# Makefiles erstellen: make + cmake

Eduard Kuhn

November 15, 2019

# Compiler in der Kommandozeile

Gebräuchliche Compiler unter Linux

| | |
|---|---|
| C: | gcc, clang |
| C++: | g++, clang++ |
| Fortran: | gfortran, ifort |

Typische Compileroptionen

| | |
|---|---|
| Objektdateien erzeugen | -c |
| Name der Ausgabedatei | -o dateiname |
| Warnungen ausgeben | -Wall -Wextra |
| Optimierungen | -O0,-O1,-O2,-O3,-Ofast |
| Debuginformationen | -g |
| Sprachstandard | -std=c++17, -std=c11, -std=f2018 |
| Prozessortyp | -march=native |

Mehr zum Beispiel unter https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

# Beispiel

main.c

```c
#include "helper.h"
int main() { hello(); return 0; }
```

helper.h

```c
void hello();
```

helper.c

```c
#include <stdio.h>
#include "helper.h"
void hello() { printf("Hello World!\n"); }
```

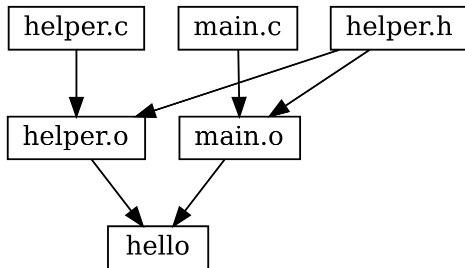
Möglichkeit 1

```
gcc -O2 -c main.c
gcc -O2 -c helper.c
gcc -O2 main.o helper.o -o hello
```

Möglichkeit 2

```
gcc -O2 main.c helper.c -o hello
```

# Abhängigkeitsbaum




```
Makefile
    hello: main.o helper.o
      gcc main.o helper.o -o hello
    main.o: main.c helper.h
      gcc -c main.c
    helper.o: helper.c helper.h
      gcc -c helper.c
```

# Makefiles

```
Makefile
    hello: main.o helper.o
      gcc main.o helper.o -o hello
    main.o: main.c helper.h
      gcc -c main.c
    helper.o: helper.c helper.h
      gcc -c helper.c
```

- Makefile besteht aus Regeln der Form

```
Ziel: Voraussetzung1 ... VoraussetzungN
[TAB]Befehl welcher Ziel erzeugt
```

- Hierbei bezeichnen `Ziel`, `Voraussetzung1` etc. Dateien

# Makefiles

- Der Befehl `make Ziel` versucht die Datei `Ziel` zu erstellen, wird kein Ziel angegeben wird automatisch das erste in der Makefile benutzt
- Dateinamen für Makefile sind `makefile`, `Makefile`, `GNUmakefile`
- `make -j N` erlaubt `N` Jobs gleichzeitig (eventuell große Zeitersparniss bei größeren Projekten)

# Variablen

- Man kann Variablen in Makefiles definieren:

```
CC       = gcc
CC_OPTS = -O2 -Wall -Wextra -std=c11
HEADERS = header1.h header2.h
```

- Und dann benutzen:

```
main.o: main.c $(HEADERS)
  $(CC) $(CC_OPTS) -c main.c
```

# Automatische Variablen

- `$@` Name das Ziels
- `$^` Name aller Voraussetzungen, getrennt durch Leerzeichen
- `$<` ist die erste Voraussetzung

Beispiele:

```
hello: main.o helper.o
  $(CC) $(CC_OPTS) $^ -o $@
main.o: main.c $(HEADERS)
  $(CC) $(CC_OPTS) -c $<
```

# Allgemeine Regeln

- Man kann Regeln die einem Muster folgen mit einer allgemeinen Regel zusammenfassen:

  ```
  %.o: %.c $(HEADERS)
  	$(CC) $(CC_OPTS) -c $<
  ```

- Es gibt viele Regeln, die implizit in make enthalten sind, siehe `https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html`

# Phony Targets

- Manchmal will man make nur benutzen um ein paar Befehle auszuführen, hier löscht `make clean` die auszuführbare Datei und alle Objektdateien:

```
clean:
  rm -f hello
  rm -f *.o
```

- Hier erzeugt der Befehl aber keine Datei mit Namen `clean`
- Existiert aber eine Datei mit Namen `clean` funktioniert `make clean` nicht mehr
- Um das das zu beheben müssen solche unechten Regeln mit `.PHONY` gekennzeichnet werden:

```
.PHONY: clean
```

# Externe Bibliotheken

Einige relevante Compileroptionen

| | |
|---|---|
| Linkt externe Bibliothek `libtest.so` | `-ltest` |
| Sucht nach Headern im Pfad | `-I/usr/local/include` |
| Sucht nach Bibliotheken im Pfad | `-L/usr/local/lib` |

Beispiel dass fftw3 benutzt

example.c

```c
#include <fftw3.h>
int main () {
  fftw_complex *in =
           fftw_malloc(sizeof(fftw_complex) * 10);
}
```

Dann kompilieren mit

```
gcc -c example.c -I/ordner/der/fftw3.h/enthält/
gcc -o example example.o -lfftw3
                  -L/ordner/der/libfftw3.so/enthält/
```

# Make Alternativen

- SCons
- Waf
- Automake
- CMake
- ninja

# CMake Erste Schritte

CMakeLists.txt für Beispiel 1

```
cmake_minimum_required(VERSION 2.8)

# Projektnamen festlegen
project(Hello)

# Ausführbare Datei hinzufügen
add_executable(hello main.c helper.c)
```

Dann die auszuführbare Datei erstellen über

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug # oder Release für
                                 # Optimierungen
make # oder cmake --build .
```

# CMake Weitere Befehle

```
# Standard festlegen
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Include Pfad hinzufügen
target_include_directories(hello
                    "${PROJECT_SOURCE_DIR}/include")
# Mit Bibliothek linken
target_link_libraries(hello mylib)

# Bibliothek erstellen
add_library(mylib mylib.c)
```

Siehe auch https://cmake.org/cmake/help/latest/guide/tutorial/index.html