CMake: Eine Einführung

Jörg Faschingbauer

Table of Contents

FASCHINGBAUER

Overview

FASCHINGBAUER

Software Build — Geschichte (1)

FASCHINGBAUER

Am Anfang war Make ...

- Löst die grundlegendsten Probleme
- Abhängigkeitsgraph → Automatik statt Handarbeit
- Sehr mächtig: GNU Make ist eine eigene Programmiersprache
- → man kann damit alles machen (siehe Linux Kernel)

Aber:

- Größere Projektstrukturen wollen mehr Aufmerksamkeit
- Kompliziertere Makefiles
- Man muss jedes Feature selber programmieren

Software Build — Geschichte (2)

FASCHINGBAUER

Fehlende Features in GNU Make

- Automatische Dependencies auf Headerfiles
  - Händische ($\rightarrow$ *unvollständige*) Listen
- Out-of-Source Build
- Packaging
- Aufruf von Unittests

Klar: Make ist nicht auf C/C++ beschränkt

Software Build — Geschichte (3)

FASCHINGBAUER

Alternative/Aufsätze:

- Autotools (autoconf, automake libtool): de-facto Standard Build Tools für GNU und andere Open Source Pakete.
  - Schwer zu lernen (Shell, M4, ... → perverse Ästhetik)
  - Generiert Makefiles
- SCons
  - Build-Instruktionen in Python
  - Benutzt nicht Make, sondern baut selbst
- M$ Visual Studio
  - (ohne Worte)
- Eclipse CDT
  - Kann C/C++ Projektverwaltung
  - Aber: Integration mit CMake
- CMake
  - ...

Also: CMake

FASCHINGBAUER

Fokus

- “Besseres Automake” (lt. KitWare)
- Generiert ausser Makefiles auch noch andere Sachen (z.B. Eclipe CDT)

Features

- Out-of-Source Build
- Shared Libraries, ohne Libtool (Fluch&Segen)
- Installation
- Eigene Sprache
- Packaging durch ein Zusatz-Paket

Overview

FASCHINGBAUER

Schon wieder: “Hello World”

FASCHINGBAUER

Source

```
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 0;
}
```

Händisch

```
$ gcc -o hello hello.c
```

“Hello World”, mit Overengineering

FASCHINGBAUER

Plan:

- Build-Tool weiss von C/C++
- Man will ihm *deklarativ* sagen, was man haben will
- Hier:
  - Ein “Projekt” namens `hello`
  - und darin ein Executable namens `hello` aus dem File `hello.c`

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6)
project(hello)
add_executable(hello hello.c)
```

“Hello World”, gebaut

FASCHINGBAUER

Out-of-Source Build

- Source-Tree ist *read-only*
- Build ist in separatem Directory, komplett woanders → kann getrost gelöscht werden

Generieren der Makefiles

```
$ cd /path/to/build
$ cmake /path/to/source
```

Bauen

```
$ make
```

Installieren (default: `/usr/local`)

```
$ make install
```

Variationen

FASCHINGBAUER

CMake gibt nur das nötigste aus. Will man mehr wissen (z.B., wie der Compileraufruf ist, oder welche Libraries gelinkt werden),

Lauter bauen

```
$ make VERBOSE=1
```

Will man debuggen,

Private Installation

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ...
$ cmake -DCMAKE_BUILD_TYPE=Release ...
```

Overview

FASCHINGBAUER

Libraries! Wozu? Wie?

FASCHINGBAUER

Wozu?

- Code, der von mehr als einem Programm benutzt wird
- "Module" getrennt voneinander entwickeln (Modul $\Longleftrightarrow$ Library)

Wie? Antwort: it depends ...

- Statische Library
  - Archiv von compilierten Files, aus dem sich der Linker bedient
  - Keine Ressourcenersparnis (jede Executable hat eine eigene Kopie), nur übersichtliche Struktur
- Shared Library
  - Gelinkte Einheit (wie ein Executable, nur mit mehreren Einsprungspunkten)
  - Code liegt nur einmal im Speicher $\rightarrow$ Ressourcenersparnis
  - Versionierung!

Shared Libraries

FASCHINGBAUER

By default werden statische Libraries gebaut. Will man Shared Libraries, sagt man es CMake folgendermassen,

```
$ cmake -DBUILD_SHARED_LIBS=1 /path/to/source
$ cd /path/to/build
$ make
...
$ ls -l lib*
-rwxr-xr-x 1 jfasch jfasch 7572 Feb 12 17:14 libgreet.so
```

Achtung: unter Umständen ist Versionierung eine gute Idee

“Hello World”, mit “i18n”

FASCHINGBAUER

```
.
|-- CMakeLists.txt
|-- deutsch.c
|-- deutsch.h
|-- english.c
|-- english.h
‘-- hello.c
```

Was will man?

- Library mit `deutsch.{h,c}` und `english.{h,c}`
- Executable `hello` aus dem File `hello.c`, das die Library braucht

```
add_library(greet deutsch.h deutsch.c english.h english.c)

add_executable(hello hello.c)
target_link_libraries(hello greet)
```

Overview

FASCHINGBAUER

Komplexere Strukturen (1)

FASCHINGBAUER

Die wenigsten Projekte sind so trivial, dass man den Code in einem einzigen Directory unterbringen kann ...

```
.
|-- CMakeLists.txt
|-- libgreet
|   |-- CMakeLists.txt
|   |-- deutsch.c
|   |-- deutsch.h
|   |-- english.c
|   `-- english.h
`-- programs
    |-- CMakeLists.txt
    `-- hello.c
```

Komplexere Strukturen (2)

FASCHINGBAUER

Toplevel `CMakeLists.txt` verzweigt nur in Subdirectories.

```
add_subdirectory(libgreet)
add_subdirectory(programs)
```

`hello.c` includiert Headers, die nun in `libgreet` liegen → *Includepfad* (`-I` Compiler-Option) in `programs/CMakeLists.txt`,

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

(→ CMake Manual: mehr Variablen)

Overview

FASCHINGBAUER

Optionaler Code

FASCHINGBAUER

Paket soll bedingt compilerten Code enthalten. Z.B. soll das `hello` Paket Schweizerisch grüßen können, was aber by Default abgeschaltet ist.

```
option(TALK_DUETSCH "Do Duetsch greeting" OFF)
```

- Schalter `TALK_DUETSCH` $\rightarrow$ CMake-Variable
- Code muss darauf reagieren:
  - In C: Schweizerisch darf nur gegrüßt werden, wenn eingeschaltet
  - In CMake: bedingte Compilierung von Schweizerisch

CMake Variablen

FASCHINGBAUER

Bedingte Compilation

- Basierend auf einer CMake-Variable `TALK_DUETSCH`
- werden in `libgreet/` die Files `duetsch.h` und `duetsch.c` nur bedingt in den Build aufgenommen.

```
if (TALK_DUETSCH)
    set(DUETSCH_SOURCES swiss.h swiss.c)
endif(TALK_DUETSCH)

add_library(greet
    deutsch.h deutsch.c
    english.h english.c
    ${DUETSCH_SOURCES})
```

C: Bedingte Compilation (1)

FASCHINGBAUER

Plan: die "Will-Duetsch" Information nach C hineinreichen und dort darauf reagieren.

- Das Programm darf die Schweizerische Grussfunktion nur dann aufrufen, wenn sie verfügbar ist (`TALK_DUETSCH` in CMake definiert ist)
- Frage: wie sieht man diese Variable in C-Code?
- → Man lässt CMake ein Headerfile generieren, das man dort includiert, wo man es wissen will.
- → **Gottseidank haben wir den Präprozessor!**

```
configure_file(
    ${PROJECT_SOURCE_DIR}/hello-config.h.in
    ${PROJECT_BINARY_DIR}/hello-config.h
)
include_directories(${PROJECT_BINARY_DIR})
```

C: Bedingte Compilation (2)

FASCHINGBAUER

hello-config.h.in

```
#cmakedefine TALK_DUETSCH
```

Präprozessor sei Dank: bedingte Compilation

- Generiertes Headerfile includieren
- Die Information daraus beziehen und darauf reagieren

programs/hello.c

```
#include <hello-config.h>

#ifdef TALK_DUETSCH
...
#endif
```

Optionaler Code: Benutzung (1)

FASCHINGBAUER

Wie wird die Variable `TALK_DUETSCH` **gesetzt?** → zwei Möglichkeiten ...

Commandline

```
$ cmake -DTALK_DUETSCH=1 /path/to/source
```

CMake bietet keine Möglichkeit, die verfügbaren Optionen (`option()` am Terminal anzuzeigen.

Empfehlung (nicht nur deswegen, sonderen wegen "Projektgesundheit"):

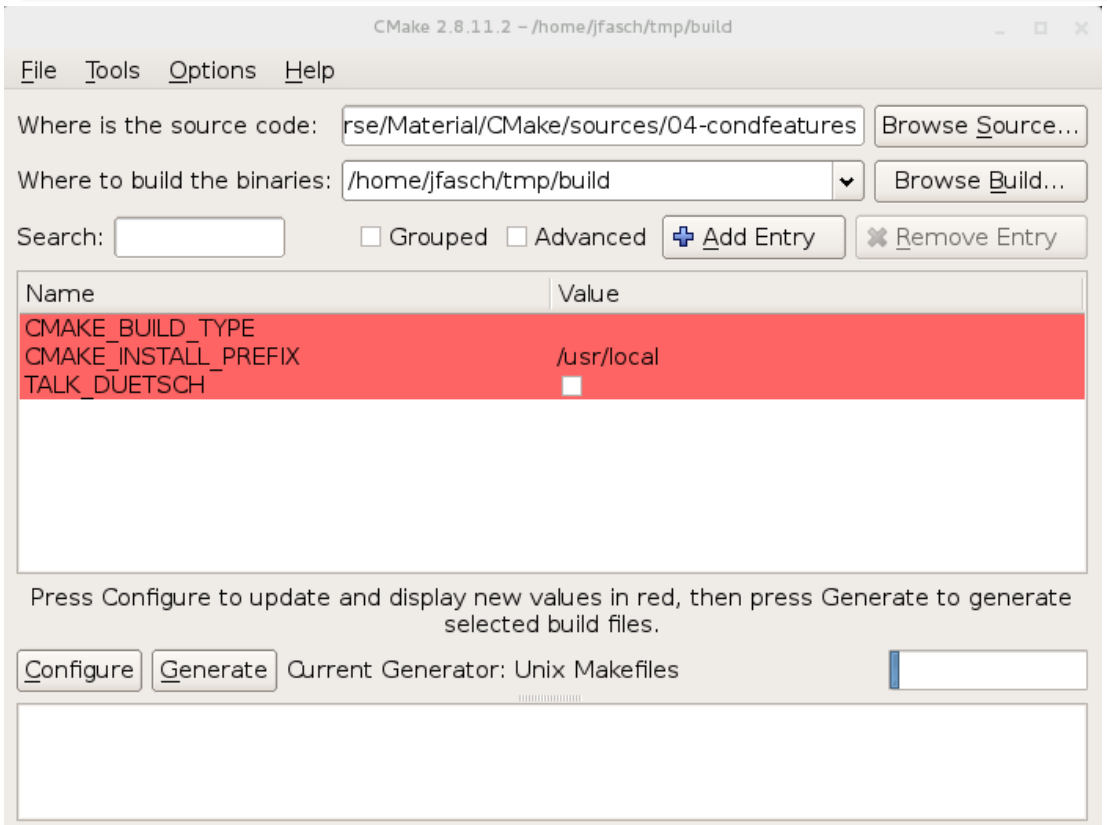
- Zentrale Dinge ("Projekteigenschaften") wie `option()` und `configure_file()` ins Toplevel `CMakeLists.txt`
- Toplevel nichts bauen
- Build passiert in Subdirectories

Optionaler Code: Benutzung (2)

FASCHINGBAUER

GUI

```
$ cmake-gui /path/to/source
```



Generierte Headerfiles: was geht noch? (1)

FASCHINGBAUER

Zweck von generierten Headers: Information aus CMake nach C reichen → da kann man kreativer sein!

- `#cmakedefine` reicht boolesche CMake-Variable rein → bequem
- Aber: man kann beliebige Werte expandieren
- Zum Beispiele: Paketversion — `hello-5.1`

Quelle der information: toplevel `CMakeLists.txt` (wie immer)

`CMakeLists.txt`

```
set(hello_MAJOR 5)
set(hello_MINOR 1)
```

Generierte Headerfiles: was geht noch? (2)

FASCHINGBAUER

Nach C reichen: im generierten Headerfile *expandieren*

`hello-config.h.in`

```
#define hello_MAJOR @hello_MAJOR@
#define hello_MINOR @hello_MINOR@
```

→ Macros verfügbar durch includieren von `hello-config.h`

```
#include <hello-config.h>
...
if (strcmp(argv[1], "--help") == 0)
    printf("hello version %d.%d\n",
           hello_MAJOR, hello_MINOR);
```

Overview

FASCHINGBAUER

Installieren: was ist das?

FASCHINGBAUER

Pakete haben etwas, das Benutzer benutzen wollen ...

- Programme
- Headerfiles und Libraries ($\rightarrow$ Paket offenbar für Entwickler gedacht)
- Dokumentation
- Internationalisierungsdatenbank
- ...

Diese "Artefakte" müssen an einen Platz gebracht werden, wo sie die Benutzer sehen $\rightarrow$ "Installieren"

Installieren: wohin?

FASCHINGBAUER

"Prefix": Ort, an den alle Pakete installiert werden

- Voreingestellte Pfade zeigen dorthin
  - `PATH`, Suchpfad für Programme
  - Shared Libraries ($\rightarrow$ `/etc/ld.so.conf`)

Z.B. Prefix `/usr`,

```
/usr
|-- bin
|-- lib
`-- share
```

Installieren: CMake

FASCHINGBAUER

Unser `hello` Paket hat nicht viel zu bieten ...

```
install(TARGETS hello DESTINATION bin)
```

Will man die Library installieren (falls noch jemand nett grüßen will, ...)

```
install(
    TARGETS greet
    DESTINATION lib)
install(
    FILES deutsch.h english.h duetsch.h
    DESTINATION include)
```

Installieren: Benutzung

FASCHINGBAUER

Installieren geht einfach ...

```
$ cmake /path/to/source
$ make
$ make install
```

Default Prefix ist `/usr/local` → nur für Root schreibbar. Explizites Prefix:

```
$ cmake \
    -DCMAKE_INSTALL_PREFIX=/home/ich/private-installation \
    /path/to/source
$ make
$ make install
```

Overview

FASCHINGBAUER

Softwarelebenszyklus

FASCHINGBAUER

Man kann programmieren und macht keine Fehler, oder?

- Das ist eine Lüge
- ... krasse Selbstüberschätzung
- Händisches Testen ist mühsam
  - Formales Testprotokoll — wer macht sowas?
  - Man vergisst die Hälfte
  - Man ist schlampig
  - Man ist faul
- $\rightarrow$ *Software stirbt mit jeder Änderung ein wenig mehr*

Unittests

FASCHINGBAUER

Unittests

- Teil der Software → kein Testprotokoll, das keiner kennt, weils keiner findet
- Werden mit der Software mitgebaut → immer up-to-date
- Sind aus Software, wie der Rest des Paketes → Programmierer tun sich leichter damit (Spassfaktor)
- Automatisch ausgeführt → keine Handarbeit (Faulheitsfaktor)
- Änderungen können ohne Angst, etwas kaputt zu machen, gemacht werden
- ... sofern die "Testabdeckung" stimmt
- Thema "Unit Testing" füllt eine eigene Schulung

Unittests in CMake (1)

FASCHINGBAUER

- Unittests sind Programme
- Beliebig komplex aufgebaut → Frameworks wie z.B. *Boost Test Library*
- Hier wird nur der Build-Aspekt behandelt

Plan: wir wollen sehen, ob sich das Programm `hello` korrekt verhält (→ einfachster Unittest)

- Lässt es sich überhaupt starten?
- Kann es grüßen?

→ Zwei Tests ...

Unittests in CMake (2)

FASCHINGBAUER

- Testfunktionalität muss man explizit einschalten.
- Wo? $\rightarrow$ Toplevel!

```
enable_testing()
```

Unittests in CMake (3)

FASCHINGBAUER

Test 1: Lässt sich das Programm starten?

```
add_test(hello_runs hello --help)
```

Test 2: Kann es Englisch grüßen?

```
add_test(hello_english hello english)
set_tests_properties(
    hello_english
    PROPERTIES PASS_REGULAR_EXPRESSION "hello world")
```

Unittests in CMake (4)

FASCHINGBAUER

Ausführen von Tests

```
$ make test
Running tests...
Test project /path/to/build
    Start 1: hello_runs
1/2 Test #1: hello_runs ..................... Passed 0.00 sec
    Start 2: hello_english
2/2 Test #2: hello_english .................. Passed 0.00 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.01 sec
```

Overview

FASCHINGBAUER

Packaging

FASCHINGBAUER

Packaging: wozu?

- Benutzer hat nicht immer Zugriff auf das Source-Repository
- $\rightarrow$ kann nicht von Source bauen
- Benutzer ist mental nicht in der Lage, von Source zu bauen (muss er ja nicht sein)

Verschiedene Arten von Paketen:

- Sourcepakete
- Binärpakete

Overview

FASCHINGBAUER

CMake kann mehr als Make

FASCHINGBAUER

Makefiles sind nur eine Möglichkeit, Software zu bauen. Alternativen:

- Eclipse CDT
- Visual Studio
- ...

Siehe `cmake --help` für eine komplette Liste

Demotime!