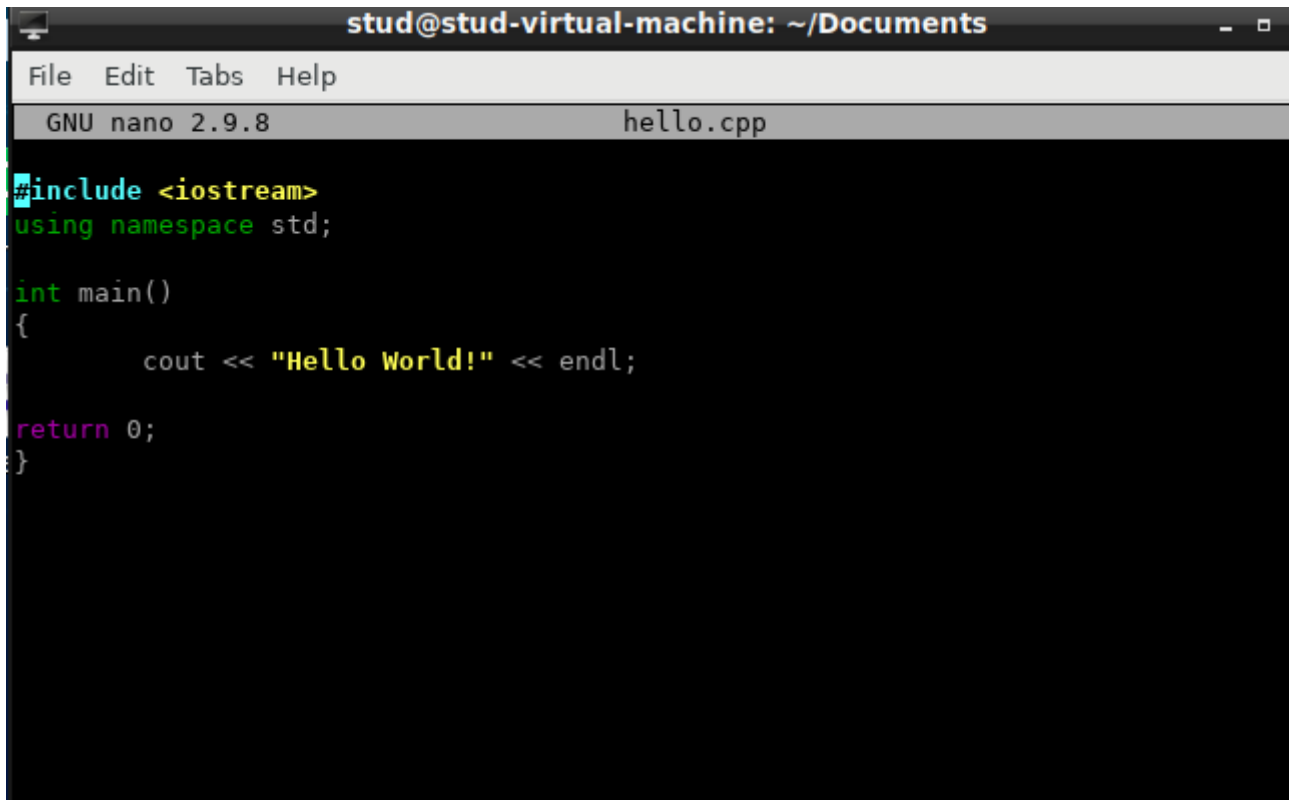


[Wiki »](#)

Exercise 1 The First Makefile

Exercise 1.1 The "Hello World" program

Herunder ser vi et billede af Hello World programmet, samt et billede at at det eksekveres:

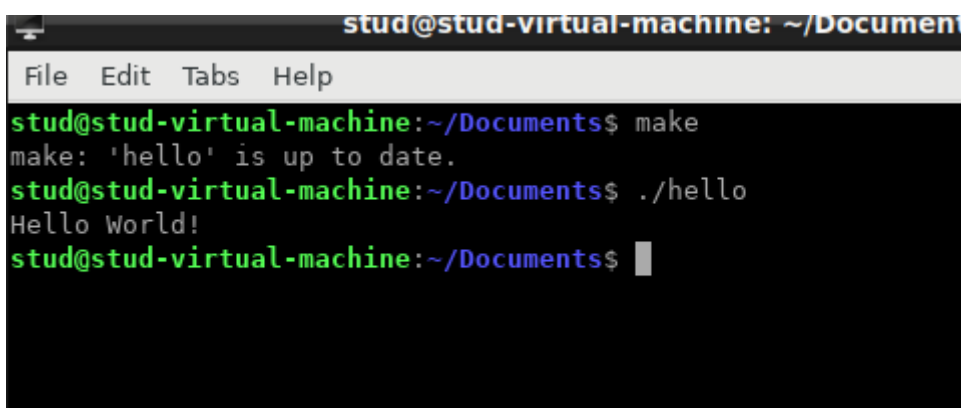


```
stud@stud-virtual-machine: ~/Documents
File Edit Tabs Help
GNU nano 2.9.8 hello.cpp

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!" << endl;

return 0;
}
```



```
stud@stud-virtual-machine: ~/Documents
File Edit Tabs Help

stud@stud-virtual-machine:~/Documents$ make
make: 'hello' is up to date.
stud@stud-virtual-machine:~/Documents$ ./hello
Hello World!
stud@stud-virtual-machine:~/Documents$
```

Exercise 1.2: Makefile basics

- **What is a target?**

Target er navnet på den fil der er genereret af et program fx. executables eller object files.

- **What is a dependant and how is it related to a target?**

En fil der bruges som input til target. Target kan være afhængig af flere filer.

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

JÅ - TABS skal bruges til eksekverbar kode ved begyndelsen af hver recipe linje.

- **How do you define and use a variable in a makefile?**

Definition: EXECUTABLE = hello Use: \$(EXECUTABLE)

- **Why use variables in a makefile?**

Det gør Makefiles simple. For eksempel kan man i stedet for at kalde en lang liste af filer, som man skal indtaste manuelt (og måske taste forkert), lave en variabel med alle disse filer, som man kan kalde på én gang. Eller hvis du vil skifte compiler, kan du have en variabel til compileren, så du så bare skal ændre variablen, i stedet for en masse kode.

- **How do you use a created makefile?**

make (kan også specificeres: Make "navn")

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

\$@: navn på target / fil. \$<: første prerequisite (normalt source filen). \$^: alle prerequisites incl. navne på directories de findes i.

\$(CC) and \$(CXX):

- **What do they refer to?**

Compilerne

- **How do they differ from each other?**

\$(CC) er variabel der bruges til C compiler. \$(CXX) er en variabel der bruges til C++ compiler.

\$(CFLAGS) and \$(CXXFLAGS):

- **What do they refer to?**

Specificere flag til compilation

- **How do they differ from each other?**

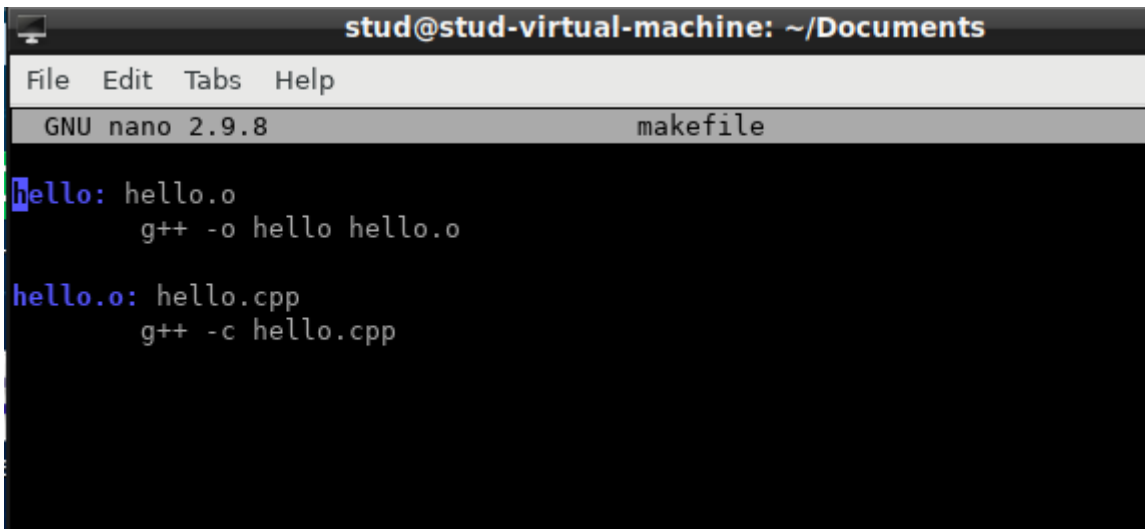
CFLAGS er til at specificere flag til C compilation.

CXXFLAGS er til at specificere flag til C++ compilation.

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

Den refererer til variablen SOURCES, og retter .cpp til en reference til .o. Der skal ikke være mellemrum.

Exercise 1.3 Writing the makefile



```

stud@stud-virtual-machine: ~/Documents
File Edit Tabs Help
GNU nano 2.9.8 makefile

hello: hello.o
    g++ -o hello hello.o

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

Excercise 2: Makefiles - compiling for host

Exercise 2.1 Using makefiles - Next steps

Herunder kan koden til vores makefile ses, hvor vi har tilføjet 'all' som target, en variabel til executable og compiler, samt help og clean:

```

SOURCES=hello.cpp
OBJECTS=$(SOURCES:.cpp=.o)
CXX=g++
EXECUTABLE=hello
CXXFLAGS= -ggdb -I.
  
```

```

all: (OBJECTS)

$(EXECUTABLE): $(OBJECTS)
    $(CXX) -o $(EXECUTABLE) $^

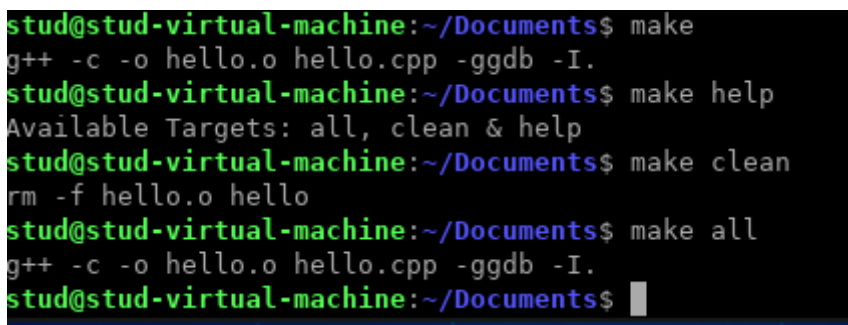
%.o: %.cpp
    $(CXX) -c -o $@ $^ $(CXXFLAGS)

clean:
    rm -f $(OBJECTS) $(EXECUTABLE)

help:
    @echo "Available Targets: all, clean & help"

```

Herunder kan det ses at vi kan kalde make, clean og help i terminalen:



```

stud@stud-virtual-machine:~/Documents$ make
g++ -c -o hello.o hello.cpp -ggdb -I.
stud@stud-virtual-machine:~/Documents$ make help
Available Targets: all, clean & help
stud@stud-virtual-machine:~/Documents$ make clean
rm -f hello.o hello
stud@stud-virtual-machine:~/Documents$ make all
g++ -c -o hello.o hello.cpp -ggdb -I.
stud@stud-virtual-machine:~/Documents$

```

Exercise 2.2 Program based on multiple files

Exercise 2.2.1 Being explicit

Her laver vi en simpel makefile til vores parts:

```

CXX=g++
EXECUTABLE=parts
SOURCES=part1.cpp part2.cpp main.cpp

all: $(EXECUTABLE)

$(EXECUTABLE): main.o part1.o part2.o
    $(CXX) -o $(EXECUTABLE) main.o part1.o part2.o

main.o: main.cpp
    $(CXX) -c main.cpp

part1.o: part1.cpp part1.h
    $(CXX) -c part1.cpp

part2.o: part2.cpp part2.h
    $(CXX) -c part2.cpp

.PHONY: clean
clean:
    rm $(EXECUTABLE) part1.o part2.o main.o

```

```
help:
    @echo all, parts, clean and help
```

Her kaldes makefile kommandoerne i terminalen:

```
stud@stud-virtual-machine:~/Documents/Excercisel$ make all
g++ -c main.cpp
g++ -c part1.cpp
g++ -c part2.cpp
g++ -o parts main.o part1.o part2.o
stud@stud-virtual-machine:~/Documents/Excercisel$ make clean
rm parts part1.o part2.o main.o
stud@stud-virtual-machine:~/Documents/Excercisel$ make help
all, parts, clean and help
stud@stud-virtual-machine:~/Documents/Excercisel$
```

Exercise 2.2.2 Using pattern matching rules

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

all: ${EXECUTABLE}

%.o: %.cpp %
    ${CXX} -c -o $@ $^ ${CXXFLAGS}

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

.PHONY: clean
clean:
    rm ${EXECUTABLE} ${OBJECTS}

help:
    @echo "Available Targets: all, clean, ${OBJECTS}, ${EXECUTABLE}\n"
```

What makes this an improved solution as opposed the previous one?

Det er hurtigere og simplere at tilføje nye filer, da man skal kun skal tilføje/ændre et enkelt sted i koden (sources).

Exercise 2.3 Problem...

- **How are the source files compiled to object files, what happens?**

n.o filer laves automatisk fra n.cpp filer med g++ som default compiler.

- **When would you expect make to recompile our executable prog - be specific?**

Hvis en af makefilens dependencies (.cpp) ændres, Linkeren laver et dependency tree til projektet, så den ved om der er flere filer der skal compiles.

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

Header filerne håndteres ikke i programmets dependency list.

- **Why is this dependency issue a serious problem**

Det vil betyde at ændringer i header filer ikke medtages ved en re-make.

Exercise 2.4 Solution

Analyze the listing 2.2:

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

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

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

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

-include $(DEPS)
```

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

Her er makefilen ændret, så når der laves .d filer (part1.d) med target part1.o, som har dependencies part1.cpp og part1.h. Den finder altså ud af hvad sources er afhængige af, så .hpp filerne også tjekket for opdateringer.

In particular what does the command `$(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $(SOURCES)` do?

Den laver .d filer om til .o filer, og finder ud af hvilke dependencies .o filerne har.

Exercise 3 Cross compilation & Makefiles

Exercise 3.1 First try - KISS

Consider:

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

```
make -f makefile.target
```

- 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?

Vi er nødt til at cleane, når vi går fra den ene til den anden.

```
stud@stud-virtual-machine:~/i3isu_f2019/lecture2/exercise2.1$ make
g++ -c hello.cpp
g++ -o hello hello.o
stud@stud-virtual-machine:~/i3isu_f2019/lecture2/exercise2.1$ make -f makefile.target
make: Nothing to be done for 'all'.
stud@stud-virtual-machine:~/i3isu_f2019/lecture2/exercise2.1$ make clean
rm hello hello.o
stud@stud-virtual-machine:~/i3isu_f2019/lecture2/exercise2.1$ make -f makefile.target
arm-rpizw-g++ -c hello.cpp
arm-rpizw-g++ -o hello hello.o
stud@stud-virtual-machine:~/i3isu_f2019/lecture2/exercise2.1$
```

3.2 The full Monty - Bye bye KISS

Things to alter:

- Objects placement - now

As it is now, where are the objects placed? Why is this bad?

De placeres samme sted som de eksekverbare filer. Det er ikke smart, da det kan gøre det uoverskueligt at finde de korrekte filer i dette directory.

- Objects placement - after change

Where are they to be placed now?

Explain how this is achieved.

Efter at have implementeret build/target og build/host sorteres objecterne i disse directories.

- Program file

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

- Place all generated object files in build/target or build/host (this is what has already begun in the above makefile listing) respectively.
- Place the executable in bin/target or bin/host respectively.

```
stud@stud-virtual-machine:~/Documents/Excercisel$ make ARCH=host
mkdir -p ./build/host
g++ -MTbuild/host/main.o -MM -I. main.cpp > build/host/main.d
mkdir -p ./build/host
g++ -MTbuild/host/part2.o -MM -I. part2.cpp > build/host/part2.d
mkdir -p ./build/host
g++ -MTbuild/host/part1.o -MM -I. part1.cpp > build/host/part1.d
mkdir -p ./build/host
g++ -c part1.cpp -o build/host/part1.o -I.
mkdir -p ./build/host
g++ -c part2.cpp -o build/host/part2.o -I.
mkdir -p ./build/host
g++ -c main.cpp -o build/host/main.o -I.
mkdir -p ./bin/host
g++ -I. -o bin/host/prog ./build/host/part1.o ./build/host/part2.o ./build/host/main.o
stud@stud-virtual-machine:~/Documents/Excercisel$
```

Herunder ses den færdige kode:

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

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
```

```
$(BIN_DIR)/$(EXE): $(DEPS) $(OBJECTS)
    mkdir -p $(BIN_DIR)
    $(CXX) $(CXXFLAGS) -o $@ $(OBJECTS)

$(BUILD_DIR)/%.d: %.cpp
    mkdir -p $(BUILD_DIR)
    $(CXX) -MT$(@:.d=.o) -MM $(CXXFLAGS) $^ > $@

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

clean:
    -rm -rf bin/
    -rm -rf build/

help:
    @echo "$(SOURCES)"

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

4.1 Using libraries

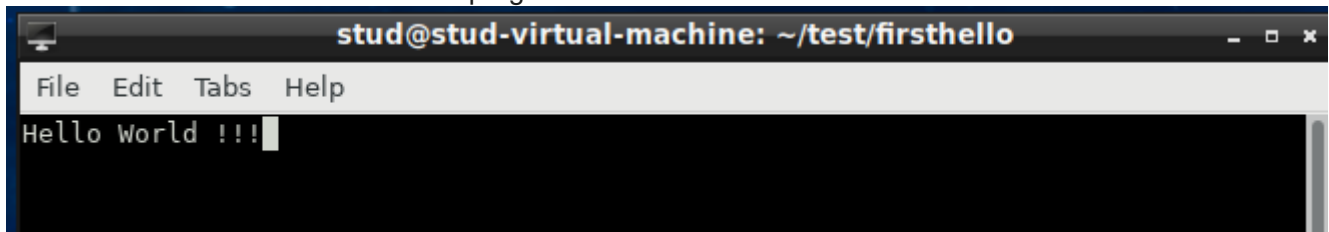
How do you link a library to a program?

Man linker library'et ncurses til et program ved at tilføje `-lncurses` i sin makefile, som set på koden nedenfor.

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

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

Herunder ses et billede af helloworld-programmet med ncurses.



When linking a library to a given program one obviously need to know the name of the file. However, is the name of the file as found on the disk exactly the same characterwise as when supplying it to gcc?

- Tilføjet af Glenn Laursen for cirka en måned siden

Der vises god forståelse for makefiles hele vejen igennem exercisen. og alle spørgsmål ser ud til at være tilfredsstillende besvaret. Flot!

[call.PNG](#) (16,9 KB) Mikkel Mahler, 2019-02-13 08:38

[helloworld.PNG](#) (16,9 KB) Mikkel Mahler, 2019-02-13 08:38

[makefile.PNG](#) (13,1 KB) Mikkel Mahler, 2019-02-13 08:48

[makefile2.0.PNG](#) (17,7 KB) Mikkel Mahler, 2019-02-13 08:48

[2.1.PNG](#) (20,3 KB) Mikkel Mahler, 2019-02-13 10:29

[2.2.1.PNG](#) (19,9 KB) Mikkel Mahler, 2019-02-13 11:42

[Opg2.4.PNG](#) (101 KB) Mikkel Mahler, 2019-02-18 13:18

[makefile_target.png](#) (28 KB) Andreas Ventzel, 2019-02-18 14:35

[Opg3.2_make.PNG](#) (30,7 KB) Mikkel Mahler, 2019-02-18 16:10

[helloworld_ncurses.PNG](#) (10,9 KB) Jonas Jesper Tække, 2019-02-19 13:44