- Kapitel 3 -

Der Einstieg in CMake

Jetzt gehts los

Im letzten Kapitel habe ich gezeigt, wie man über die Kommandozeile ein ausführbares Programm mit der Hilfe von CMake (und einem passenden Build-System) erzeugen kann. In diesem Kapitel will ich nun genauer auf die Syntax des CMake-Scripts *CMakeLists.txt* eingehen und welche zusätzlichen Keywords Sie in den Befehlen verwenden können.

3.1. Die erste CMakeLists.txt – Es reichen drei Befehle

Das Hello-World-Programm aus dem vorigen Kapitel lässt sich mit der folgenden *CMakeLists.txt* erzeugen.

Code 3.1: *CMakeLists.txt* für das Hello-World-Programm

```
1 cmake_minimum_required(VERSION 3.7)
2
3 project(
4   code_3-1
5   LANGUAGES CXX
6   )
7
8 add_executable(
9   hello_world
```

```
10      main.cpp
11      )
```

Insgesamt werden nur drei Befehle in dieser *CMakeLists.txt*-Datei benötigt:

- **cmake_minimum_required**() in Zeile 1
- **project**() in Zeile 3–6
- **add_executable**() in Zeile 8–11

Diese Befehle gehen ich jetzt einmal Schritt für Schritt durch, wobei ich mich erst einmal nur auf die hier dargestellten Versionen der Befehle konzentriere. Das heißt, ich betrachte erst einmal nur die in Code 3.1 verwendeten Keywords. In Kapitel 3.3 gehe ich dann noch einmal genauer auf die drei genannten Befehle ein.

Codeschnipsel 3.1: Der **cmake_minimum_required**()-Befehl

```
1  cmake_minimum_required(VERSION <min>)
```

CMake wird beständig aktualisiert und erweitert, um neue Werkzeuge, Plattformen und Funktionen unterstützen zu können. Die Abwärtskompatibilität von CMake wird mit jeder neuen Version aufrechterhalten, sodass Benutzer, die eine neuere Version von CMake nutzen, ältere Projekte wie zuvor erstellen können. Manchmal muss ein bestimmtes CMake-Verhalten geändert werden oder es werden strengere Prüfungen und Warnungen in neueren Versionen eingeführt. Anstatt von allen CMake-Projekten zu verlangen, dass sie sich sofort damit auseinandersetzen, stellt CMake sogenannte Policies zur Verfügung. Der Begriff Policy kann in etwa mit „Richtlinien“ übersetzt werden. Wenn eine neue Funktionalität oder ein neues Verhalten in CMake implementiert wird, dann gibt es dazu auch eine entsprechende Policy, die die neue Funktionalität oder das neue Verhalten definiert. Die Policies können dann von CMake intern so gesetzt werden, dass sich CMake automatisch wie eine bestimmte CMake-Version verhält. Dies erlaubt CMake, intern Fehler zu beheben und neue Funktionen einzuführen, aber trotzdem das zu erwartende Verhalten einer bestimmten älteren Version beizubehalten. In diesem Kontext übernimmt der Befehl **cmake_minimum_required**() zwei Aufgaben:

- Die CMake-Version hinter dem Keyword **VERSION** spezifiziert, welche CMake-Version mindestens benötigt wird, um diese *CMakeLists.txt*-Datei auszuführen. In Code 3.1 wäre das Version 3.7. Der Befehl **cmake_minimum_required**()

vergleicht die angegebene CMake-Version mit der CMake-Version auf Ihrem Computer. Falls die verwendete CMake-Version zu alt ist, im Falle von Code 3.1 Version 3.6 oder älter, so wird ein entsprechender Fehler ausgegeben. Das Keyword **VERSION** ist nicht optional und muss daher angegeben werden.

- Die korrekten Policies für die angegebene CMake-Version werden gesetzt und damit verhält sich CMake wie die im Befehl **cmake_minimum_required**() angegebene Version, egal welche Version nun genau auf Ihrem Rechner installiert ist. Das heißt, Sie können Ihre *CMake Lists.txt* auch ganz einfach mit verschiedenen CMake-Versionen testen, indem Sie einfach die Versionsnummer hinter dem Keyword **VERSION** ändern. In Code 3.1 würde sich CMake also wie die CMake-Version 3.7 verhalten, egal ob Sie Version 3.7, 3.11, 3.20 etc. auf Ihrem Rechner installiert haben.

Der Befehl **cmake_minimum_required**() sollte zu Beginn einer jeden *CMake Lists.txt*-Datei stehen, um die verwendete CMake-Version zu überprüfen und die korrekten Policies zu setzen. CMake gibt sogar eine Warnung aus, falls dieser Befehl fehlt oder nicht am Anfang der *CMake Lists.txt*-Datei steht.

Codeschnipsel 3.2: Der **project**()-Befehl

```
1 project(
2   <ProjectName>
3   [LANGUAGES <Programmiersprache1> ...]
4   )
```

Jede *CMake Lists.txt* sollte auch den **project**()-Befehl beinhalten, idealerweise direkt nach dem Befehl **cmake_minimum_required**(). Das erste Argument <ProjectName> innerhalb des **project**()-Befehls spezifiziert den Namen des CMake-Projektes. In Code 3.1 hat das CMake-Projekt den Namen code_3-1. Der Projektname wird insbesondere von Entwicklungsumgebungen genutzt, um Projekte innerhalb der Entwicklungsumgebung zu benennen. In diesem Buch benenne ich die Projekte anhand ihrer Nummerierung im Buch.
Mittels des optionalen Keywords **LANGUAGES** werden die Programmiersprachen <Programmiersprache1> usw. angegeben, die zum Erstellen dieses Projektes nötig sind. Falls das Keyword **LANGUAGES** weggelassen wird, werden automatisch C und

CXX (also C++) als verwendete Programmiersprachen gesetzt. Da C in diesem CMake-Projekt (und auch in allen weiteren in diesem Buch) nicht benötigt wird, wurde das **LANGUAGES**-Keyword verwendet, um lediglich C++ als verwendete Programmiersprache zu setzen. CMake überprüft bei Ausführung des **project**()-Befehls, ob entsprechende Compiler für die angegebenen Programmiersprachen installiert sind und zur Verfügung stehen. Zudem werden einige CMake-Variablen und -Eigenschaften entsprechend der gewählten Programmiersprachen gesetzt. Weitere mögliche Programmiersprachen sind zum Beispiel Assembler (ASM), Fortran, CUDA (ab CMake-Version 3.8) oder C# (CSharp, ab CMake-Version 3.8). Eine vollständige Liste unterstützter Programmiersprachen findet man unter anderem auf stackoverflow.com [47].

Codeschnipsel 3.3: Der **add_executable**()-Befehl

```
1 add_executable(
2   <DateiName>
3   [<SourceDatei1>] [<SourceDatei2> ...]
4   )
```

Mit dem Befehl **add_executable**() fügt CMake dem verwendeten Build-System die Anweisung hinzu, eine ausführbare Datei aus den angegebenen Source-Dateien <SourceDatei1>, <SourceDatei2> usw. zu erzeugen. Schauen Sie sich diesen Prozess bei Bedarf auch noch einmal in Abbildung 2.1 an. Als Erstes folgt der Name der ausführbaren Datei <DateiName> respektive hello_world im obigen Code 3.1. Mit dem angegebenen Namen wird diese ausführbare Datei innerhalb des CMake-Projektes identifiziert. Das bedeutet damit auch, dass dieser Name innerhalb des CMake-Projektes nur einmal verwendet werden darf und somit für CMake eindeutig zuzuordnen ist.
Dahinter folgen eine oder mehrere Source-Dateien <SourceDatei1>, <SourceDatei2> usw. Da in Code 3.1 die zu erzeugende ausführbare Datei lediglich aus einer Datei besteht, folgt nur die Source-Datei main.cpp.[1] Ab CMake 3.11 ist es auch erlaubt, die Source-Dateien wegzulassen und diese später mittels des Befehls **target_sources**() der ausführbaren Datei <DateiName> hinzuzufügen. Wird dies jedoch nicht getan, also wird versucht eine ausführbare Datei ohne Source-Dateien zu erzeugen, gibt CMake am Ende des Scripts einen entsprechenden Fehler aus. Mehr zum **target_sources**()-Befehl

[1] Im weiteren Verlauf des Buches wird es noch mehrere Beispiele mit mehr als einer Source-Datei geben.

finden Sie in Kapitel 5.1.
Der angegebene Name <DateiName> dient CMake als Grundlage für den Namen der erstellten Datei. Dieser kann, je nach System, jedoch etwas unterschiedlich ausfallen. So wird zum Beispiel auf Windows aus hello_world eine *hello_world.exe*-Datei und auf Linux-basierten Systemen wird die Datei *hello_world* ohne die Endung *.exe* erzeugt. Im Übrigen ist es kein Problem, mehrere ausführbare Dateien innerhalb eines Projektes zu erzeugen. Dazu muss lediglich der **add_executable**()-Befehl mehrfach verwendet werden.

Merkkasten

- Für ein einfaches CMake-Script sind bereits die folgenden drei Befehle ausreichend: **cmake_minimum_required**(), **project**() und **add_executable**().
- Der Befehl **cmake_minimum_required**() überprüft nicht nur die verwendete CMake-Version, sondern setzt auch die notwendigen Policies und sollte immer am Anfang eines CMake-Scriptes stehen.
- Der **project**()-Befehl sollte als zweiter Befehl folgen.
- Unter anderem überprüft der **project**()-Befehl, ob die entsprechenden Compiler für die ausgewählten Programmiersprachen zur Verfügung stehen.
- Mit dem Befehl **add_executable**() können ausführbare Dateien erzeugt werden.

3.2. Kommentare – Auch in CMake wichtig

Kommentare gibt es in fast jeder Programmiersprache und auch CMake bildet da keine Ausnahme. Allgemein sollte CMake wie jede andere Programmiersprache behandelt werden, das heißt Code sollte entsprechend kommentiert werden, eine Versionskontrolle eingerichtet werden usw. Ich bin überzeugt davon, dass Kommentare stets auf Englisch erfolgen sollten, außer es handelt sich um kleinere private Projekte, die nicht von anderen Personen bearbeitet und/oder gelesen werden. Ich mache hier jedoch eine Ausnahme und werde die folgende CMake-Datei auf Deutsch kommentieren, da es sich