Wiki »

Exercise2: Building C/C++ programs for host n target ¶

Exercise 1: The first Makefile

Exercise 1.1: The "hello world" program

```
#include <iostream>

using namespace std;

int main(){

cout << "hello" << endl;
}
```

hello.cpp compileres ved at skrive:

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

i terminalen

Exercise 1.2: Makefile basics

What is a target?

- Et target er det som står til venstre for kolon i en recipe. Det er enten navnet på en fil som bliver genereret, eller en handling som skal udføres

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

- En dependant er det som står til højre for kolon i en recipe. Det er en fil der bliver brugt som input til at lave target. Når der bliver ændret i de filer som er sat dependant til et target, vil target blive kompileret.

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

- JA! Recipes virker kun, hvis handlingen blivver rykket ind med et tab.

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

- En variabel defineres ved at skrive et navn og sætte den lig med noget. Eks: MitNavn = main.cpp
- Herefter kan variablen kaldes: $(variabelnavn)

Why use variables in a makefile?

- Der skal skrives mindre og der er mindre sandsynlighed for fejl. Man skal kun ændre et sted og så ændrer variablen sig over det hele

How do you use a created makefile?

- Man skriver 'make' i shellen inde i det directory hvor makefilen er placeret

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

- $@ refererer til targetnavn. $^ refererer til dependencies. $< refererer til den første dependant.
- $(CC) refererer til C-compileren. $(CXX) refererer til C++-compileren
- $(CFLAGS) refererer til ekstra flag, som tildeles til C-compileren. $(CXXFLAGS) refererer til ekstra flag, som tildeles C++-compileren

What does $(SOURCES:.cpp=.o) mean) any spaces in the text??

- Det betyder, at alle steder hvor der står .cpp i variablen SOURCES bliver det lavet om til .o. Der skal ikke være nogle mellemrum i teksten.

Exercise 1.3: Writing the makefile

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

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

.PHONY: clean

clean:
     rm *.o output
```

Exercise 2: Makefiles - compiling for host

Exercise 2.1: Using makefiles - next steps

```
EXE=output
CXX=g++

all: hello.cpp
    $(CXX) -o $(EXE) $<

clean:
    rm $(EXE) *.o

help:
    @echo "Available targets: clean, all\n"
```

Exercise 2.2: Program based on multiple filed

Exercise 2.2.1: Being explicit

```
CXX=g++
EXE=output

all: main.cpp part1.h part1.cpp part2.h part2.cpp
```

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

.PHONY: clean

clean:
	rm *.o output

help: main.cpp part1.h part1.cpp part2.h part2.cpp
	echo "Available targets: all, help"
```

Exercise 2.2.2: using pattern matching rules

```
SOURCES= main.cpp part1.cpp part2.cpp
OBJECTS=$(SOURCES:.cpp=.o)
CXX=g++
EXE=output
CXXFLAGS=-ggdb -I.

all: main.o part1.o part2.o
	$(CXX) -o $(EXE) $^ $(CXXFLAGS)

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

$(EXE): $(OBJECTS)
	$(CXX) -o $@ $^

.PHONY: clean

clean:
	rm *.o output

help: main.cpp part1.cpp part1.h part2.cpp part2.h
	echo "Available targets: all, clean"
```

Pattern matching er en god løsning idet man kun skal ændre navnene på cpp filerne et sted (i variablen), så der er mindre sandsynlighed for at brugeren laver fejl (skriv mindre=færre fejl)

Exercise 2.3: Problem...

> 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`'.

How are the source files compiled to object files? what happens?

- Reglen over siger, at .o filer bliver lavet automatisk sålænge .cpp filerne eksisterer, når man laver en recipe på formen som beskrevet over.

when would you expect 'make' to recompile our executable prog - be specific?

- Når der bliver lavet ændringer i en af filerne og 'make' kommandoen bliver kørt i terminalen.

make fails using this particular makefile in that not all dependencies are handled by the chosen approach Whic ones are not?

- Det er .hpp filerne der ikke bliver håndteret, når 'make' bliver kørt. Det er ikke specificeret i reglen, at .hpp filerne bliver recompileret, når der er ændringer i disse filer.

why is this dependency issue a serious problem?

- Hvis .hpp filerne bliver ændret siger makefilen, at executable filen er "up-to-date"

Exercise 2.4: Solution

Describe and verify what it (the makefile snippet in the exercise) does and how it alleviates our prior dependency problems!

- Når der laves en .d fil, finder den ud af hvad sources er afhængig af og derved bliver .hpp filerne også tjekket for opdateringer.

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

- Kommandoen laver .d filer om til .o filer og finder ud af hvilke filer de specifikke .o filer er afhængige af.

Exercise 3: Cross compilation and makefiles

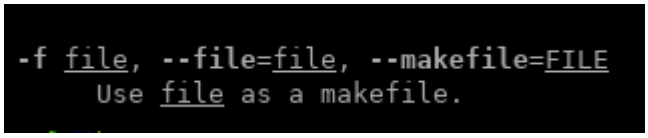
Exercise 3.1: First try - KISS

Do you have to do something special to invoke this particular makefile?
At this point we have particular makefiles in the same directory. How are you forced to handle it?

- I denne opgave havde vi flere makefiles i samme directory og det skulle derfor specificeres, hvilken makefile vi ville bruge til at bygge, da den ellers søger efter default makefile navne. Det gøres i terminalen ved, at skrive:

  ```
  make -f "filnavn"
  ```

  Hvor i vores tilfælde bygger vi makefile.target filen ved:



Ud over det kunne vi ændre navnet på den eksekverbare fil så den hedder noget andet når den compileres til RPIen, fx outputRPI

Exercise 3.2: The full monty - bye bye KISS

Objects placement - now. As it is now, where are the objects placed? Why is this bad?

- De ligger i de samme mapper som den eksekverbare fil som default. Det gør den rodet og svært at finde den eksekverbare fil

Objects placement - after change. Where are they to be placed now? Explain how this is achieved

- Alt efter hvilken ARCH der vælges, ændres compileren og filerne lægges i build/host eller build/target

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

- Den eksekverbare fil bliver lagt i samme directory som de andre filer, hvilket er forkert. Den eksekverbare fil bliver nu lagt i et seperat directory: bin/host eller bin/target alt efter hvilken ARCH. Dette gøres ved, at specificere et directory for target.

Makefile og de forskellige cpp filer kan findes i repository

Exercise 4: Libraries

Exercise 4.1 using libraries

Når der skal laves en makefile, som linker til ncurses librariet, skrives der -lncurses efter filnavnet:

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

clean:
	rm hello.o hello
```

How do you link a library to a program?

- Når et library er inkluderet i en .c fil kan den linkes i terminalen under kompilering med kommandoen (for librariet ncurses):

```
gcc hello.c -lncurses
```

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

- nej, man finder navnet med følgende kommando i terminalen (for ncurses):

```
stud@stud-virtual-machine:~/Desktop/ISU/lektion2/exercise2/2_4$ find /usr/lib -name "libncur*"
/usr/lib/x86_64-linux-gnu/libncursesw.a
/usr/lib/x86_64-linux-gnu/libncurses++.a
/usr/lib/x86_64-linux-gnu/vlc/plugins/gui/libncurses_plugin.so
/usr/lib/x86_64-linux-gnu/libncursesw.so
/usr/lib/x86_64-linux-gnu/libncurses.a
/usr/lib/x86_64-linux-gnu/libncurses.so
/usr/lib/x86_64-linux-gnu/libncurses++w.a
stud@stud-virtual-machine:~/Desktop/ISU/lektion2/exercise2/2_4$
```

- **Tilføjet af Henrik Thue Jensen for cirka en måned siden**

Learning objectives

Learning objective #1: Lave en Makefile og kompilere til host.

Gruppen viser, at de har forstået de grundlæggende termer inden for Makefiles ved at besvare spørgsmålene i Exercise 1.2 fyldestgørende. De viser, de er i stand kompilere deres cpp-fil via g++ compileren inde i Makefilen, hvor de laver deres cpp-fil om til en o-fil, som compueren er i stand til at læse, og derefter lave om til en eksekverbar fil, de kalder EXE.

Learning objective #2: Anvende pattern matching.

Gruppen anvender pattern matching til at gøre deres kode lettere at rette i. Pattern matching anvendes således, at de definerer variable i toppen af deres make-fil og definerer variablen SOURCE til at indeholde samtlige .cpp filer. OBJECTS variablen indeholder så alle .o filer som oprettes af .cpp filer.
Gruppen kunne have anvendt variablen OBJECT i stedet for main.o part1.o og part2.o som prerequisites til target ”all”, i koden ved opgave 2.2.2.

Gruppen clean target rydder fint op i .o filer og exe-filen ”output”, men de kunne have brugt deres variabler OBJECTS og EXE.

Learning objective #3: Strukturere filer i relevante mapper.

Gruppen har svaret klart og fyldestgørende på de i opgaven stillede spørgsmål. Det er tydeligt, at gruppen har fået god forståelse for, at strukturere filerne når de kører deres Makefile. Der kunne eventuelt have været inkluderet et screenshot, som viste, at filerne blev lagt i korrekte mapper. Det er godt, at gruppen har anvendt korte og konkrete svar, dette gør informationssøgning i wikien lettere.

Learning objective #4: Inkludere Libraries

Gruppen har besvaret spørgsmålene korrekt og har tydeligvis lært at inkludere libraries i deres filer. Det sidste screenshot med mapperne /usr/lib/x86_64-linux-gnu, viser, at de har forståelse for, hvor filerne ligger og hvordan de inkluderes.

Feedback

Journalen er godt lavet. Det er godt med korte og konkrete svar på spørgsmålene, samt relevante kodeudsnit. Der kunne med fordel tilføjes ekstra kommentarer til de enkelte kodeudsnit og man kunne overveje om de i opgaven stillede spørgsmål skulle markeres med fed eller lignende, så de skiller sig ud fra besvarelserne. Generelt en rigtigt god besvarelse.

Must haves

- Relevant files in repository - Done!
- Makefiles in repository - Done!
- Any special requirements stated in the particular exercise! - Ikke relevant.

Conclusion

Læringsmål, samt relevante krav er opfyldt, derfor er opgaven vurderet, som ok.

lib.png (26,9 KB) Emil Christian Foged Klemmensen, 2019-02-19 13:10

makefile.png (3,71 KB) Emil Christian Foged Klemmensen, 2019-02-19 13:11