CMake-Bibliotheken - 7 verschiedene Arten und wie man sie erstellt

CodingWithMagga (Marco Schoos)

29. März 2023

Inhaltsverzeichnis

1 Nötige Vorkenntnisse **2**

2 Normale Bibliotheken **3**

2.1 Statische Bibliotheken 4
2.2 Dynamische Bibliotheken 4
2.3 Modulare Bibliotheken 4
2.4 Beispiel 1: Erstellen von normalen CMake-Bibliotheken 4
2.5 Beispiel 2: Vergleich statische und dynamische Boost-Bibliothek 6

3 Objekt-Bibliotheken **9**

3.1 Beispiel 3: Objekt-Bibliothek erstellen und verlinken 9

4 Interface-Bibliotheken **11**

4.1 Beispiel 4: Interface-Bibliothek als Interface für mehrere Targets 12

5 Imported-Bibliothek **13**

5.1 Beispiel 5: Importieren einer dynamischen Bibliothek 14

6 Alias-Bibliothek **16**

6.1 Beispiel 6: Erstellen und Verwenden einer Alias-Bibliothek 16

7 Zusammenfassung CMake-Bibliotheken **17**

8 Weitere Informationen **17**

Bei der Entwicklung eines C/C++ Projektes kommt man ab einer gewissen Größe des Projektes an den Punkt, in dem man sein Projekt in verschiedene einzelne Targets, also ausführbare Dateien und Bibliotheken, aufteilen möchte. In diesem Artikel gehe ich auf die verschiedene Arten von Bibliotheken ein, die mit CMake erstellt werden können und gebe dazu einige Beispiele. Als Basis für die Erstellung von CMake-Bibliotheken dient der Befehl `add_library()`, der, je nachdem welche Keywords verwendet werden, insgesamt sieben verschiedene Arten von Bibliotheken erzeugen kann. Den Code, den ich in den Beispielen dieses Artikels vorstelle, könnt ihr gerne weiter verwenden, ihr findet ihn dazu auf GitHub. Alle beschriebenen CMake-Befehle, -Variablen und -Properties sind mit einem Link zur entsprechenden Stelle in der CMake-Dokumentation hinterlegt.

1 Nötige Vorkenntnisse

- CMake Grundkenntnisse: Dazu gehört etwa die Erstellung und Kompilierung eines ausführbaren Programms mit CMake. Ihr solltet dahin gehend auch mit den CMake-Befehlen `cmake_minimum_required()`, `project()` und `add_executable()` vertraut sein. Solltet ihr nicht wissen, wie ihr ein Programm mit CMake erstellt und kompiliert, schaut euch am besten zuvor diesen Artikel oder dieses YouTube-Video von mir an.
- CMake-Variablen: CMake-Variablen stelle ich sowohl in einem Artikel als auch in einem YouTube-Video vor. Zudem findet sich eine Erklärung zu CMake-Variablen auch in der CMake-Dokumentation.
- CMake-Properties: In diesem Video auf YouTube gehe ich auf Properties in CMake ein. Zudem findet man eine Auflistung von Properties in der CMake-Dokumentation, eine genauere Erläuterung, was Properties eigentlich sind, fehlt hier jedoch.
- If-Verzweigungen: If-Verzweigungen in CMake unterscheiden sich generell nicht von if-Verzweigungen in anderen Programmiersprachen und sollten daher wohl den meisten Lesern geläufig sein. If-Verzweigungen in CMake erwähne ich kurz in diesem YouTube-Video, ansonsten hilft hier natürlich auch wieder die CMake-Dokumentation weiter.
- Ich verwende zudem häufig die CMake-Befehle `target_include_directories()` und `target_link_libraries()`. Wenn euch diese Befehle nicht geläufig sind, liest am besten den ersten Abschnitt über normale Bibliotheken, bis ihr zum ersten Beispiel gelangt. Bevor ihr euch das Beispiel anschaut, informiert euch zuerst über diese beiden Befehle. Weitere Informationen findet ihr dann an den folgenden Stellen:
  - `target_include_directories()`: in der CMake-Dokumentation oder in einem YouTube-Video
  - `target_link_libraries()`: in der CMake-Dokumentation oder in einem YouTube-Video
- Der `find_package()`-Befehl: Im Grunde ist es für diesen Artikel ausreichend zu wissen, dass durch diesen Befehl externer Code in CMake eingebunden werden kann. Wer dennoch weitere Informationen benötigt, findet diese wie immer in der CMake-Dokumentation oder in diesem YouTube-Video.

- An einer Stelle verwende ich Generator Expressions. Genauere Kenntnisse sind dabei nicht unbedingt erforderlich. Mehr Informationen zu Generator Expressions findet ihr in der CMake-Dokumentation.
- Alle hier benötigten Vorkenntnisse findet ihr auch in meinem Buch CMake für Einsteiger.

2 Normale Bibliotheken

Unter dem Begriff „normale“ Bibliotheken fasse ich an dieser Stelle statische und dynamische Bibliotheken zusammen. Zusätzlich gibt es CMake-Bibliotheken, die dazu gedacht sind, zur Laufzeit eingebunden zu werden. Auf diese drei Arten von Bibliotheken gehe ich auch in diesem YouTube-Video ein. Sie können mit dem folgenden add_library()-Befehl erstellt werden:

```
1 add_library (
2     <BibliotheksName>
3     [STATIC|SHARED|MODULE]
4     [EXCLUDE_FROM_ALL]
5     [<SourceDatei1> <SourceDatei2> ...]
6 )
```

Das erste Argument des add_library()-Befehls ist der Name der Bibliothek <BibliotheksName>. Dieser kann frei gewählt werden, muss aber einzigartig innerhalb des CMake-Projektes sein, denn unter diesem Namen wird die Bibliothek im weiteren Verlauf des CMake-Scriptes angesprochen. Der Name der Bibliothek ist auch Grundlage für die Benennung der erzeugten Bibliotheksdatei auf dem System. Da sich die Namensgebung je nach Art der Bibliothek unterscheidet, komme ich später noch einmal darauf zurück.

Wenn man bei der Kompilierung des CMake-Projektes kein spezifisches Target angibt, so wird das hypothetische Target „all“ gebaut, zu dem standardmäßig alle ausführbaren Dateien und Bibliotheken hinzugefügt werden. Wird das optionale Keyword EXCLUDE_FROM_ALL angegeben, so wird die Bibliothek nicht ins Target „all“ geschrieben und infolgedessen nicht automatisch erzeugt.

Am Ende des Befehls folgen die Source-Dateien <SourceDatei1>, <SourceDatei2> usw. aus denen die Bibliothek erstellt werden soll. Ab CMake 3.11 können die Source-Dateien optional weggelassen werden, wenn diese später durch den Befehl target_sources() hinzugefügt werden. Wenn man versucht, eine Bibliothek ohne Source-Dateien zu erstellen, gibt CMake am Ende einen entsprechenden Fehler aus.

Die Verwendung eines der drei Keywords STATIC (statisch), SHARED (dynamisch) oder MODULE (modular) bestimmt den Typ der Bibliothek, die CMake erstellt. Es kann nur eines dieser Keywords an dieser Stelle verwendet werden, symbolisiert durch das „oder“-Symbol |. Wird keines dieser Keywords verwendet, so entscheidet der Wert der CMake-Variablen BUILD_SHARED_LIBS, ob eine statische oder dynamische Bibliothek erstellt wird.

2.1 Statische Bibliotheken

Bei Verwendung des Keywords STATIC wird eine statische Bibliothek erzeugt. Wird eine statische Bibliothek mit einem Target verlinkt, so werden die benötigten Programmteile dieser Bibliothek in das Target kopiert. Die Datei, die aus diesem Target erzeugt wird, benötigt entsprechend mehr Speicher auf der Festplatte, als bei einer dynamischen Bibliothek. Das liegt daran, dass die benötigten Programmteile in der Regel mehr Speicher benötigen, als die Verlinkung zu einer dynamischen Bibliothek. Unter Windows ist der Standardname dieser Bibliothek <BibliotheksName>.lib während er auf Unix basierten Betriebssystemen lib<BibliotheksName>.a lautet.

2.2 Dynamische Bibliotheken

Bei Verwendung des Keywords SHARED wird eine dynamische Bibliothek erzeugt. Im Gegensatz zu einer statischen Bibliothek werden beim Verlinken eines Targets mit einer dynamischen Bibliothek die benötigten Programmteile lediglich verlinkt und erst zur Laufzeit eingebunden bzw. geladen. Damit ist das erstellte Target in der Regel kleiner als ein äquivalentes Target, das mit einer statischen Bibliothek verlinkt wurde. Unter Windows ist der Standardname einer dynamischen Bibliothek <BibliotheksName>.dll, während er auf macOS lib<BibliotheksName>.dylib und auf anderen Unix basierten Betriebssystemen lib<BibliotheksName>.so lautet.

2.3 Modulare Bibliotheken

Das MODULE-Keyword erzeugt eine Bibliothek, die einer dynamischen Bibliothek sehr ähnlich ist. Jedoch ist die Intention bei einer solchen Bibliothek, dass diese als eine Art Plug-in zur Laufzeit geladen wird und nicht direkt mit dem ausführbaren Programm verlinkt wird.

2.4 Beispiel 1: Erstellen von normalen CMake-Bibliotheken

In diesem Beispiel erstelle ich einmal jede der drei oben genannten „normalen“ CMake-Bibliotheken.

```
1  cmake_minimum_required(VERSION 3.7...3.22)
2
3  project(normal_libs LANGUAGES CXX)
4
5  set(CMAKE_WINDOWS_EXPORT_ALL_SYMBOLS TRUE)
6
7  set(LIB_SOURCES
8      src/mathe/rechteck.cpp
9      src/mathe/quadrat.cpp
10 )
11
12 add_library(mathe_shared SHARED ${LIB_SOURCES})
13 add_library(mathe_static STATIC ${LIB_SOURCES})
14 add_library(mathe_module MODULE ${LIB_SOURCES})
```