Kostenloses eBook

LERNEN

cmake

Free unaffiliated eBook created from
Stack Overflow contributors.

#cmake

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit cmake ... 2

Bemerkungen ... 2

Versionen ... 2

Examples ... 4

CMake-Installation ... 4

Wechseln zwischen Build-Typen, z. B. Debug und Release ... 4

Einfaches "Hello World" -Projekt ... 5

"Hello World" mit mehreren Quelldateien ... 6

"Hello World" als Bibliothek ... 7

Kapitel 2: Benutzerdefinierte Build-Schritte ... 9

Einführung ... 9

Bemerkungen ... 9

Examples ... 9

Qt5 dll kopie beispiel ... 9

Ein benutzerdefiniertes Ziel ausführen ... 10

Kapitel 3: CMake-Integration in GitHub-CI-Tools ... 12

Examples ... 12

Konfigurieren Sie Travis CI mit dem Standard-CMake ... 12

Konfigurieren Sie Travis CI mit dem neuesten CMake ... 12

Kapitel 4: Datei konfigurieren ... 14

Einführung ... 14

Bemerkungen ... 14

Examples ... 15

Generieren Sie eine C ++ - Konfigurationsdatei mit CMake ... 15

Beispiel basiert auf SDL2-Steuerungsversion ... 15

Kapitel 5: Erstellen Sie Testsuiten mit CTest ... 18

Examples ... 18

Basic Test Suite ... 18

Kapitel 6: Fügen Sie dem Compiler Include Path Verzeichnisse hinzu ... 19

Syntax ... 19
Parameter ... 19
Examples ... 19
Fügen Sie das Unterverzeichnis eines Projekts hinzu ... 19
Kapitel 7: Funktionen und Makros ... 21
Bemerkungen ... 21
Examples ... 21
Einfaches Makro zum Definieren einer Variablen basierend auf der Eingabe ... 21
Makro, um eine Variable mit dem angegebenen Namen zu füllen ... 21
Kapitel 8: Hierarchisches Projekt ... 23
Examples ... 23
Einfacher Ansatz ohne Pakete ... 23
Kapitel 9: Kompilieren Sie Funktionen und die C / C ++ - Standardauswahl ... 25
Syntax ... 25
Examples ... 25
Funktionsanforderungen kompilieren ... 25
C / C ++ - Versionsauswahl ... 25
Kapitel 10: Konfigurationen erstellen ... 27
Einführung ... 27
Examples ... 27
Festlegen einer Release / Debug-Konfiguration ... 27
Kapitel 11: Projekte verpacken und verteilen ... 28
Syntax ... 28
Bemerkungen ... 28
Examples ... 28
Erstellen eines Pakets für ein erstelltes CMake-Projekt ... 28
Auswahl eines zu verwendenden CPack Generators ... 29
Kapitel 12: Suchen und verwenden Sie installierte Pakete, Bibliotheken und Programme ... 30
Syntax ... 30
Parameter ... 30
Bemerkungen ... 30

Examples ... 30
Verwenden Sie find_package und find .cmake-Module ... 31
Verwenden Sie pkg_search_module und pkg_check_modules ... 31
Kapitel 13: Testen und debuggen ... 33
Examples ... 33
Allgemeiner Ansatz zum Debuggen beim Erstellen mit Make ... 33
Lassen Sie CMake ausführliche Makefiles erstellen ... 33
Debuggen Sie find_package () -Fehler ... 33
CMake intern unterstütztes Paket / Modul ... 34
CMake-fähiges Paket / Bibliothek ... 34
Kapitel 14: Variablen und Eigenschaften ... 36
Einführung ... 36
Syntax ... 36
Bemerkungen ... 36
Examples ... 36
Zwischengespeicherte (globale) Variable ... 36
Lokale Variable ... 36
Streicher und Listen ... 37
Variablen und der Cache für globale Variablen ... 37
Anwendungsfälle für zwischengespeicherte Variablen ... 38
Hinzufügen von Profilierungsflags zu CMake, um gprof zu verwenden ... 39
Kapitel 15: Verwenden von CMake zum Konfigurieren von Pre-Prozessor-Tags ... 41
Einführung ... 41
Syntax ... 41
Bemerkungen ... 41
Examples ... 41
Verwenden von CMake zum Definieren der Versionsnummer für die Verwendung von C ++ ... 41
Kapitel 16: Ziele erstellen ... 43
Syntax ... 43
Examples ... 43
Ausführbare Dateien ... 43

Bibliotheken 43
Credits 45

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cmake

It is an unofficial and free cmake ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cmake.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit cmake

Bemerkungen

CMake ist ein Tool zum Definieren und Verwalten von Code-Builds, hauptsächlich für C ++.

CMake ist ein plattformübergreifendes Tool. Die Idee besteht darin, eine einzige Definition für die Erstellung des Projekts zu haben, die in spezifische Build-Definitionen für jede unterstützte Plattform umgesetzt wird.

Dies wird durch die Kombination mit verschiedenen plattformspezifischen Buildsystemen erreicht. CMake ist ein Zwischenschritt, der Build-Input für verschiedene spezifische Plattformen generiert. Unter Linux generiert CMake Makefiles. Unter Windows können Visual Studio-Projekte usw. erstellt werden.

Das Build-Verhalten wird in `CMakeLists.txt` Dateien definiert - eine in jedem Verzeichnis des Quellcodes. Die `CMakeLists` Datei jedes Verzeichnisses definiert, was das Buildsystem in diesem bestimmten Verzeichnis tun soll. Es definiert auch, welche Unterverzeichnisse CMake behandeln soll.

Typische Aktionen umfassen:

- Erstellen Sie aus einigen Quelldateien in diesem Verzeichnis eine Bibliothek oder eine ausführbare Datei.
- Fügen Sie dem während des Builds verwendeten Include-Pfad einen Dateipfad hinzu.
- Definieren Sie Variablen, die das Buildsystem in diesem Verzeichnis und in seinen Unterverzeichnissen verwendet.
- Generieren Sie eine Datei basierend auf der spezifischen Build-Konfiguration.
- Suchen Sie eine Bibliothek, die sich irgendwo im Quellbaum befindet.

Die endgültigen `CMakeLists` Dateien können sehr klar und unkompliziert sein, da sie jeweils nur einen begrenzten Umfang haben. Jeder verarbeitet nur so viel des Builds, wie im aktuellen Verzeichnis vorhanden ist.

Offizielle Informationen zu CMake finden Sie in der Dokumentation und dem Lernprogramm von CMake.

Versionen

Ausführung	Veröffentlichungsdatum
3.9	2017-07-18
3.8	2017-04-10
3.7	2016-11-11

Ausführung	Veröffentlichungsdatum
3.6	2016-07-07
3,5	2016-03-08
3.4	2015-11-12
3.3	2015-07-23
3.2	2015-03-10
3.1	2014-12-17
3,0	2014-06-10
2.8.12.1	2013-11-08
2.8.12	2013-10-11
2.8.11	2013-05-16
2.8.10.2	2012-11-27
2.8.10.1	2012-11-07
2.8.10	2012-10-31
2.8.9	2012-08-09
2.8.8	2012-04-18
2.8.7	2011-12-30
2.8.6	2011-12-30
2.8.5	2011-07-08
2.8.4	2011-02-16
2.8.3	2010-11-03
2.8.2	2010-06-28
2.8.1	2010-03-17
2.8	2009-11-13
2.6	2008-05-05

Examples

CMake-Installation

Besuchen Sie die CMake- Downloadseite, und erhalten Sie eine Binärdatei für Ihr Betriebssystem, z. B. Windows, Linux oder Mac OS X. Klicken Sie unter Windows doppelt auf die zu installierende Binärdatei. Führen Sie unter Linux die Binärdatei von einem Terminal aus.

Unter Linux können Sie die Pakete auch über den Paketmanager der Distribution installieren. Auf Ubuntu 16.04 können Sie die Befehlszeilen- und Grafikanwendung installieren mit:

```
sudo apt-get install cmake
sudo apt-get install cmake-gui
```

Auf FreeBSD können Sie die Befehlszeile und die Qt-basierte grafische Anwendung mit folgendem Befehl installieren:

```
pkg install cmake
pkg install cmake-gui
```

Wenn Sie unter Mac OSX einen der zur Verfügung stehenden Paketmanager verwenden, um Ihre Software zu installieren, insbesondere MacPorts (MacPorts) und Homebrew (Homebrew), können Sie CMake auch über einen dieser Programme installieren. Geben Sie im Fall von MacPorts beispielsweise Folgendes ein

```
sudo port install cmake
```

installiert CMake, während Sie den Homebrew-Paketmanager verwenden, den Sie eingeben

```
brew install cmake
```

Nachdem Sie CMake installiert haben, können Sie dies ganz einfach überprüfen

```
cmake --version
```

Sie sollten etwas Ähnliches wie das Folgende sehen

```
cmake version 3.5.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Wechseln zwischen Build-Typen, z. B. Debug und Release

CMake kennt mehrere Build-Typen, die normalerweise die Standard-Compiler- und Linker-Parameter (z. B. erstellte Debugging-Informationen) oder alternative Codepfade beeinflussen.

Standardmäßig kann CMake die folgenden Build-Typen verarbeiten:

- **Debug** : Normalerweise ein klassischer Debug-Build mit Debugging-Informationen, keine Optimierung usw.
- **Release** : Ihr typischer Release-Build ohne Debugging-Informationen und vollständige Optimierung.
- **RelWithDebInfo::** Wie *Release* , jedoch mit Debugging-Informationen.
- **MinSizeRel** : Ein spezieller *Release*- Build, der für die Größe optimiert wurde.

Wie Konfigurationen gehandhabt werden, hängt von dem verwendeten Generator ab.

Einige Generatoren (wie Visual Studio) unterstützen mehrere Konfigurationen. CMake generiert alle Konfigurationen gleichzeitig und Sie können aus der IDE oder mit `--config CONFIG` (mit `cmake --build`) `cmake --build` welche Konfiguration Sie erstellen möchten. Für diese Generatoren versucht CMake, eine Build-Verzeichnisstruktur so zu erstellen, dass Dateien aus verschiedenen Konfigurationen nicht aufeinanderfolgen.

Generatoren, die nur eine einzige Konfiguration unterstützen (wie Unix Makefiles), funktionieren anders. Hier wird die aktuell aktive Konfiguration durch den Wert der CMake-Variablen `CMAKE_BUILD_TYPE` .

Wenn Sie beispielsweise einen anderen Build-Typ auswählen möchten, können Sie die folgenden Befehlszeilenbefehle ausgeben:

```
cmake -DCMAKE_BUILD_TYPE=Debug path/to/source
cmake -DCMAKE_BUILD_TYPE=Release path/to/source
```

Ein CMake-Skript sollte es vermeiden, `CMAKE_BUILD_TYPE` selbst `CMAKE_BUILD_TYPE` , da dies in der Regel die Verantwortung des Benutzers darstellt.

Für Generatoren mit einer einzigen Konfiguration muss für die Konfiguration CMake erneut ausgeführt werden. Ein nachfolgender Build überschreibt wahrscheinlich Objektdateien, die von der vorherigen Konfiguration erstellt wurden.

Einfaches "Hello World" -Projekt

Bei einer C ++ - Quelldatei `main.cpp` , die eine `main()` Funktion definiert, wird *CMake* durch eine begleitende `CMakeLists.txt` Datei (mit folgendem Inhalt) aufgefordert, die entsprechenden Build-Anweisungen für das aktuelle System und den Standard-C ++ - Compiler zu generieren.

main.cpp *(C ++ Hello World-Beispiel)*

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

add_executable(app main.cpp)
```

Sehen Sie live auf Coliru

1. `cmake_minimum_required(VERSION 2.4)` legt eine minimale CMake-Version fest, die zum Auswerten des aktuellen Skripts erforderlich ist.

2. `project(hello_world)` startet ein neues CMake-Projekt. Dies löst eine Menge interner CMake-Logik aus, insbesondere die Erkennung des Standard-Compilers C und C ++.

3. Mit `add_executable(app main.cpp)` eine Build-Ziel- `app` erstellt, die den konfigurierten Compiler mit einigen Standardflags für die aktuelle Einstellung aufruft, um eine ausführbare `app` aus der angegebenen Quelldatei `main.cpp` zu kompilieren.

Befehlszeile *(In-Source-Build, nicht empfohlen)*

```
> cmake .
...
> cmake --build .
```

`cmake .` führt die Compiler-Erkennung aus, wertet die `CMakeLists.txt` in der angegebenen aus `.` Verzeichnis und generiert die Build-Umgebung im aktuellen Arbeitsverzeichnis.

Das `cmake --build .` Befehl ist eine Abstraktion für den erforderlichen Aufruf von build / make.

Befehlszeile *(Out-of-Source, empfohlen)*

Um den Quellcode von Build-Artefakten frei zu halten, sollten Sie Builds "out-of-source" ausführen.

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

Oder CMake kann auch die grundlegenden Befehle Ihrer Plattform-Shell vom obigen Beispiel abstrahieren:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

"Hello World" mit mehreren Quelldateien

Zuerst können wir die Verzeichnisse der Header-Dateien mit `include_directories()` angeben, dann müssen wir die entsprechenden Quelldateien der Zieldatei mit `add_executable()` und sicherstellen, dass in den Quelldateien genau eine `main()` Funktion vorhanden ist.

Es folgt ein einfaches Beispiel. Es wird angenommen, dass sich alle Dateien im Verzeichnis `PROJECT_SOURCE_DIR` .

main.cpp

```
#include "foo.h"

int main()
{
    foo();
    return 0;
}
```

foo.h

```
void foo();
```

foo.cpp

```
#include <iostream>
#include "foo.h"

void foo()
{
    std::cout << "Hello World!\n";
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_executable(app main.cpp foo.cpp)  # be sure there's exactly one main() function in the
source files
```

Wir können dasselbe Verfahren wie im obigen Beispiel befolgen, um unser Projekt zu erstellen. Dann wird die Ausführung der `app` gedruckt

```
>./app
Hello World!
```

"Hello World" als Bibliothek

Dieses Beispiel zeigt, wie Sie das Programm "Hello World" als Bibliothek bereitstellen und mit anderen Zielen verknüpfen.

Nehmen wir an, wir haben die gleichen Quell- / Header-Dateien wie im Beispiel http://www.riptutorial.com/cmake/example/22391/-hello-world--with-multiple-source-files . Anstatt aus mehreren Quelldateien zu erstellen, können wir zuerst `foo.cpp` als Bibliothek `add_library()`

indem Sie `add_library()` und anschließend mit dem Hauptprogramm mit `target_link_libraries()` .

Wir ändern **CMakeLists.txt** in

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_library(applib foo.cpp)
add_executable(app main.cpp)
target_link_libraries(app applib)
```

und nach den gleichen Schritten erhalten wir dasselbe Ergebnis.

Erste Schritte mit cmake online lesen: https://riptutorial.com/de/cmake/topic/862/erste-schritte-mit-cmake

Kapitel 2: Benutzerdefinierte Build-Schritte

Einführung

Benutzerdefinierte Erstellungsschritte sind hilfreich, um benutzerdefinierte Ziele in Ihrem Projekterstellungsvorgang auszuführen oder um Dateien einfach zu kopieren, sodass Sie sie nicht manuell ausführen müssen (vielleicht DLLs?). Hier zeige ich Ihnen zwei Beispiele. Das erste ist das Kopieren von DLLs (insbesondere Qt5-DLLs) in das Binärverzeichnis Ihres Projekts (entweder Debug oder Release) und das zweite ist das Ausführen eines benutzerdefinierten Ziels (in diesem Fall Doxygen) in Ihrer Lösung (wenn Sie Visual Studio verwenden).

Bemerkungen

Wie Sie sehen, können Sie mit benutzerdefinierten Erstellungszielen und Schritten in cmake eine Menge tun, aber Sie sollten vorsichtig damit umgehen, insbesondere beim Kopieren von DLLs. Dies ist zwar praktisch, kann aber manchmal zu etwas führen, das liebevoll "dll hell" genannt wird.

Im Grunde bedeutet dies, dass Sie sich verlaufen können, in welcher DLL-Datei Ihre ausführbare Datei tatsächlich davon abhängt, von welcher Datei sie geladen wird und welche sie ausführen muss (möglicherweise aufgrund der Pfadvariablen Ihres Computers).

Abgesehen von der oben genannten Einschränkung, können Sie benutzerdefinierte Ziele tun lassen, was Sie wollen! Sie sind leistungsstark und flexibel und ein unschätzbares Werkzeug für jedes cmake-Projekt.

Examples

Qt5 dll kopie beispiel

Nehmen wir an, Sie haben ein Projekt, das von Qt5 abhängt, und Sie müssen die entsprechenden DLLs in Ihr Build-Verzeichnis kopieren und möchten dies nicht manuell tun. Sie können folgendes tun:

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

# add the executable
add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
```

```
    )
```

Jedes Mal, wenn Sie Ihr Projekt erstellen und die Ziel-DLLs geändert haben, die Sie kopieren möchten, werden sie nach dem `copy_if_different` Ihres Ziels (in diesem Fall der Hauptprogrammdatei) kopiert (beachten Sie den Befehl `copy_if_different`). Andernfalls werden sie nicht kopiert.

Beachten Sie außerdem die Verwendung von Generatorausdrücken . Der Vorteil bei der Verwendung dieser Optionen ist, dass Sie nicht explizit angeben müssen, wo DLLs oder welche Varianten verwendet werden sollen. Um diese jedoch verwenden zu können, muss das von Ihnen verwendete Projekt (in diesem Fall Qt5) Ziele importiert haben.

Wenn Sie in Debug bauen, kann CMake (basierend auf dem importierten Ziel) die Dateien Qt5Cored.dll, Qt5Guid.dll und Qt5Widgetsd.dll in den Ordner Debug des Build-Ordners kopieren. Wenn Sie ein Release erstellen, werden die Release-Versionen der DLL-Dateien in den Release-Ordner kopiert.

Ein benutzerdefiniertes Ziel ausführen

Sie können auch ein benutzerdefiniertes Ziel erstellen, das ausgeführt werden soll, wenn Sie eine bestimmte Aufgabe ausführen möchten. Dies sind normalerweise ausführbare Dateien, die Sie ausführen, um verschiedene Aufgaben auszuführen. Etwas, das besonders nützlich sein kann, ist das Ausführen von Doxygen , um Dokumentation für Ihr Projekt zu erstellen. Dazu können Sie in Ihrer `CMakeLists.txt` Folgendes `CMakeLists.txt` (der Einfachheit halber setzen wir unser Qt5-Projektbeispiel fort):

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )

#Add target to build documents from visual studio.
set(DOXYGEN_INPUT ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile)
#set the output directory of the documentation
set(DOXYGEN_OUTPUT_DIR ${CMAKE_CURRENT_SOURCE_DIR}/docs)
# sanity check...
message("Doxygen Output ${DOXYGEN_OUTPUT_DIR}")
find_package(Doxygen)

if(DOXYGEN_FOUND)
    # create the output directory where the documentation will live
    file(MAKE_DIRECTORY ${DOXYGEN_OUTPUT_DIR})
```

```
    # configure our Doxygen configuration file. This will be the input to the doxygen
    # executable
    configure_file(${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in
    ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile @ONLY)

# now add the custom target. This will create a build target called 'DOCUMENTATION'
# in your project
ADD_CUSTOM_TARGET(DOCUMENTATION
  COMMAND ${CMAKE_COMMAND} -E echo_append "Building API Documentation..."
  COMMAND ${CMAKE_COMMAND} -E make_directory ${DOXYGEN_OUTPUT_DIR}
  COMMAND ${DOXYGEN_EXECUTABLE} ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile
    WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND ${CMAKE_COMMAND} -E echo "Done."
  WORKING_DIRECTORY ${DOXYGEN_OUTPUT_DIR})

endif(DOXYGEN_FOUND)
```

Wenn Sie nun unsere Lösung erstellen (vorausgesetzt, Sie verwenden Visual Studio), haben Sie ein Erstellungsziel namens `DOCUMENTATION` , das Sie erstellen können, um die Dokumentation Ihres Projekts neu zu erstellen.

Benutzerdefinierte Build-Schritte online lesen:
https://riptutorial.com/de/cmake/topic/9537/benutzerdefinierte-build-schritte

Kapitel 3: CMake-Integration in GitHub-CI-Tools

Examples

Konfigurieren Sie Travis CI mit dem Standard-CMake

Travis CI hat CMake 2.8.7 vorinstalliert.

Ein minimales `.travis.yml` Skript für einen Out-of-Source-Build

```
language: cpp

compiler:
  - gcc

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Konfigurieren Sie Travis CI mit dem neuesten CMake

Die auf Travis vorinstallierte CMake-Version ist sehr alt. Sie können die offiziellen Linux-Binärdateien verwenden , um eine neuere Version zu erstellen.

Hier ist ein Beispiel `.travis.yml` :

```
language: cpp

compiler:
  - gcc

# the install step will take care of deploying a newer cmake version
install:
  # first we create a directory for the CMake binaries
  - DEPS_DIR="${TRAVIS_BUILD_DIR}/deps"
  - mkdir ${DEPS_DIR} && cd ${DEPS_DIR}
  # we use wget to fetch the cmake binaries
  - travis_retry wget --no-check-certificate https://cmake.org/files/v3.3/cmake-3.3.2-Linux-
x86_64.tar.gz
```

```
  # this is optional, but useful:
  # do a quick md5 check to ensure that the archive we downloaded did not get compromised
  - echo "f3546812c11ce7f5d64dc132a566b749 *cmake-3.3.2-Linux-x86_64.tar.gz" > cmake_md5.txt
  - md5sum -c cmake_md5.txt
  # extract the binaries; the output here is quite lengthy,
  # so we swallow it to not clutter up the travis console
  - tar -xvf cmake-3.3.2-Linux-x86_64.tar.gz > /dev/null
  - mv cmake-3.3.2-Linux-x86_64 cmake-install
  # add both the top-level directory and the bin directory from the archive
  # to the system PATH. By adding it to the front of the path we hide the
  # preinstalled CMake with our own.
  - PATH=${DEPS_DIR}/cmake-install:${DEPS_DIR}/cmake-install/bin:$PATH
  # don't forget to switch back to the main build directory once you are done
  - cd ${TRAVIS_BUILD_DIR}

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

CMake-Integration in GitHub-CI-Tools online lesen:
https://riptutorial.com/de/cmake/topic/1445/cmake-integration-in-github-ci-tools

Kapitel 4: Datei konfigurieren

Einführung

configure_file ist eine CMake-Funktion zum Kopieren einer Datei an einen anderen Speicherort und zum Ändern des Inhalts. Diese Funktion ist sehr nützlich zum Erzeugen von Konfigurationsdateien mit Pfaden, benutzerdefinierten Variablen, unter Verwendung einer generischen Vorlage.

Bemerkungen

Kopieren Sie eine Datei an einen anderen Ort und ändern Sie ihren Inhalt.

```
configure_file(<input> <output>
           [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
           [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

Kopiert eine Datei in eine Datei und ersetzt die im Dateiinhalt referenzierten Variablenwerte. Wenn es sich um einen relativen Pfad handelt, wird er in Bezug auf das aktuelle Quellverzeichnis ausgewertet. Das muss eine Datei sein, kein Verzeichnis. Wenn es sich um einen relativen Pfad handelt, wird er in Bezug auf das aktuelle Binärverzeichnis ausgewertet. Wenn ein vorhandenes Verzeichnis benannt wird, wird die Eingabedatei mit ihrem ursprünglichen Namen in diesem Verzeichnis abgelegt.

Wenn die Datei geändert wird, führt das Buildsystem CMake erneut aus, um die Datei neu zu konfigurieren und das Buildsystem erneut zu generieren.

Dieser Befehl ersetzt alle Variablen in der Eingabedatei, auf die als $ {VAR} oder @ VAR @ verwiesen wird, durch ihre von CMake bestimmten Werte. Wenn eine Variable nicht definiert ist, wird sie durch nichts ersetzt. Wenn COPYONLY angegeben ist, findet keine Variablenerweiterung statt. Wenn ESCAPE_QUOTES angegeben ist, werden alle ersetzten Anführungszeichen im C-Stil gespeichert. Die Datei wird mit den aktuellen Werten der CMake-Variablen konfiguriert. Wenn @ONLY angegeben wird, werden nur Variablen der Form @ VAR @ ersetzt und $ {VAR} wird ignoriert. Dies ist nützlich zum Konfigurieren von Skripts, die $ {VAR} verwenden.

Eingabedateizeilen der Form "#cmakedefine VAR ..." werden entweder durch "#define VAR ..." oder / * #undef VAR * / ersetzt, abhängig davon, ob VAR in CMake auf einen Wert gesetzt ist, der nicht als falsch gilt konstant durch den if () Befehl. (Der Inhalt von "..." wird, falls vorhanden, wie oben verarbeitet.) Eingabedateizeilen der Form "# cmakedefine01 VAR" werden auf ähnliche Weise durch "#define VAR 1" oder "#define VAR 0" ersetzt.

Mit NEWLINE_STYLE konnte das Zeilenende angepasst werden:

```
'UNIX' or 'LF' for \n, 'DOS', 'WIN32' or 'CRLF' for \r\n.
```

COPYONLY darf nicht mit NEWLINE_STYLE verwendet werden.

Examples

Generieren Sie eine C ++ - Konfigurationsdatei mit CMake

Wenn wir ein C ++ - Projekt haben, das eine Konfigurationsdatei config.h mit einigen benutzerdefinierten Pfaden oder Variablen verwendet, können wir es mit CMake und einer generischen Datei config.h.in generieren.

Die config.h.in kann Teil eines git-Repositorys sein, während die generierte Datei config.h niemals hinzugefügt wird, da sie aus der aktuellen Umgebung generiert wird.

```
#CMakeLists.txt
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)

SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

SET(${PROJ_NAME}_DATA     ""     CACHE PATH "This directory contains all DATA and RESOURCES")
SET(THIRDPARTIES_PATH    ${CMAKE_CURRENT_SOURCE_DIR}/../thirdparties      CACHE PATH "This
directory contains thirdparties")

configure_file ("${CMAKE_CURRENT_SOURCE_DIR}/common/config.h.in"
            "${CMAKE_CURRENT_SOURCE_DIR}/include/config.h" )
```

Wenn wir eine config.h.in wie folgt haben:

```
cmakedefine PATH_DATA "@myproject_DATA@"
cmakedefine THIRDPARTIES_PATH "@THIRDPARTIES_PATH@"
```

Die vorherigen CMakeLists erzeugen einen C ++ - Header wie folgt:

```
#define PATH_DATA "/home/user/projects/myproject/data"
#define THIRDPARTIES_PATH "/home/user/projects/myproject/thirdparties"
```

Beispiel basiert auf SDL2-Steuerungsversion

Wenn Sie ein `cmake` Modul haben. Sie können einen Ordner erstellen, `in` dem alle Konfigurationsdateien gespeichert werden.

Sie haben beispielsweise ein Projekt namens `FOO` . Sie können eine Datei `FOO_config.h.in` wie `FOO_config.h.in` erstellen:

```
//=============================================================================
//  CMake configuration file, based on SDL 2 version header
// ============================================================================

#pragma once

#include <string>
#include <sstream>
```

```
namespace yournamespace
{
  /**
 *  \brief Information the version of FOO_PROJECT in use.
 *
 *  Represents the library's version as three levels: major revision
 *  (increments with massive changes, additions, and enhancements),
 *  minor revision (increments with backwards-compatible changes to the
 *  major revision), and patchlevel (increments with fixes to the minor
 *  revision).
 *
 *  \sa FOO_VERSION
 *  \sa FOO_GetVersion
 */
typedef struct FOO_version
{
    int major;        /**< major version */
    int minor;        /**< minor version */
    int patch;        /**< update version */
} FOO_version;

/* Printable format: "%d.%d.%d", MAJOR, MINOR, PATCHLEVEL
*/
#define FOO_MAJOR_VERSION   0
#define FOO_MINOR_VERSION   1
#define FOO_PATCHLEVEL      0

/**
 *  \brief Macro to determine FOO version program was compiled against.
 *
 *  This macro fills in a FOO_version structure with the version of the
 *  library you compiled against. This is determined by what header the
 *  compiler uses. Note that if you dynamically linked the library, you might
 *  have a slightly newer or older version at runtime. That version can be
 *  determined with GUCpp_GetVersion(), which, unlike GUCpp_VERSION(),
 *  is not a macro.
 *
 *  \param x A pointer to a FOO_version struct to initialize.
 *
 *  \sa FOO_version
 *  \sa FOO_GetVersion
 */
#define FOO_VERSION(x)                          \
{                                       \
    (x)->major = FOO_MAJOR_VERSION;                 \
    (x)->minor = FOO_MINOR_VERSION;                 \
    (x)->patch = FOO_PATCHLEVEL;                    \
}

/**
 *  This macro turns the version numbers into a numeric value:
 *  \verbatim
    (1,2,3) -> (1203)
    \endverbatim
 *
 *  This assumes that there will never be more than 100 patchlevels.
 */
#define FOO_VERSIONNUM(X, Y, Z)                     \
    ((X)*1000 + (Y)*100 + (Z))

/**
```

```
 *  This is the version number macro for the current GUCpp version.
 */
#define FOO_COMPILEDVERSION \
    FOO_VERSIONNUM(FOO_MAJOR_VERSION, FOO_MINOR_VERSION, FOO_PATCHLEVEL)

/**
 *  This macro will evaluate to true if compiled with FOO at least X.Y.Z.
 */
#define FOO_VERSION_ATLEAST(X, Y, Z) \
    (FOO_COMPILEDVERSION >= FOO_VERSIONNUM(X, Y, Z))

}

// Paths
#cmakedefine FOO_PATH_MAIN "@FOO_PATH_MAIN@"
```

Diese Datei erstellt eine `FOO_config.h` im Installationspfad mit einer in c `FOO_PATH_MAIN` definierten Variable aus der cmake-Variablen. Um es zu generieren, müssen Sie es `in` Datei in Ihrer CMakeLists.txt aufnehmen, wie folgt (Pfade und Variablen setzen):

```
MESSAGE("Configuring FOO_config.h ...")
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/common/in/FOO_config.h.in"
"${FOO_PATH_INSTALL}/common/include/FOO_config.h" )
```

Diese Datei enthält die Daten aus der Vorlage und die Variable mit Ihrem tatsächlichen Pfad. Beispiel:

```
// Paths
#define FOO_PATH_MAIN "/home/YOUR_USER/Respositories/git/foo_project"
```

Datei konfigurieren online lesen: https://riptutorial.com/de/cmake/topic/8304/datei-konfigurieren

Kapitel 5: Erstellen Sie Testsuiten mit CTest

Examples

Basic Test Suite

```
# the usual boilerplate setup
cmake_minimum_required(2.8)
project(my_test_project
        LANGUAGES CXX)

# tell CMake to use CTest extension
enable_testing()

# create an executable, which instantiates a runner from
# GoogleTest, Boost.Test, QtTest or whatever framework you use
add_executable(my_test
               test_main.cpp)

# depending on the framework, you need to link to it
target_link_libraries(my_test
                      gtest_main)

# now register the executable with CTest
add_test(NAME my_test COMMAND my_test)
```

Das Makro `enable_testing()` hat viel Magie. In erster Linie schafft sie eine eingebaute `test` (für GNU machen; `RUN_TESTS` für VS), die, wenn sie ausgeführt wird , *CTest* ausführt.

Der Aufruf von `add_test()` registriert schließlich eine beliebige ausführbare Datei bei *CTest. Daher* wird die ausführbare Datei *jedes Mal ausgeführt* , wenn wir das `test` aufrufen.

Erstellen Sie nun das Projekt wie gewohnt und führen Sie schließlich das Testziel aus

GNU Make	Visual Studio
`make test`	`cmake --build . --target RUN_TESTS`

Erstellen Sie Testsuiten mit CTest online lesen:
https://riptutorial.com/de/cmake/topic/4197/erstellen-sie-testsuiten-mit-ctest

Kapitel 6: Fügen Sie dem Compiler Include Path Verzeichnisse hinzu

Syntax

- include_directories ([NACH OBEN] [SYSTEM] dir1 [dir2 ...])

Parameter

Parameter	Beschreibung
`dirN`	ein oder mehrere relative oder absolute Pfade
`AFTER` , `BEFORE`	(optional) ob die angegebenen Verzeichnisse vor oder am Ende der aktuellen Liste der Include-Pfade eingefügt werden sollen; Das Standardverhalten wird von `CMAKE_INCLUDE_DIRECTORIES_BEFORE` definiert
`SYSTEM`	(optional) weist den Compiler an, die angegebenen Verzeichnisse als *System-Include-Verzeichnisse aufzurufen* , die eine spezielle Behandlung durch den Compiler auslösen können

Examples

Fügen Sie das Unterverzeichnis eines Projekts hinzu

Angesichts der folgenden Projektstruktur

```
include\
  myHeader.h
src\
  main.cpp
CMakeLists.txt
```

die folgende Zeile in der Datei `CMakeLists.txt`

```
include_directories(${PROJECT_SOURCE_DIR}/include)
```

fügt das `include` Verzeichnis zum *Include-Suchpfad* des Compilers für alle in diesem Verzeichnis definierten Ziele hinzu (und alle über `add_subdirectory()` enthaltenen Unterverzeichnisse).

Somit wird die Datei `myHeader.h` im Projekt `include` kann Unterverzeichnis über einbezogen werden `#include "myHeader.h"` in der `main.cpp` - Datei.

Fügen Sie dem Compiler Include Path Verzeichnisse hinzu online lesen:

https://riptutorial.com/de/cmake/topic/5968/fugen-sie-dem-compiler-include-path-verzeichnisse-hinzu

Kapitel 7: Funktionen und Makros

Bemerkungen

Der Hauptunterschied zwischen *Makros* und *Funktionen* besteht darin, dass *Makros* im aktuellen Kontext ausgewertet werden, während *Funktionen* einen neuen Bereich im aktuellen Kontext öffnen. Daher sind innerhalb von *Funktionen* definierte Variablen nach Auswertung der Funktion nicht bekannt. Im Gegensatz dazu werden Variablen innerhalb von *Makros* noch definiert, nachdem das Makro ausgewertet wurde.

Examples

Einfaches Makro zum Definieren einer Variablen basierend auf der Eingabe

```
macro(set_my_variable _INPUT)
  if("${_INPUT}" STREQUAL "Foo")
    set(my_output_variable "foo")
  else()
    set(my_output_variable "bar")
  endif()
endmacro(set_my_variable)
```

Verwenden Sie das Makro:

```
set_my_variable("Foo")
message(STATUS ${my_output_variable})
```

wird drucken

```
-- foo
```

während

```
set_my_variable("something else")
message(STATUS ${my_output_variable})
```

wird drucken

```
-- bar
```

Makro, um eine Variable mit dem angegebenen Namen zu füllen

```
macro(set_custom_variable _OUT_VAR)
  set(${_OUT_VAR} "Foo")
endmacro(set_custom_variable)
```

Verwenden Sie es mit

```
set_custom_variable(my_foo)
message(STATUS ${my_foo})
```

was gedruckt wird

```
-- Foo
```

Funktionen und Makros online lesen: https://riptutorial.com/de/cmake/topic/2096/funktionen-und-makros

Kapitel 8: Hierarchisches Projekt

Examples

Einfacher Ansatz ohne Pakete

Beispiel, das eine ausführbare Datei (Editor) erstellt und eine Bibliothek (Hervorhebung) mit ihr verknüpft. Die Projektstruktur ist unkompliziert. Für jedes Teilprojekt ist eine Master-CMakeLists und ein Verzeichnis erforderlich:

```
CMakeLists.txt
editor/
    CMakeLists.txt
    src/
        editor.cpp
highlight/
    CMakeLists.txt
    include/
        highlight.h
    src/
        highlight.cpp
```

Die Master-Datei CMakeLists.txt enthält globale Definitionen und `add_subdirectory` Aufruf `add_subdirectory` für jedes Teilprojekt:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

add_subdirectory(highlight)
add_subdirectory(editor)
```

CMakeLists.txt für die Bibliothek weist Quellen und Include-Verzeichnisse zu. Durch die Verwendung von `target_include_directories()` anstelle von `include_directories()` die Include-Verzeichnisse an Bibliotheksbenutzer weitergegeben:

```
cmake_minimum_required(VERSION 3.0)
project(highlight)

add_library(${PROJECT_NAME} src/highlight.cpp)
target_include_directories(${PROJECT_NAME} PUBLIC include)
```

CMakeLists.txt für die Anwendung weist Quellen zu und verknüpft die Markierungsbibliothek. Pfade zu den Binär- und Includes des Hightlighter werden automatisch von cmake gehandhabt:

```
cmake_minimum_required(VERSION 3.0)
project(editor)

add_executable(${PROJECT_NAME} src/editor.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC highlight)
```

Hierarchisches Projekt online lesen: https://riptutorial.com/de/cmake/topic/1443/hierarchisches-projekt

Kapitel 9: Kompilieren Sie Funktionen und die C / C ++ - Standardauswahl

Syntax

- target_compile_features (*target* **PRIVATE | PUBLIC | INTERFACE** *feature1* [*feature2* ...])

Examples

Funktionsanforderungen kompilieren

Erforderliche Compiler-Features können mit dem Befehl target_compile_features auf einem Ziel angegeben werden :

```
add_library(foo
    foo.cpp
)
target_compile_features(foo
    PRIVATE          # scope of the feature
    cxx_constexpr    # list of features
)
```

Die Funktionen müssen Bestandteil von CMAKE_C_COMPILE_FEATURES oder CMAKE_CXX_COMPILE_FEATURES sein . Ansonsten meldet cmake einen Fehler. Cmake fügt den Kompilierungsoptionen des Ziels alle erforderlichen Flags wie `-std=gnu++11` .

In dem Beispiel werden die Features als `PRIVATE` deklariert: Die Anforderungen werden dem Ziel hinzugefügt, nicht jedoch seinen Verbrauchern. Um die Anforderungen automatisch zu einem Zielgebäude gegen foo hinzuzufügen, sollte `PUBLIC` oder `INTERFACE` anstelle von `PRIVATE` :

```
target_compile_features(foo
    PUBLIC    # this time, required as public
    cxx_constexpr
)

add_executable(bar
    main.cpp
)
target_link_libraries(bar
    foo       # foo's public requirements and compile flags are added to bar
)
```

C / C ++ - Versionsauswahl

Die gewünschte Version für C und C ++ kann mit den Variablen `CMAKE_C_STANDARD` (`CMAKE_C_STANDARD` Werte sind 98, 99 und 11) und `CMAKE_CXX_STANDARD` (`CMAKE_CXX_STANDARD` Werte sind 98, 11 und 14) global angegeben werden:

```
set(CMAKE_C_STANDARD 99)
set(CMAKE_CXX_STANDARD 11)
```

`-std=c++11` werden die erforderlichen Kompilierungsoptionen für Ziele `-std=c++11` (z. B. `-std=c++11` für gcc).

Die Version kann als Anforderung festgelegt werden, indem die Variablen `CMAKE_C_STANDARD_REQUIRED` und `CMAKE_CXX_STANDARD_REQUIRED` auf `ON` `CMAKE_CXX_STANDARD_REQUIRED` werden.

Die Variablen müssen vor der Zielerstellung festgelegt werden. Die Version kann auch pro Ziel angegeben werden:

```
set_target_properties(foo PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED ON
)
```

Kompilieren Sie Funktionen und die C / C ++ - Standardauswahl online lesen: https://riptutorial.com/de/cmake/topic/5297/kompilieren-sie-funktionen-und-die-c---c-plusplus---standardauswahl

Kapitel 10: Konfigurationen erstellen

Einführung

In diesem Thema wird die Verwendung verschiedener CMake-Konfigurationen wie Debug oder Release in verschiedenen Umgebungen beschrieben.

Examples

Festlegen einer Release / Debug-Konfiguration

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

# Configuration types
SET(CMAKE_CONFIGURATION_TYPES "Debug;Release" CACHE STRING "Configs" FORCE)
IF(DEFINED CMAKE_BUILD_TYPE AND CMAKE_VERSION VERSION_GREATER "2.8")
  SET_PROPERTY(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS  ${CMAKE_CONFIGURATION_TYPES})
ENDIF()

SET(${PROJ_NAME}_PATH_INSTALL     "/opt/project"                    CACHE PATH "This
directory contains installation Path")
SET(CMAKE_DEBUG_POSTFIX "d")

# Install
#-----------------------------------------------------#
INSTALL(TARGETS ${PROJ_NAME}
    DESTINATION  "${${PROJ_NAME}_PATH_INSTALL}/lib/${CMAKE_BUILD_TYPE}/"
    )
```

Bei den folgenden Builds werden zwei verschiedene Ordner ('/ opt / meinprojekt / lib / Debug' '/ opt / meinprojekt / lib / release') mit den Bibliotheken generiert:

```
$ cd /myproject/build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
$ sudo make install
$ cmake _DCMAKE_BUILD_TYPE=Release ..
$ make
$ sudo make install
```

Konfigurationen erstellen online lesen: https://riptutorial.com/de/cmake/topic/8319/konfigurationen-erstellen

Kapitel 11: Projekte verpacken und verteilen

Syntax

- # Packen Sie ein Build-Verzeichnis
  pack [PFAD]
- # Verwenden Sie einen bestimmten Generator
  cpack -G [GENERATOR] [PFAD]
- # Optionale Überschreibungen angeben
- cpack -G [GENERATOR] -C [KONFIGURATION] -P [PACKAGE NAME] -R [PACKAGE-VERSION] -B [PACKAGE-VERZEICHNIS]

Bemerkungen

CPack ist ein externes Tool, das das schnelle Packen von erstellten CMake-Projekten ermöglicht, indem alle erforderlichen Daten direkt aus den Dateien `CMakeLists.txt` und den verwendeten Installationsbefehlen wie `install_targets()` .

Damit CPack ordnungsgemäß funktioniert, muss die Datei `CMakeLists.txt` Dateien oder Ziele enthalten, die mithilfe des `install` installiert werden sollen.

Ein minimales Skript könnte so aussehen:

```
# Required headers
cmake(3.0)

# Basic project setup
project(my-tool)

# Define a buildable target
add_executable(tool main.cpp)

# Provide installation instructions
install_targets(tool DESTINATION bin)
```

Examples

Erstellen eines Pakets für ein erstelltes CMake-Projekt

Um ein weitervertreibbares Paket (z. B. ein ZIP-Archiv oder ein Setup-Programm) zu erstellen, genügt es normalerweise, CPack mit einer dem Aufruf von CMake sehr ähnlichen Syntax aufzurufen:

```
cpack path/to/build/directory
```

Abhängig von der Umgebung werden dadurch alle für das Projekt erforderlichen / installierten

Dateien gesammelt und in einem komprimierten Archiv oder einem selbstentpackenden Installationsprogramm abgelegt.

Auswahl eines zu verwendenden CPack Generators

Um ein Paket mit einem bestimmten Format zu erstellen, können Sie den zu verwendenden **Generator** auswählen.

Ähnlich wie bei CMake kann dies mit dem Argument **-G** geschehen:

```
cpack -G 7Z .
```

Wenn Sie diese Befehlszeile verwenden, wird das erstellte Projekt im aktuellen Verzeichnis im 7-Zip-Archivformat gepackt.

Zum Zeitpunkt des Schreibens unterstützt CPack Version 3.5 standardmäßig folgende Generatoren:

- `7Z` 7-Zip-Dateiformat (Archiv)
- `IFW` Qt Installer Framework (ausführbare Datei)
- `NSIS` Null Soft Installer (ausführbare Datei)
- `NSIS64` Null Soft Installer (64-Bit, ausführbar)
- `STGZ` Selbstentpackende Tar GZip-Komprimierung (Archiv)
- `TBZ2` Tar BZip2-Komprimierung (Archiv)
- `TGZ` Tar GZip-Komprimierung (Archiv)
- `TXZ` Tar XZ-Komprimierung (Archiv)
- `TZ` Tar Compression-Komprimierung (Archiv)
- `WIX` MSI-Dateiformat über WiX-Tools (ausführbares Archiv)
- `ZIP` ZIP-Dateiformat (Archiv)

Wenn kein expliziter Generator bereitgestellt wird, versucht CPack, die beste verfügbare Leistung in Abhängigkeit von der tatsächlichen Umgebung zu ermitteln. Beispielsweise wird es bevorzugt, eine selbstextrahierende ausführbare Datei unter Windows zu erstellen und ein ZIP-Archiv nur zu erstellen, wenn kein geeignetes Toolset gefunden wird.

Projekte verpacken und verteilen online lesen:
https://riptutorial.com/de/cmake/topic/4368/projekte-verpacken-und-verteilen

Kapitel 12: Suchen und verwenden Sie installierte Pakete, Bibliotheken und Programme

Syntax

- find_package (pkgname [version] [EXACT] [QUIET] [ERFORDERLICH])
- include (FindPkgConfig)
- pkg_search_module (Präfix [ERFORDERLICH] [QUIET] pkgname [otherpkg ...])
- pkg_check_modules (Präfix [ERFORDERLICH] [QUIET] pkgname [otherpkg ...])

Parameter

Parameter	Einzelheiten
Version (optional)	Mindestversion des Pakets, definiert durch eine Hauptnummer und optional eine Neben-, Patch- und Tweak-Nummer, im Format major.minor.patch.tweak
EXACT (optional)	Geben Sie an, dass die in Version angegebene `version` genau die zu suchende `version` ist
ERFORDERLICH (optional)	Löst automatisch einen Fehler aus und stoppt den Prozess, wenn das Paket nicht gefunden wird
QUIET (optional)	Die Funktion sendet keine Nachricht an die Standardausgabe

Bemerkungen

- Der `find_package` Weg ist auf allen Plattformen kompatibel, während der `pkg-config` Weg nur auf Unix-ähnlichen Plattformen wie Linux und OSX verfügbar ist.
- Eine vollständige Beschreibung der `find_package` zahlreichen Parameter und Optionen finden Sie im Handbuch .
- Obwohl es möglich ist, viele optionale Parameter anzugeben, z. B. die Version des Pakets, verwenden nicht alle Suchmodule alle diese Parameter ordnungsgemäß. Wenn ein undefiniertes Verhalten auftritt, kann es erforderlich sein, das Modul im Installationspfad von CMake zu finden und dessen Verhalten zu beheben oder zu verstehen.

Examples

Verwenden Sie find_package und find .cmake-Module

Die Standardmethode zum Suchen installierter Pakete mit CMake ist die Verwendung der `find_package` Funktion in Verbindung mit einer `Find<package>.cmake` Datei. Der Zweck der Datei besteht darin, die `<package>_FOUND` für das Paket zu definieren und verschiedene Variablen festzulegen, z. B. `<package>_FOUND` , `<package>_INCLUDE_DIRS` und `<package>_LIBRARIES` .

Viele `Find<package>.cmake` Dateien vom `Find<package>.cmake` sind bereits standardmäßig in CMake definiert. Wenn jedoch keine Datei für das von Ihnen benötigte Paket vorhanden ist, können Sie immer Ihre eigene schreiben und in `${CMAKE_SOURCE_DIR}/cmake/modules` (oder in ein anderes Verzeichnis, wenn `CMAKE_MODULE_PATH` überschrieben wurde)

Eine Liste der Standardmodule finden Sie im Handbuch (v3.6) . Es ist unbedingt erforderlich, das Handbuch anhand der im Projekt verwendeten Version von CMake zu überprüfen, da sonst Module fehlen. Es ist auch möglich, die installierten Module mit der `cmake --help-module-list` .

Es gibt ein schönes Beispiel für einen `FindSDL2.cmake` auf Github

Hier ist eine grundlegende `CMakeLists.txt` , die SDL2 erfordern würde:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

set(CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH} ${CMAKE_SOURCE_DIR}/cmake/modules")
find_package(SDL2 REQUIRED)

include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Verwenden Sie pkg_search_module und pkg_check_modules

Unter Unix-ähnlichen Betriebssystemen können Sie mit dem Programm `pkg-config` Pakete suchen und konfigurieren, die eine `<package>.pc` Datei enthalten.

Um `pkg-config` , müssen Sie `include(FindPkgConfig)` in einer `CMakeLists.txt` . Dann gibt es 2 mögliche Funktionen:

- `pkg_search_module` , das nach dem Paket `pkg_search_module` und das erste verfügbare verwendet.
- `pkg_check_modules` , die alle entsprechenden Pakete prüfen.

Hier ist eine grundlegende `CMakeLists.txt` , die `pkg-config` , um SDL2 mit der Version über 2.0.1 zu suchen:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

include(FindPkgConfig)
pkg_search_module(SDL2 REQUIRED sdl2>=2.0.1)
```

```
include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Suchen und verwenden Sie installierte Pakete, Bibliotheken und Programme online lesen: https://riptutorial.com/de/cmake/topic/6752/suchen-und-verwenden-sie-installierte-pakete--bibliotheken-und-programme

Kapitel 13: Testen und debuggen

Examples

Allgemeiner Ansatz zum Debuggen beim Erstellen mit Make

Angenommen, das `make` schlägt fehl:

```
$ make
```

Starten Sie es stattdessen mit `make VERBOSE=1` , um die ausgeführten Befehle `make VERBOSE=1` . Führen Sie dann direkt den Linker- oder Compiler-Befehl aus, den Sie sehen. Versuchen Sie, dies durch das Hinzufügen notwendiger Flags oder Bibliotheken zum Laufen zu bringen.

Überlegen Sie sich dann, was Sie ändern müssen, damit CMake die richtigen Argumente an den Compiler / Linker-Befehl übergeben kann:

- Was muss im System geändert werden (welche Bibliotheken müssen installiert werden, welche Versionen und Versionen von CMake selbst)
- Wenn vorherige fehlschlägt, welche Umgebungsvariablen oder Parameter an CMake übergeben werden sollen
- Andernfalls ändern Sie die Änderungen in der `CMakeLists.txt` des Projekts oder in den Erkennungsskripten für Bibliotheken wie `FindSomeLib.cmake`

Um dies zu `CMakeLists.txt` , fügen Sie in `CMakeLists.txt` oder `*.cmake` `message(${MY_VARIABLE})` `*.cmake` , um zu überprüfende Variablen zu debuggen.

Lassen Sie CMake ausführliche Makefiles erstellen

Sobald ein CMake-Projekt über `project()` initialisiert wurde, kann die Ausgabebosität des resultierenden Build-Skripts angepasst werden:

```
CMAKE_VERBOSE_MAKEFILE
```

Diese Variable kann beim Konfigurieren eines Projekts über die Befehlszeile von CMake festgelegt werden:

```
cmake -DCMAKE_VERBOSE_MAKEFILE=ON <PATH_TO_PROJECT_ROOT>
```

Für GNU make hat diese Variable den gleichen Effekt wie das Ausführen von `make VERBOSE=1` .

Debuggen Sie find_package () -Fehler

Hinweis: Die angezeigten CMake-Fehlermeldungen enthalten bereits das Update für "Nicht-Standard" -Bibliothek / Tool-Installationspfade. Die folgenden Beispiele zeigen lediglich

ausführlichere CMake `find_package()` .

CMake intern unterstütztes Paket / Modul

Wenn der folgende Code (ersetzen Sie das `FindBoost` Modul durch Ihr fragliches Modul)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Boost REQUIRED)
```

gibt einen Fehler wie

```
CMake Error at [...]/Modules/FindBoost.cmake:1753 (message):
  Unable to find the requested Boost libraries.

  Unable to find the Boost header files.  Please set BOOST_ROOT to the root
  directory containing Boost or BOOST_INCLUDEDIR to the directory containing
  Boost's headers.
```

Und Sie fragen sich, wo Sie die Bibliothek gefunden haben. Sie können überprüfen, ob Ihr Paket eine `_DEBUG` -Option hat wie das `Boost` Modul, um ausführlichere Ausgaben zu erhalten

```
$ cmake -D Boost_DEBUG=ON ..
```

CMake-fähiges Paket / Bibliothek

Wenn der folgende Code (ersetzen Sie den `Xyz` durch Ihre `Xyz` Bibliothek)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Xyz REQUIRED)
```

gibts die etwas fehler wie

```
CMake Error at CMakeLists.txt:4 (find_package):
  By not providing "FindXyz.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Xyz", but
  CMake did not find one.

  Could not find a package configuration file provided by "Xyz" with any of
  the following names:

    XyzConfig.cmake
    xyz-config.cmake

  Add the installation prefix of "Xyz" to CMAKE_PREFIX_PATH or set "Xyz_DIR"
  to a directory containing one of the above files.  If "Xyz" provides a
  separate development package or SDK, be sure it has been installed.
```

Sie fragen sich, wo die Bibliothek `CMAKE_FIND_DEBUG_MODE` wurde. Sie können die undokumentierte globale Variable `CMAKE_FIND_DEBUG_MODE` , um eine ausführlichere Ausgabe zu erhalten

```
$ cmake -D CMAKE_FIND_DEBUG_MODE=ON ..
```

Testen und debuggen online lesen: https://riptutorial.com/de/cmake/topic/4098/testen-und-debuggen

Kapitel 14: Variablen und Eigenschaften

Einführung

Die Einfachheit grundlegender CMake-Variablen lässt die Komplexität der vollständigen Variablensyntax nicht erkennen. Diese Seite dokumentiert die verschiedenen Fälle mit Beispielen und weist auf die zu vermeidenden Fallstricke hin.

Syntax

- set (Wert der Variablenname [CACHE-Typbeschreibung [FORCE]])

Bemerkungen

Variablennamen unterscheiden zwischen Groß- und Kleinschreibung. Ihre Werte sind vom Typ string. Der Wert einer Variablen wird referenziert über:

```
${variable_name}
```

und wird in einem zitierten Argument ausgewertet

```
"${variable_name}/directory"
```

Examples

Zwischengespeicherte (globale) Variable

```
set(my_global_string "a string value"
    CACHE STRING "a description about the string variable")
set(my_global_bool TRUE
    CACHE BOOL "a description on the boolean cache entry")
```

Ist eine zwischengespeicherte Variable bereits im Cache definiert, wenn CMake die entsprechende Zeile verarbeitet (z. B. wenn CMake erneut ausgeführt wird), wird diese nicht geändert. Um den Standardwert zu überschreiben, hängen Sie `FORCE` als letztes Argument an:

```
set(my_global_overwritten_string "foo"
    CACHE STRING "this is overwritten each time CMake is run" FORCE)
```

Lokale Variable

```
set(my_variable "the value is a string")
```

Standardmäßig ist eine lokale Variable nur im aktuellen Verzeichnis definiert, und alle Unterverzeichnisse, die mit dem Befehl `add_subdirectory` hinzugefügt werden.

Um den Gültigkeitsbereich einer Variablen zu erweitern, gibt es zwei Möglichkeiten:

1. `CACHE` es, die es global verfügbar machen wird
2. Verwenden Sie `PARENT_SCOPE` , um es im übergeordneten Bereich verfügbar zu machen. Der übergeordnete Bereich ist entweder die `CMakeLists.txt` Datei im übergeordneten Verzeichnis oder der Aufrufer der aktuellen Funktion.

   Technisch gesehen ist das übergeordnete Verzeichnis die `CMakeLists.txt` Datei, die die aktuelle Datei über den Befehl `add_subdirectory` .

Streicher und Listen

Es ist wichtig zu wissen, wie CMake zwischen Listen und einfachen Strings unterscheidet. Wenn du schreibst:

```
set(VAR "ab c")
```

Sie erstellen eine **Zeichenfolge** mit dem Wert `"ab c"` . Aber wenn Sie diese Zeile ohne Anführungszeichen schreiben:

```
set(VAR abc)
```

Sie erstellen stattdessen eine **Liste** mit drei Elementen: `"a"` , `"b"` und `"c"` .

Nicht-Listenvariablen sind eigentlich auch Listen (eines einzelnen Elements).

Listen können mit dem Befehl `list()` bearbeitet werden, der das Verketten von Listen, das Durchsuchen von Listen, das Aufrufen von beliebigen Elementen usw. ermöglicht (Dokumentation von list ()).

Etwas verwirrend ist eine **Liste** auch eine **Zeichenfolge** . Die Linie

```
set(VAR abc)
```

ist äquivalent zu

```
set(VAR "a;b;c")
```

Um Listen zu verketten, können Sie auch den Befehl `set()` :

```
set(NEW_LIST "${OLD_LIST1};${OLD_LIST2})"
```

Variablen und der Cache für globale Variablen

Meist werden Sie "normale Variablen" verwenden :

```
set(VAR TRUE)
set(VAR "main.cpp")
```

```
set(VAR1 ${VAR2})
```

CMake kennt aber auch globale "zwischengespeicherte Variablen" (in `CMakeCache.txt` persistiert). Wenn im aktuellen Bereich normale und zwischengespeicherte Variablen mit demselben Namen vorhanden sind, verbergen normale Variablen die zwischengespeicherten Variablen:

```
cmake_minimum_required(VERSION 2.4)
project(VariablesTest)

set(VAR "CACHED-init" CACHE STRING "A test")
message("VAR = ${VAR}")

set(VAR "NORMAL")
message("VAR = ${VAR}")

set(VAR "CACHED" CACHE STRING "A test" FORCE)
message("VAR = ${VAR}")
```

Ausgabe des ersten Laufs

```
VAR = CACHED-init
VAR = NORMAL
VAR = CACHED
```

Ausgabe des zweiten Laufs

```
VAR = CACHED
VAR = NORMAL
VAR = CACHED
```

Anmerkung: Mit der Option `FORCE` wird auch die normale Variable aus dem aktuellen Bereich zurückgesetzt.

Anwendungsfälle für zwischengespeicherte Variablen

Normalerweise gibt es zwei Anwendungsfälle (bitte nicht für globale Variablen missbrauchen):

1. Ein Wert in Ihrem Code sollte vom Benutzer Ihres Projekts `cmakegui` , z. B. mit den `cmakegui` , `ccmake` oder `cmake -D ...` :

   CMakeLists.txt / MyToolchain.cmake

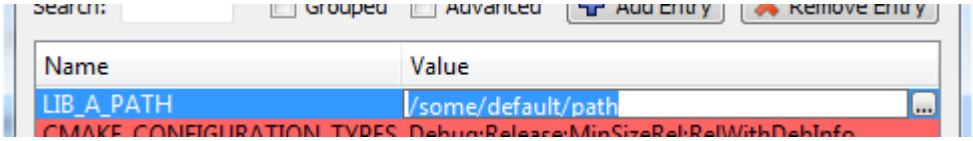
   ```
   set(LIB_A_PATH "/some/default/path" CACHE PATH "Path to lib A")
   ```

   Befehlszeile

   ```
   $ cmake -D LIB_A_PATH:PATH="/some/other/path" ..
   ```

Dadurch wird dieser Wert im Cache voreingestellt und die obige Zeile ändert ihn nicht.

CMake GUI



In der GUI startet der Benutzer zunächst den Konfigurationsprozess, kann dann jeden zwischengespeicherten Wert ändern und endet mit dem Start der Generierung der Build-Umgebung.

2. Darüber hinaus speichert CMake die Such- / Test- / Compiler-Identifizierungsergebnisse im Cache (muss also nicht erneut ausgeführt werden, wenn die Konfigurations- / Generierungsschritte erneut ausgeführt werden)

```
find_path(LIB_A_PATH libA.a PATHS "/some/default/path")
```

Hier wird `LIB_A_PATH` als zwischengespeicherte Variable angelegt.

Hinzufügen von Profilierungsflags zu CMake, um gprof zu verwenden

Die Veranstaltungsreihe hier soll wie folgt funktionieren:

1. Kompilieren Sie den Code mit der Option -pg
2. Verknüpfen Sie den Code mit der Option -pg
3. Programm ausführen
4. Das Programm generiert die Datei gmon.out
5. Führen Sie das Programm gprof aus

Um Profilierungsflags hinzuzufügen, müssen Sie Ihrer CMakeLists.txt Folgendes hinzufügen:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pg")
SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -pg")
SET(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -pg")
```

Das muss Flags zum Kompilieren und Verknüpfen hinzufügen und nach der Ausführung das Programm verwenden

gprof ./my_exe

Wenn Sie eine Fehlermeldung erhalten, wie:

```
gmon.out: No such file or directory
```

Das bedeutet, dass die Kompilierung Profilierungsinformationen nicht ordnungsgemäß hinzugefügt hat.

Variablen und Eigenschaften online lesen: https://riptutorial.com/de/cmake/topic/2091/variablen-

und-eigenschaften

Kapitel 15: Verwenden von CMake zum Konfigurieren von Pre-Prozessor-Tags

Einführung

Die korrekte Verwendung von CMake in einem C ++ - Projekt kann es dem Programmierer ermöglichen, sich weniger auf die Plattform, die Programmversionsnummer und mehr auf das eigentliche Programm selbst zu konzentrieren. Mit CMake können Sie Präprozessor-Tags definieren, die eine einfache Überprüfung der Plattform oder anderer Präprozessor-Tags ermöglichen, die Sie im aktuellen Programm benötigen. Wie die Versionsnummer, die in einem Protokollsystem verwendet werden könnte.

Syntax

- #define preprocessor_name "@ cmake_value @"

Bemerkungen

Es ist wichtig zu verstehen, dass nicht jeder Präprozessor in der `config.h.in` definiert sein `config.h.in` . Präprozessor-Tags werden im Allgemeinen nur verwendet, um den Programmierern das Leben zu erleichtern, und sollten mit Diskretion verwendet werden. Sie sollten prüfen, ob bereits ein Präprozessor-Tag vorhanden ist, bevor Sie ihn definieren, da Sie auf anderen Systemen undefiniertes Verhalten feststellen können.

Examples

Verwenden von CMake zum Definieren der Versionsnummer für die Verwendung von C ++

Die Möglichkeiten sind endlos. Sie können dieses Konzept verwenden, um die Versionsnummer von Ihrem Build-System abzurufen. wie git und verwenden Sie diese Versionsnummer in Ihrem Projekt.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(project_name VERSION "0.0.0")

configure_file(${path to configure file 'config.h.in'}
include_directories(${PROJECT_BINARY_BIN}) // this allows the 'config.h' file to be used
throughout the program

...
```

config.h.in

```
#ifndef INCLUDE_GUARD
#define INCLUDE_GUARD

#define PROJECT_NAME "@PROJECT_NAME@"
#define PROJECT_VER  "@PROJECT_VERSION@"
#define PROJECT_VER_MAJOR "@PROJECT_VERSION_MAJOR@"
#define PROJECT_VER_MINOR "@PROJECT_VERSION_MINOR@"
#define PTOJECT_VER_PATCH "@PROJECT_VERSION_PATCH@"

#endif // INCLUDE_GUARD
```

main.cpp

```
#include <iostream>
#include "config.h"
int main()
{
    std::cout << "project name: " << PROJECT_NAME << " version: " << PROJECT_VER << std::endl;
    return 0;
}
```

Ausgabe

```
project name: project_name version: 0.0.0
```

Verwenden von CMake zum Konfigurieren von Pre-Prozessor-Tags online lesen: https://riptutorial.com/de/cmake/topic/10885/verwenden-von-cmake-zum-konfigurieren-von-pre-prozessor-tags

Kapitel 16: Ziele erstellen

Syntax

- add_executable (target_name [EXCLUDE_FROM_ALL] source1 [source2 ...])
- add_library (lib_name [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] source1 [source2 ...])

Examples

Ausführbare Dateien

Um ein Build-Ziel zu erstellen, das eine ausführbare Datei erzeugt, sollte der Befehl `add_executable` :

```
add_executable(my_exe
               main.cpp
               utilities.cpp)
```

Dadurch wird ein `make my_exe` erstellt, z. B. `make my_exe` für GNU make mit den entsprechenden Aufrufen des konfigurierten Compilers, um eine ausführbare Datei `my_exe` aus den beiden Quelldateien `main.cpp` und `utilities.cpp` zu erstellen.

Standardmäßig werden alle ausführbaren Ziele hinzugefügt , um die builtin `all` Ziel (`all` für GNU machen, `BUILD_ALL` für MSVC).
Um auszuschließen, dass eine ausführbare Datei mit dem Standard- `all` Ziel erstellt wird, kann der optionale Parameter `EXCLUDE_FROM_ALL` direkt nach dem `EXCLUDE_FROM_ALL` :

```
add_executable(my_optional_exe EXCLUDE_FROM_ALL main.cpp)
```

Bibliotheken

Verwenden Sie den Befehl `add_library` um ein Build-Ziel zum Erstellen einer Bibliothek zu `add_library` :

```
add_library(my_lib lib.cpp)
```

Die CMake-Variable `BUILD_SHARED_LIBS` steuert, wann eine statische (`OFF`) oder eine gemeinsam genutzte (`ON`) Bibliothek erstellt wird, beispielsweise mit `cmake .. -DBUILD_SHARED_LIBS=ON` . Sie können jedoch explizit eine gemeinsam genutzte oder eine statische Bibliothek , indem zu bauen gesetzt `STATIC` oder `SHARED` nach dem Zielnamen:

```
add_library(my_shared_lib SHARED lib.cpp) # Builds an shared library
add_library(my_static_lib STATIC lib.cpp) # Builds an static library
```

Die tatsächliche Ausgabedatei unterscheidet sich zwischen den Systemen. Beispielsweise wird eine gemeinsam genutzte Bibliothek auf Unix-Systemen normalerweise als `libmy_shared_library.so` , unter Windows jedoch `my_shared_library.dll` und `my_shared_library.lib` .

`add_executable` wie `add_executable EXCLUDE_FROM_ALL` vor der Liste der Quelldateien hinzu, um sie vom `all` Ziel auszuschließen:

```
add_library(my_lib EXCLUDE_FROM_ALL lib.cpp)
```

Bibliotheken, die zur Laufzeit geladen werden sollen (z. B. Plugins oder Anwendungen, die etwa `dlopen`), sollten `MODULE` anstelle von `SHARED` / `STATIC` :

```
add_library(my_module_lib MODULE lib.cpp)
```

Unter Windows gibt es beispielsweise keine `.lib` (`.lib`), da die Symbole direkt in die `.dll` Datei exportiert werden.

Ziele erstellen online lesen: https://riptutorial.com/de/cmake/topic/3107/ziele-erstellen

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit cmake	Amani, arrowd, ComicSansMS, Community, Daniel Schepler, dontloo, Fantastic Mr Fox, fedepad, Florian, greatwolf, Mario, Neui, OliPro007, Torbjörn, Ziv
2	Benutzerdefinierte Build-Schritte	Developer Paul
3	CMake-Integration in GitHub-CI-Tools	ComicSansMS
4	Datei konfigurieren	Jav_Rock, Shihe Zhang, vgonisanz
5	Erstellen Sie Testsuiten mit CTest	arrowd, ComicSansMS, Torbjörn
6	Fügen Sie dem Compiler Include Path Verzeichnisse hinzu	kiki, Torbjörn
7	Funktionen und Makros	Torbjörn
8	Hierarchisches Projekt	Adam Trhon, Anedar, Clare Macrae, Robert
9	Kompilieren Sie Funktionen und die C / C ++ - Standardauswahl	wasthishelpful
10	Konfigurationen erstellen	Jav_Rock
11	Projekte verpacken und verteilen	Mario, Meysam, Neui
12	Suchen und verwenden Sie installierte Pakete, Bibliotheken und Programme	OliPro007

13	Testen und debuggen	Florian, Torbjörn, Velkan
14	Variablen und Eigenschaften	arrowd, CivFan, Florian, Torbjörn, Trilarion, Trygve Laugstøl, vgonisanz
15	Verwenden von CMake zum Konfigurieren von Pre-Prozessor-Tags	JVApen, Matthew
16	Ziele erstellen	arrowd, Neui, Torbjörn