CMake Tutorial

Für C++, Java, LaTeX und Linux

Dr. Günter Kolousek

2016-2019

Inhaltsverzeichnis

1 Überblick

Bei CMake handelt es sich um ein plattformübergreifendes Programm zum Generieren von Buildsystemen. Es erzeugt aus einer Beschreibungsdatei `CMakeLists.txt` eines Projektes eine Datei `Makefile` für das unter Unix meist verwendete `make` (auch unsere Wahl!). CMake unterstützt (je nach verwendeter Plattform) eine Vielzahl an weiteren Generatoren, die z.B. ein Visual Studio-Projekt, ein Eclipse-Projekt oder auch ein XCode-Projekt erzeugen.

Hilfe zu CMake gibt es entweder auf der Homepage `http://www.cmake.org` zu finden, aber auch die Option `--help` kann weiterhelfen:

```
cmake --help
```

Wie man den zu verwendeten Generator explizit setzen kann ist im Abschnitt Explizite Angabe des Generators zu sehen.

Weitere Möglichkeiten CMake aufzurufen:

- Eine graphischen Benutzeroberfläche steht mit `cmake-gui` zur Verfügung.
- Eine textuelle Oberfläche für das Terminal steht mit `ccmake` zur Verfügung.

Beide Varianten sind wie das Kommando `cmake` zu verwenden.

2 Eine erste `CMakeLists.txt`

Nehmen wir an, dass wir ein klassisches "Hello World" Programm schreiben wollen und es demzufolge in der folgenden Art und Weise programmiert ist:

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
}
```

Für dieses "Projekt" wird dieses Programm in einem eigenen Verzeichnis, dem Projektverzeichnis, abgespeichert und dazu kommt weiters die folgende minimale `CMakeLists.txt` Datei:

```
cmake_minimum_required(VERSION 3.5)
project(hello)
add_executable(hello hello.cpp)
```

Zuerst wird die minimale Version von `cmake` angegeben, dann wird ein Projektname ("hello") vergeben, wobei standardmäßig die Programmiersprachen `C` und `C++` (also `CXX`) von CMake für das Projekt angenommen werden. Wie andere Programmiersprachen verwendet werden können ist im Abschnitt Spezifizieren von Projektinformationen zu finden.

Zum Schluss wird festgelegte, dass es eine ausführbare Datei geben soll, die `hello` heißt und aus der Datei `hello.cpp` zu übersetzen ist.

Die aktuelle Version von `cmake` kann so ermittelt werden:

```
cmake --version
```

Ist diese Versionsnummer nicht mindestens so hoch wie in der Datei `CMakeLists.txt` angegeben, dann wird der folgende Aufruf (ausgeführt im Verzeichnis der Datei `CMakeLists.txt`) mit einer Fehlermeldung fehlschlagen:

```
cmake .
```

Anderenfalls analysiert `cmake` die `CMakeLists.txt` im aktuellen Verzeichnis (→ Verzeichnis `.`) und sucht danach im System nach den erforderlichen Tools (Compiler, Linker,...) und erstellt danach die notwendigen Dateien zum eigentlichen "Builden" des Programmes.

Außerdem werden die verschiedensten Meldungen am Bildschirm, die die gefundenen Tools angeben. Abgeschlossen wird diese Liste der Meldungen bei Erfolg mit:

```
Build files have been written to:...
```

Das kannst du dir die angelegten Dateien gerne mit einem `ls` in der Shell deiner Wahl ansehen.

Da wir auch `make` verwenden, wird auch die entscheidende Datei `Makefile` generiert, die du dir gerne einmal mit dem Kommando `cat` ansehen kannst.

Das eigentliche "Builden" wird unter Linux (bei Verwendung von `make`) folgendermaßen gestartet:

```
make
```

Plattformunabhängig kann das "Builden" folgendermaßen gestartet werden:

```
cmake --build .
```

Damit wird das "Builden" gestartet und am Ende liegt ein ausführbares Programm mit dem Namen "hello" vor.

Es wird automatisch die Datei `hello.cpp` in eine Objektdatei übersetzt und mit Standardbibliothek gebunden und im Dateisystem als eine ausführbare Datei `hello` abgelegt.

Bei jeder Änderung in einer Source-Code-Datei wie `hello.cpp` muss nur noch mehr `make` eingegeben werden. Der Rest funktioniert automatisch. Dies werden wir noch zu schätzen lernen, wenn unsere Projekte aus mehreren (vielen) Dateien bestehen.

Gratulation, erstes `cmake` - Projekt erfolgreich erstellt.

Eine Kleinigkeit ist bezüglich der Syntax von `CMakeLists.txt` Dateien zu beachten: Die Groß-/Kleinschreibung ist bei Direktiven nicht von Belang, allerdings spielt diese bei Variablennamen und Argumenten sehr wohl eine Rolle!

3 Eigenes Build-Verzeichnis

Das Vermischen von Source-Code-Dateien, CMake-Dateien und erzeugten Dateien wie ausführbaren Programmen ist nicht sinnvoll. Besser ist es, alle irgendwie erzeugten Dateien in ein eigenes Verzeichnis auszulagern. Dieses Verzeichnis nennen wir `build` und legen in dieses in unserem Projektverzeichnis an. Außerdem verschieben wir im gleichem Schritt die `.cpp` - Dateien auch in ein eigens angelegtes Verzeichnis `src`. Der Verzeichnisbaum für das neue Projekt `hello2` im Verzeichnis `hello2` sieht jetzt folgendermaßen aus (nachdem du alle erzeugten Dateien manuell gelöscht hast):

```
hello2
  CMakeLists.txt
  build
  src
    hello.cpp
```

Bitte auch die `CMakeLists.txt` an den neuen Projektnamen und and den geänderten Ort von `hello.cpp` anpassen:

```
cmake_minimum_required(VERSION 3.5)
project(hello2)
add_executable(hello src/hello.cpp)
```

Zum Übersetzen wechselst du jetzt in das Verzeichnis `build` und startet `cmake` mit dem Verzeichnis, das die `CMakeLists.txt` enthält:

```
cd build
cmake ..
```

Jetzt werden alle notwendigen Dateien im Verzeichnis `build` erzeugt und hier kann auch der eigentliche Build-Vorgang gestartet werden:

```
make
```

Das Programm kann man danach wie gewohnt mit `hello` starten.

Abgesehen von der verbesserten Struktur können wir mit einem Schlag alle erzeugten Dateien löschen, indem wir den gesamten Inhalt des Verzeichnes `build` löschen.

Solche Verzeichnisstrukturen werden *out-of-source builds* genannt.

4 Explizite Angabe des Generators

Wie schon anfangs erwähnt, unterstützt CMake eine Vielzahl an verschiedenen Generatoren. Jetzt zeige ich wie man z.B. den Generator `Ninja` verwendet:

```
cmake -G Ninja ..
```

Alternativ kann man in der `CMakeLists.txt` durch Setzen der Variable `CMAKE_GENERATOR` sich die explizite Angabe über die Kommandozeile ersparen.

Eine Liste der unterstützten Generatoren erhält man durch Aufruf von `cmake --help`.

CMake erkennt allerdings den von der aktuellen Plattform unterstützten Generator alleine, womit eine explizite Angabe des Generators nur notwendig ist, wenn mehrere vorhanden sind und man einen bestimmten auswählen will.

5 Mehrere Source-Code-Dateien und Verwendung von CMake-Variablen

5.1 Erste Version

Nehmen wir einmal an, dass unser Beispielprojekt aus den Dateien `main.cpp` und `hello.cpp` besteht, wobei die Ausgabe unseres glorreichen "Hello world!" in eine eigene Funktion (!) `say_hello` in der Datei `hello.cpp` ausgelagert wird. Diese Änderungen werden wir in einem weiteren Projekt `hello3` (in dem entsprechenden Verzeichnis) vornehmen.

D.h. die Datei `hello.cpp` sieht so aus:

```
#include <iostream>

using namespace std;

void say_hello() {
    cout << "Hello world!" << endl;
}
```

In der Datei `main.cpp` wird lediglich die Funktion `say_hello` aufgerufen:

```
void say_hello();

int main() {
    say_hello();
}
```

Klarerweise muss dem `cmake` dieser Zusammenhang jetzt mitgeteilt werden:

```
cmake_minimum_required(VERSION 3.5)
project(hello3)
add_executable(hello src/main.cpp src/hello.cpp)
```

D.h. alle Source-Code-Dateien werden durch jeweils ein Leerzeichen getrennt hinten an der `add_executable` Direktive angehängt.

Der Rest funktioniert wie gehabt.

5.2 Zweite Version

Allerdings ist es natürlich mühsam alle zusammengehörigen Source-Code-Dateien in der `add_executable` Direktive zu erfassen. Hier kann ich in einfacher Art und Weise 2 Lösungen anbieten:

1. Erfasse alle Dateien, die zu einer zusammengehörigen Einheit gehören in einer Variable `SOURCES` und verwende diese Variable in `add_exeutable`:

   ```
   set(SOURCES src/main.cpp src/hello.cpp)
   add_executable(hello ${SOURCES})
   ```

   Beachte zwei Dinge:

   - In einer CMake-Variable können auch mehrere Werte gespeichert werden. Dazu werden diese einfach getrennt durch Leerzeichen angeschrieben. Diese Werte werden danach also Liste interpretiert (Listen sind in CMake Werte, die durch ein Semikolon (`;`) getrennt sind; hier allerdings nicht notwendit).
   - Beachte weiters wie du auf den Wert der Variable zugreifst.

   Der Rest funktioniert wie gehabt.

2. Natürlich entbindet uns die vorhergehende Lösung nicht davon, die Variable entsprechend zu verwalten. Bei jedem Hinzufügen oder Entfernen einer Datei muss der Wert entsprechend adaptiert werden.

   Besser ist die Dateien von `cmake` zu ermitteln lassen, indem du das Setzen der Variable durch folgende Zeile ersetzt:

   ```
   file(GLOB_RECURSE SOURCES "src/*.cpp")
   ```

   Hier funktioniert das `GLOB_RECURSE` wie auch in der Shell das "globbing" funktioniert, also nichts Neues, abgesehen davon, dass dies auch die Unterverzeichnisse rekursiv durchsucht. Ist das nicht gewünscht, dann `GLOB_RECURSE` durch `GLOB` ersetzen. Das einzige, das bei dieser Lösung bedacht werden muss ist, dass das `cmake` Kommande bei jeder Änderung einmal aufgerufen werden muss.

   Aber bedenke: Danach muss zumindest wieder `cmake ..` ausgeführt werden. Manchmal ist aber auch der CMake-Cache nicht mehr gültig. Daher ist es, bei Verwendung von Unix Makefiles, besser das folgende Kommando auszuführen:

   ```
   make rebuild_cache
   ```

   Also: Das erste Mal ein CMake Projekt mittels `cmake ..` initialisieren und danach jedes Mal `make rebuild_cache` ausführen, wenn neue Quellcodeateien hizugefügt worden sind!

6 Ausgabe von Meldungen auf der Konsole und Setzen von Kommentaren

Oft ist es notwendig, den Ablauf der Abarbeitung der Datei `CMakeLists.txt` mitzuverfolgen. Dafür bietet sich das CMake-Kommando `message` an. Will man z.B. sehen, welche Werte in der gesetzten Variable `SOURCES` stehen, kann man das folgendermaßen lösen:

```
message(In SOURCES steht: ${SOURCES})
```

Damit sieht man beim Aufruf von `cmake` die entsprechende Meldung. Benötigt man die Information nicht mehr, kann man diese Zeile natürlich wieder löschen. Will man unter Umständen zu einem späteren Zeitpunkt sich die Information anzeigen lassen, dann ist die einfachste Möglichkeit, diese in einem Kommentar zu verpacken. Dies geschieht einfach dadurch, dass ein Hashzeichen vorzustellen (wie in Python oder der Shellprogrammierung unter Unix üblich):

```
# message(In SOURCES steht: ${SOURCES})
```

7 Headerdateien

Nehmen wir an, das wir jetzt auch über eine Headerdatei `hello.h` verfügen, die die Schnittstelle unseres glorreichen Moduls `hello.cpp` enthält, nämlich den Prototypen der Funktion `say_hello`:

```
#ifndef HELLO_H
#define HELLO_H

void say_hello();

#endif
```

Diese Headerdatei gehört eindeutig in ein anderes Unterverzeichnis unseres Projektes. Hier bietet sich `include` an. Damit sieht unser Verzeichnisbaum jetzt folgendermaßen aus (wenn du ein neues Projekt im Verzeichnis `hello4` angelegt hast):

```
hello4
  CMakeLists.txt
  build
  include
    hello.h
  src
    hello.cpp
```

Um das Modul richtig zu implementieren, muss auch noch die Datei `hello.cpp` angepasst werden:

```
#include <iostream>

#include "hello.h"

using namespace std;

void say_hello() {
    cout << "Hello world!" << endl;
}
```

Letztendlich muss natürlich auch noch `main.cpp` angepasst werden:

```
#include "hello.h"

int main() {
    say_hello();
}
```

Das ist ja alles gut und schön, aber jetzt muss dem Compiler noch mitgeteilt werden wo die Header-Dateien liegen, sonst wirst du Fehlermeldungen bekommen. Auch hier hilft `cmake` weiter. Adaptiere dazu deine `CMakeLists.txt` indem du die folgende Zeile nach `add_executable` anfügst:

```
target_include_directories(hello PRIVATE include)
```

Ein neues `cmake` und alles ist wieder in Ordnung!

Damit können wir schon einfache Programme auf der Basis von *out-of-source builds* erstellen, bei dem auch die Headerdateien in einem eigenem Verzeichnis zusammengefasst sind.

PRIVATE bedeutet in diesem Zusammenhang, dass die angegebenen Verzeichnisse nur für das angegebene Target (also "hello") Verwendung finden. Das ist im Falle eines Executables auch sinnvoll. Aber bei einer Library...

8 Erstellen einer "shared library" und Installation eines Targets

Nehmen wir an, dass wir unsere fantastische say_hello in eine shared library verpacken wollen, damit wir diese in die ungezählten, zukünftigen, extrem wichtigen Projekte verwenden können.

Erstelle daher ein Verzeichnis hello_shared und in diesem wie gewohnt je ein Verzeichnis src, include und build. Ins src Verzeichnis kommt nur unser hello.cpp und in das Verzeichnis include die entsprechende Headerdatei.

Die Datei CMakeLists.txt beinhaltet folgendes:

```
cmake_minimum_required(VERSION 3.5)
project(hello_shared)

file(GLOB_RECURSE SOURCES "src/*.cpp")

add_library(hello SHARED ${SOURCES})
target_include_directories(hello PUBLIC include)
install(TARGETS hello DESTINATION ~/lib)
```

Wir sehen, dass wir jetzt eine Library erstellen, die offensichtlich als "shared" gekennzeichnet ist und sich aus den Dateien (eigentlich nur eine) aus dem Verzeichnis src zusammensetzen soll.

Weiters ist zu beachten, dass bei target_include_directories jetzt PUBLIC angegeben ist. Das ist wichtig, da die in include enthaltene Datei hello.h sowohl für die Implementierung der Library als auch für die Verwendung der Library benötigt wird (also den abhängigen Targets).

Eine shared library sollte (muss aber nicht) auch irgendwohin installiert werden, daher haben wir hier auch eine install Direktive verwendet. Der Einfachheit halber habe ich gesagt, dass diese in das Verzeichnis ~/lib (also lib im Homeverzeichnis) installiert werden soll, da wir dort auch sicher Schreibrechte haben.

```
cmake ..
make
make install
```

Das make install bewirkt eben die "Installation" eigentlich das Kopieren der entstandenen Library in das angegebene Verzeichnis. Natürlich kann man dies auch manuell erledigen.

Natürlich sollte man eine shared library auch verwenden, aber davon mehr im nächsten Abschnitt.

9 Verwenden einer "shared library"

Im vorhergehenden Abschnitt haben wir eine shared library erstellt, jetzt wollen wir diese in einem Projekt hello5 auch verwenden.

Kopiere deshalb hello4 auf hello5 und lösche die Datei hello.cpp aus dem src Verzeichnis.

Die CMakeLists.txt soll jetzt folgendermaßen aussehen:

```
cmake_minimum_required(VERSION 3.5)
project(hello5)
```

```
file(GLOB_RECURSE SOURCES "src/*.cpp")

link_directories(~/lib)
# alternativ, wenn direkt verwendet:
#   link_directories(../hello_shared/build)
add_executable(hello ${SOURCES})
target_include_directories(hello PRIVATE ../hello_shared/include)
target_link_libraries(hello PRIVATE libhello.so)
```

`link_directories` legt die Verzeichnisse fest in denen Bibliotheken zu finden sind, in unserem Fall das Verzeichnis `lib` im Homeverzeichnis und `target_link_libraries` legt fest gegen welche Bibliothek das ausführbare Programm gelinkt werden soll.

Die Direktive `target_include_directories` legt geht hier davon aus, dass alle unsere Projekte in einem gemeinsamen Verzeichnis enthalten sind (dem übergeordnetem Verzeichnis) und innerhalb dieses Projektes wird direkt auf dieses Verzeichnis zugegriffen. Damit benötigt `hello5` auch kein eigenes Verzeichnis `include`.

Was aber, wenn sich die shared library sich nicht in `~/lib` (oder in einem anderen mittels `link_directories` spezifiziertem Verzeichnis) befindet? Lösche diese von dort und probiere jetzt das Programm `hello` wieder zu starten.

Eh klar, das geht nicht, aber was ist dann zu tun?

```
export LD_LIBRARY_PATH=../../hello_shared/build
```

Dann lässt sich das Programm wieder starten, da der Loader sich auch den Inhalt der Umgebungsvariable `LD_LIBRARY_PATH` ansieht. Besser wäre es natürlich hier einen absoluten Pfad zu verwenden, wie z.B. `~/lib`, aber ich wollte hier explizit auch einmal einen relativen Pfad einsetzen.

Man sieht, dass die Verwendung einer shared library eben nicht nur Vorteile hat (kleinere Excecutables, leichtere Änderungen der Funktion durch Austauschen der shared library, geringerer Verbrauch an Festplattenpeicher und Hauptspeicher), sondern auch Nachteile (fehlende shared libraries oder falsche Versionen). Deshalb weiter zum nächsten Punkt.

10 Erstellen einer "static library"

An sich alles gleich wie bei einer shared library nur die `CMakeLists.txt` ist ein bisschen anders:

```
cmake_minimum_required(VERSION 3.5)
project(hello_static)

file(GLOB_RECURSE SOURCES "src/*.cpp")

add_library(hello STATIC ${SOURCES})
target_include_directories(hello PUBLIC include)

# nicht unbedingt notwendig:
install(TARGETS hello DESTINATION ~/lib)
```

11 Verwenden einer "static library"

Das Verwenden einer static library funktioniert ähnlich wie bei einer shared library, nur der Name Library in der Datei `CMakeLists.txt` gehört geändert:

```
cmake_minimum_required(VERSION 3.5)
project(hello6)

file(GLOB_RECURSE SOURCES "src/*.cpp")

link_directories(../hello_static/build)
# alternativ, wenn statische Library installiert:
#   link_directories(~/lib)
add_executable(hello ${SOURCES})
target_include_directories(hello PRIVATE ../hello_static/include)
target_link_libraries(hello PRIVATE libhello.a)
```

12 Erstellen und verwenden einer "header-only library"

Gehen wir von folgender Projektstruktur aus:

```
hello8
  CMakeLists.txt
  hello_headeronly
    CMakeLists.txt
    include
      hello.h
  src
  build
```

Schauen wir uns zuerst die Datei `hello.h` an:

```
#ifndef HELLO_H
#define HELLO_H

#include <iostream>

inline void say_hello() {
    std::cout << "Hello, World!" << std::endl;
}
#endif
```

Die Datei `CMakeLists.txt` der header-only Bibliothek ist simpel:

```
add_library(say_hello_headeronly INTERFACE)
target_include_directories(say_hello_headeronly INTERFACE include)
```

Hier hat `INTERFACE` in `add_library` die Bedeutung, dass eine header-only Bibliothek erzeugt wird und daher darf es darin auch keine Angabe von Source-Dateien geben. Mittels `target_include_directories` wird das Verzeichnis mit den Header-Dateien mitgeteilt.

Damit kommen wir auch schon zu der Verwendung dieser header-only Bibliothek. Die entsprechende `CMakeLists.txt` sieht ganz unspektakulär folgendermaßen aus:

```
cmake_minimum_required(VERSION 3.5)
project(hello8)

add_subdirectory(hello_headeronly)

file(GLOB_RECURSE SOURCES "src/*.cpp")

add_executable(hello ${SOURCES})
target_compile_features(hello PUBLIC cxx_std_14)
target_link_libraries(hello say_hello_headeronly)
```

13 Installationsverzeichnis spezifizieren

Wir haben uns schon angesehen, dass man bei der `install`-Direktive einen Parameter `DESTINATION` angeben kann. Wie wir gesehen haben kann man hier einen absoluten Pfad angeben, aber es besteht auch die Möglichkeit einen relativen Pfad zu verwenden.

Um dies gut ausnutzen zu können, kann man `cmake` auch folgendermaßen starten:

```
cmake .. -DCMAKE_INSTALL_PREFIX=../install
```

Dann werden die `DESTINATION` Parameter, wenn diese relative Pfade darstellen, als relative Pfade zu dem angegebenen "Präfix" verstanden. Damit kann man Libraries z.B. in einem Verzeichnis mit dem Namen `lib` abspeichern, wenn die `install`-Direktive in der `CMakeLists.txt` der Library so aussieht:

```
install(TARGETS hello DESTINATION lib)
```

Das Executable könnte in einem Verzeichnis `bin` abgelegt werden oder zusätzliche Header-Dateien, die bei der Entwicklung einer Bibliothek für den Benutzer essentiell sind, könnten folgendermaßen in einem Verzeichnis `include` "installiert" werden:

```
install(FILES include/hello.h DESTINATION include)
```

14 Explizite Angabe der Programmiersprache

Standardmäßig aktiviert `cmake` die Verarbeitung von `C` und `C++` Projekten, allerdings können die benötigten Sprachen auch in der `project`-Direktive angegeben werden:

```
project(hello LANGUAGES CXX)
```

15 Explizite Angabe des Compilers

Um einen speziellen Compiler einzusetzen, ist die Variable `CMAKE_CXX_COMPILER` wie folgt zu setzen:

```
set(CMAKE_CXX_COMPILER g++)
```

16 Explizite Angabe der `C++` Version

Eine aktuelle CMake-Version vorausgesetzt, kann man die benötigte `C++` Version auf folgende Weise angeben:

```
target_compile_features(hello PUBLIC cxx_std_14)
```

Damit wird für das Target `hello` der C++ Standard 14 verwendet, wobei `PUBLIC` bedeutet, dass sowohl Features zum Übersetzen sowohl von `.cpp` als auch von `.h` Dateien.

Klarerweise gibt es auch `cxx_std_11` bzw. `cxx_std_17`, aber es ist mit `target_compile_features` auch möglich spezielle Features von C++ explizit anzufordern (wie z.B., dass `constexpr` unterstützt wird: `cxx_constexpr`).

Alternativ kann man auch die verwendete Compiler-Version generell für das gesamte Projekt setzen:

```
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

Damit wird `cmake` das Projekt mit C++ 14 übersetzen, wenn C++ Code zu übersetzen ist (für jeden verwendeten Compiler). Außerdem wird das Konfigurieren mit einer Fehlermeldung abgebrochen, wenn kein C++ 14-fähiger Compiler zur Verfügung steht.

Bei etwas älterer CMake-Version so vorzugehen, wie in in Abschnitt Explizites Setzen von Compileroptionen gezeigt wird.

17 Explizites Setzen von Compileroptionen

Will man spezielle Compileroptionen verwenden, dann kann man diese wie z.B. auch die Angabe der C++ Version und die Anforderungen (viele) Warnungen anzuzeigen, dann kann man dies auf folgende Art und Weise erledigen:

```
add_compile_options(-std=c++14 -Wall)
```

Das setzt (natürlich) allerdings voraus, dass der verwendete Compiler diese Option in dieser Form unterstützt, wie dies für den g++ oder den `clang` der Fall ist!

Die Direktive `add_compile_definitions` bezieht sich auf das Verzeichnis und die darunterliegenden Verzeichnisse. Besser ist es allerdings `target_compile_options` wie folgt zu verwenden (empfohlen):

```
target_compile_options(hello PUBLIC -std=c++14 -Wall)
```

Alternativ (nicht empfohlen!!!) kann man auch die CMake-Variable `CMAKE_CXX_FLAGS` dafür setzen:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")
```

Will man jedoch den verwendeten Sprachstandard von C++ setzen, dann soll man auf jeden Fall wie in Abschnitt Explizite Angabe der C++ Version gezeigt vorgehen!

Bei Verwendung von Release- und Debugversionen (siehe Übersetzen als Release- oder Debugversion) macht es Sinn die eingesetzten Compilerversionen in Abhängigkeit von Release- bzw. Debugversion unterschiedlich anzugeben. Das kann auf folgende Art und Weise realisiert werden:

```
set(MY_DEBUG_OPTIONS -O0 -Wall -Werror)
set(MY_RELEASE_OPTIONS -O3)

target_compile_options(foo PUBLIC "$<$<CONFIG:Debug>:${MY_DEBUG_OPTIONS}>")
target_compile_options(foo PUBLIC "$<$<CONFIG:Release>:${MY_RELEASE_OPTIONS}>")
```

Das vorhergehende Codesnippet setzt voraus, dass in den CMake-Variablen `MY_DEBUG_OPTIONS` und `MY_RELEASE_OPTIONS` im Vorhinein die eigentlichen Optionen abgespeichert worden sind.

18 Plattformspezifische Aktivierung von Warnungen

Prinzipiell kann die Ausgabe von Warnungen mittels der in Abschnitt Explizites Setzen von Compileroptionen gezeigten Methode erreicht werden.

Um das CMake-Projekt auf Windows und Unix-Systemen verwenden zu können, ist es sinnvoll eine CMake `if`-Anweisung einzusetzen:

```
if (MSVC)
  add_compile_options(/W4)
else()
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

19 Setzen von Präprozessordefinitionen

Dem Präprozessor können Definitionen – unter `g++` und `clang` mittels der Option `-D` – mitgegeben werden. Diese entsprechen dann einer `#define` Präprozessordirektive.

Will man diese in der `CMakeLists.txt` (plattformübergreifend) angeben, dann kann man dies folgendermaßen erreichen:

```
add_definitions(DEBUG)
```

Eine Definition mit Wert sieht dann so aus:

```
add_definitions(DEBUG=1)
```

20 Übersetzen als Release- oder Debugversion

Das kann in einfacher Art und Weise so veranlasst werden, dass beim Aufruf von `cmake` eine Option mitgegeben wird:

- Als Releaseversion:

  ```
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```

  Der Pfad zur `CMakeLists.txt` ist natürlich entsprechend anzupassen.

- Als Debugversion

  ```
  cmake -DCMAKE_BUILD_TYPE=Debug ..
  ```

  Damit kann das Programm debuggt werden.

Folgende Möglichkeiten stehen in der Regel zur Verfügung:

- Debug: nicht optimierter Code mit Debugsymbolen
- Release: optimierter Code ohne Debugsymbolen
- RelWithDebInfo: optimierter Code mit Debugsymbolen

Will man eine der Versionen immer verwenden, dann kann man diese Variable auch direkt in der Datei `CMakeLists.txt` setzen:

```
set(CMAKE_BUILD_TYPE Debug)
```

Besser ist es allerdings folgenden CMake-Anweisungen an das Ende der Datei `CMakeLists.txt` anzufügen:

```
ADD_CUSTOM_TARGET(debug
  COMMAND ${CMAKE_COMMAND} -DCMAKE_BUILD_TYPE=Debug ${CMAKE_SOURCE_DIR}
  COMMAND ${CMAKE_COMMAND} --build ${CMAKE_BINARY_DIR} --target all
  COMMENT "Switch CMAKE_BUILD_TYPE to Debug"
  )

ADD_CUSTOM_TARGET(release
  COMMAND ${CMAKE_COMMAND} -DCMAKE_BUILD_TYPE=Release ${CMAKE_SOURCE_DIR}
  COMMAND ${CMAKE_COMMAND} --build ${CMAKE_BINARY_DIR} --target all
  COMMENT "Switch CMAKE_BUILD_TYPE to Release"
  )
```

Damit kann man eine der beiden Varianten direkt mittels `make` erzeugen: `make debug` oder eben `make release`.

21 Spezifizieren von Projektinformationen

Abgesehen von dem Projektnamen kann man bei dem Kommando `project` auch noch die Projektversion als auch die verwendeten Programmiersprachen angeben.

Die Grundversion von `project` sieht ja so aus, dass man nur den Projektnamen angibt. Damit werden von CMake automatisch die CMake-Variablen `PROJECT_SOURCE_DIR` als auch `PROJECT_BINARY_DIR` gesetzt. Auf die top-level Projektverzeichnisse kann immer mittels der Variablen `CMAKE_SOURCE_Dir` und `CMAKE_BINARY_DIR` zugegriffen werden.

Man kann für das Projekt auch eine Versionsnummer vergeben, entweder als `major`, als `major.minor`, als `major.minor.patch` oder auch als `major.minor.patch.tweak`:

```
project(hello VERSION 1.0)
```

Wobei die Versionsnummer die folgende Form annehmen muss: `<major>[.<minor>[.<patch>[.<tweak>]]]`, wobei jeder der Angaben eine nicht-negative Zahl sein muss. Damit sind die üblichen Versionsbezeichnungen realisierbar.

Diese Versionsinformationen stehen dann in den folgenden Variablen zur Verfügung: `PROJECT_VERSION`, `PROJECT_VERSION_MAJOR`, `PROJECT_VERSION_MINOR`, `PROJECT_VERSION_PATCH`, `PROJECT_VERSION_TWEAK`.

Weiters kann auch die verwendete Programmiersprache angegeben werden:

```
project(hello VERSION 1.0.1 LANGUAGES CXX)
```

Wird keine Programmiersprache angegeben, dann wird `C` und `C++` von CMake angenommen.

22 Unterprojekte

Es kann durchaus sinnvoll sein, sein Projekt aus Unterprojekten zusammensetzen zu lassen. Ein derartiges Unterprojekt ist nichts anderes als ein Unterverzeichnis in unserem aktuellen Projektverzeichnis, das genauso aufgebaut ist, wie unser derzeitiges Projektverzeichnis. Zusätzlich "erbt" das Unterprojekt die Konfiguration des Hauptprojektes.

Für unser Hello World-Beispiel könnte es so sein, dass ein Unterprojekt die statische Bibliothek ist. Der Verzeichnisbaum für unser Projekt `hello7` mit inkludierter Bibliothek sieht dann folgendermaßen aus:

```
hello7
  CMakeLists.txt
  hello_static
    CMakeLists.txt
    src
    include
  src
    main.cpp
  build
```

Die `CMakeLists.txt` von `hello_static` ist einfach aufgebaut, da diese von der übergeordneten `CMakeLists.txt` "erbt". Damit braucht diese nur folgendermaßen aussehen:

```
include_directories(include)

file(GLOB_RECURSE SOURCES "src/*.cpp")

add_library(hello_static STATIC ${SOURCES})
```

Die `CMakeLists.txt` des Hauptprojektes muss natürlich auch das Unterprojekt spezifizieren und das sieht dann folgendermaßen aus:

```
cmake_minimum_required(VERSION 3.5)
project(hello7)

add_subdirectory(hello_static)

file(GLOB_RECURSE SOURCES "src/*.cpp")

add_executable(hello ${SOURCES})
target_compile_features(hello PUBLIC cxx_std_14)  # also _11 and _17
target_include_directories(hello PRIVATE hello_static/include)
target_link_libraries(hello hello_static)
```

Zwei Aspekte sind an dieser `CMakeLists.txt` interessant:

- `add_subdirectory` fügt eben das Unterprojekt hinzu.
- Bei `target_link_libraries` kann einfach nur der Name der "target name" des Unterprojektes angegeben werden (d.h. den von `add_library`).

23 Verwenden von Threads

Um Threads in einem C++ Programm verwenden zu können, muss die entsprechende Bibliothek hinzugefügt werden und auch der Sprachstandard entsprechend gesetzt werden.

Plattformübergreifend funktioniert das auf folgende Art und Weise:

```
cmake_minimum_required(VERSION 3.5)
project(my_threads)

find_package(Threads REQUIRED)
add_executable(my_threads ${SOURCES})
target_compile_features(hello PUBLIC cxx_std_14)
target_link_libraries(my_threads ${CMAKE_THREAD_LIBS_INIT})
```

24 Unit-Tests mit C++

CMake bietet von Haus aus Unterstützung an, um Tests in die Build-Umgebung einzubinden.

Verwendet man die header-only Bibliothek Catch, dann kann das folgendermaßen aussehen:

```
CMakeLists.txt
catch
  CMakeLists.txt
  include
    catch.hpp
src
  calc.cpp
include
  calc.h
```

Im Verzeichnis `catch/include` befindet sich lediglich die header-only Bibliothek `catch.hpp`. Diese wurde im Verzeichnis `catch` als header-only CMake Projekt realisiert (siehe Erstellen und verwenden einer "header-only library"). Dazu sieht die zugehörige `CMakeLists.txt` folgendermaßen aus:

```
add_library(catch INTERFACE)
target_include_directories(catch INTERFACE include)
```

Im Verzeichnis `src` befindet sich der Programmcode für das Testprogramm, das in unserem Fall (mit `Catch`) folgendermaßen aussieht:

```
#define CATCH_CONFIG_MAIN

#include "catch.hpp"

#include <iostream>

#include "calc.h"

int sum(int a, int b) {
    return a + b;
}

TEST_CASE("sums are computed", "[arithmetic]") {
    REQUIRE(sum(0, 0) == 0);
    REQUIRE(sum(2, 3) == 5);
    REQUIRE(sum(3, 2) == 5);
    //...
}
```

Die noch nicht erwähnte `CMakeLists.txt` des Hauptprojektes sieht folgendermaßen aus:

```
cmake_minimum_required(VERSION 3.5)
project(hello9)

add_subdirectory(catch)

file(GLOB_RECURSE SOURCES "src/*.cpp")

add_executable(calc_test ${SOURCES})
target_include_directories(calc_test PUBLIC include)
```

```
target_compile_features(calc_test PUBLIC cxx_std_14)
target_link_libraries(calc_test catch)

enable_testing()
add_test(NAME first_test COMMAND calc_test)
```

Das Projekt ist aufgebaut, dass es die header-only Bibliothek `catch` aus dem Unterprojekt `catch` verwendet. Der einzige entscheidende Teil ist der Schluss. Mittels `enable_testing()` wird CMake angewiesen, dass die Funktionalität zum Erstellen von Tests aktiviert werden soll und mittels `add_test` kann eben ein benannter Test hinzugefügt werden, der das Executable `calc_test` starten soll.

Das CMake-Projekt wird wie üblich erzeugt. Danach steht weiters die Möglichkeit zur Verfügung die Tests mittels `ctest` oder auch (bei Verwendung von Unix Makefile: `make test`) zu starten:

```
$ ctest
Running tests...
Test project /home/knslnto/workspace/school/scripts/cmake_tutorial/hello9/build
    Start 1: first_test
1/1 Test #1: first_test ......................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.00 sec
```

Mittels `ctest -V` kann der geschwätzige Modus von `ctest` aktiviert werden, sodass genauere Fehlermeldungen zur Verfügung stehen.

25 CMake mit Netbeans

Hier eine kurze Anleitung wie ein vorhandenes out-of-source-directory CMake Projekt in Netbeans eingebunden werden kann:

1. CMake-Projektverzeichnis mit der `CMakeLists.txt` und den Unterverzeichnissen `src` (oder anderen Unterverzeichnissen), `include`, `build` einrichten. Das CMake-Projektverzeichnis sollte so heißen wie das Projekt in der `CMakeLists.txt`!

   In einem Source-Datei-Verzeichnis (z.B. `src`) sollte zumindest schon eine `.cpp` Datei enthalten sein.

2. Danach geht es in Netbeans weiter.

3. Ein neues Projekt anlegen (`File` → `New Project`). Im Dialog "Choose Project" ist die Kategorie `C/C++` mit Projekttyp `C/C++ Project with Existing Sources` auszuwählen. Jetzt geht es weiter mit `Next`.

4. Im darauffolgenden `Select Mode` Dialog ist das Source-Verzeichnis (also z.B. Projektverzeichnis selber) auszuwählen. Man kann auch gleich das eigentliche `src` (oder eines mit einem anderen Namen) eingeben. Allerdings kann in einem später Schritt noch beliebig viele weitere Source-Verzeichnisse hinzugefügt werden. Deshalb meine Empfehlung: Hier nur das eigentliche CMake-Projektverzeichnis angeben (dann kann man hier auch schon eine `main.cpp` angeben, wenn man will).

   Außerdem ist unterhalb der "Configuration Mode" `Custom` auszuwählen. Dann weiter mit `Next`.

5. Der weitere Dialog "Pre-Build Action" benötigt ein "Hakerl" bei `Pre-Build Step is Required`, samt der Eingabe des `build` Verzeichnis in dem Eingabefeld `Run in Folder` und eine Aus-

wahl des Radio-Button `Predefined Command` mit obligatorischen Auswählen von `CMake` bei der Dropdown-Box `Script-Type` und Auswahl der existierenden `CMakeLists.txt` Datei im Open-Dialog zur Auswahl des `Script`.

6. Im darauffolgenden Dialog "Build Actions" ist nur das `Working Directory`, also unser `build` Verzeichnis, auszuwählen. Dann geht es wieder weiter mit `Next`.
7. Im Dialog "Source Files" kann man zusätzliche Verzeichnisse mit `.cpp` Dateien angeben. Hat man lediglich das CMakeProjekt-Verzeichnis im Schritt "Configuration Mode" angegeben, dann ist hier eine gute Gelegenheit weitere Verzeichnisse wie z.B. ein `src` Verzeichnis hinzuzufügen. Dann geht es weiter mit einem beherzten Klicken auf `Next`.
8. In "Code Assistance Configuration" ist keine spezielle Aktion notwendig. Weiter mit `Next`.
9. Im letzten Dialog "Project Name and Location" wird der Projektname für Netbeans und der Ort des Projektes angegeben. Hier ist es sinnvoll den gleichen Namen in `Project Name` einzutragen wie derjenige, der in der `CMakeLists.txt` schon vergeben worden ist.

   Als Projektverzeichnis `Project Location` empfehle ich das **übergeordnete** Verzeichnis des CMake-Projektes selber anzulegen. Netbeans wird daraufhin in dem CMake-Projekt-Verzeichnis das Verzeichnis `nbproject` anlegen.
10. Wird das Verzeichnis mit einem Versionsverwaltungssystem verwaltet, dann ist es sinnvoll sowohl das `build` Verzeichnis als auch das `nbproject` Verzeichnis zu den ignorierten Dateien hinzuzufügen. Das hat **außerhalb** von Netbeans zu geschehen. Z.B. indem man, wenn man Mercurial verwendet, in die Datei `.hgignore` die beiden Verzeichnisse `build` und `nbproject` mit anführt.

    Fertig!

    Achtung! Für Netbeans ist die Compileroption `-no-pie` anzugeben:

    ```
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14 -no-pie")
    ```

26 Java verwenden

CMake kann auch mit anderen Programmiersprachen als `C` und `C++` verwendet werden (siehe auch Abschnitt Spezifizieren von Projektinformationen).

Um aus Java-Quelldateien eine ausführbare JAR-Datei zu machen ist folgendermaßen vorzugehen:

```
cmake_minimum_required(VERSION 3.5)
project(hello_java LANGUAGES Java)

find_package(Java 1.8 REQUIRED COMPONENTS Development)
include(UseJava)

add_jar(hello HelloWorld.java)
```

Die folgenden Punkte sind wichtig:

- Hiermit wird mittels dem Kommando `project` die Sprache Java festgelegt.
- Mittels `find_package` wird nach der Java-Unterstützung gesucht. Diese soll in der mindestens in der Version 1.8 vorliegen und ist notwendig (`REQUIRED`). Gefordert ist speziellen die Entwicklungsvariante (`COMPONENTS Development`) vorliegen. Will man nur Java-Programme ausführen, dann wäre `Runtime` anstatt `Development` ausreichend.
- Durch `include(UseJava)` wird die gesuchte (und gefundene) Java-Unterstützung aktiviert.

- Am Ende wird mittels `add_jar` wiederum ein Target erzeugt, sodass ein JAR `hello.jar` erzeugt wird, das aus der übersetzten Klasse `HelloWorld` besteht.

Nach dem Übersetzen steht `hello.jar` zur Verfügung und kann ganz "normal" gestartet werden:

```
java -cp hello.jar HelloWorld
```

Will man ein direkt ausführbares JAR erzeugen, dann ist die ausführbare Klasse anzugeben. Dies kann so erreicht werden, dass das Kommando `add_jar` folgendermaßen abgeändert wird:

```
add_jar(hello HelloWorld.java ENTRY_POINT HelloWorld)
```

Damit kann das JAR direkt gestartet werden (auch mittels Doppelklick im Dateimanager, wenn dies unterstützt wird):

```
java -jar hello.jar
```

Bemerkung: Natürlich muss immer der gesamte Klassenname angegeben werden. Wenn die Klasse im Paket `com.example` liegt, dann ist der vollständige Klassenname `com.example.HelloWorld`. Damit ist als Einsprungspunkt `com/example/HelloWorld` anzugeben.

Besteht das Projekt aus vielen Java-Dateien, dann kann genauso verfahren werden, wie auch bei `C++` Projekten vorgegangen wird:

```
file(GLOB_RECURSE SOURCES "src/*.java")

add_jar(hello ${SOURCES} ENTRY_POINT HelloWorld)
```

Will man einzelne Compileroptionen für `javac` setzen, dann funktioniert dies durch Setzen einer Variable folgendermaßen:

```
set(CMAKE_JAVA_COMPILE_FLAGS -Xlint:unchecked)
```

In diesem Fall wird eine einzelne Compileroption gesetzt, die bewirkt, dass der Compiler zusätzliche Detailinformationen zu den Warnungen ausgibt, die angeben, dass der Compiler nicht in der Lage ist eine typsichere Konvertierung sicherzustellen (so etwas will man nicht...).

27 Java mit Unit-Tests

CMake bietet von Haus aus Unterstützung an, um Tests in die Build-Umgebung einzubinden.

Verwendet man die Bibliothek Junit 4 dann kann das folgendermaßen aussehen:

```
CMakeLists.txt
JUnit.cmake
src
  Hello.java
  HelloWorld.java
tests
  hamcrest-core-x.x.jar
  junit-4.xx.jar
  TestHelloWorld.java
```

Schauen wir uns diese Verzeichnishierarchie an und beginnen mit dem einfachen Teil, nämlich dem Verzeichnis `src`. Dieses enthält den Code, der zu testen ist. Das ist in unserem Fall die Klasse `Hello`:

```
public class Hello {
    String message() {
        return "Hello, World";
```

```
    }

    String message(String guy) {
        return "Hello, " + guy;
    }
}
```

Die Klasse `HelloWorld.java` ist die eigentliche Applikation, die für unsere Testsituation eigentlich unwichtig ist, aber der Vollständigkeit halber hier angegeben wird:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(new Hello().message());
    }
}
```

Im Verzeichnis `tests` befindet sich der Programmcode für das Testprogramm, das in unserem Fall folgendermaßen aussieht:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class TestHelloWorld {
    private Hello hello;

    @Before
    public void setUp() {
        hello = new Hello();
    }

    @Test
    public void test_default_message() {
        assertEquals(hello.message(), "Hello, World");
    }

    @Test
    public void test_custom_message() {
        assertEquals(hello.message("Bob"), "Hello, Bob");
    }
}
```

Weiters befindet sich im Verzeichnis `tests` die eigentlichen Jar-Dateien für JUnit.

Im Wurzelverzeichnis unseres Projektes ist noch die Datei `JUnit.cmake` platziert, die die Funktionalität zum Testen mittels JUnit für CMake zur Verfügung stellt.

Jetzt fehlt nur mehr `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.5)
project(hello_java LANGUAGES Java)

enable_testing()

find_package(Java 1.8 REQUIRED COMPONENTS Development)
include(UseJava)
```

```
# Teil 1
file(GLOB_RECURSE SOURCES "src/*.java")

add_jar(hello ${SOURCES} ENTRY_POINT HelloWorld)

# Teil 2
# Teil 2a
get_target_property(jarFile hello JAR_FILE)

file(GLOB_RECURSE TESTS "tests/*.java")
set(ALL ${SOURCES} ${TESTS})

file(GLOB_RECURSE JUNIT_JAR_FILE "tests/junit*.jar")
file(GLOB_RECURSE HAMCREST_JAR_FILE "tests/hamcrest*.jar")

# Teil 2b
add_jar(helloTest ${ALL} INCLUDE_JARS ${JUNIT_JAR_FILE} ENTRY_POINT TestHelloWorld)
get_target_property(junitJarFile helloTest JAR_FILE)

add_test(NAME helloTest COMMAND ${Java_JAVA_EXECUTABLE} -cp .:${junitJarFile}:${JUNIT_JAR_FILE}
```

Der erste Teil (*Teil 1*) ist wieder ganz normal für die eigentliche Java Applikation, während der zweite Teil (*Teil 2*) für das Testen verantwortlich ist. Für jede Testklasse ist ein eigener *Teil 2b* zu kopieren und anzupassen!

Das CMake-Projekt wird wie üblich erzeugt. Danach steht weiters die Möglichkeit zur Verfügung die Tests mittels `ctest` oder auch (bei Verwendung von Unix Makefile: `make test`) zu starten:

```
$ ctest
Test project /home/knslnto/workspace/school/scripts/cmake_tutorial/hello_junit/build
    Start 1: helloTest
1/1 Test #1: helloTest .......................   Passed    0.21 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.21 sec
```

Mittels `ctest -V` kann der geschwätzige Modus von `ctest` aktiviert werden, sodass genauere Fehlermeldungen zur Verfügung stehen.

28 Java: Hinzufügen von externen JARs

Will man externe JAR-Dateien zu einem Projekt hinzufügen, dann geht man folgendermaßen vor, vorausgesetzt die externen `.jar` Dateien werden in einem Unterverzeichnis `extern` abgelegt:

```
cmake_minimum_required(VERSION 3.5)
project(hello_java LANGUAGES Java)

find_package(Java 1.8 REQUIRED COMPONENTS Development)
include(UseJava)
```

```
file(GLOB_RECURSE EXTERN_JARS "extern/*.jar")
add_jar(hello ${SOURCES} INCLUDE_JARS ${EXTERN_JARS} ENTRY_POINT HelloWorld)
```

Damit werden die externen jar-Dateien in der `INCLUDE_JARS` Klausel *nur* zum Compilieren der Java-Dateien herangezogen. Das heißt, dass die Class-Dateien schon im Classpath enthalten sein müssen. Wenn dem nicht so ist, dann ist die Situation in Java mit den jar-Dateien etwas kompliziert. Ein Lösung dafür ist, dass man alle jar-Dateien zusammenpackt. Dies ist im Abschnitt Java: Packen von jar-Dateien für den Einsatz beschrieben.

29 Java: Packen von jar-Dateien für den Einsatz

Um Java-Programme wirklich verteilen zu können, müssen alle Artefakte zusammengeführt werden und gemeinsam verteilt werden. Dazu eignet sich das Tool packr, das jar-Dateien und Ressourcen samt einer JVM für Windows, Linux und macOS in einem Verzeichnis bereitstellen kann und auch ein entsprechendes ausführbares Programm erstellt.

Für ein entsprechendes CMake-Projekt schlage ich folgende Struktur vor, das als externe jar-Datei den sqlite3-Treiber von xerial einbindet und von dieser Quelle auch das Beispielprogramm `HelloJDBC.java` verwendet:

```
hello_java_jar
  CMakeLists.txt
  src
    HelloJDBC.java
  tools
    packr.jar
  extern
    sqlite-jdbc-3.21.0.jar
  build
    bin
```

Im Verzeichnis `build/bin` befindet sich danach das ausfürhbare Programm `hello` samt allen notwendigen jar-Dateien. Dieses Verzeichnis kann an eine beliebige Stelle kopiert oder verteilt werden.

Und hier ist die entsprechende Datei `CMakeLists.txt`, die so von mir erstellt wurde, dass nur der Anfang der Datei verändert werden muss:

```
cmake_minimum_required(VERSION 3.5)
project(hello_java_jar LANGUAGES Java)

# begin_of_changes: change as appropriate!

set(TARGET_NAME "hello")  # name used for naming the jar-file and the executable
set(TARGET_MAIN_CLASS "HelloJDBC")  # name used for naming the main class
# platform: windows32, windows64, linux32, linux64, or mac
set(PLATFORM "linux32")
# will be used to create the directory "bin"
set(JDK_HOME "/usr/lib/jvm/java-8-openjdk/")

find_package(Java 1.8 REQUIRED COMPONENTS Development)
include(UseJava)

file(GLOB EXTERN_JARS "extern/*.jar")
```

```
# find packr jar: https://github.com/libgdx/packr
find_jar(PACKR_JAR packrjar "packr" PATHS "tools")

# end_of_changes

################################################################
# don't touch the following part unless you're really brave #
################################################################

file(GLOB SOURCES "src/*.java")

# build jar for the application
add_jar(${TARGET_NAME}
  SOURCES ${SOURCES}
  INCLUDE_JARS ${EXTERN_JARS}
  ENTRY_POINT ${TARGET_MAIN_CLASS})
get_target_property(TARGET_JAR_FILE ${TARGET_NAME} JAR_FILE)

# build packed application to directory "bin"
add_custom_command(
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/bin/${TARGET_NAME}"
  COMMAND ${CMAKE_COMMAND} -E remove_directory bin
  COMMAND java -jar ${PACKR_JAR} --platform ${PLATFORM} --jdk ${JDK_HOME}
    --executable ${TARGET_NAME} --classpath ${TARGET_NAME}.jar
    ${EXTERN_JARS} --mainclass ${TARGET_MAIN_CLASS} --output bin
  DEPENDS "${TARGET_JAR_FILE}"
)

add_custom_target(
  ${TARGET_NAME}_bin
  ALL DEPENDS "${CMAKE_CURRENT_BINARY_DIR}/bin/${TARGET_NAME}"
)
```

30 Java: Hinzufügen von Ressourcen

Wird die Applikation als `jar` Datei gepackt und werden Ressourcen verwendet, dann müssen die verwendeten Ressourcen auch zur `.jar` Datei hinzugefügt werden. Das passiert folgendermaßen, wenn alle Ressourcen im Verzeichnis `resources` abgelegt werden:

```
file(GLOB RESOURCES "resources/*")

add_jar(${TARGET_NAME}
  SOURCES ${SOURCES} ${RESOURCES}
  ...
)
```

31 Verwenden von LaTeX

Um LaTeXmit CMake vernünftig verwenden zu können, ist ein weiteres Paket notwendig.

Lade daher von `http://public.kitware.com/Wiki/CMakeUserUseLATEX` die Datei `UseLATEX.cmake` herunter und kopiere diese in dein Projektverzeichnis. Wichtig: Bei Verwendung von `UseLATEX.cmake` ist ein out-of-source build zwingend notwendig!

```
cmake_minimum_required(VERSION 3.5)
project(mydoc NONE)

find_package(LATEX REQUIRED COMPONENTS XELATEX)  # empfohlen
set(PDFLATEX_COMPILER ${XELATEX_COMPILER})  # empfohlen
include(UseLATEX.cmake)

add_latex_document(src/mini_xelatex.tex)
```

Bei `project` ist zwingend `NONE` anzugeben, da es sich weder um ein `C++` noch um Java,... handelt.

Ich empfehle XeLaTeXzu verwenden, wenn nicht gewünscht, dann werden die entsprechende Zeilen einfach weggelassen.

Die Verwendung von Bibliographie, Index,... ist in `UseLATEX.pdf` nachzulesen!

32 Setzen von Versionsstring (und dgl.)

Angenommen man will eine Versionsbezeichnung in den Quelldateien zentral konfigurieren, dann kann man diese in der `CMakeLists.txt` mittels `configure_file` wie folgt definieren:

```
cmake_minimum_required(VERSION 3.9)
project(version)

set (VERSION v1.0)  # hier wird die Version festgelegt

file(GLOB_RECURSE SOURCES "src/*.cpp")

# aus 'config.h.in' soll 'config.h' werden
configure_file (
  "src/config.h.in"
  "${PROJECT_BINARY_DIR}/config.h"
  )

include_directories(${CMAKE_CURRENT_BINARY_DIR})

add_executable(version ${SOURCES})
```

In `config.h.in` kann dann folgendes geschrieben stehen:

```
#ifndef CONFIG_H
#define CONFIG_H

#include <string>

const std::string version = "@VERSION@";

#endif
```

Alle durch `@` eingeschlossenen Strings werden ersetzt! `VERSION` ist in der `CMakeLists.txt` gesetzt worden.

Das Programm kann dann folgendermaßen aussehen:

```
#include <string>
#include <iostream>

#include "config.h"

using namespace std;

int main() {
    cout << version << endl;
}
```

33 Auslesen der Versionsinformationen aus Mercurial

Meist ist es sinnvoller die Versionsinformation aus einem Versionsverwaltungssystem auszulesen. CMake bietet auch Unterstützung für Mercurial an.

Die notwendigen Anpassungen für `CMakeLists.txt` sind hier zu finden:

```
cmake_minimum_required(VERSION 3.9)
project(version)

file(GLOB_RECURSE SOURCES "src/*.cpp")

find_package(Hg REQUIRED)
if(HG_FOUND)
  HG_WC_INFO(${PROJECT_SOURCE_DIR} Project)
  set(REVISION ${Project_WC_REVISION})
  set(CHANGESET ${Project_WC_CHANGESET})
endif()

configure_file(
  "src/config.h.in"
  "${PROJECT_BINARY_DIR}/config.h"
  )

include_directories(${CMAKE_CURRENT_BINARY_DIR})

add_executable(version ${SOURCES})
```

Die entsprechende Datei `config.h.in` sieht dann folgendermaßen aus:

```
#ifndef CONFIG_H
#define CONFIG_H

#include <string>

const std::string revision = "@REVISION@";
const std::string changeset = "@CHANGESET@";

#endif
```

Und verwendet kann dies folgenderweise werden.

```cpp
#include <string>
#include <iostream>

#include "config.h"

using namespace std;

int main() {
    cout << "revision: " << revision << endl;
    cout << "changeset: " << changeset << endl;
}
```

Nach jedem Commit und anschließenden Builden (z.B. mittels `make`) werden die relevanten Versionsinformationen aus dem Repository in das Executable einkompiliert.