Gruppe 33 - Exercise2 - # 46

Building C/C++ programs for host n target

Læringsmål for øvelsen:

1. Skrive en simpel Makefile
2. Benytte og forstå pattern matching
3. Implementere og håndtere Cross compilation i makefilen
4. Linke et libary til et program

Exercise 1 The first Makefile

Exercise 1.1 The "Hello World" program

Indledning
"In the file hello.cpp write a small "Hello World!" C++ program. Use direct compiler invocation of the compiler g++ to compile your program to an executable hello. Correct any errors you may have, then execute your program."

Design og implementering

I filen hello.cpp skrives følgende program:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

I teminalen skrives følgende for at compile programmet med g++:

g++ -o hello hello.cpp

Dette skaber filen hello, som vil execute ved følgende teminal input:

./hello

Nu udføres programmet og "Hello World!" skrives i terminalen. Resultatet er vist nedenfor:

```
stud@stud-virtual-machine:~/test/ISU/exercis1_1$ g++ -o hello hello.cpp
stud@stud-virtual-machine:~/test/ISU/exercis1_1$ ./hello
Hello World!
```

Exercise 1.2 Makefile basics

Indledning
"Answer the following questions and remember to use code snippets if it serves to pave the way for better understanding."

Nedenfor vises en illustration fra c++_Programming_in_linux.pdf(undervistningsmateriale til ISU lektion 2), for en bedre forståelse og reference.

```
target1: prereq1 prereq2 ...
    command1
    command2
    ...

target2: prereq1 prereq2 ...
    command1
    command2
    ...
```

• What is a target?

Et target er den fil som ønskes at build/rebuild, og illustret syntax som vist ovenfor, hvor target1 er navnet efterfult af:

• What is a dependant and how is it related to a target?

Dependant eller Prerequisites som vist i illustrationen ovenfor er de filer som kræves for at build/rebuild target. Ændres nogle af disse skal target rebuild.

• Does it matter whether one uses tabs or spaces in a makefile?

Recipes er de commands som ønskes udført, på illustrationen command 1 og command2. Inden disse benyttes tabs netop for at markere at dette er recipes.

• How do you define and use a variable in a makefile?

en variabel kunne være SOURCES. Her vil man øverst i makefilen kunne skrive således:

```
SOURCES=main.cpp fil1.cpp fil2.cpp fil.cpp
```

I makefilen vil vi nu kunne benytte ${SOURCES} de steder vi ønsker, istedet for at indtaste hver enkelt filnavn.

• Why use variables in a makefile?

Tilføjes endnu en fil, kan det skrives i SOURCES og vil automatisk medgå de steder ${SOURCES} benyttes, frem for at skulle indsætte filnavnet flere steder i makefilen.

• How do you use a created makefile?
makefilen bruges ved at skrive "make" i terminalen. Eksempelvis:

```
stud@stud-virtual-machine:~/test/ISU/exercise1_2$ make
g++ -c hello.cpp
g++ hello.o -o output
```

I dette eksempel kan filen output executes ved:

```
stud@stud-virtual-machine:~/test/ISU/exercise1_2$ ./output
Hello World!c
```

Man kan vælge specifikke targets i makefilen ved:

```
stud@stud-virtual-machine:~/test/ISU/exercise1_2$ make clean
```

• In the makefile scripting language they often refer to built-in variables such as these
– $@, $< and $^ - explain what each of these represent.

Til besvarelsen af disse benyttes følgende kilde: >https://courses.cs.duke.edu/cps108/doc/makefileinfo/sample.html

$@ - Refererer til target filnavnet
$< - Refererer til den første dependency
$^ - Refererer til listen af dependencies adskilt med mellemrum uden duplicates
Til besvarelsen af disse benyttes følgende kilde:

https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html#Catalogue-of-Rules
– $(CC) and $(CXX):
What do they refer to?

$(CC) - "n.o is made automatically from n.c with a recipe of the form $(CC)"
$(CXX) - "n.o is made automatically from n.cc, n.cpp, or n.C with a recipe of the form" '$(CXX)
Der er altså til om en recipe til at compile c og cpp filer

How do they differ from each other?"

Ønskes der en automatisk compilering af .c filer benyttes $(CC)
Ønskes der en automatisk compilering af .cpp filer benyttes $(CXX)

- $(CFLAGS) and $(CXXFLAGS):
What do they refer to?

$(CFLAGS) referere til tilføjelsen af flags ved compilation af c filer

```
$(CC)  $(CFLAGS)
```

$(CXXFLAGS) referere til tilføjelsen af flags ved compilation af cpp filer

```
$(CXX) $(CPPFLAGS)
```

How do they differ from each other?

Igen er det i forhold til om der arbejdes med .c eller .cpp filer.

• What does $(SOURCES:.cpp=.o) mean? Any spaces in this text???

For variablen SOURCES laves reglen at .cpp = .o
: er en separator char, for at lave en "rule"
.cpp=.o fortæller at for .cpp filerne skal der laves filer lig med, = ny separator char, .o filer

```
SOURCES=file1.cpp file2.cpp
OBJECTS=${SOURCES:.cpp=.o}
```

her bliver variablen OBJECTS= file1.o og file2.o
Det hele skrives uden mellemrum(whitespace)

Exercise 1.3 Writing the makefile

Indledning
"Write a makefile for the program hello you just created"

Design og implementering

makefilen skrives på følgende måde:

```
output: hello.o
    g++ hello.o -o output

hello.o: hello.cpp
    g++ -c hello.cpp

clear:
    rm *.o output
```

"Test and verify that make does what you want it to do"

Make skrives i terminalen og programmet compiles.

```
stud@stud-virtual-machine:~/test/ISU/exercise1_3$ make
g++ -c hello.cpp
g++ hello.o -o output
```

Programmet executes som før ved:

```
stud@stud-virtual-machine:~/test/ISU/exercise1_3$ ./output
Hello World!
```

make clear kan benyttes til at fjerne hello.o og ouput filen således:

```
stud@stud-virtual-machine:~/test/ISU/exercise1_3$ make clear
rm *.o output
```

Exercise 2 Makefiles - compiling for host

Exercise 2.1 Using makefiles - Next steps

Indledning
"Rewrite your first makefile. Add a target all that compiles your program, furthermore use variables to specify the following:
• The name of the executable
• The used compiler."

Design og implementering
makefilen fra tidligere opgave omskrives således:

```
EXECUTABLE=edit
CXX=g++

#http://wiki.maemo.org/Documentation/Maemo_5_Developer_Guide/GNU_Build_System
#phony all example
.PHONY: all
all:${EXECUTABLE}
${EXECUTABLE}: hello.o
    ${CXX} -o $@ $^

hello.o: hello.cpp
    ${CXX} -c hello.cpp

clear:
    rm *.o ${EXECUTABLE}
```

Programmet compiles nu i terminalen med make og filen edit executes som tidligere

```
stud@stud-virtual-machine:~/test/ISU/exercise2_1$ make
g++ -c hello.cpp
g++ -o edit hello.o

stud@stud-virtual-machine:~/test/ISU/exercise2_1$ ./edit
Hello World!
```

Nu udbygges makefilen yderligere ud fra følgende beskrivelse fra opgavesættet:
"Add two targets to your makefile; clean that removes all object files as well as the executable. Add a target help that prints a list of available targets1"

"clear " target omskrives nu til "clean" og "help" implementeres som vist nedenfor:

```
EXECUTABLE=edit
CXX=g++
#http://wiki.maemo.org/Documentation/Maemo_5_Developer_Guide/GNU_Build_System
#phony all example
```

```
.PHONY: all
all:${EXECUTABLE}
${EXECUTABLE}: hello.o
    ${CXX} -o $@ $^

hello.o: hello.cpp
    ${CXX} -c hello.cpp

clean:
    rm *.o ${EXECUTABLE}

help:
    echo all ${EXECUTABLE} hello.o clean $@
```

Til clean, som tidligere, benyttes rm *.o der sletter alle .o filer ved "wildcarded" *
Til help benyttes echo til at udskrive navnene på all targes i makefilen.

De nye implementeringer testes således:

```
stud@stud-virtual-machine:~/test/ISU/exercise2_1$ make
g++ -c hello.cpp
g++ -o edit hello.o

stud@stud-virtual-machine:~/test/ISU/exercise2_1$ make help
echo all edit hello.o clean help
all edit hello.o clean help

stud@stud-virtual-machine:~/test/ISU/exercise2_1$ ./edit
Hello World!

stud@stud-virtual-machine:~/test/ISU/exercise2_1$ make clean
rm *.o edit
```

Exercise 2.2 Program based on multiple files

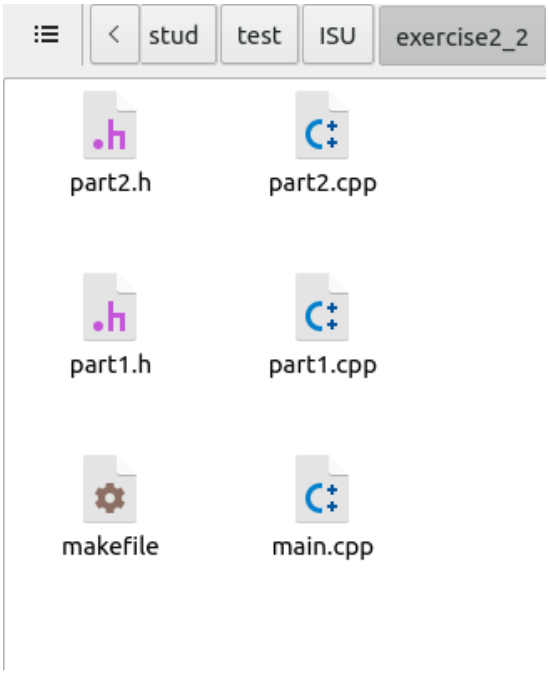
Exercise 2.2.1 Being explicit

Indledning
Create a simple program parts consisting of 5 files:
• part1.cpp
contains 1 simple function part1() that prints "This is part 1!" on stdout
• part1.h
contains the definition of part1()
• part2.cpp
contains 1 simple function part2() that prints "This is part 2!" on stdout
• part2.h
contains the definition of part2()
• main.cpp
contains main() which calls part1() and part2()

> Create a makefile for parts. As in Exercise 2.1 and specify the executable and the used compilerby means of variables. Add targets all, clean and help as in Exercise 2.1."

Design og implementering

Først oprettes de 5 Filer til parts med den ønskede funktionalitet.



Herefter skrives makefilen således:

```
SOURCES=main.cpp part1.cpp part2.cpp
OBJECTS=${SOURCES:.cpp=.o}
EXECUTABLE=edit
CXX=g++

#http://wiki.maemo.org/Documentation/Maemo_5_Developer_Guide/GNU_Build_System
#phony all example
.PHONY: all
all:${EXECUTABLE}
${EXECUTABLE}: ${OBJECTS}
    ${CXX} -o $@ $^
#programming_in_linux.pdf pattern matching rules
%.o: %.cpp
    ${CXX} -c -o $@ $^ ${CXXFLAGS}

clean:
    rm -f ${EXECUTABLE}  *.o

help:
    echo all ${EXECUTABLE} *.o ${SOURCES} clean help
```

Og testes dette i terminalen opnås følgende resultat:

```
stud@stud-virtual-machine:~/test/ISU/exercise2_2$ make
g++    -c -o main.o main.cpp
g++    -c -o part1.o part1.cpp
g++    -c -o part2.o part2.cpp
g++ -o edit main.o part1.o part2.o

stud@stud-virtual-machine:~/test/ISU/exercise2_2$ ./edit
this is part1!
this is part2!

stud@stud-virtual-machine:~/test/ISU/exercise2_2$ make help
echo all edit *.o main.cpp part1.cpp part2.cpp part1.h part2.h  clean help
all edit main.o part1.o part2.o main.cpp part1.cpp part2.cpp  clean help

stud@stud-virtual-machine:~/test/ISU/exercise2_2$ make clean
rm -f edit  *.o
```

```
stud@stud-virtual-machine:~/test/ISU/exercise2_2$ ls
main.cpp  makefile  part1.cpp  part1.h  part2.cpp  part2.h
```

Programmet kan compileres og executes, samt alle .o bliver korrekt fjernet ved clean.

Exercise 2.2.2 Using pattern matching rules

Indledning
"The makefile created in the previous exercise is very explicit and rather large. In this exercise the idea is to use the same but shrink it down and make it less error prone. To achieve this pattern-matching will be employed. It has a special syntax involving the % character for representing wildcards. In other words, one writes a general rule that applies to many situations alleviating the need to write a rule for each and every file.

In this version of the makefile two extra variables are needed:
• Source files
• Object files (acquired from the source file variable - how?)"

${SOURCES} og ${OBJECTS} ipmlementering til makefilen er allerede implementeret i løsningen til Exercise 2.2.1.
pattern matching benyttes ligeledes til at erstatte target .o med %.0: %.cpp som vist her:

```
SOURCES=main.cpp part1.cpp part2.cpp
OBJECTS=${SOURCES:.cpp=.o}
EXECUTABLE=edit
CXX=g++
CXXFLAGS=-ggdb -I.

#http://wiki.maemo.org/Documentation/Maemo_5_Developer_Guide/GNU_Build_System
#phony all example

.PHONY: all
all:${EXECUTABLE}
${EXECUTABLE}: ${OBJECTS}
    ${CXX} -o $@ $^
#programming_in_linux.pdf
%.o: %.cpp
    ${CXX} -c -o $@ $^ ${CXXFLAGS}

clean:
    rm -f ${EXECUTABLE}  *.o

help:
    echo all ${EXECUTABLE} *.o ${SOURCES} clean help
```

Source fileren skrives til variablen SOURCES således:

```
SOURCES=main.cpp part1.cpp part2.cpp part1.h part2.h
```

Object filerne skrive til variablen OBJECTS og gives udfra SOURCE med reglen om at .cpp blot skal ændres til .o således:

```
OBJECTS=${SOURCES:.cpp=.o}
```

Som vist i løsningen til Exercise 2.2.1 benyttes ${SOURCES} og ${OBJECTS} på samme måde som ${EXECUTABLE} og ${CXX} til at forenkle makefilen.
En løsning uden denne implementering vil kræve at all part.cpp og part.o filerne skulle opskrives i stedet for blot ${SOURCES} og ${OBJECTS}. Samtidig hvis der tilføjes nye filer, kan de blot skrives ind i SOURCES og ${SOURCES} og ${OBJECTS} vil automatisk være up to date. På denne måde skal det kun ændres et sted.

Den samlede kildekode er i Repository, exercise2_2

Exercise 2.3 Problem..

Indledning
"The below makefile compiles and produces a working executable (IT DOES). This is obviously assuming that the said files exist and

are adequately sane. In this particular scenario it is assumed that the following files exist (You may need to create some to figure out what happens...):

• server.hpp & server.cpp
• data.hpp & data.cpp
• connection.hpp & connection.cpp"

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 $@ $^
```

Questions to consider:
• How are the source files compiled to object files, what happens?

Følgende kilde benyttes
__ https://www.gnu.org/software/make/manual/make.pdf)
"Compiling C++ programs"

"n.o is made automatically from n.cc, n.cpp, or n.C with a recipe of the form '$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c'. We encourage you to use the suffix '.cc' for C++ source files instead of '.C'"

$(CXX) er en built-in rule, som gør at .o filerne automatisk generes ud fra cpp filerne

• When would you expect make to recompile our executable prog - be specific / precise with respect to file names?

Make forventes at recompile hvis server.cpp data.cpp eller connection.cpp ændres

• Make fails using this particular makefile in that not all dependencies are considered by the chosen approach. Which ones are not?

.cpp filernes dependencies mangler ved denne løsning, eksempelvis header filerne

• Why is this dependency issue a serious problem?

Debugging bliver sværere da der ikke bliver noteret ændring i disse dependencies. Den færdige applikation får derfor måske ikke den ønskede funktionalitet.

Exercise 2.4 Solution

Indledning
"Analyze the listing"

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   $(CXX) -MT$@ -MM $(CXXFLAGS) $< > $@
14   $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $< >> $@
15
16
17 -include $(DEPS)
```

"Describe and verify what it does and how it alleviates our prior dependency problems! In particular what does the command $(CXX) -MT$(:.d=.o) -MM $(CXXFLAGS) $(SOURCES) do See your man files... Hint try to run the command part by hand and examine its output. Note that the option -MT$(:.d=.o) does not do much here, however it is important onward which is why you need to figure out how it works."

"Describe and verify what it does and how it alleviates our prior dependency problems!"

> DEPS, ligesom OBJECTS, benytter alle SOURCES og erstatter .cpp nu med .d
> Samtidig benyttes pattern matching ved %.d som target, med all %.cpp dependencies.

$(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $(SOURCES)

> Der bliver her lavet en rule for target(%.d) ift. .d filerne = .o filer
> på samme måde som reglen for OBJECTS=$(SOURCES:.cpp=.o)

> Problemet fra tidligere opgave er derfor løst.

Exercise 3 Cross compilation Makefiles

Exercise 3.1 First try - KISS

Indledning
"Cross compiling is simple in the sense that g++, in our case, is exchanged with another g++ that supports the desired target architecture, e.g. ARM and thus the compiler arm-rpizw-g++. That being said lets revisit exercise 2.1 in which a simple makefile was made. Start by making a copy of this makefile called makefile.arm and change it such that it uses thearm compiler. Having done that invoke it using make.
Consider:

• Do you have to do something special to invoke this particular makefile?

> Da der er 2 makefiles vælges makefile.arm specifikt således:

```
stud@stud-virtual-machine:~/test/ISU/exercise2_1$ make -f makefile.arm
arm-rpizw-g++ -c hello.cpp
arm-rpizw-g++ -o edit hello.o
```

• 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(Hint: Think object / bin files)?"

> Dette kan løses ved at oprette en mapper til hver makefile og de forskellige bin og .o filer, og på denne måde holde det adskilt og forsimplet.

Exercise 3.2 The full Monty - Bye bye KISS

Indledning
"At this point we want to develop the final makefile that handles all the issues encountered in one go. Take your starting point in the listing 3.1 and finalize the missing parts such that we get a makefile that attains the desired functionality."

Listing 3.1: Handling cross compiling properly

```
SOURCES=main.cpp part1.cpp part2.cpp
OBJECTS=$(SOURCES:.cpp=.o)
DEPS=$(SOURCES:.cpp=.d)
EXE=prog
CXXFLAGS=-I.

ARCH?=x86-64

# Making for x86-64 e.g. x86-64 (the architecture employed)
# > make ARCH=x86-64
ifeq (${ARCH},x86-64)
CXX=g++
BUILD_DIR=build/x86-64
endif

# Making for target
# > make ARCH=target
ifeq (${ARCH},arm)
CXX=arm-rpizw-g++
BUILD_DIR=build/arm
endif


$(EXE): $(DEPS) $(OBJECTS)   # << Check the $(DEPS) new dependency
  $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)

# %.cpp -> %.o needs to be added! Target is NOT just %.o...

# Rule that describes how a .d (dependency) file is created from a .cpp
    file
# Similar to the assigment that you just completed %.cpp -> %.o
$(BUILD_DIR)/$%.d: %.cpp
  $(CXX) -MT$@ -MM $(CXXFLAGS) $< > $@
  $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $< >> $@

ifneq ($(MAKECMDGOALS),clean)
-include $(DEPS)
endif
```

Things to alter:
• Objects placement - now
As it is now, where are the objects placed (main.o etc)? Why is this bad?

Alle.o filer bliver placeret i projekt mappen. Vi ønsker .o filerne opdelt i mappen for hhv. build/x86-64 og build/arm

• Objects placement - after desired change e.g. new placement
Where are they to be placed now (See .d file generation placement in 3.1)?
Explain how this is to be achieved (Hint: Think target name).

vi benytter target BUILD_DIR sådan:

```
$(BUILD_DIR)/%.o : %.cpp
    $(CXX) $(CXXFLAGS) -o $@ -c $<
```

• Program file
Is the current placement the correct one? Hardly; what to do and where to place it?

Executable "prog" ønskes placeret hhv. i bin/arm og bin/x86-64 således:

```
OBJ_Prefix := $(addprefix $(BUILD_DIR)/,$(OBJECTS))

$(BIN_DIR)/$(EXE): $(OBJ_Prefix) $(DEPS)
    $(CXX) $(CXXFLAGS) -o $@ $^
```

hints og suggestions fra øvelsesbeskrivelsen er efterfulgt.
nu benyttes makefile med ARCH=arm og efterfølgende uden:

```
stud@stud-virtual-machine:~/test/ISU/exercise3_1$ make ARCH=arm
mkdir -p build/arm
mkdir -p bin/arm
arm-rpizw-g++ -I. -o build/arm/main.o -c main.cpp
arm-rpizw-g++ -I. -o build/arm/part1.o -c part1.cpp
arm-rpizw-g++ -I. -o build/arm/part2.o -c part2.cpp
arm-rpizw-g++ -I. -o bin/arm/prog build/arm/main.o build/arm/part1.o build/arm/part2.o
stud@stud-virtual-machine:~/test/ISU/exercise3_1$ make
mkdir -p build/x86-64
mkdir -p bin/x86-64
g++ -I. -o build/x86-64/main.o -c main.cpp
g++ -I. -o build/x86-64/part1.o -c part1.cpp
g++ -I. -o build/x86-64/part2.o -c part2.cpp
g++ -I. -o bin/x86-64/prog build/x86-64/main.o build/x86-64/part1.o build/x86-64/part2.o
stud@stud-virtual-machine:~/test/ISU/exercise3_1$
```

Den samlede kildekode er i Repository, exercise3_1

Exercise 4 Improving code quality...

Exercise 4.1 clang-format

Indledning
"clang-format simply formats your code, such it conforms to some predetermined layout. IT does not change casing or any other aspect of your code that would imply that your code has actually changed. E.g. needs recompilation."
Følgende tilføjes til makefile:

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

Og køres det i terminalen sker følgende:

```
stud@stud-virtual-machine:~/test/ISU/exercise3_1$ make format
Formatting file 'main.cpp'
Formatting file 'part1.cpp'
Formatting file 'part2.cpp'
```

Det antages nu at de gældende filer er formateret udfra clang-format config filen.

Den samlede kildekode er i Repository, exercise3_1

Exercise 4.1 clang-tidy

Indledning
"clang-tidy checks whether your code follows certain standards as well as whether you employ some bad practices. IT does not change your code (how we use it) but rather output errors when your code does not conform or uses bad practises."
Følgende tilføjes til makefile:

```
tidy: ${SOURCES:.cpp=.tidy}
%.tidy: %.cpp
    @echo "Tidying file '$<'"
    @clang-tidy $< -- $(CXXFLAGS)
    @echo "" > $@
```

Og køres det i terminalen sker følgende:

```
stud@stud-virtual-machine:~/test/ISU/exercise3_1$ make tidy
Tidying file 'main.cpp'
Tidying file 'part1.cpp'
Tidying file 'part2.cpp'
```

Det antages nu at de gældende filer er formateret udfra clang-tidy config filen.

Exercise 4.1 Makefile QoL

Følgende linje i makefilen udskiftes:

```
ifneq ($(MAKECMDGOALS),clean)
```

Og erstattes med:

```
ifneq ($(MAKECMDGOALS),clean)
```

Den samlede kildekode er i Repository, exercise3_1

Exercise 5 Libraries

Exercise 5.1 Using libraries

Pick out one of your already created makefiles and modify it such that you may link and afterwards run the program.

• How do you link a library to a program?

Fra NCURSES https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/
"To link the program with ncurses the flag -lncurses should be added."

Og i makefile bliver det:

```
${EXECUTABLE}: ${OBJECTS}
    ${CXX} -o $@ $^ -lncurses
```

Tilføjes dette ikke i vores makefile er resultatet følgende:

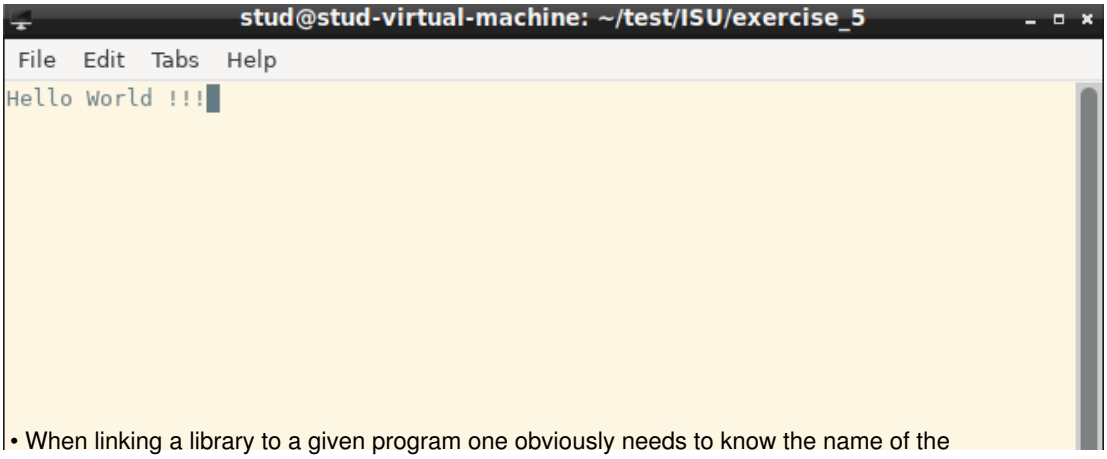
```
hello.cpp:(.text+0x9): undefined reference to `initscr'
/usr/bin/ld: hello.cpp:(.text+0x1a): undefined reference to `printw'
/usr/bin/ld: hello.cpp:(.text+0x1f): undefined reference to `refresh'
/usr/bin/ld: hello.cpp:(.text+0x26): undefined reference to `stdscr'
/usr/bin/ld: hello.cpp:(.text+0x2e): undefined reference to `wgetch'
/usr/bin/ld: hello.cpp:(.text+0x33): undefined reference to `endwin'
```

Istedet hvis -lncurses er tilføjet korrekt i makefile:

```
stud@stud-virtual-machine:~/test/ISU/exercise_5$ make
g++ -o edit hello.o -lncurses
```

Og programmet kan nu køres:

```
stud@stud-virtual-machine:~/test/ISU/exercise_5$ ./edit
```



• When linking a library to a given program one obviously needs to know the name of the file. However, one thing is the actual file name another is how it is denoted when linking... Whats the difference?

Fra https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

> "-llibrary
> -l library
> Search the library named library when linking."

Så for at linke et givent biblotek tilføjes -l <libary name>

> "The -l option is passed directly to the linker by GCC. Refer to your linker documentation for exact details. The general description below applies to the GNU linker.
> The linker searches a standard list of directories for the library. The directories searched include several standard system directories plus any that you specify with -L."

Med -l<libary name> fortælles hvilket libary linkeren skal søge efter. Denne søgning fortages så i ene rækkes standatd system directories.

Filer

1_1.png	27,2 KB	11.09.2021	Rasmus Baunvig Aagaard
2_2.png	11,8 KB	11.09.2021	Rasmus Baunvig Aagaard
2_3.PNG	13 KB	11.09.2021	Rasmus Baunvig Aagaard
2_4.PNG	45 KB	11.09.2021	Rasmus Baunvig Aagaard
1_2.PNG	4,31 KB	11.09.2021	Rasmus Baunvig Aagaard
3_1.PNG	82,1 KB	11.09.2021	Rasmus Baunvig Aagaard
5.PNG	10,3 KB	17.09.2021	Rasmus Baunvig Aagaard