CMake-Einführung

Andreas Naumann

11. Dez. 14

Wieso Buildsysteme?

Idee: löse $\Delta u = 1$ in $[0,1]^2$, $u(\partial[0,1]^2) = 0$, dense, Makefile

- mit dichter LU aus der MTL4

Makefile:

```
SOFTWARE=$(HOME)/Software/
CPPFLAGS+=-I$(SOFTWARE)/mtl4/
all: laplace
```

Wieso Buildsysteme?

Idee: löse $\Delta u = 1$ in $[0,1]^2$, $u(\partial[0,1]^2) = 0$, sparse, Makefile

- mit sparse-LU Umfpack-Binding aus der MTL4
- deren Features & Abhängigkeiten.

Makefile:

```
SW=$(HOME)/Software/
S_DIR=$(SW)/suitesparse-4.2.1
UMF_LIB=$(S_DIR)/lib
UMF_H=$(S_DIR)/include

CPPFLAGS+=-I$(SW)/mtl4/
CPPFLAGS+=-DMTL_HAS_UMFPACK\
          -I$(UMF_H)
LDLIBS+=-L$(UMF_LIB) -lumfpack\
               -lamd -lcholmod\
               -lcolamd -lblas\
               -lsuitesparseconfig

all: laplace
```

Wieso Buildsysteme?

Idee: löse $\Delta u = 1$ in $[0,1]^2$, $u(\partial[0,1]^2) = 0$, sparse, Makefile

- mit sparse-LU Umfpack-Binding aus der MTL4
- deren Features & Abhängigkeiten.

Weitere Nutzer $\rightarrow$

- Wo sind die Bibliotheken?
- Welche Distribution?
- externe Abhängigkeiten?
- Anderes OS
- Einbindung in IDE's

Makefile:

```
SW=$(HOME)/Software/
S_DIR=$(SW)/suitesparse-4.2.1
UMF_LIB=$(S_DIR)/lib
UMF_H=$(S_DIR)/include

CPPFLAGS+=-I$(SW)/mtl4/
CPPFLAGS+=-DMTL_HAS_UMFPACK\
          -I$(UMF_H)
LDLIBS+=-L$(UMF_LIB) -lumfpack\
               -lamd -lcholmod\
               -lcolamd -lblas\
               -lsuitesparseconfig

all: laplace
```

CMake, Grundlegendes

- Eine Datei: **CMakeLists.txt**
- Variable $\neq$ VARIABLE, FUNCTION(args) == function(args)
- Grundlegende Befehle:
  - `project(NAME)`
  - `set(VAR <Wert>)`
  - `include_directories(<liste-von-verzeichnissen>)`
  - `add_(library|executable)(TGT <liste-von-quellen>)`
  - `target_link_libraries(TGT <liste-von-bibliotheken>)`
  - `add_subdirectory(<dirname>)`

```
project(laplace)
cmake_minimum_required(VERSION 2.8)
set(SOFTWARE "$ENV{HOME}/Software/")
include_directories("${SOFTWARE}/mtl4/")
add_executable(laplace laplace.cc)
```

Introspektion

- `find_library(VAR <lib> HINTS ....)`
- `find_file(VAR <file> HINTS ....)`
- `find_path(VAR <file> HINTS ....)`
- `find_package(PKG <OPT> <COMP>)`

```
project(laplace)
cmake_minimum_required(VERSION 2.8)
find_package(MTL REQUIRED UMFPACK)
include_directories(${MTL_INCLUDE_DIRS})
add_definitions(${MTL_CXX_DEFINITIONS})
add_executable(laplace laplace.cc)
target_link_libraries(laplace ${MTL_LIBRARIES})
```

Introspektion

- `find_library(VAR <lib> HINTS ....)`
- `find_file(VAR <file> HINTS ....)`
- `find_path(VAR <file> HINTS ....)`
- `find_package(PKG <OPT> <C>)`

Ergebnis:

- Build unabhängig von:
  - Systempfaden
  - Konfiguration
- Modularisierung

```
project(laplace)
cmake_minimum_required(VERSION 2.8)
find_package(MTL REQUIRED UMFPACK)
include_directories(${MTL_INCLUDE_DIRS})
add_definitions(${MTL_CXX_DEFINITIONS})
add_executable(laplace laplace.cc)
target_link_libraries(laplace ${MTL_LIBRARIES})
```

Paketsuche

- `<PKG>Config.cmake`
- gegeben: `<PKG>_DIR`

- `Find<PKG>.cmake`
- unbekannt: `<PKG>_DIR`

- Ergebnis: `<PKG>_FOUND`
- Befehle: `find_*`
- Variablen:
  - `<PKG>_<C>_FOUND`
  - `<PKG>_<C>_CXX_DEFINITIONS`
  - `<PKG>_INCLUDE_DIRS`
  - `<PKG>_<C>_LIBRARIES`

Beispiel

Paket Datacontainer:

- Abhängigkeit: MTL4
- `include/data.h`
- `lib/libData1.so`
- `lib/libData2.so`
- `lib/Datacontainer.cmake`

Code3:

- Abhängigkeit: Container & MTL
- `src/laplace.cc`
- `CMakeLists.txt`

Beispiel, CMakeLists.txt

```
project(laplace)
cmake_minimum_required(VERSION 2.8)
find_package(MyIOLib REQUIRED)
find_package(MTL REQUIRED UMFPACK)
include_directories(${MyIOLib_INCLUDE_DIRS} ${MTL_INCLUDE_DIRS})
add_definitions(${MyIOLib_DEFINITIONS} ${MTL_CXX_DEFINITIONS})
add_executable(laplace src/laplace.cc)
target_link_libraries(laplace ${MyIOLib_LIBRARIES} ${MTL_LIBRARIES})
```

Beispiel, MyIOLibConfig.cmake

```
#describes the io-library
#variables: MyIOLib_INCLUDE_DIRS, MyIOLib_LIBRARIES, MyIOLib_DEFINITIONS
find_package(MTL REQUIRED)
find_library(MyIOLib_T_LIB MyIOLibB HINTS ${MyIOLib_DIR} NO_DEFAULT_PATH)
find_library(MyIOLib_B_LIB MyIOLibT  HINTS ${MyIOLib_DIR} NO_DEFAULT_PATH)
get_filename_component(MyIOLib_INCLUDE_DIRS "${MyIOLib_DIR}/../include" ABS
list(APPEND MyIOLib_INCLUDE_DIRS ${MTL_INCLUDE_DIRS})
list(APPEND MyIOLib_LIBRARIES ${MyIOLib_T_LIB} ${MyIOLib_B_LIB} ${MTL_LIBRA
list(APPEND MyIOLib_DEFINITIONS "-std=c++11" ${MTL_CXX_DEFINITIONS})
if(NOT MyIOLib_T_LIB)
        message(FATAL_ERROR "wrong installation")
endif()
if(NOT MyIOLib_B_LIB)
        message(FATAL_ERROR "wrong installation")
endif()
if( NOT EXISTS "${MyIOLib_DIR}/../include/MyIOLib.h")
        message(FATAL_ERROR "wrong installation")
endif()
```

Konfiguration, mit INTERFACE_*

```
project(laplace)
cmake_minimum_required(VERSION 2.8)
find_package(MyIOLib REQUIRED)
find_package(MTL REQUIRED UMFPACK)

include_directories(${MTL_INCLUDE_DIRS})
add_definitions(${MTL_CXX_DEFINITIONS})
add_executable(laplace src/laplace.cc)
target_link_libraries(laplace ${MyIOLib_LIBRARIES} ${MTL_LIBRARIES})
```

Beispiel, MyIOLibConfig.cmake

```
find_package(MTL REQUIRED)
find_library(MyIOLib_T_LIB MyIOLibT HINTS ${MyIOLib_DIR} NO_DEFAULT_PATH)
find_library(MyIOLib_B_LIB MyIOLibB HINTS ${MyIOLib_DIR} NO_DEFAULT_PATH)
get_filename_component(MyIOLib_INCLUDE_DIRS "${MyIOLib_DIR}/../include" ABS
list(APPEND MyIOLib_INCLUDE_DIRS ${MTL_INCLUDE_DIRS})
add_library(MyIOLib::T SHARED IMPORTED)
set_target_properties(MyIOLib::T PROPERTIES
        IMPORTED_LOCATION             ${MyIOLib_T_LIB}
        INTERFACE_INCLUDE_DIRECTORIES ${MyIOLib_INCLUDE_DIRS}
        INTERFACE_COMPILE_OPTIONS "-std=c++11")
add_library(MyIOLib::B SHARED IMPORTED)
set_target_properties(MyIOLib::B PROPERTIES
        IMPORTED_LOCATION ${MyIOLib_B_LIB}
        INTERFACE_INCLUDE_DIRECTORIES ${MyIOLib_INCLUDE_DIRS}
        INTERFACE_COMPILE_OPTIONS "-std=c++11")
list(APPEND MyIOLib_LIBRARIES MyIOLib::T MyIOLib::B ${MTL_LIBRARIES})

if(NOT MyIOLib_T_LIB)
        message(FATAL_ERROR "wrong installation")
endif()
if(NOT MyIOLib_B_LIB)
```

Hinweies & Weitere Features

- Bibliotheken, Includepfade: absolute Namen
- Ändere NIEMALS den Compiler $\rightarrow$ Buildverzeichnisse, Wrapper
- Feinsteuerun: `set_*_properties`, `get_*_property`
- compile & Link-Tests: `try_compile`, CheckCXXCompilerFlag
- Dateierzeugung: `file(...)`, `configure_file(...)`
- Externe-Projekte einbinden `ExternalProject_Add`
- Paketerstellung (rpm, deb, tar.gz) `cpack`
- Test (ctest, `add_test(...)` & Dashboard `cdash`
- Generatoren: Eclipse, VS, XCode, ninja, ...

Zusammenfassung

- Pro:
  - einfache Sprache
  - einfache Modularisierung
  - Schnittstelle zu pkgConfig
  - große Auswahl an Paketen
  - Systemunabhängig, Auswahl an Generatoren
  - aktive Entwickler & Community
- Kontra:
  - Inkonsistente Variablen in Modulen
  - Variablenauswertung in if(...)
  - Trennung Statischer & dynamischer Bibliotheken bei Suche schwierig