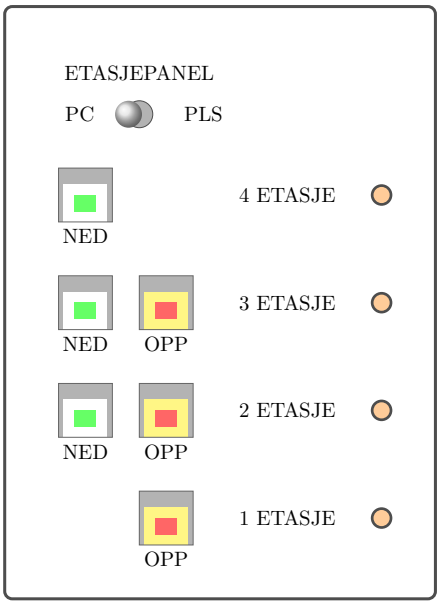
1.3.1 Heismodell

Heismodellen er illustrert til høyre i bildet på forsiden av dette dokumentet, og består av en heisstol som kan beveges opp og ned langs en stolpe. Dette tilsvarer henholdsvis heisrommet- og sjakten i en virkelig heis.

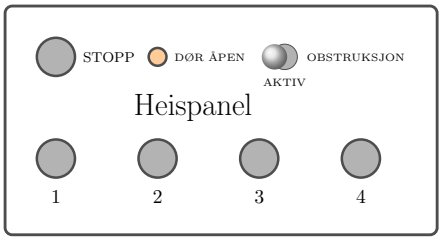
Langs heisbanen er det montert fire Hall-effektsensorer som fungerer som heisens etasjer. Over øverste etasje, og under nederste etasje er det også montert endestoppbrytere, som vil kutte motorpådraget dersom heisen kjører utenfor sitt lovlige område. Dette er for å beskytte heisens motor mot skade. Om heisen skulle treffe en av endestoppene, må heisstolen manuelt skyves bort fra bryterne før en kan be motoren om et nytt pådrag.

1.3.2 Betjeningsboks

Betjeningsboksen er delt i to; *etasjepanel* og *heispanel*. Disse er illustrert i figur 1 og figur 2. Øverst på betjeningsboksen finnes en bryter som velger om datamaskinen eller PLSen skal styre heismodellen. Denne skal stå i "PC" gjennom hele oppgaven.



Figur 1: Etasjepanel.



Figur 2: Heispanel.

Etasjepanelet finnes på oversiden av betjeningsboksen. På dette panelet finner dere bestillingsknapper for opp- og nedretning fra hver etasje. Hver av knappene er utstyrt med lys som skal indikere om en bestilling er mottatt eller ei. Etasjepanelet har også ett lys for hver etasje for å indikere hvilken

etasje heisen befinner seg i.

Heispanelet finnes på kortsiden av betjeningsboksen og representerer de knappene man forventer å finne inne i heisrommet til en vanlig heis. Her har man bestillingsknapper for hver etasje, samt en stoppknapp for nødstans. Alle knappene er utstyrt med lys som kan settes via styreprogrammet. I tillegg til knappene er det inkludert et ekstra lys, markert "Dør Åpen", som indikerer om heisdøren er åpen. Heispanelet har også en obstruksjonsbryter, men denne skal vi ikke bruke i dette prosjektet.

1.3.3 Motorstyringsboks

Styringsboksen er ansvarlig for å forsyne effekt til heismodellen, og for å forsterke pådraget som settes av datamaskinen. Motoren kan forsynes med mellom 0- og 5 V, som henholdsvis er minimalt- og maksimalt pådrag. Retningen til motoren settes ved et ekstra retningsbit i styringsboksens grensesnitt. Alt dette gjøres via funksjonskall i styringsprogrammet.

Det er også mulig å hente ut et analogt tachosignal, samt en digital encoderverdi, for å lese av hastighet- og posisjon fra styringsboksen. Disse målesignalene trenger dere ikke ta stilling til i dette prosjektet, men de nevnes for fullstendighetens skyld.

1.4 Virkemåte og oppkobling

Heismodellen er laget for å konseptuelt oppføre seg som en virkelig heis. Det er et par punkter man bør merke seg for å få den til å fungere som ønsket:

- Alle lys må settes eksplisitt. Det er ingen automatikk mellom Hall-effektsensoren i hver etasje og tilhørende etasjeindikator.
- Om endestoppbryterne aktiveres vil pådrag til heisen kuttes. Om det skjer må heisstolen manuelt skyves vekk fra endestopp.
- Rød og blå ledning forsyner effekt til motoren. Disse kobles henholdsvis til *M+* og *M-* som dere finner på motorstyringsboksen.

2 Kravspesifikasjon

Kravspesifikasjonen i denne seksjonen beskriver punktene styringskoden skal oppfylle. Ved uklarheter kan dere betrakte studass eller vitass som kunden av heissystemet, og spørre om oppklaring.

2.1 Oppstart

Punkt	Beskrivelse
O 1	Ved oppstart skal heisen alltid komme til en definert tilstand. En definert tilstand betyr at styresystemet vet hvilken etasje heisen står i.
O 2	Om heisen starter i en udefinert tilstand, skal heissystemet ignorere alle forsøk på å gjøre bestillinger, før systemet er kommet i en definert tilstand.
O 3	Heissystemet skal ikke ta i betraktning urealistiske start-betingelser, som at heisen er over fjerde etasje, eller under første etasje idet systemet skrus på.

2.2 Håndtering av bestillinger

Punkt	Beskrivelse
H 1	Det skal ikke være mulig å komme i en situasjon hvor en bestilling ikke blir tatt. Alle bestillinger skal betjenes, selv om nye bestillinger opprettes.
H 2	Heisen skal ikke betjene bestillinger fra utenfor heisrommet om heisen er i bevegelse i motsatt retning av bestillingen.
H 3	Når heisen først stopper i en etasje, skal det antas at alle som venter i etasjen går på eller av. Dermed skal alle ordre i etasjen være regnet som ekspedert.
H 4	Om heissystemet ikke har noen ubetjente bestillinger, skal heisen stå stille.

2.3 Bestillingslys og etasjelys

Punkt	Beskrivelse
L 1	Når en bestilling gjøres, skal lyset i bestillingsknappen lyse helt til bestillingen er utført. Dette gjelder både bestillinger inne i heisen, og bestillinger utenfor.

L 2	Om en bestillingsknapp ikke har en tilhørende bestilling, skal lyset i knappen være slukket.
L 3	Når heisen er i en etasje skal korrekt etasjelys være tent.
L 4	Når heisen er i bevegelse mellom to etasjer, skal etasjelyset til etasjen heisen sist var i være tent.
L 5	Kun ett etasjelys skal være tent av gangen.
L 6	Stoppknappen skal lyse så lenge denne er trykket inne. Den skal slukkes straks knappen slippes.

2.4 Døren

Punkt	Beskrivelse
D 1	Når heisen ankommer en etasje det er gjort bestilling til, skal døren åpnes i 3 sekunder, for deretter å lukkes.
D 2	Om heisen ikke har ubetjente bestillinger, skal heisdøren være lukket.
D 3	Hvis stoppknappen trykkes mens heisen er i en etasje, skal døren åpne seg. Døren skal forholde seg åpen så lenge stoppknappen er aktivert, og ytterligere 3 sekunder etter at stoppknappen er sluppet. Deretter skal døren lukke seg.

2.5 Sikkerhet

Punkt	Beskrivelse
S 1	Heisen skal alltid stå stille når døren er åpen.
S 2	Heisdøren skal aldri kunne åpne seg om heisen ikke står i en etasje.
S 3	Heisen skal aldri kjøre utenfor området definert av første- og fjerde etasje.
S 4	Om stoppknappen trykkes, skal heisen stoppe momentant.
S 5	Om stoppknappen trykkes, skal alle heisens ubetjente bestillinger slettes.
S 6	Så lenge stoppknappen holdes inne, skal heisen ignorere alle forsøk på å gjøre bestillinger.
S 7	Etter at stoppknappen er blitt sluppet, skal heisen stå i ro til den får nye bestillinger.

S 8	Det er ikke tillatt å kjøre en kalibreringsrunde etter at stoppknappen er blitt trykket; heissystemet skal vite hvor heisen befinner seg selv om stoppknappen brukes.

2.6 Robusthet

Punkt	Beskrivelse
R 1	Obstruksjonsbryteren skal ikke påvirke systemet.
R 2	Det skal ikke være nødvendig å periodisk starte programmet på nytt som følger av eksempelvis udefinert oppførsel, at programmet krasjer, eller minnelekkasje.

3 Oppgaver

Dere står fritt til å velge arbeidsmetode, så lenge dere er i stand til å levere det som er beskrevet i seksjon 3.1. Det anbefales derimot sterkt å benytte seg av V-modellen for å strukturere arbeidet. Fremgangsmåten i V-modellen er beskrevet i seksjon 3.2.

3.1 Hva skal leveres inn?

Alt i denne seksjonen skal leveres inn for å få full vurdering av heisprosjektet. Dere skal levere én `.zip`-fil til slutt, som skal ha følgende struktur:

```
levering.zip
  |-- Makefile
  |
  |-- source
  |   |-- .h-filer og .c-filer
  |
  |-- docs
      |-- .pdf-fil(er) av UML
```

Vær så snill, lever én `.zip`-fil til slutt.

I mappen kalt `source` skal all kode nødvendig for å kjøre heisen ligge. Dette vil også inkludere utleverte drivere. Alle UML-diagrammer putter dere i mappen `docs`, i `.pdf`-format. `Makefilen` skal inneholde regler for å bygge styreprogrammet deres, basert på filene i `source`.

3.1.1 Dokumentasjon

Alle offentlige APIer skal være dokumentert med Doxygen. Med offentlige APIer menes alle funksjoner, structer, enumer, etc som ligger tilgjengelig i headerfiler (`.h`-filer).

Om dere har private hjelpefunksjoner, eller structer som kun brukes i implementasjonfiler (`.c`-filer) er det ikke noe krav at de skal være dokumentert, men det skader ikke.

Introduksjon til hvordan dere bruker Doxygen finner dere i seksjon 5.

I tillegg til Doxygen-kommentert kode skal dere også levere følgende UML-diagrammer:

- Overordnet **klassediagram** som viser hver av modulene som inngår i designet deres, og hvilke relasjoner som finnes mellom dem.
- Minst ett **sekvensdiagram**; ihvertfall et som viser denne sekvensen:
  1. Heisen står stille i 2. etasje med døren lukket.
  2. En person bestiller heisen fra 1. etasje.
  3. Når heisen ankommer, går personen inn i heisen og bestiller 4. etasje.
  4. Heisen ankommer 4. etasje, og personen går av.
  5. Etter 3 sekunder lukker dørene til heisen seg.
- Et **tilstandsdiagram** som viser oppførselen til heisen basert på hvilke input som kommer inn, og hvilke output heisen selv genererer.

Dere står fritt til å legge ved flere diagrammer om dere mener det hjelper til å forklare designet deres, og valgene dere har gjort. Dette kan potensielt telle positivt, men husk at et utall lite forklarende diagrammer fortsatt er mindre verdt enn ett eneste godt forklarende diagram.

Ved tvil om hva dere skal levere, og hvilket format dere skal levere i, vær vennlig å les denne seksjonen igjen, eller spør studass eller vitass.

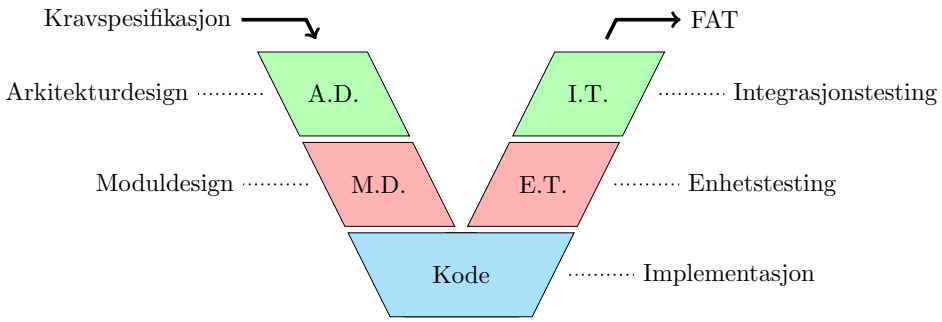
3.1.2 Kode

Kode legger dere i mappen `source`, som dere deretter zipper. All kode dere trenger for å kompilere styringsprogrammet og kjøre heisen skal ligge i denne mappen - også utleverte drivere.

Det skal være mulig å kalle `make` for å kompilere programmet deres fra innlevert kode, uten at endringer i selve koden skal være nødvendige.

For kodekvalitet bør dere ta en titt på seksjon 6.

3.2 Anbefalt fremgangsmåte



Figur 3: Illustrasjon av V-modellen.

V-modellen er illustrert i figur 3. Det kan være fristende å hoppe direkte inn i implementasjonsfasen, men dere bør ikke ta for lett på hverken analyse og design, eller testing. Det er mye bedre med noen få linjer gjennomtenkt og veltestet kode, enn mange linjer med spaghetti som "funket i går".

Hver av stegene i V-modellen er beskrevet under.

3.2.1 Arkitekturdesign

Dere bør først forsikre dere om at dere forstår punktene i kravspesifikasjonen. Når dere er sikre på at dere vet hva som kreves av sluttsystemet, tenker dere gjennom hvilke implikasjoner dette har for koden dere skal skrive senere.

På dette stadiet ønsker vi å bestemme en *arkitektur* som vil oppfylle kravene fra spesifikasjonen. Det innebærer å legge abstraksjonsnivået forholdsvis høyt, og ignorere implementasjonsdetaljer enn så lenge.

Eksempel: Heisen skal kunne huske ordre helt til de blir ekspedert. Ikke tenk "dette skal jeg implementere som en lenket liste", men heller "på arkitekturnivå trenger vi et køsystem". Detaljene om hvordan et eventuelt køsystemet er implementert kommer ikke inn i bildet på dette stadiet.

Resultatet av dette stadiet bør være ett eller flere klassediagrammer som illustrerer hvilke moduler styringssystemet skal bestå av. For å få en idé om hvilken funksjonalitet hver modul må tilby kan det også være lurt å sette opp et par sekvensdiagrammer som illustrerer hvordan de forskjellige modulene samarbeider.

Om dere føler behovet, kan dere også supplere klasse- og sekvensdiagrammene med kommunikasjonsdiagrammer for å få en bedre oversikt over grensesnit-

tet mellom hver modul.

3.2.2 Moduldesign

Når dere har en overordnet tanke om hvilke moduler som kreves for å oppfylle kravspesifikasjonen, er det på tide å skissere hvordan hver modul skal se ut. Et spørsmål som er lurt å ta stilling til her er om hver modul trenger å lagre tilstand eller ikke.

Eksempel: Et køsystem trenger åpenbart å lagre tilstand, mens en modul som utelukkende setter pådrag til motorstyringsboksen kanskje kan skrives uten å ha hukommelse.

En modul som ikke husker ting mellom hvert funksjonskall vil alltid ha færre måter den kan feile på enn en tilsvarende modul som lagrer tilstand. Det kan derfor være lurt å skille ut delene av systemet som trenger hukommelse i en dedikert tilstandsmaskin, og beholde hjelpemoduler så enkle som mulig.

Her bør dere benytte dere av klassediagrammer og tilstandsdiagrammer. Motivasjonen for å gjøre dette er at det er mye lettere å eksperimentere og endre på designet på diagramnivå, enn med halvferdig kode.

3.2.3 Implementasjon

Det er på dette stadiet dere skriver kode. Hvis dere har lagt inn en grei innsats i designfasen, vil dette stadiet koke ned til å oversette diagrammene til kode. Så feilfritt går det selvsagt aldri, men et godt forarbeid kan spare dere for mye unødvendig hodebry. Vær heller ikke redd for å gå tilbake for å endre på arkitekturen eller modulsammensetningen om dere finner mer hensiktsmessige måter å gjøre noe på.

Det kan også være interessant å nevne forskjellen mellom å "programmere inn i et språk" og "programmere i et språk". Om man programmerer "i et språk" vil man begrense abstraksjonskonseptene og tankesettet sitt til de primitivene som språket direkte støtter. Om man deretter programmerer "inn i et språk" vil man først bestemme seg for hvilke konsepter man ønsker å strukturere programmet inn i, og deretter finne måter å implementere konseptene på i språket man skriver.

Eksempel: C er ikke i utgangspunktet objektorientert. Allikevel kan man se på hver klasse i et klassediagram som en egen modul, hvor alle funksjonene som modulen gjør tilgjengelig svarer til offentlige medlemsfunksjoner i en klasse, og så videre.

Selvsagt er det en fordel å benytte seg av de primitivene et språk gjør tilgjengelig, fremfor å prøve å tvinge inn funksjonalitet som ikke er støttet[1], men det er alltid greit å tenke gjennom et program som en abstrakt oppskrift på hvordan man løser et problem - før man tenker "dette kan implementeres som en klasse som arver fra en annen".

3.2.4 Enhetstesting

Enhetstesting speiler moduldesignfasen. Her gjør dere tester som forsikrer dere om at hver modul oppfører seg som den skal. I første omgang er det greit å gjøre små, veldig veldefinerte tester, som tester ut én bestemt funksjon fra modulen dere prøver ut.

Antallet tester er ikke et bra mål på hvor godt testet en modul er, så sikt heller på å teste forskjellige ting. Randtilfeller ("Border cases") er stort sett en langt større kilde til feil enn vanlige tilfeller, så det er altså mye mer verdifullt med *tester som tester forskjellige ting* enn med *forskjellige tester som tester samme ting.*

Eksempel:

> "A test engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.
>
> First real customer walks in asks where the bathroom is. The bar bursts into flames, killing everyone."

3.2.5 Integrasjonstesting

Enhetstesting foregår på modulnivå og svarer på spørsmålet "Fungerer denne modulen som den skal?". Integrasjonstesting speiler arkitekturdesignfasen, og svarer på spørsmålet "Fungerer denne modulen sammen med denne andre modulen?". Her vil dere typisk prøve ut hele- eller nesten hele programmet på en spesifikk funksjonalitet. Om dere har tatt dere tid til å lage fornuftige sekvensdiagrammer, kommer disse godt med i denne fasen.

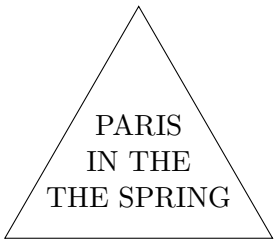
Integrasjonstesting kan enten være en særdeles enkel oppgave, eller et sant helvete, avhengig av graden kobling dere har mellom modulene i programmet. Moduler som avhenger sterkt av andre moduler blir nødvendigvis både vanskeligere å teste, og å vedlikeholde. Derfor er det ønskelig at moduler kun vet om- og kommuniserer med akkurat de modulene den trenger.

[1] "You can write FORTRAN in any language"

Eksempel: 23. September 1999 ble *Mars Climate Orbiter* tapt etter at fartøyet enten gikk i stykker i Mars' atmosfære, eller spratt tilbake til en utilsiktet heliosentrisk bane. Styringssystemet til sonden bestod av software skrevet av Lockheed Martin og NASA. Begge hadde testet sine egne moduler, men Lockheed Martin sine moduler opererte med *pund per sekund* ($\mathrm{lb} \cdot \mathrm{s}$), mens NASA sine moduler opererte med *newton per sekund* ($\mathrm{N} \cdot \mathrm{s}$). Manglende integrasjonstesting[2] endte totalt opp med å koste NASA JPL omlag 330 millioner amerikanske dollar.

3.2.6 Kommentar om testing

Når dere tester er det lett å tro at man har testet alt det går an å teste, eller å tro at en gitt del av kodebasen må være feilfri. Som mennesker har vi også en ulempe når det kommer til testing - vi har en tendens til å oppfatte bare det vi ønsker å se, eller gjøre en subjektiv farging av input vi får. Ta en titt på figur 4 for en illustrasjon.



Figur 4: Hva er det som står her?

[2] Og enheter som kun U-land bruker

4 Factory Acceptance Test

Factory Acceptance Test er sluttprøven som bestemmer i hvilken grad dere har implementert et korrekt system. FATen er en direkte gjenspeiling av kravspesifikasjonen fra seksjon 2, så om dere oppfyller alle kravene som er satt av den, har dere implementert et fullverdig system.

5 Dokumentasjon med Doxygen

Doxygen er et verktøy som kan ta kode med spesielt formaterte kommentarer og automatisk generere dokumentasjon basert på disse. Fordelen med at dokumentasjonen er basert på kommentarer fremfor et eget dokument er at det blir mer sannsynlig at dokumentasjonen forholder seg korrekt, selv med endrende implementasjon - så fremst man oppdaterer kommentarene, og ikke bare koden, selvsagt.

I heisprosjektet skal dere levere Doxygen-generert dokumentasjon av det offentlige APIet til alle modulene dere har implementert. Med dette menes alle funksjoner og variabler som er synlige utenfor en modul - det er ikke nødvendig å dokumentere private hjelpefunksjoner, om dere skulle ha dem, men det skader ikke.

Prosessen for å generere dokumentasjon med Doxygen er illustrert via et eksempel, beskrevet i seksjonene 5.1 til 5.5.

5.1 Eksempelprosjekt

Vi har et prosjekt som består av en "memory"-modul, som definerer noen enkle operasjoner på lister av heltall. Denne modulen er implementert i `memory_library.h` og `memory_library.c`, og prosjekttreet vårt ser slik ut:

```
.
|-- [Makefile og eventuelle andre filer]
|
|-- source
    |-- main.c
    |-- memory_library.h
    |-- memory_library.c
```

Det eneste modulen består av er funksjonene `memory_reverse_copy()` og `memory_multiply_elements()`. Det er disse vi ønsker å dokumentere.

5.2 Doxygen config

Når dere kjører Doxygen, forventer programmet å finne en konfigurasjonsfil som forteller hva det skal gjøre. For å opprette en konfigurasjonsfil kaller vi `doxygen -g doxconfig`. Dette vil opprette en fil kalt `doxconfig`, med en del parametre dere kan sette for å bestemme oppførselen til Doxygen senere.

Når vi har generert denne filen redigerer vi den med en teksteditor av fritt valg. Endringene vi gjør i dette eksempelet er som følger:

```
PROJECT_NAME = "Memory Library Example"
OPTIMIZE_OUTPUT_FOR_C = YES
INPUT = source/
SOURCE_BROWSER = YES
```

Alle parametre dere kan sette er forklart med kommentarer i konfigurasjonsfilen. Prosjekttreet vårt ser nå slik ut:

```
.
|-- [Makefile og eventuelle andre filer]
|-- doxconfig
|
|-- source
    |-- main.c
    |-- memory_library.h
    |-- memory_library.c
```

5.3 Kommenter koden

For å dokumentere koden vår, legger vi inn spesielt formaterte kommentarer som består av kommandoer som sier hvordan kommentaren skal tolkes. I toppen av hver fil vi vil at Doxygen skal se gjennom, må vi inkludere `file`-kommandoen, for å inkludere den i dokumentasjonen. I utgangspunktet ser `memory_library.h` slik ut:

```
#ifndef MEMORY_LIBRARY_H
#define MEMORY_LIBRARY_H

int memory_reverse_copy(const int * p_from, int * p_to, int
↪  size);

void memory_multiply_elements(int * p_buffer, int factor, int
↪  size);

#endif
```

For å fortelle Doxygen at det skal genereres dokumentasjon for denne filen, må vi markere den med Doxygen sin `file`-kommando, som vi putter i toppen av `h`-filen:

```
/**
 * @file
 * @brief A simple library for doing operations on memory
 * buffers consisting of integers
 */
```

I tillegg til `@file`, har vi også spesifisert `@brief`, som simpelthen gir en kort oppsummering av denne filens funksjon.

Vi skal nå dokumentere hva de to funksjonene i filen gjør. Dokumentasjonen putter vi i `h`-filen, men her har dere implementasjonen av de to funksjonene for å skjønne hva de gjør:

```
#include "memory_library.h"
#include <stdlib.h>

int memory_reverse_copy(const int * p_from, int * p_to, int
↪  size){
    if(from == NULL || to == NULL){
        return 1;
    }

    for(int i = 0; i < size; i++){
        to[size - i - 1] = from[i];
    }

    return 0;
}

void memory_multiply_elements(int * p_buffer, int factor, int
↪  size){
    if(p_buffer == NULL){
        exit(1);
    }

    for(int i = 0; i < size; i++){
        p_buffer[i] *= factor;
    }
}
```

5.3.1 `int memory_reverse_copy(...)`

Som vi ser fra funksjonsdefinisjonen vil `memory_reverse_copy` ta inn to buffere, og kopiere fra det første inn i det andre, samtidig som rekkefølgen på elementene blir reversert. I pseudokode har vi altså:

```
array_one = {1, 2, 3, 4}
array_two = [space for four integers]
memory_reverse_copy(array_one, array_two, 4)
array_two == {4, 3, 2, 1}
```

For å dokumentere denne funksjonen, skriver vi følgende kommentar inn rett over `memory_reverse_copy` i `h`-filen:

```
/**
 * @brief Copy a list of integers from one buffer to another,
 * reversing the order of the output in the process.
 *
 * @param[in]  p_from Source buffer.
 * @param[out] p_to   Destination buffer.
 * @param[in]  size Number of integers in the buffer.
 *
 * @return 0 on success, 1 if either @p p_from or @p p_to
 * is a @c NULL pointer.
 */
```

Som før gir `@brief` kun en kort oppsummering av hva funksjonen gjør. Kommandoen `@param` forteller hva en funksjonsparameter sin oppgave er. Det er frivillig å legge ved `[in]`, `[out]` eller `[in,out]` ved en parametererklæring; denne vil simpelthen fortelle om parameteren blir modifisert av funksjonen eller ikke. Dette er nyttig for ting som kalles med referanse i C++, eller via pekere i C. Følgende konvensjon brukes:

- `[in]`: Parameteren brukes inne i funksjonen, men vil ikke endres på utsiden av funksjonskroppen.
- `[out]`: Parameteren brukes ikke direkte inne i funksjonen, men vil være endret på utsiden av funksjonen når den returnerer.
- `[in,out]`: Verdien til parameteren brukes direkte i funksjonen, og den vil være endret på utsiden av funksjonen når den returnerer.

Deretter ser vi kommandoen `@return`, som naturlig nok forteller oss hva funksjonen returnerer. Denne trenger man ikke spesifisere for `void`-funksjoner.

Vi ser også bruk av `@c`, og `@p`. `@c` vil ta det neste ordet og formatere det som kode i dokumentasjonen. Altså vil ordet "NULL" formateres som `NULL` i dokumentasjonen. `@p` er en referanse til en parameter, og vil også formateres som kode i den ferdige dokumentasjonen.

5.3.2 `void memory_multiply_elements(...)`

Det er ikke mye ny magi for å dokumentere denne funksjonen. Det eneste vi gjør forskjellig her er å introdusere `@warning`, som vi kan bruke for å advare brukeren mot oppførsel som kanskje ikke er åpenbar:

```
/**
 * @brief Multiply all the elements in @p p_buffer, of size
```

```
 * @p size with the supplied @p factor.
 *
 * @param[in,out] p_buffer Buffer of integers to be multiplied
 * with @p factor.
 *
 * @param[in]     factor Factor to multiply each of the
 * elements in @p p_buffer with.
 *
 * @param[in]     size Size of @p p_buffer.
 *
 * @warning If @p p_buffer is @c NULL, the function will
 * abruptly terminate the program with exit code @c 1.
 */
```

5.3.3 Heads up

For en komplett liste over hvilke kommandoer Doxygen støtter, kan dere ta en titt på doxygen.nl.

Legg også merke til at det er mulig å bruke "\" istedenfor "`@`" for å starte kommandoer i Doxygen. For C og C++ anbefales "`@`" fordi tegnet ikke kolliderer med *escape characters* (som `\n` eller `\t`) i koden, som gjør det enklere å søke gjennom etter Doxygen-kommentarer senere.

Det er lett å glemme at hver fil må markeres med `@file` i toppen for å bli inkludert. Da vil dere ende opp med tom dokumentasjon.

5.4 Generer dokumentasjonen

Fra samme mappe som konfigurasjonsfilen ("`doxconfig`") ligger i, kaller vi `doxygen doxconfig`. Dette vil generere dokumentasjon i både HTML- og LaTeX-format, i hver sin mappe. Når dette er gjort, vil prosjektmappen se slik ut:

```
.
|-- [Makefile og eventuelle andre filer]
|-- doxconfig
|
|-- source
```

```
|   |-- main.c
|   |-- memory_library.h
|   |-- memory_library.c
|
|-- html
|   |-- index.html
|   |-- [...]
|
|-- latex
    |-- refman.tex
    |-- [...]
```

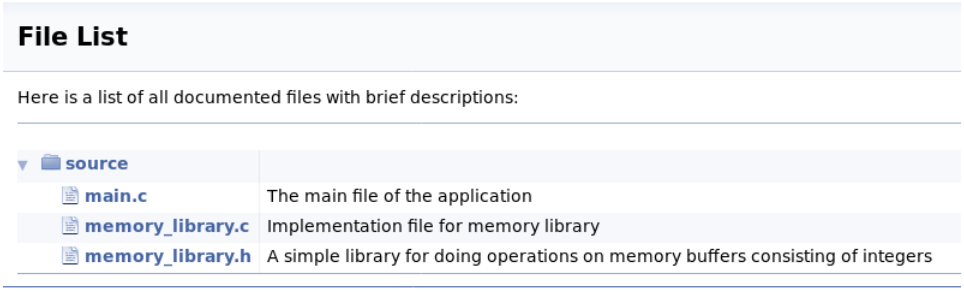
Dere kan velge hvilke formater Doxygen skal generere dokumentasjon på ved å endre på parametrene `GENERATE_LATEX` eller `GENERATE_HTML` i konfigurasjonsfilen.

Dere kan også endre hvor Doxygen plasserer den genererte dokumentasjonen, ved å endre på parameteren `OUTPUT_DIRECTORY`.

5.5 Nyt sexy dokumentasjon

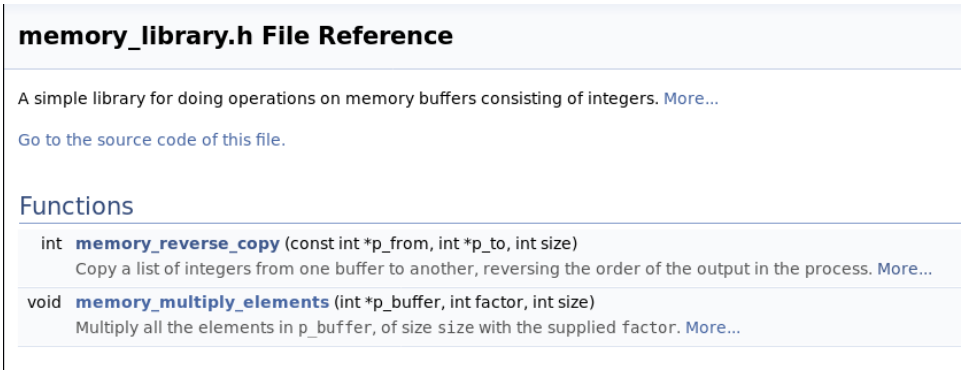
Åpne `html/index.html` i en egnet nettleser ved å enten kalle noe i duren av `firefox html/index.html` fra terminalen, eller trykke `Ctrl + O` inne i nettleseren og åpne `index.html` derfra.

I utgangspunktet vil dere bli møtt av en ganske tom side, fordi vi ikke har brydd oss om å sette så mye under "`Project related configuration options`" i konfigurasjonsfilen. Dette står dere fritt til å endre på som dere vil. Uansett vil dere ha en lenke til `Files` oppe i venstre hjørne. Denne kan dere følge for å få en oversikt som illustrert i figur 5.



Figur 5: Oversikt over hvilke filer prosjektet består av.

Om dere herfra følger lenken til `memory_library.h`, vil dere få dokumentasjonen til denne filen; gjengitt i figur 6.



memory_library.h File Reference

A simple library for doing operations on memory buffers consisting of integers. More...

Go to the source code of this file.

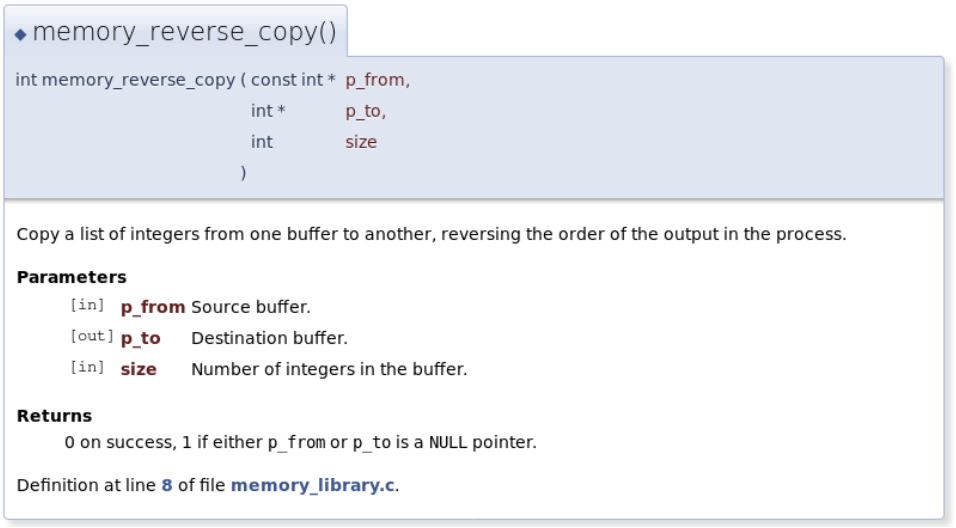
Functions

int **memory_reverse_copy** (const int *p_from, int *p_to, int size)
Copy a list of integers from one buffer to another, reversing the order of the output in the process. More...

void **memory_multiply_elements** (int *p_buffer, int factor, int size)
Multiply all the elements in `p_buffer`, of size `size` with the supplied `factor`. More...

Figur 6: Dokumentasjon for `memory_library.h`.

Herfra kan dere også følge lenkene som heter "`More...`" for å få dokumentasjonen til hver enkelt funksjon. Dette er gjengitt i figur 7 og figur 8.



◆ memory_reverse_copy()

```
int memory_reverse_copy ( const int *  p_from,
                          int *        p_to,
                          int          size
                        )
```

Copy a list of integers from one buffer to another, reversing the order of the output in the process.

Parameters

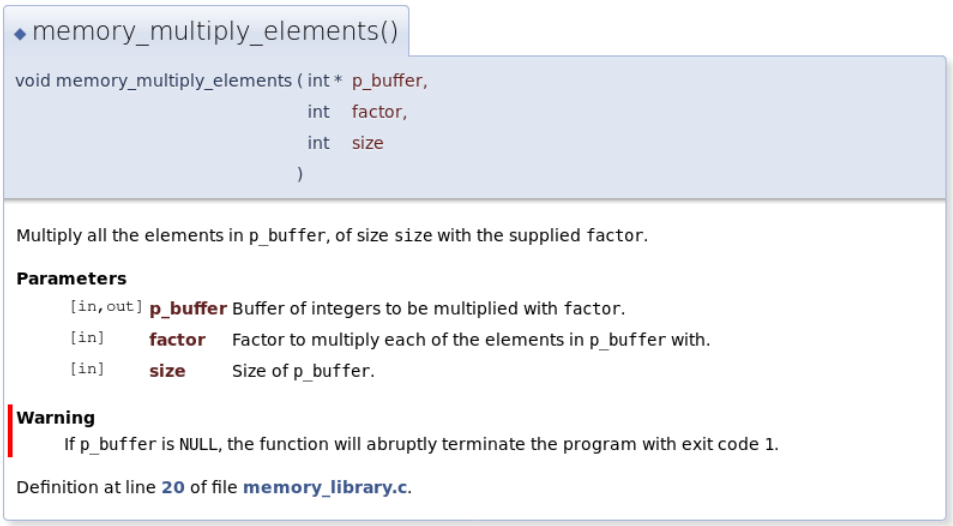
[in]	**p_from**	Source buffer.
[out]	**p_to**	Destination buffer.
[in]	**size**	Number of integers in the buffer.

Returns

0 on success, 1 if either `p_from` or `p_to` is a `NULL` pointer.

Definition at line **8** of file **memory_library.c**.

Figur 7: Dokumentasjon for `int memory_reverse_copy( [...] )`.



◆ memory_multiply_elements()

```
void memory_multiply_elements ( int * p_buffer,
                                int   factor,
                                int   size
                              )
```

Multiply all the elements in p_buffer, of size size with the supplied factor.

Parameters

[in,out]	**p_buffer**	Buffer of integers to be multiplied with factor.
[in]	**factor**	Factor to multiply each of the elements in p_buffer with.
[in]	**size**	Size of p_buffer.

Warning

If p_buffer is NULL, the function will abruptly terminate the program with exit code 1.

Definition at line **20** of file **memory_library.c**.

Figur 8: Dokumentasjon for `void memory_multiply_elements( [...] )`.

6 Kodekvalitet

For deres egen del bør dere ha kodekvalitet i bakhodet hele tiden. Hva som er god kode finnes det mange meninger om, men felles for dem alle er at de gir lesbar kode som er lettere å vedlikeholde enn annen kode.

Ute i virkelige settinger bruker man betraktelig mye mer tid på å lese kode enn å skrive kode selv. Uavhengig om det er noen andre sin kode, eller deres egen kode en del frem i tid, er det mye greiere om den er skrevet for leserens skyld - enn at den er skrevet for å spare nanosekunder på ubetydelige steder.

Under ligger en liste av hva som regnes som god kodekvalitet i dette faget. Listen er i stor grad basert på Code Complete 2 av Steve McConnell, men noen ekstra punkter som er nyttige for C er også lagt til. Det er helt greit å være uenig i deler av- eller hele listen, men da bør dere ha en bedre konvensjon selv, og dere bør i alle fall være helt konsekvente.

6.1 Moduler

- Alle funksjonene i en modul bør ha samme abstraksjonsnivå. Ikke bland lavnivå funksjonalitet med høynivå funksjonalitet.
- En modul har som formål å gjemme noe bak et grensesnitt. Implementasjonsdetaljene til modulen bør altså ikke lekke opp til overflaten. Ideelt sett skal dere kunne bruke modulen uten å ha den ringeste anelse om hvordan innmaten dens ser ut.

- Hver modul skal ha en sentral oppgave. En modul bør dermed ikke håndtere flere vidt forskjellige ansvarsområder.
- Grensesnittet til hver modul bør gjøre det helt åpenbart hvordan modulen skal brukes. I dette ligger det også å navngi modulfunksjoner logisk.
- Moduler bør snakke med så få andre moduler som mulig, og samarbeidet mellom moduler bør være så lett koblet som mulig. Dere skal altså kunne fjerne én modul uten å måtte endre på alle andre som inngår i programmet.
- For klasser er det ønskelig at alle medlemsvariabler er definerte etter at konstruktøren har kjørt. Tilsvarende, for moduler er det ønskelig at all medlemsdata er definert etter en eventuell initialiseringsfunksjon.

6.2 Funksjoner

- Den viktigste grunnen til å opprette en funksjon er ikke kodegjenbruk, men å gi dere en måte å håndtere kompleksitet på. Det kan fint hende at dere kommer over tilfeller hvor det er mer lesbart å repetere dere selv et par ganger, fremfor å trekke noen få linjer ut i en egen funksjon.
- Alle funksjoner bør ha ett eneste ansvarsområde - én oppgave som funksjonen gjør bra.
- Navnet på en funksjon bør beskrive alt funksjonen gjør.
- Sterke verb foretrekkes fremfor svake- og vage verb. For eksempel bør dere sky ord som "handle" eller "manage" som pesten.
- Kohesjon er et viktig begrep for å klassifisere funksjoner:
  - **Sekvensiell kohesjon** beskriver funksjoner hvor stegene som tas innad i funksjonen må gjøres i den bestemte rekkefølgen de er satt opp i.
  - **Kommunikasjonskohesjon** er når en funksjon benytter samme data til å gjøre forskjellige ting, men hvor bruken av dataen ellers er urelatert.
  - **Tidsavhengig kohesjon** har man hvis en funksjon inneholder mange forskjellige operasjoner som gjøres til samme tid, men som ellers ikke har noe med hverandre å gjøre.
  - **Funksjonell kohesjon** har man i funksjoner som inneholder instruksjoner for å gjøre én ting. Tingene funksjonen gjør er altså nødvendige for å utføre en bestemt oppgave.

  Av disse er funksjonell kohesjon den mest ønskelige.

- Motsatte verb bør være presise og opptre i veldefinerte par som "begin - end", "create - destroy", "open - close" eller "next - previous".
- Funksjoner har også kobling mot hverandre i den forstand at de avhenger av returverdien til andre funksjoner. Denne koblingen bør være så løs som mulig. Om dere endrer én funksjon skal det ikke være nødvendig å endre mange andre.

6.3 Variabler

- Unngå å bruke for mange "arbeidsvariabler" - variabler som opprettes i starten av en funksjon for så å muteres gjennom hele funksjonens levetid.
- Navnekvaliteten til en variabel bør speile variabelens levetid. En iteratorvariabel kan fint hete "i", mens en global variabel bør ha et virkelig godt navn.

6.4 Kommentarer

- En kommentar er en erkjennelse om at koden som kommentaren beskriver ikke er åpenbar. Åpenbar kode trenger ikke kommentarer, og er bedre enn esoterisk kode med kommentarer.
- Alle kommentarer må være oppdatert. Det er fort gjort å endre kode, uten å endre kommentarene rundt. Kode med ukorrekte kommentarer har mindre verdi enn kode uten kommentarer.

6.5 Øvrig

- Funksjoner bør prefikses med navnet på modulen sin for å gjøre det åpenbart hvor funksjonen kommer fra, og for å gjøre samme jobb som et namespace.
- Variabler kan med fordel prefikses med `p_` om de er pekere, `pp_` om de er pekere til pekere, `m_` om de er modulvariabler begrenset med kodeordet `static`, og `g_` om de globale.
- Selv om C er relativt lavnivå er det fullt mulig å ivareta god softwareutviklingspraksis. Allikevel kommer man alltid til å skrive noe "ondskapsfull" kode - det gjelder bare å velge det beste alternativet tilgjengelig.
- Det er helt greit å være uenig i denne listen, eller ha andre konvensjoner dere heller vil følge, så lenge dere kan argumentere for at de bedre ivaretar lesbarhet og vedlikeholdbarhet.