Effective dependency management with CMake

Meetup C++, München

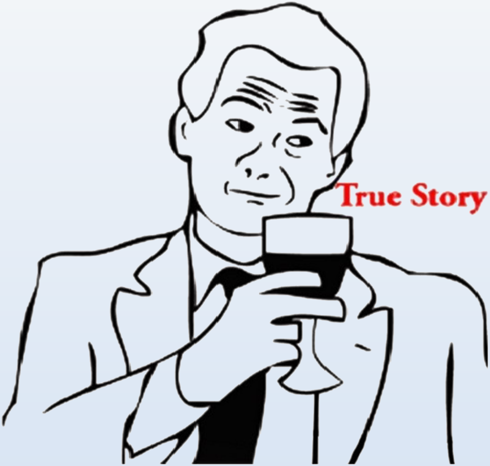
Kai Wolf

`http://kai-wolf.me`

`http://effective-cmake.com`

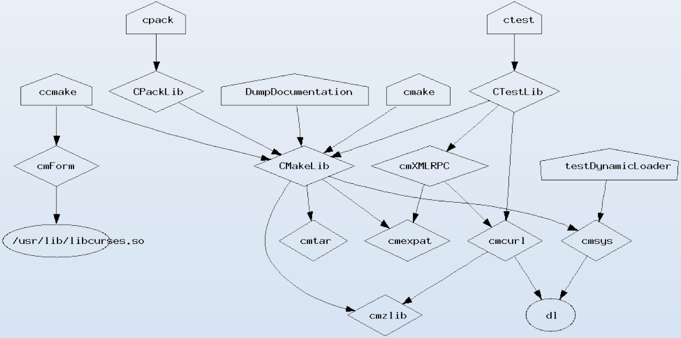
06.06.2017

Let’s talk about build systems



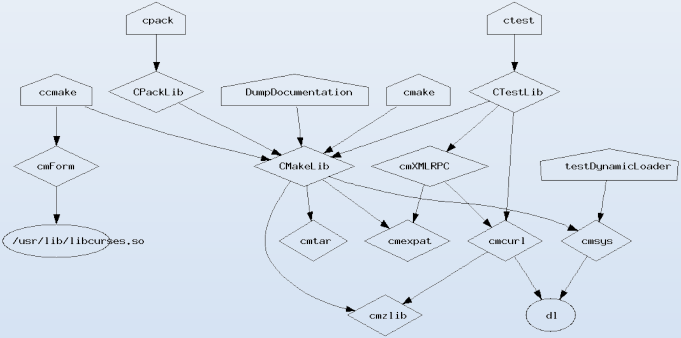
Dependency management is hard (1/5)

- Dozens of different build systems for C/C++ (CMake, Autoconf, QMake, Make, Gyp, SCons, Bazel, . . .)
- Different VCSs (CVS, SVN, Git, Mercurial, . . .)
- No standard project layout (include paths, src/source)
- Transitive dependencies

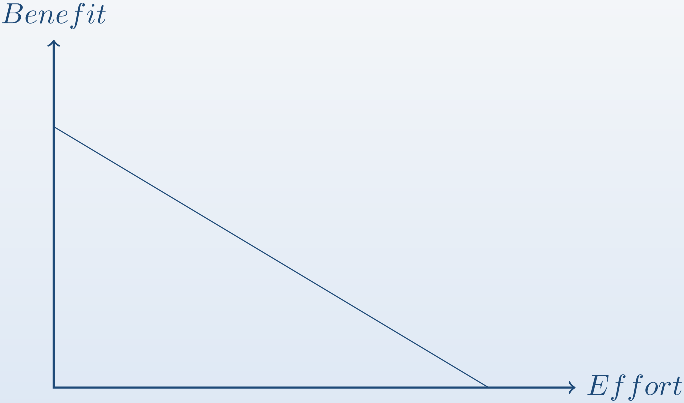


Dependency management is hard (2/5)

- Dependency conflicts
- Thirdparty libraries may need to be treated specially:
  - Custom configuration
  - Only specific parts of a thirdparty library needed
  - Specific compiler (e.g. Intel ICC) and flags
- Partial linking
- Patched versions



Dependency management is hard (3/5)



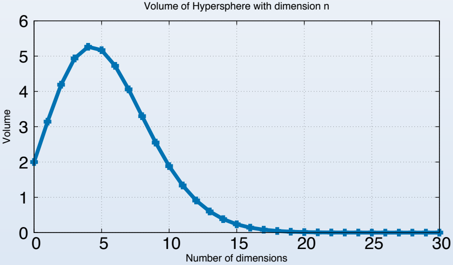
Conjecture

The advantage of using thirdparty libraries is inverse proportional to the amount of effort spent in any software project

Dependency management is hard (4/5)

Curse of Dimensionality

- What applies to ML seems to apply here as well
  → With increasing number of dimensions, generalization suffers
- Build configuration consists of
  - Target OS
  - Compiler
  - Compiler flags
  - Build targets
  - Mandatory thirdparty dependencies
  - Optional: Dedicated compiler
    flags for specific source files



- Number of possible combinations grows exponentially
- CMake cache and toolchain files to the rescue

Dependency management is hard (5/5)

- find_package() is fine, if you don't need full control
- Lowest common denominator is ExternalProject_Add()
  - Works fine with any other build system
  - Thirdparty libraries configurable for own project needs
  - Integrates nicely using ALIAS targets
  - Works with toolchain files
  - No need to keep thirdparty libraries locally available (though recommended)
- Conan-esque solution without introducing another dependency (Python)

CMake initialization

- A large part of CMake’s internal logic is located inside the `<..>/share/cmake-${VERSION}/Modules/` folder
- Which files are parsed depends on following factors
  - Host and target OS
  - Target compiler
  - Host computer’s environment
  - Project specific CMake files which may include
    - toolchain file
    - selected programming language

```
$ find <..>/cmake/Modules/ -regex ".*/CMake[A-Za-z]*Information.cmake"
    <..>/Modules/CMakeASMInformation.cmake
    <..>/Modules/CMakeCInformation.cmake
    <..>/Modules/CMakeCSharpInformation.cmake
    <..>/Modules/CMakeCUDAInformation.cmake
    <..>/Modules/CMakeCXXInformation.cmake
    <..>/Modules/CMakeFortranInformation.cmake
    <..>/Modules/CMakeJavaInformation.cmake
    <..>/Modules/CMakeRCInformation.cmake
    <..>/Modules/CMakeSwiftInformation.cmake
```

CMake initialization

```
$ cmake --debug-output --trace <path>
```

1. Target OS detection
   - Generator dependent
   - Checks system on which CMake runs
   - Searches for sysroot
   - OSX-specific: XCode related stuff

```
<..>/share/cmake/Modules/CMakeUnixFindMake.cmake
<..>/share/cmake/Modules/CMakeDetermineSystem.cmake
<..>/share/cmake/Modules/CMakeSystemSpecificInitialize.cmake
<..>/share/cmake/Modules/Platform/Darwin-Initialize.cmake
```

CMake initialization

❷ Compiler detection

- Starts with `project()`
- Determines compiler executable's location
- Mainly following variables will be defined
  - `CMAKE_CXX_COMPILER`
  - `CMAKE_CXX_SOURCE_FILE_EXTENSIONS`
  - `CMAKE_CXX_IGNORE_EXTENSIONS`
  - `CMAKE_CXX_COMPILER_ENV_VAR`

```
<..>/share/cmake/Modules/CMakeDetermineCXXCompiler.cmake
<..>/share/cmake/Modules/CMakeDetermineCompiler.cmake
<..>/share/cmake/Modules/Platform/Darwin-Determine-CXX.cmake
<..>/share/cmake/Modules/CMakeDetermineCompilerId.cmake
<..>/share/cmake/Modules/Compiler/ADSP-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/ARMCC-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/AppleClang-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Borland-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Clang-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Comeau-CXX-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Cray-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Embarcadero-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Fujitsu-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/HP-CXX-DetermineCompiler.cmake
<..>/share/cmake/Modules/Compiler/Intel-DetermineCompiler.cmake
...
```

CMake initialization

3. Compiler verification
    - Calls compiler to determine its id
    - Searches for C/C++ related tools, such as archiver, linker etc.
    - In the following case AppleClang is chosen

```
<..>/share/cmake/Modules/CMakeFindBinUtils.cmake
<..>/share/cmake/Modules/Compiler/AppleClang-CXX.cmake
<..>/share/cmake/Modules/Platform/Darwin-AppleClang-CXX.cmake
<..>/share/cmake/Modules/CMakeCommonLanguageInclude.cmake
<..>/share/cmake/Modules/CMakeTestCXXCompiler.cmake
<..>/share/cmake/Modules/CMakeDetermineCompilerABI.cmake
<..>/share/cmake/Modules/CMakeParseImplicitLinkInfo.cmake
<..>/share/cmake/Modules/CMakeDetermineCompileFeatures.cmake
<..>/share/cmake/Modules/Internal/FeatureTesting.cmake
<..>/share/cmake/Modules/Compiler/AppleClang-CXX-FeatureTests.cmake
<..>/share/cmake/Modules/Compiler/Clang-CXX-TestableFeatures.cmake
<..>/share/cmake/Modules/Compiler/AppleClang-CXX-FeatureTests.cmake
<..>/share/cmake/Modules/CMakeDetermineCompileFeatures.cmake
```

CMake initialization

4. Project configuration files
    - `-C <initial-cache>`
      May be used to preset values, such as library search paths
    - `CMAKE_TOOLCHAIN_FILE`
      Mainly used for cross-compiling, but can be exploited for presetting values for specific compiler toolchains (stay tuned)
    - `PreLoad.cmake`
      Undocumented. More or less same as *initial cache*. No command line option, has to be in the same directory as your project's CMakeLists.txt
    - `CMAKE_USER_MAKE_RULES_OVERRIDE`
      Modify non-cached default values after automatic detection by CMake

```cmake
# MakeRulesOverwrite.cake
list(APPEND CMAKE_CXX_SOURCE_FILE_EXTENSIONS c)
```

```
$ cmake -DCMAKE_USER_MAKE_RULES_OVERRIDE=../MakeRulesOverwrite.cmake ..
```

CMake initialization

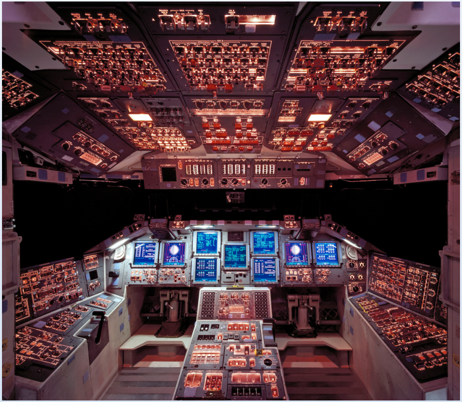
5. Toolchain file
    - Read multiple times while determining the system, compiler etc.
        - Read for each `try_compile()`
        - If the toolchain file is changed, CMake will re-trigger the compiler detection

After this initial configuration, everything else comes from the cache. This includes cached variables as well, resulting in much faster reconfiguration runs.

Compiler flags management

Challenge

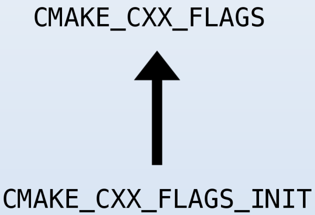
Full control over all set compiler flags, on a source file basis



Compiler flags management

- Compiler flag variables are first initialized after calling `project()`
    - `CMAKE_<language>_FLAGS` is used to invoke the compiler for `<language>`
    - `CMAKE_<language>_FLAGS` is initialized with the content of `CMAKE_<language>_FLAGS_INIT` and placed into the cache (`CMakeCache.txt`)

Example



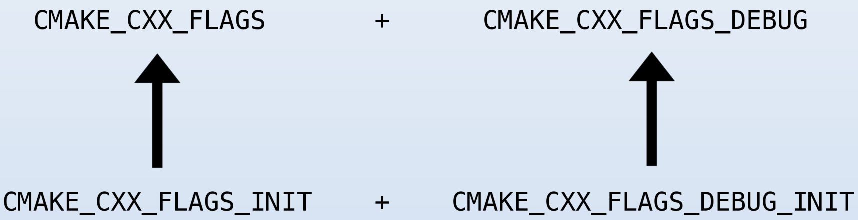
CMAKE_CXX_FLAGS

CMAKE_CXX_FLAGS_INIT

Compiler flags management (cont.)

- If a build type is specified the variable `CMAKE_<language>_FLAGS_<build>` is appended to the variables above
- This variable is initialized from `CMAKE_<language>_FLAGS_<build>_INIT` and also gets cached

Example

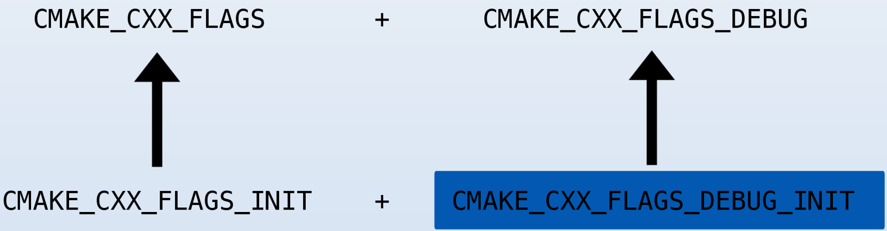


```
CMAKE_CXX_FLAGS          +     CMAKE_CXX_FLAGS_DEBUG

CMAKE_CXX_FLAGS_INIT     +     CMAKE_CXX_FLAGS_DEBUG_INIT
```

Compiler flags management (cont.)

- If CMake knows about the compiler, it will automatically add appropriate flags to the `CMAKE_<language>_FLAGS_<build>_INIT` variable
- For instance, CMake will add `-g` to `CMAKE_C_FLAGS_DEBUG_INIT` if GCC has been selected

Example



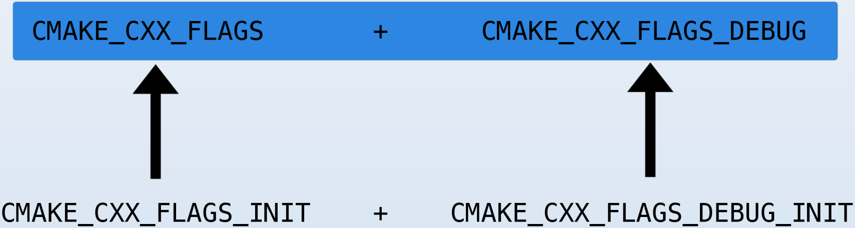
CMAKE_CXX_FLAGS + CMAKE_CXX_FLAGS_DEBUG

CMAKE_CXX_FLAGS_INIT + CMAKE_CXX_FLAGS_DEBUG_INIT

Compiler flags management (cont.)

- Concatenation of the `CMAKE_<language>_FLAGS_<build>` variable to `CMAKE_<language>_FLAGS` is done on a per-file basis

Example



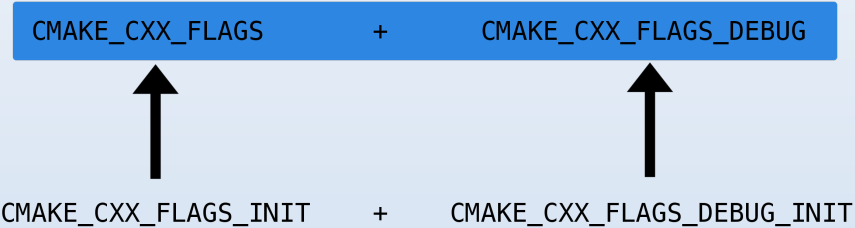
CMAKE_CXX_FLAGS + CMAKE_CXX_FLAGS_DEBUG

CMAKE_CXX_FLAGS_INIT + CMAKE_CXX_FLAGS_DEBUG_INIT

Compiler flags management (cont.)

- Thus, in order to override the compiler flags for a single source (via `CMAKE_<language>_FLAGS_<build>`), the variable `CMAKE_<language>_FLAGS_<build>` needs to be set to the empty string and the `COMPILE_FLAGS` property for the source file has to be assigned accordingly

Example



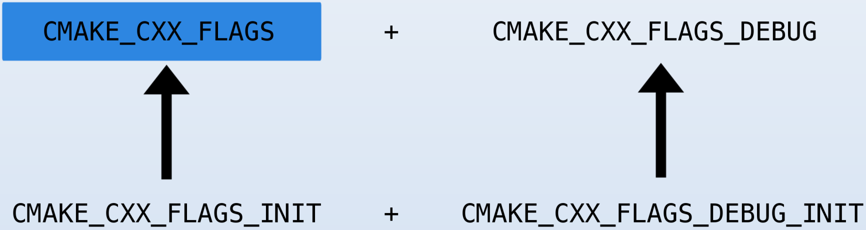
CMAKE_CXX_FLAGS + CMAKE_CXX_FLAGS_DEBUG

CMAKE_CXX_FLAGS_INIT + CMAKE_CXX_FLAGS_DEBUG_INIT

Compiler flags management (cont.)

- The flags set there will be concatenated to the contents of the `CMAKE_<language>_FLAGS` variable as the flags used to compile the file
- Note that `COMPILE_FLAGS` has to be set for every file in the scope where `CMAKE_<language>_FLAGS_<build>` was cleared

Example



CMAKE_CXX_FLAGS + CMAKE_CXX_FLAGS_DEBUG

CMAKE_CXX_FLAGS_INIT + CMAKE_CXX_FLAGS_DEBUG_INIT

Compiler flags management (cont.)

- Unfortunately, CMake's `set_source_files_properties()` can only add additional compile flags, but not replace them entirely
- One workaround is to set `CMAKE_CXX_FLAGS_<build>` for each file individually
- Either the predefined or the custom defined ones, iff a special variable in the form `<FILENAME>_CXX_FLAGS_<build>` is set

```
set(SOURCE_FILES
  foo.cc foo.h bar.cc bar.h)

set(FOO_CC_FLAGS_RELEASE -O4)
set(FOO_CC_FLAGS_DEBUG -pedantic -Wall)

set_atomic_source_file_properties(SOURCE_FILES)
```

Compiler flags management (cont.)

```cmake
macro(make_upper var src_file)
  get_filename_component(var ${src_file} NAME)
  string(REGEX REPLACE "\\.|/" "_" var ${var})
  string(TOUPPER ${var} var)
endmacro()
```

Compiler flags management (cont.)

```
function(set_atomic_source_file_properties sources)
  foreach(SRC_FILE IN ITEMS ${${sources}})
    make_upper(SRC_FILE_NAME ${SRC_FILE})
    set(SRC_FILE_FLAGS
      ${SRC_FILE_NAME}_FLAGS_${UC_BUILD_TYPE})
    if(DEFINED ${SRC_FILE_FLAGS})
      set_source_files_properties(${SRC_FILE}
        PROPERTIES COMPILE_FLAGS ${${SRC_FILE_FLAGS}})
    else()
      set_source_files_properties(${SRC_FILE}
        PROPERTIES COMPILE_FLAGS
          ${CMAKE_CXX_FLAGS_${UC_BUILD_TYPE}})
    endif()
  endforeach()
endfunction()
```

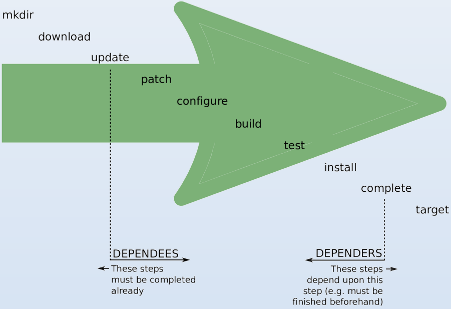
Integration of thirdparty libraries

```
# Toplevel thirdparty/CMakeLists.txt
option(LIBXML_SUPPORT "Build without libxml" ON)
option(LIBPNG_SUPPORT "Build without libpng" ON)
...

if(LIBXML_SUPPORT)
  add_subdirectory(libxml)
endif()
if(LIBPNG_SUPPORT)
  add_subdirectory(libpng)
endif()
```

```
|-- include
|-- src
|-- test
|-- thirdparty
|   |-- libpng
|   |   |-- libpng-2.3.4
|   |   \-- CMakeLists.txt
|   |-- libxml
|   |   |-- libxml-1.2.3
|   |   \-- CMakeLists.txt
|   \-- CMakeLists.txt
|   ...
|-- tools
\-- CMakeLists.txt
```

Integration of thirdparty libraries

```
# Example thirdparty/libxml/CMakeLists.txt
include(ExternalProject)

if(NOT DEFINED LIBXML_VERSION)
  set(LIBXML_VERSIOM libxml-1.2.3)
endif()
set(LIBXML_SOURCE_PATH
  ${CMAKE_CURRENT_SOURCE_DIR}/${LIBXML_VERSION)}

# set custom configure options and compile flags, alt.
# use inherited (cached) variables
...

set_directory_properties(PROPERTIES EP_PREFIX
  ${CMAKE_CURRENT_BINARY_DIR}/${LIBXML_VERSION})

# platform dependent configure steps
if(WIN32)
  ...
else()
  ...
endif()

ExternalProject_Add(libxml_library
  URL ${LIBXML_SOURCE_PATH}
  CONFIGURE_COMMAND ${CONF_COMMAND}
  BUILD_COMMAND ${BUILD_COMMAND}
  BUILD_BYPRODUCTS ${LIBRARIES}
  INSTALL_COMMAND ${INSTALL_COMMAND})
```


mkdir
download
update
patch
configure
build
test
install
complete
target
DEPENDEES
These steps must be completed already
DEPENDERS
These steps depend upon this step (e.g. must be finished beforehand)

Integration of thirdparty libraries

```
function(setup_thirdparty_library TP_NAME)
# header only
add_library(thirdparty::${TP_NAME}
  INTERFACE IMPORTED)
set_target_properties(thirdparty::${TP_NAME}
  PROPERTIES
    INTERFACE_INCLUDE_DIRECTORIES ...)
# full library
add_library(thirdparty::${TP_NAME} STATIC IMPORTED)
set_target_properties(thirdparty::${TP_NAME}
  PROPERTIES
    IMPORTED_LOCATION ...
    INTERFACE_LINK_LIBRARIES ...
    INTERFACE_INCLUDE_DIRECTORIES ...)
endfunction()
```

Dealing with system libraries in CMake

```cmake
macro(add_system_library LIB_NAME LIB_FILE)
  if(NOT TARGET ${LIB_NAME})
    add_library(${LIB_NAME} INTERFACE IMPORTED)
    set_target_properties(${LIB_NAME} PROPERTIES
      INTERFACE_LINK_LIBRARIES "${LIB_FILE}")
  endif()
endmacro()

if(CMAKE_SYSTEM_NAME STREQUAL "Darwin")
  add_system_library(system::OpenGL
    "-framework OpenGL")
elseif(CMAKE_SYSTEM_NAME STREQUAL "Linux")
  add_system_library(system::OpenGL GL)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Windows")
  add_system_library(system::OpenGL Opengl32.lib)
```

Using system libraries

```
# Usage
target_link_libraries(some_target
  PRIVATE system::OpenGL)

# Also works with generator expressions
target_link_libraries(another_target
  thirdparty::libpng thirdparty::libxml
  $<$<PLATFORM_ID:Darwin>:system::Security>)
```

Transitive usage requirements in CMake

- According to CMake documentation:
  "The usage requirements of a target can transitively propagate to dependents"
- **Example:**
  Add the system's math library as a dependency to your target

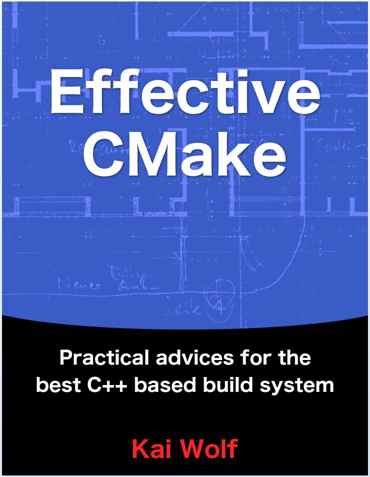
  ```
  target_link_libraries(your_target system::m)
  ```

- This dependency can propagate to any target that has `your_target` added via `target_link_libraries()`
- There's is a difference between *dynamic* and *static* libraries here which is not documented but reasonable

Transitive usage requirements in CMake (cont.)

- **Case: Dynamic library**
  Any library dependency added as `PUBLIC` to `your_target` will be propagated to any target linking against `your_target`
  - A dynamic library (usually) records its own dependencies
  - Thus, no need to explicitly link against its dependencies when linking against it
- **Case: Static library**
  Library dependencies added via `PUBLIC` or `PRIVATE` will be propagated to any target linking against `your_target`
  - A static library cannot record its library dependencies

There will be a book



Effective CMake

Practical advices for the best C++ based build system

Kai Wolf

`http://effective-cmake.com`