Prerequisites:

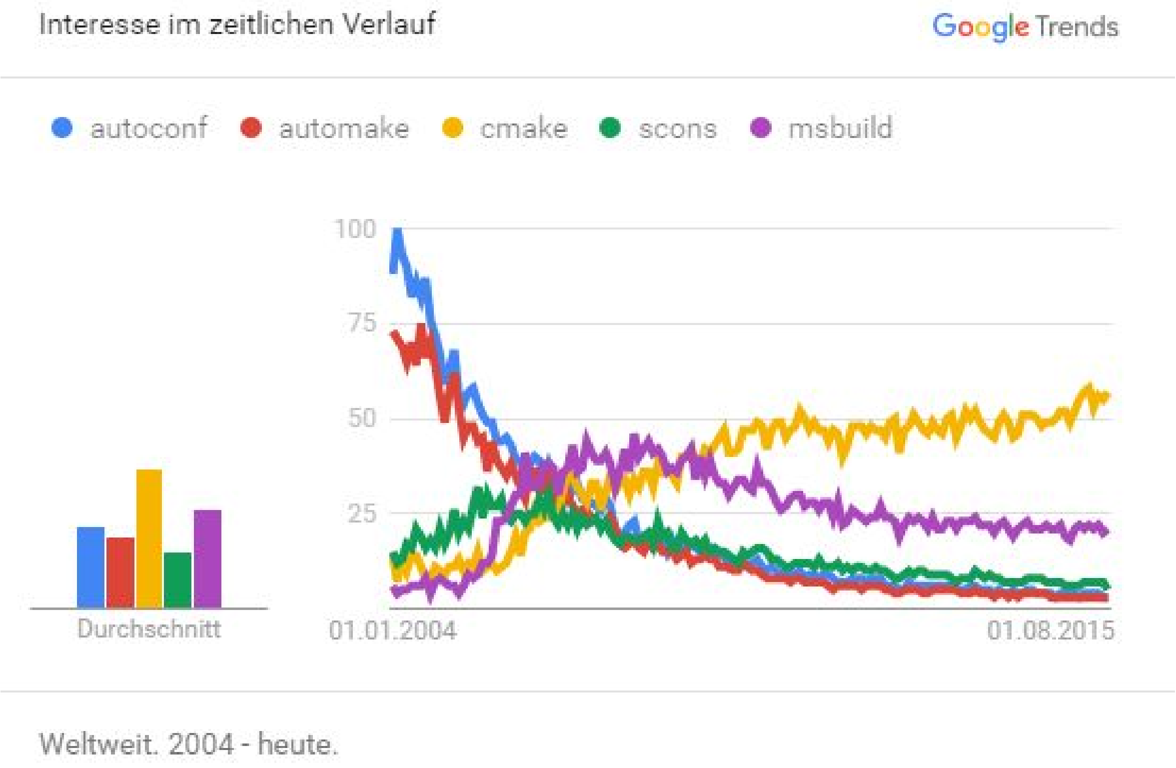
- cmake >= 3.5
- git
- https://github.com/toeb/moderncmake.git

Modern CMake

an Introduction

Motivation

- Knowing your Build System is important but often neglected
  - Key to Automation / Continuous Integration
    - easy entry for new developers
    - time is saved
    - quality can be assured
- Why Cross Platform?
  - Writing cross platform code is good for quality
  - Larger audience
  - Migration to new Compiler Versions
  - Backward Compatibility
- Why **C**ross Platform **Make**
    - The defacto Standard cross Platform “build system” for C/C++



Interesse im zeitlichen Verlauf
Google Trends
autoconf
automake
cmake
scons
msbuild
100
75
50
25
Durchschnitt
01.01.2004
01.08.2015
Weltweit. 2004 - heute.

Content

- Introduction
- Using CMake
  - Command Line
  - Syntax and Library
- Anatomy of a CMake Project
  - Scaffolding
  - Targets
  - Tests
  - Installation
  - Consuming External Projects

“CMake is an **open-source**, **cross-platform** family of tools designed to build, test and package software.

CMake is used to control the software compilation process using **simple platform and compiler independent configuration files**, and **generate native makefiles and workspaces** that can be used in the compiler environment of your choice. “

What is CMake?

- open-source, cross platform Generator for Build Systems
    - Takes a single definition of a project structure
      and translates it to a concrete build system
    - CMake is integrated in newer IDEs
    - Provides a generic interface to control your build process
    - Allows “out of source” builds
- “Cross Platform Bash”
    - Cross Platform Scripting
    - A subset of well known Commands available in Shells
    - Useful for Continuous integration/delivery pipelines

What Build Systems?

Visual Studio 14 2015
Visual Studio 12 2013
Visual Studio 11 2012
Visual Studio 10 2010
Visual Studio 9 2008
Visual Studio 8 2005
Visual Studio 7 .NET 2003
Visual Studio 7
Visual Studio 6
Borland Makefiles
NMake Makefiles
NMake Makefiles JOM
Green Hills MULTI
MSYS Makefiles
MinGW Makefiles
Unix Makefiles
Ninja
Watcom WMake

CodeBlocks - MinGW Makefiles
CodeBlocks - NMake Makefiles
CodeBlocks - Ninja
CodeBlocks - Unix Makefiles
CodeLite - MinGW Makefiles
CodeLite - NMake Makefiles
CodeLite - Ninja
CodeLite - Unix Makefiles
Eclipse CDT4 - MinGW Makefiles
Eclipse CDT4 - NMake Makefiles
Eclipse CDT4 - Ninja
Eclipse CDT4 - Unix Makefiles
Kate - MinGW Makefiles
Kate - NMake Makefiles
Kate - Ninja
Kate - Unix Makefiles
Sublime Text 2 - MinGW Makefiles
Sublime Text 2 - NMake Makefiles
Sublime Text 2 - Ninja
Sublime Text 2 - Unix Makefiles

*on my machine

First Project

- Just create (`CmakeLists.txt`)

```
cmake_minimum_required(VERSION 3.5)
project(sample)

add_executable(sample_exe src/main.cpp)
```

- Generate (bash/PS/cmd)

```
mkdir build | cd
cmake ..
cmake --build .
```

- Run Executable
- Done.

Using CMake

Using the CMake Command Line

- The command line is the central way to use cmake (IMHO the only way)
    - *there is also a GUI
- Commands:

```
cmake [<options>] (<path-to-source> | <path-to-existing-build>)
```

configures a build system for the specified cmake project dir

```
cmake [(-D<var>=<value>)...] -P <cmake-script-file>
```

executes a cmake script file

```
cmake --build <dir> [<options>] [-- <build-tool-options>...]
```

executes the build process through a generic interface

```
cmake -E <command> [<options>...]
```

gives you cross platform commands

CMake Syntax

- CMake's Syntax
  - is very simple, designed for lists of source files
  - also more powerful than you might think
- every line is a command invocation
- commands do not return values
- commands cannot be nested
- all arguments are strings `abc “abc” 1;2;3;4`
- variables can be scoped but inherit their parent scope
- variables can be evaluated using `${<var-name>}`
- control structures: `if(...)` | `elseif(...)` | `else()` | `endif()`, `foreach(...)` | `endforeach()`, `while(...)` | `endwhile()`, `break()`, `function(...)`, `endfunction()`, `return()` , ...
- example: template_compile.cmake

https://cmake.org/cmake/help/v3.5/manual/cmake-language.7.html#syntax

CMake Commands

- CMake provides alot of scripting functionality out of the box [1]
  - `string(...)`
  - `file(...)`
  - `math(...)`
  - `...`
- using `include(<module-name>)` you can also add alot of functionality by contributers [2]
  - `GenerateExportHeader`
  - `TestBigEndian`
  - `...`
  - (you can also contribute yourself)
- Shameless plug: I developed cmakepp which is pure CMake code but adds alot of functionality and is easy to use

[1] https://cmake.org/cmake/help/v3.5/manual/cmake-commands.7.html
[2] https://cmake.org/cmake/help/v3.5/manual/cmake-modules.7.html

Useful Variables

- You have access to a very large list of Variables inside a CMake file
- Almost every aspect of your build is available in variables
- Examples:
  - `UNIX, WIN32, APPLE, MSYS, ...`
  - `CMAKE_CURRENT_LIST_DIR`
  - `CMAKE_CURRENT_SOURCE_DIR`
  - `CMAKE_CURRENT_BINARY_DIR`
  - `...`

https://cmake.org/cmake/help/v3.5/manual/cmake-variables.7.html

Create and run a CMake Script file

- create a file called `fizzbuzz.cmake`

```
# fizzbuzz in cmake
function(fizzbuzz last)
    foreach(i RANGE 0 ${last})
        math(EXPR notFizz "${i} % 3")
        math(EXPR notBuzz "${i} % 5")
        if(NOT notFizz AND notBuzz)
            message("fizz")
        elseif(notFizz AND NOT notBuzz)
            message("buzz")
        elseif(NOT notFizz AND NOT notBuzz)
            message("fizzbuzz")
        else()
            message("${i}")
        endif()
    endforeach()
endfunction()
fizzbuzz(${n})
```

- execute with `cmake -Dn=15 -P fizzbuzz.cmake`

Download, Build and Install Google Test

```
git clone https://github.com/google/googletest.git
cd googletest
mkdir build | cd
cmake .. [-G "Visual Studio 14 2015"] -DCMAKE_INSTALL_PREFIX=stage -DBUILD_SHARED_LIBS=On
cmake --build . --target install --config Release
```

- Results:
  - Downloaded the gtest repository
  - created a “out of source” build folder
  - Configured Build System Visual Studio 15 in Shared Library mode
  - Compiled it in Release Configuration and installed it to `CMAKE_INSTALL_PREFIX`
  - Compilation Result is in googletest/build/stage

Anatomy of a CMake Project

When writing your project configuration never assume to know on which toolchain/platform you are building

Keep your Project definitions Explicit

Anatomy of a CMake Project - Scaffolding

- `CMakeLists.txt` is the description of your project
- Always starts with `cmake_minimum_required(VERSION x.x)`
  - This allows CMake to be backwards compatible
- Always should contain `project(<project-name>)`
  - Names the Solution, does a bit off setup
  - Never assume that your project is the root project
- Afterwards you may write whatever you like.
  - You should probably define a target else you will not build anything

Anatomy of a CMake Project - Scaffolding

- `add_subdirectory(<dir>)`
  Allows you to add subprojects
  - This allows you to easily create a recursive project structure
  - Subdirectory must contain a `CMakeLists.txt`
  - All paths specified are evaluated from the dir of the current `CMakeLists.txt`
  - Never assume that your project is the root project

Anatomy of a CMake Project - Targets

- A target is a node inside the dependency graph of your project
- A target is an executable, static lib, shared lib, header only lib or custom
  - ```
    add_executable(<name> <sourcefile>...)
    add_library(<name> [SHARED|STATIC|INTERFACE]<sourcefile>...)
    add_custom_target(<name> ...)
    (install)
    ```
    ```

Anatomy of a CMake Project - Targets

- `add_library(<name> [SHARED|STATIC|INTERFACE] <sourcefile>...)`
  - no option → creates either shared or static depending on `BUILD_SHARED_LIBS`
  - `SHARED` → creates a shared library
  - `STATIC` → creates a static library
  - `INTERFACE` creates a header only library
- `add_library(<name> ALIAS <original>)`

  allows you to put a library into a custom namespace
- `add_library(<name> <SHARED|STATIC|MODULE|UNKNOWN> IMPORTED [GLOBAL])`

  allows you to define a library target for a external library

Anatomy of a CMake Project - Targets

- You should use the following functions to control your targets
  - target_include_directories(<target-name> [PUBLIC|INTERFACE|PRIVATE] <include-dir>...)

    adds directories to the include search path
  - target_compile_definitions(<target-name> [PUBLIC|INTERFACE|PRIVATE] <definition>...)

    adds preprocessor definitions
  - target_compile_options(<target-name> [PUBLIC|INTERFACE|PRIVATE] <include-dir>...)

    adds compiler options (-Wall, /bigobj ...)
  - target_compile_features(<target-name> [PUBLIC|INTERFACE|PRIVATE] <include-dir>...)

    adds necessary compiler features. (cxx_constexpr, cxx_variadic_templates)
  - target_sources(<target-name> [PUBLIC|INTERFACE|PRIVATE] <source-file>...)

    adds more source files
  - target_link_libraries(<target-name> [PUBLIC|INTERFACE|PRIVATE] <other-target>...)

    add library dependency

Anatomy of a CMake Project - Targets

- Scoping
  - `PUBLIC` causes the property to be available in current target and in all targets depending on it
  - `INTERFACE` causes the property to be available only in targets depending on it
  - `PRIVATE` causes the property to be available only in the current target

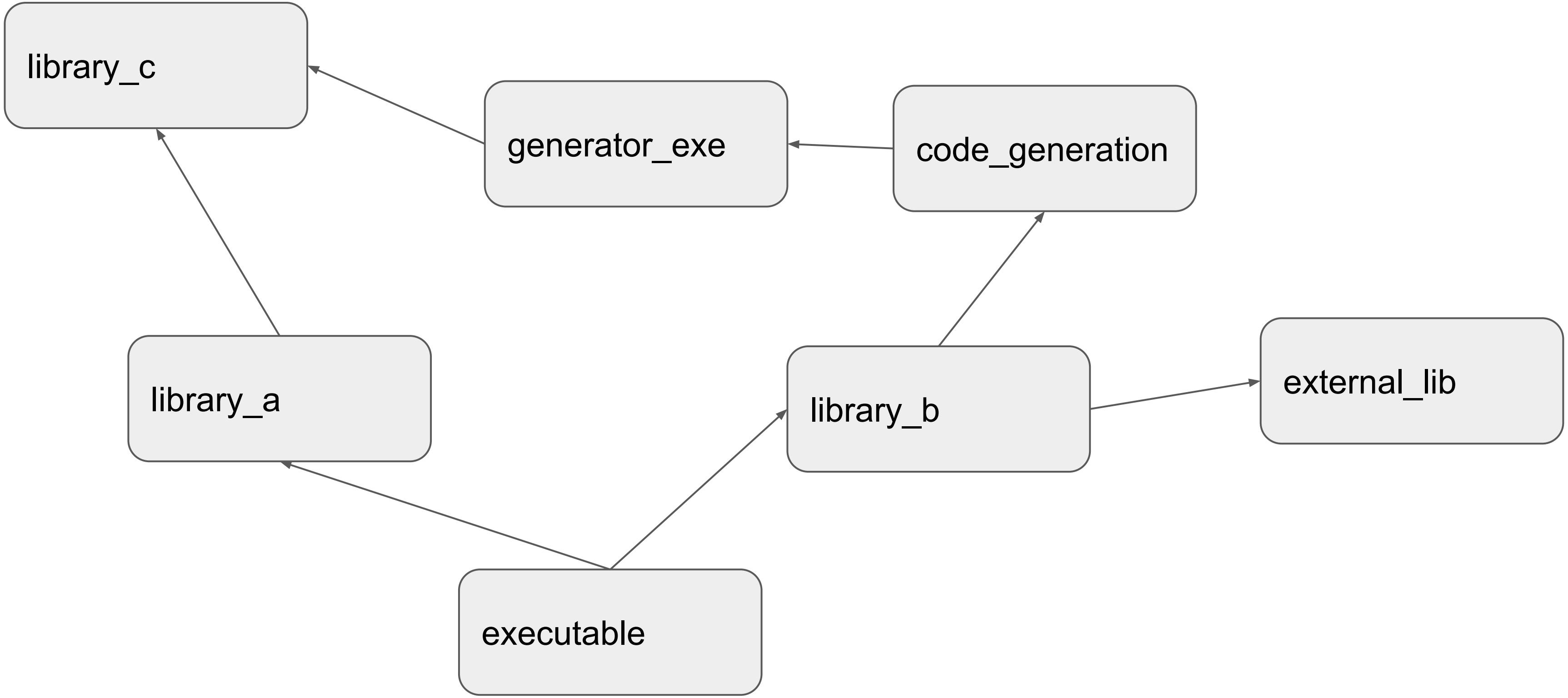
Anatomy of a CMake Project - Targets

- Examples of Scoping
  - `target_include_directories(myTarget PUBLIC ./include)`

    causes the directory `./include` to be searched for include files by `myTarget` and in all targets which depend on it via `target_link_libraries`
  - `target_include_directories(myTarget PRIVATE ./src)`

    causes the directory `./src` to be searched for include files only by `myTarget`
  - `target_compile_definitions(myTarget INTERFACE USE_MYTARGET)` causes the preprocessor definition `USE_MYTARGET` to be defined in all targets depending on `myTarget` but not in `myTarget` itself

Anatomy of a CMake Project - Targets



Example

Create a Project with 3 libraries

- Greeter (shared and static)
- Fareweller (shared and static)
- Conversation depends on Greeter and Fareweller

```
.
|   CMakeLists.txt
|
+---conversation
|       main.cpp
|
+---fareweller
|   |   CMakeLists.txt
|   |
|   +---include
|   |   \---fareweller
|   |           config.h
|   |           make_farewell.h
|   |
|   +---src
|   |       make_farewell.cpp
|   |
|   \---tests
|           fareweller_test.cpp
|
\---greeter
    |   CMakeLists.txt
    |
    +---include
    |   \---greeter
    |           config.h
    |           make_greeting.h
    |
    +---src
    |       localheader.h
    |       make_greeting.cpp
    |
    \---tests
            greeter_test.cpp
```

Anatomy of a CMake Project - Tests

- CMake allows you to wrap executables as test cases
- It allows you or your Build Server to simply execute tests
- It is possible to enable upload of test results to CDash
  - CDash can collect all build information and test run results from every client that builds a project
- Usage
  - to enable add `enable_testing()` to your `CMakeLists.txt`
  - Create an executable which runs your test using `add_executable()`
  - Create a test by using `add_test(<testname> <command> [arg…])`
  - Unit testing Frameworks like gtest are easily integrated
- On Command Line:
  - `ctest`

Example

Create a Project with 3 libraries

- Greeter add Test
- Fareweller add Test
- Conversation enable testing

```
.
|   CMakeLists.txt
|
+---conversation
|       main.cpp
|
+---fareweller
|   |   CMakeLists.txt
|   |
|   +---include
|   |   \---fareweller
|   |           config.h
|   |           make_farewell.h
|   |
|   +---src
|   |       make_farewell.cpp
|   |
|   \---tests
|           fareweller_test.cpp
|
\---greeter
    |   CMakeLists.txt
    |
    +---include
    |   \---greeter
    |           config.h
    |           make_greeting.h
    |
    +---src
    |       localheader.h
    |       make_greeting.cpp
    |
    \---tests
            greeter_test.cpp
```

Anatomy of a CMake Project - Installation

- After the build process is complete you want to deploy your application
- An Installation is normally a collection of files in a specific directory structure
- Installing is the transformation of build results to this specific structure
- ie
  - deploy include files
  - deploy binaries (shared libs and executables)
  - deploy libs
  - deploy resources / docs
- CMake provides a mechanism which creates an installation target

Anatomy of a CMake project - Installation

- `install(...)`
  causes the build system to do something when the install target is executed
- `install(FILES <file>...  DESTINATION <dir>)`
  copies the files from the source directory to the prefix directory
- `install(TARGETS <target>... DESTINATION <dir>)`
  copies the files from the source directory to the prefix directory
- `install(EXPORT <target> NAMESPACE <name> DESTINATION <dir>)`
  creates an import file for your installation that other cmake projects can use
- ...

https://cmake.org/cmake/help/v3.5/command/install.html

Example

Create a Project with 3 libraries

- Fareweller add installation target
- run installation using

```
cmake .. -DCMAKE_INSTALL_PREFIX=stage
cmake --build . --target install
```

```
.
|   CMakeLists.txt
|
+---conversation
|       main.cpp
|
+---fareweller
|   |   CMakeLists.txt
|   |
|   +---include
|   |   \---fareweller
|   |           config.h
|   |           make_farewell.h
|   |
|   +---src
|   |       make_farewell.cpp
|   |
|   \---tests
|           fareweller_test.cpp
|
\---greeter
    |   CMakeLists.txt
    |
    +---include
    |   \---greeter
    |           config.h
    |           make_greeting.h
    |
    +---src
    |       localheader.h
    |       make_greeting.cpp
    |
    \---tests
            greeter_test.cpp
```

Anatomy of a CMake project - External Dependency

- The Export file we created in the installation step can now be include in a different project
- Copy the installed files to another project (or Create a package)
- `include(<path-to-export-file>)`
  in another CMake project to have all installed targets available
- or use `add_library(<name> IMPORTED)`
  - `set_property(TARGET <name> INTERFACE_INCLUDE_DIRECTORIES <include-dir>...)`
  - `set_property(TARGET <name> IMPORTED_LOCATION <path-to-lib-or-dll>...)`
  - `set_property(TARGET <name> IMPORTED_IMPLIB <path-to-lib-dll-lib>...)`
  - `...`

Example

- Use previously generated installation in new executable
- by including generated export file
- by manually creating imported target

Uncovered Topics

- Per File Properties
- Third Party Modules
  - GenerateExportHeader
  - ExternalProject
- External Dependencies with
  - find_package
  - ProjectConfig.cmake
  - package managers
- Precompiled Headers
- Unity Builds
- CPack
- CDash setup
- ...

Sources

[1] https://cmake.org/
[2] http://www.slideshare.net/DanielPfeifer1/cmake-48475415
[3] https://steveire.wordpress.com/
[4] https://www.google.com/trends/explore?date=all&q=autoconf,automake,bjam,cmake,scons

Also alot of other blog posts, and personal experience...

Questions?