Wiki »

Building CC++ programs for host n target

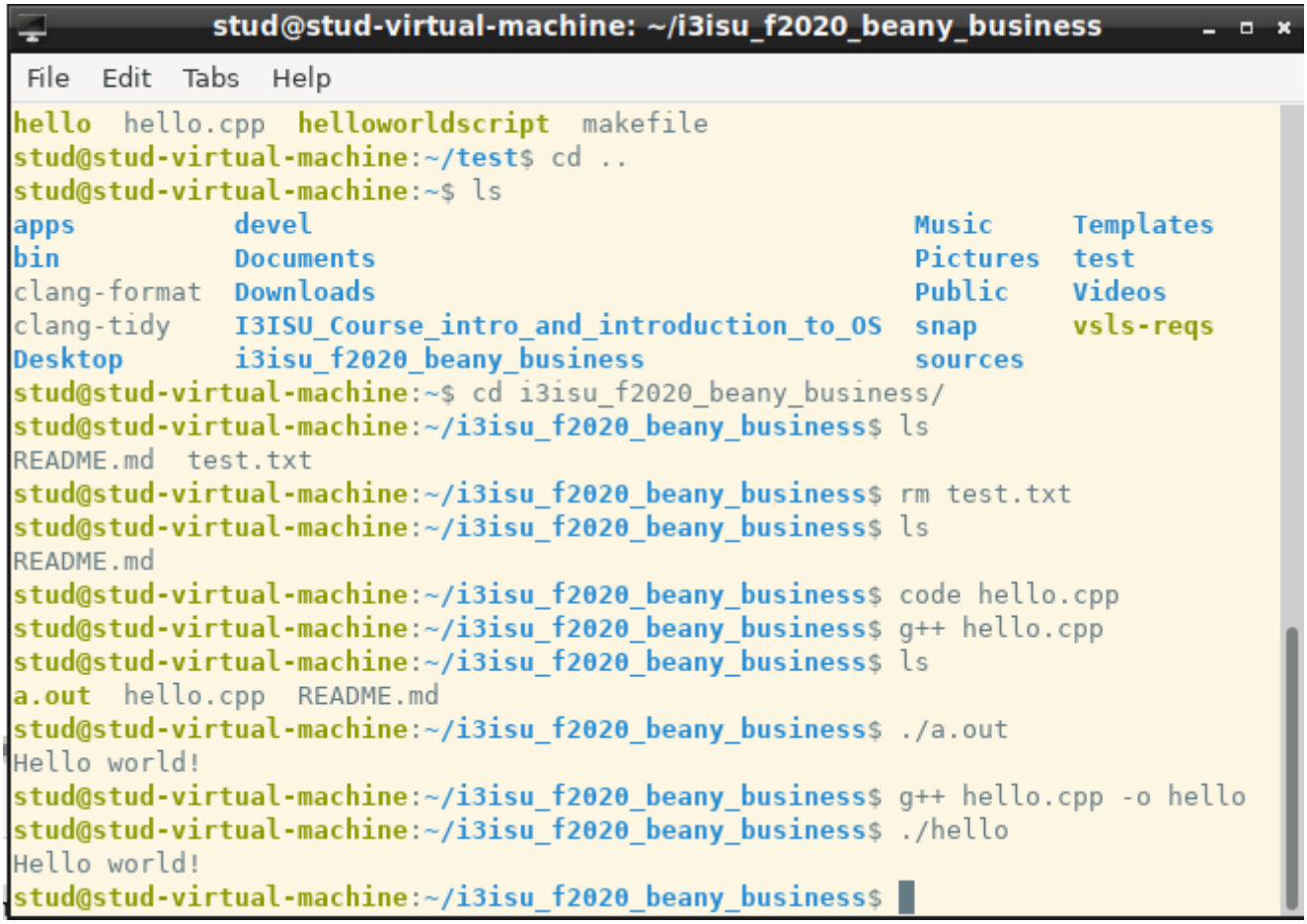
Exercise 1 The first Makefile

Exercise 1.1 The ”Hello World” program

Først åbnes programmet kate fra terminalen og programmet implementeres og gemmes.

```
#include <iostream>
using namespace std;
int main()
{
    cout<<"Hello world!"<<endl;
    return 0;
}
```

På nedestående billede af terminalen kan det ses hvordan programmet oprettes, kompileres og til sidst eksekveres.



Exercise 1.2 Makefile basics

- What is a target?
  Det er målet i en Makefile med givende regler. Ofte er navnet på taget den fil som bliver eller ændret efter udførslen.

- What is a dependant and how is it related to a target?
  Det en samling af filer som skal være opdateret for at kunne lave et target. Hvis en dependant er ændret eller ikke kompileret før, så vil handlingen blive udført.

- Does it matter whether one uses tabs or spaces in a makefile?
  Ja. Tabs angiver en recipe til et target, hvis tab ikke er brugt vil det ikke blive anset som et target. Spaces bruges til at separere kommandoer fra argumenter.

- How do you define and use a variable in a makefile?
  Der er to måder at gøre dette på:

```
x = værdi
y:= værdi
```

X er erklæret som en rekursiv variabel, dvs. at hver gang der refereres til variablen, vil variablens værdi ekspanderes og evalueres. Variablens værdi kan altså ændres, uden at der skrives en ny værdi til x, hvis variablen afhænger af en anden variabel der ændres.
Y er erklæret som en simpel variabel. Dvs. at y's værdi evalueres når den erklæres, og derefter antager denne værdi, indtil/hvis y tilskrives en ny værdi.

- Why use variables in a makefile?
  Variabler gør at et program kan være meget mere fleksibel. Istedet for at skrive data direkte ind i programmet, kan variabler bruges til at repræsentere de data. Variabler spiller derfor en stor rolle i programmering. For for eksempel at ændre navnene på filer der kompileres i en makefile, kan variable

benyttes, så man kun ændrer filnavnene i en variabel. Derefter kan man skrive kode der bruger den pågældende variabel til at kompilere det ønskede program uafhængigt af navnene på de enkelte filer.

- How do you use a created makefile?
  Ved at bruge kommandoen *make* i terminalen når man står i den samme mappe, som selve makefilen ligger i. Hvis man kun skriver *make* vil den søge efter "makefile" og udføre det med default target. Derfor hvis køre med en specifik makefile skrives *make navn*, hvor navn står for navnet på den fil.

- In the makefile scripting language they often refer to built-in variables such as these
  - `$, $< and $^ - explain what each of these represent.`
    $ den referer til det nuværende target.
    $< referer altid til den første dependancy i det nuværende target.
    $^ giver en string med alle dependamcies.

- $(CC) and $(CXX):
  - What do they refer to?
    De referer til systemets compiler.
  - How do they differ from each other?
    CC gør det til c og og CXX gør det til c++.

- $(CFLAGS) and $(CXXFLAGS):
  - What do they refer to?
    De referer til flag for komplireingen. Det er for at sikre at alle de har de samme flag, flagene bliver allokeret i FLAGS-variablen, som så vil blive brugt til kompileringen.
  - How do they differ from each other?
    CFLAGS gør det for c og CXXFLAGS for c++.

- What does $(SOURCES : .cpp = .o) mean? Any spaces in this text???
  Det betyder at den værdi variablen SOURCES er modtaget, og at får hver .cpp-fil vil der blive oprettet en output fil med samme navn bare .o istedet for .cpp.

Exercise 1.3 Writing the makefile

Målet ved at lave en makefile er så man ikke behøver manuelt compliere alle ens filer selv, men herved istedet automatisere det. Dette er især anvendeligt hvis man arbejder på projekter med bare mere end en fil.

```
SOURCES= hello.cpp
OBJECTS= $(SOURCES:.cpp=.o)
CXX=g++
hello: $(OBJECTS)
    $(CXX) $^ -o $@

%.o: %.cpp
    $(CXX) -c $^ -o $@
```

Exercise 2 Makefiles - compiling for host

Exercise 2.1 Using makefiles - Next steps

Vi har lavet vores makefile relativ generel ved hjælp af **%-funktionen**, og herved gør det muligt nemt at slette og tilføje filer. Derudover da både *clean* og *help* funktionerne ikke skal oprette en outputfil, bruges kommandoen **.PHONY** for at fortælle makefilen dette.

```
SOURCES= hello.cpp
OBJECTS= $(SOURCES:.cpp=.o)
EXE=hello
CXX=g++
all: $(OBJECTS)
```

```
    $(CXX) $^ -o $(EXE)

%.o: %.cpp
    $(CXX) -c $^ -o $@

.PHONY: clean
clean:
    rm $(OBJECTS) $(EXE)
.PHONY: help
help:
    @echo "all clean help"  $(OBJECTS)
```

Exercise 2.2 Program based on multiple files

Exercise 2.2.1 Being explicit

Først oprettes de simple filer til parts:

parts1.cpp

```
#include "part1.h"
#include <iostream>
using namespace std;
void part1()
{
    cout<<"This is part 1"<<endl;
}
```

part1.h

```
#ifndef PART1_H
#define PART1_H

void part1();

#endif
```

part2.cpp

```
#include "part2.h"
#include <iostream>
using namespace std;
void part2()
{
    cout<<"This is part 2"<<endl;
}
```

part2.h

```
#ifndef PART2_H
#define PART2_H

void part2();

#endif
```

main.cpp

```
#include "part1.h"
#include "part2.h"
int main(void)
{
    part1();
    part2();
    return 0;
}
```

Inkluderer de nyoprettede filer og opdater makerfile fra Exercise 2.1

```
SOURCES= part1.cpp part2.cpp main.cpp
DEPENDENCIES=part1.h part2.h
OBJECTS= $(SOURCES:.cpp=.o)
EXE=exe
CXX=g++
all: $(OBJECTS)
    $(CXX) $^ -o $(EXE)

%.o: %.cpp %.h
    $(CXX) -c $< -o $@

.PHONY: clean
clean:
    rm $(OBJECTS) $(EXE)
.PHONY: help
help:
    @echo "all clean help"  $(OBJECTS)
```

Exercise 2.2.2 Using pattern matching rules

Ved brug af pattern matching rule formindsker man både mængde af hvad der skal skrive for at kunne eksekvere koden rigtigt, men vigtigst af alt mindsker man chancen for at lave fejl, f.eks. stavefejl.
Derudover til at løse denne opgave bruger vi den samme kode som før, da vi allerede havde gjort brug af dette hack. Men for god orden skyld kommer koden her igen:

```
SOURCES= part1.cpp part2.cpp main.cpp
DEPENDENCIES=part1.h part2.h
OBJECTS= $(SOURCES:.cpp=.o)
EXE=exe
CXX=g++
all: $(OBJECTS)
    $(CXX) $^ -o $(EXE)

%.o: %.cpp %.h
    $(CXX) -c $< -o $@

.PHONY: clean
clean:
    rm $(OBJECTS) $(EXE)
.PHONY: help
help:
    @echo "all clean help"  $(OBJECTS)
```

Exercise 2.3 Problem...

Der antages først og fremmest at følgende filer eksisterer og er fyldestgørende nok til at compilere:
server.cpp og server.hpp
data.cpp og data.hpp
connection.cpp og connection.hpp

Derefter tages der udgangspunkt i denne makefile:

Listing 2.1: Simple makefile creating a simple program executable called prog

```
1 EXE=prog
2 OBJECTS=server.o data.o connection.o
3
4 $(EXE): $(OBJECTS)
5       $(CXX) -o $@ $^
```

Følgende spørgsmål besvares for at skabe en bedre forståelse for hvad implicitte regler er og hvorfor det er vigtigt at kende dem, samt deres begrænsninger.

Questions to consider:

- How are the source files compiled to object files, what happens?
  I denne makefile gøres brug af implicitte regler, da der ikke specifikke regler. Dette kan gøres fordi der er en generel konvention om at navngive sine header og source filer f.eks. data.cpp og data.hpp, altså ens navn men forskellige suffix. Derfor vil programmet godt kunne bygge en executable fil.
  Den implicitte recipe ville i dette tilfælde være: '$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c' da det er et C++ program der compileres.

- When would you expect make to recompile our executable prog - be specific / precise with respect to file names?
  Når en ændring blev foretaget i en af .cpp filerne ville der blive recompileret.

- Make fails using this particular makefile in that not all dependencies are considered by the chosen approach. Which ones are not?
  Her opstår et problem, der vil nemlig ikke blive recompileret hvis der ændres i nogle af .hpp filerne, da disse ikke er inkluderet som dependencies i makefile.

- Why is this dependency issue a serious problem?
  Når der ikke bliver recompileret, selvom der er lavet ændringer i .hpp filerne ville det skabe problemer i programmet. Så snart der er lavet ændringer i en .hpp fil skal den tilsvarende .cpp og .o fil recompileres.

Exercise 2.4 Solution

Listing 2.2: Using finesse to ensure that dependencies are always met

```
1  SOURCES=main.cpp part1.cpp part2.cpp
2  OBJECTS=$(SOURCES:.cpp=.o)
3  DEPS=$(SOURCES:.cpp=.d)
4  EXE=prog
5  CXXFLAGS=-I.
6
7  $(EXE): $(DEPS) $(OBJECTS)   # << Check the $(DEPS) new dependency
8    $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)
9
10 # Rule that describes how a .d (dependency) file is created from a .cpp
       file
11 # Similar to the assigment that you just completed %.cpp -> %.o
12 %.d: %.cpp
13   $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $^ > $@
14
15 -include $(DEPS)
```

Givet den ovenstående kode vil vi nu gennemgå det og forklare hvad der gøres og hvorfor det afhjælper vores problem med dependencies i den tidligere opgave.

I linje 1-5 angives først de source files der arbejdes med og de gemmes i variablen SOURCES, dette er smart da vi i de næste linjer erklæres variablen OBJECTS som tager variablen SOURCES og skifter suffixet .cpp med .o, det samme sker når DEPS variablen erklæres her bliver suffixet .d.
I linje 4 erklæres variablen EXE, deri angives navnet for vores executable.
I linje 5 erklæres CXXFLAGS til -I, som angiver i hvilket directory der skal søges efter makefiles, i dette tilfælde med et punktum som angiver der skal søges i 'current' directory.

Linje 7-8 angiver en recipe hvor der udelukkende er brugt variabler til at definere executable navnet, .o filer, dependencies og flag. Der er altså taget højde for at en ændring i dependencies også ville tvinge en recompilering.

I linje 12 laves der tilsvarende .d filer til alle .cpp filer.
I linje 13 ses først $(CXX) der henviser til compileren, -MT ændrer target til den string man vælger, i dette tilfælde vælges .o filer. -MM skifter outputtet fra resultatet af preprocesseren til en regel til make der beskriver dependencies fra source filen hvilket svarer til hvad -M ville gøre. -MM undlader dog også system header filer, og kun lokale filer inkluderes. Til sidst skiftes der således outputtet gemmes i target med > $@ delen.

I sidste linje, nummer 15, inkluderes alle .d filer. Hvis en dependency fil ikke er blevet generet vil make nu sørge for at det gøres.

Exercise 3 Cross compilation & Makefiles

Exercise 3.1 First try - KISS

Til at starte denne opgave oprettes der en fil kaldet *makefile.target*, hvor alt fra *makefile* opgave 2.1 kopiers over i.

- Do you have to do something special to invoke this particular makefile?
  I den oprindelige fil Makefile bruges g++ compileren da det skal køre på Linux, men for at få Makefile.target til at fungere på RPI bruges funktionen:

```
CXX=arm-rpizw-g++
```

Herefter for at kunne tilgå og make Makefile.target, bruges kommandoen *-f*, så compileren ved at det er en makefile. Vi skriver derfor således i terminalen:

```
make -f Makefile.taget
```

- At this point we have two makefiles in the same dir. How does this present a problem in the current setup and how are you forced to handle it?
  Problemet ved at have to makefils i samme directory, kan være hvis den ene er kørt og derfor har oprettet outputfilerner. Det giver et problem da den tilbageværende makefil ser det ikke nødvendigt at gen-compile de outputfiler, selvom de tilhører en anden makefile. For at undgå dette problem bruges kommandoen:

```
make clean
```

Dette kalder "clean" target i makefilen, der fjerner evt. objekter og det eksekverbare program.

Exercise 3.2 The full Monty - Bye bye KISS

- **Objects placement** - now
  Et af problemerne som gør koden uhensigtsmæssigt er at flere filer med det samme navn, derudover vil filerne blive placeret i samme mappe som source-koden, for ellers vil det ikke være muligt at compile rigtigt, fordi når den første compiles vil systemet efterfølgende antage at de filer som allerede er compilet ikke behøver at opdateres.

- **Objects placement** - after desired change e.g. new placement
  For at imødekomme ovenstående problem bruges variablen *BUILD_DIR*. Den skifter mellem mapperne afhængigt om der skrives til target eller host, så nu

```
${BUILD_DIR}/%.d:%.cpp
${BUILD_DIR}/%.o:%.cpp
```

Samme princip udnyttes ved EXE-filerne. Det gøres ved at bruge kommandoen *BIN_DIR*, så der i bin oprettes to undermapper, en til target og en til host.

Nedenstående kode evaluerer variablen *ARCH* der gives en værdi ved kald af make (enten ARCH=host eller ARCH=target). Ud fra dette vælges den rigtige compiler, samt den ønskede sti for de genererede filer.

```
ifeq (${ARCH},host)
CXX=g++
BUILD_DIR =./build/host
BIN_DIR = ./bin/host
endif

ifeq (${ARCH},target)
CXX=arm-rpizw-g++
BUILD_DIR =./build/target
BIN_DIR = ./bin/target
endif
```

Efter dette tjekkes om de bestemte mapper findes eller ej, og hvis der er mapper som mangler skal de oprettes, for at kunne gøre dette bruges kommandoen *mkdir -p*:

```
mkdir -p $(BUILD_DIR)
mkdir -p $(BIN_DIR)
```

Kommandoen addprefix, benyttes til at tilføje stien for de pågældene mapper til Depencies, Objects og EXE.

```
OBJECTS=$(addprefix ${BUILD_DIR}/, $(SOURCES:.cpp=.o))
DEPS=$(addprefix ${BUILD_DIR}/,$(SOURCES:.cpp=.d))
```

```
EXE=$(addprefix ${BIN_DIR}/, prog)
```

- **Program file**
  Det eksekverbare program lægges i det directory der findes i variablen *BIN_DIR* Dette kan simpelt udføres ved kommandoen

```
EXE=$(addprefix ${BIN_DIR}/, prog)
```

Samt at sørge for at mappen oprettes i det target der bygger EXE:

```
$(EXE): $(DEPS) $(OBJECTS)
    @# create directories if needed:
    mkdir -p $(BIN_DIR)
    $(CXX) $(CXXFLAGS ) -o $@ $(OBJECTS)
```

Ved kald af make og en gyldig værdi for *ARCH* bygges programmet, hvor objekter og det eksekverbare program lægges i de ønskede mapper - se nedenstående billede:

```
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe2/exe32$ make ARCH=host
mkdir -p ./build/host
g++ -c main.cpp -o build/host/main.o
mkdir -p ./build/host
g++ -c part1.cpp -o build/host/part1.o
mkdir -p ./build/host
g++ -c part2.cpp -o build/host/part2.o
mkdir -p ./bin/host
g++  -o bin/host/prog ./build/host/main.o ./build/host/part1.o ./build/host/part2.o
stud@stud-virtual-machine:~/i3isu_f2020_beany_business/exe2/exe32$ make clean ARCH=host
rm ./build/host/main.o ./build/host/part1.o ./build/host/part2.o ./bin/host/prog
```

Det kan ses at kommandoen sikrer at de ønskede mapper er oprettet, og at objekterne gemmes i disse. Hvis man kalder clean med en gyldig værdi for *ARCH* slettes filerne for enten host eller target.

Exercise 4 Improving code quality...

Exercise 4.1 clang-format

Først blev følgende kode tilføjet til makefilen fra tidligere:

```
format: $(SOURCES:.cpp=.format)
%.format: %.cpp
    @echo "Formatting file '$<'"...
    @clang-format -i $<
    @echo " " > $@
```

Derefter blev 'make format' kommandoen brugt hvilket gav følgende svar i terminalen:

```
stud@stud-virtual-machine:~$ make format
Formatting file 'part1.cpp'...
Formatting file 'part2.cpp'...
Formatting file 'main.cpp'...
```

For at sammenligne før og efter formatering blev der taget screenshots som viste følgende:

Det er forholdsvis små ændringer der bliver foretaget, men når det kommer til formatering kan selv små ændringer gøre en stor forskel. Specielt forståelse og læsning af kode bliver nemmere for både sig selv og andre programmører der evt. skal bruge koden.

Exercise 4.2 clang-tidy

For at vise hvad tidy funktionen gør ændres en funktionskald til stort forbogstav. Derefter bruges 'make tidy' funktionen i terminalen.
Det følgende vises så i outputtet:

```
/home/stud/part1.cpp:4:6: warning: invalid case style for function 'Partone' [readability-identifier-naming]
void Partone() { std::cout << "This is part one!" << std::endl; }
     ^~~~~~
     partone
```

Her forslås altså at rette "Partone" til "partone". Dette er selvfølgelig en fin ændring men det viser også at tidy funktionen ikke gør alt arbejdet da det her ville være mere tydeligt at skrive "partOne".
Så selvom vi kan få hjælp til formatering skal vi stadig selv være opmærksomme.

Exercise 4.3 Makefile QoL

Nu tilføjes nogle ekstra linje kode til makefilen som sikrer at hverken clean, format eller tidy funktionerne generer dependency filer når de bliver brugt.
Dette gør at vi kan bruge funktionerne uden at der spildes unødvendig tid og generes dependency filer på uhensigtsmæssigt tidspunkter.

Følgende kode tilføjes til makefilen:

```
ifneq ($(filter -out clean format tidy ,$(MAKECMDGOALS)) ,)
-include $(DEPS)
endif
```

Exercise 5 Libraries

I denne del af øvelsen skal der arbejdes med biblioteker. I et bibliotek kan der være en masse funktionalitet, dette kan udnyttes så man ikke selv behøver at kode alt fra bunden.
I dette eksempel vil vi f.eks. bruge *ncurses* til at pifte et Hello world program lidt op, så der ikke bare udskrives i terminalen.

Exercise 5.1 Using libraries

Først tilføjes *ncurses* som en include fil i main.cpp.

```
1   #include "part1.h"
2   #include "part2.h"
3   #include <ncurses.h>
4
5   int main()
6   {
7
8     partOne();
9     partTwo();
10
11    return 0;
12  }
```

Derefter linkes der i makefilen vha. *-lncurses*, denne metode gøre brug af -l flaget som bruges til at linke til libraries som vist herunder:

```
SOURCES= part1.cpp part2.cpp main.cpp
DEPENDENCIES=part1.h part2.h
OBJECTS= $(SOURCES:.cpp=.o)
EXE=exe
CXX=g++
all: $(OBJECTS)
    $(CXX) $^ -o $(EXE) -lncurses

%.o: %.cpp %.h
    $(CXX) -c $< -o $@
```

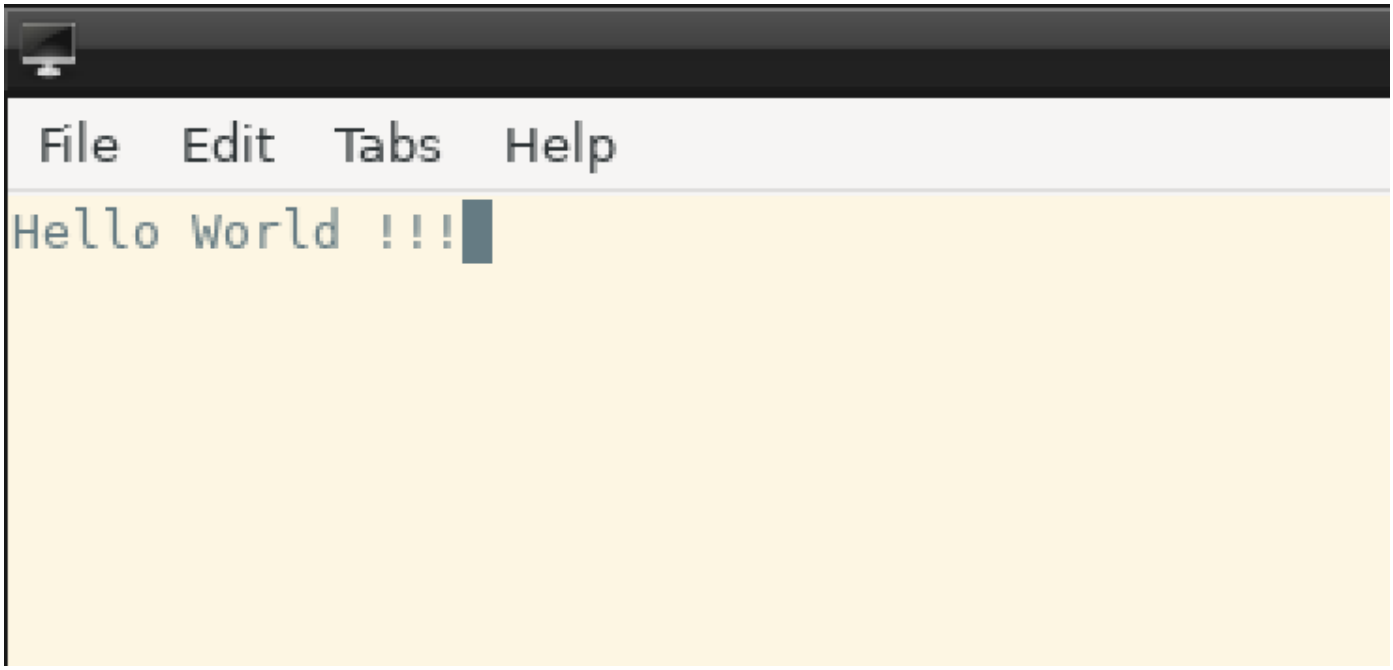
Der findes andre metoder til at linke, man kan bl.a. gøre det mere manuelt, ved at linke vha. filens sti.
Når der er linket vil man kunne se at biblioteket er med når man bruger 'make' kommandoen:

```
stud@stud-virtual-machine:~$ make
g++ -c part1.cpp -o part1.o
g++ -c part2.cpp -o part2.o
g++    -c -o main.o main.cpp
g++ part1.o part2.o main.o -o exe -lncurses
```

Selvom der nu er linket, har vi endnu ikke brugt noget af funktionaliteten fra biblioteket.
Der tilføjes derfor følgende kode fra tldp.org's how to side om *ncurses*:

```
    initscr();            /* Start curses mode           */
    printw("Hello World !!!");    /* Print Hello World          */
    refresh();            /* Print it on to the real screen */
    getch();            /* Wait for user input */
    endwin();            /* End curses mode          */
```

Dette stykke kode initialiserer *ncurses* og printer "Hello World !!!" i et separat vindue fra terminalen som vist her:



Det er en meget lille ændring som kun lige akkurat viser at det kan lade sige gøre at manipulere sit interface. Det er muligt at dykke mere ned mulighederne på dette link.
http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/helloworld.html

- **Tilføjet af Martin Holme Elsborg for 4 måneder siden**

Review

Learning objectives

The first Makefile

I de besvarer alle spørgsmål med en passende uddybelse, samt de får lavet hello world programmet og eksekverer det.

Makefiles - compiling for host

Der er taget kode med fra alle delopgaver, så man nemt kan forstå hvordan de har lavet selve opgaven. Det fungere rigtig godt, og der er desuden blevet besvaret tilstrækkeligt på spørgsmålene i denne opgave. Det eneste der kunne mangle, var måske nogle screenshots inde fra terminalen af, men det andet fungerer også helt klart.

Cross compilation & Makefiles

De formår at forklare spørgsmålene til denne opgave, samt at konfigurer makefilen efter det behov, som den skal have for at kunne fungere optimalt. Derudover har de massere af eksempler med, der viser hele processen for at få makefilen til at virker, og hvordan de compiler.

Improving code quality...

De kan finde ud at at anvende clang-formatet og det viser de med nogle gode eksempler, de konkluderer desuden hvordan man stadig skal passe på selvom man bruger clang-tid, hvilket er en super observation. Måden de viser billederne af koden, er en super måde, da det giver en god forståelse af hvordan de har gjort.

Using Libraries

De viser at de kan finde ud af at bruge et bibliotek, og dermed inkludere det i deres kode, samt udføre koden i terminalen.

Feedback

Opgaven er velskrevet og godt struktureret, da de viser relevante billeder, samt tekst baseret på de understøttende billeder. Alle spørgsmålene fra opgaverne af er velformuleret. Hvis det skulle gøres noget bedre, ville det måske være at viser noget mere fra terminalen.

Must haves

- Relevant files in repository ✓
- Makefiles in repository ✓

Conclusion

Alt I alt er det en rigtig gode opgave, de opfylder alle kravene, og derfor giver vi ”ok”.

terminal png.png (67,3 KB) Jens Nørby Kristensen, 2020-02-03 13:54
2.3makefile.png (41,2 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-11 22:21
2.4solution.png (139 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-11 22:27
clang.png (163 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-14 17:14
format.png (18,8 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-14 17:14
tidyformat.png (24,7 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-14 18:41
include.png (30,6 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-15 14:22
makecurses.png (20,4 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-16 09:17
Curse.png (14,3 KB) Joachim Krøyer Leth-Jørgensen, 2020-02-16 09:33
byebyeKISS.jpg (29,1 KB) Jens Nørby Kristensen, 2020-02-16 13:37