Modern CMake

szczecin::cpp #2
13.06.2019

Thomas Geymayer, Senior Software Engineer at GlobalLogic

What do you already know?

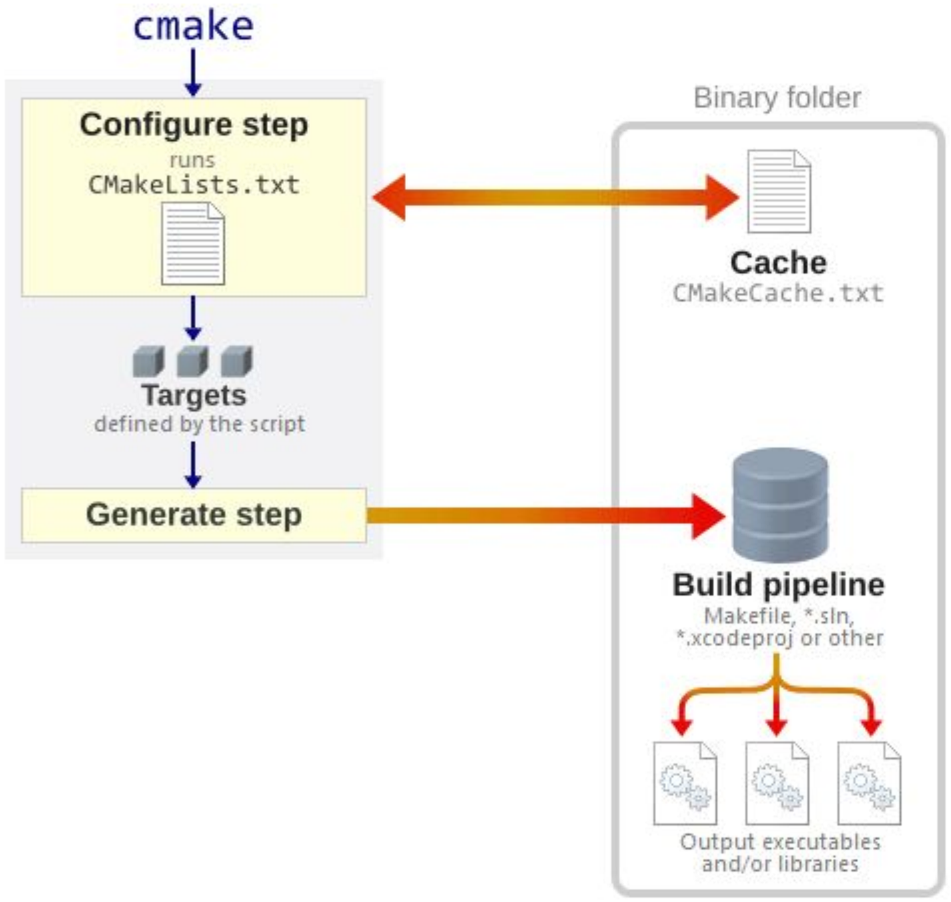
CMake

- Is not a build system
- Custom language and modules
    - Describe build
    - Find dependencies
- Generate build pipeline
    - Cross-platform
    - IDEs, command line

CMake

- Is not a build system
- Custom language and modules
  - Describe build
  - Find dependencies
- Generate build pipeline
  - Cross-platform
  - IDEs, command line

- Steps
  1. Configure
  2. Generate
  3. Build
  4. Install



https://preshing.com/20170511/how-to-build-a-cmake-based-project/

Basic Example

```
cmake_demo
├── CMakeLists.txt
└── src
    └── main.cpp
```

cmake_demo/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(Demo)

add_executable(demo main.cpp)
```

Basic Example

```
cmake_demo
├── CMakeLists.txt
└── src
    └── main.cpp
```

cmake_demo/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(Demo)

add_executable(demo main.cpp)

find_package(Boost 1.67 REQUIRED program_options)

# Don't do the following:
include_directories(${Boost_INCLUDE_DIR})
target_link_libraries(demo ${Boost_LIBRARIES})
```

Basic Example

```
cmake_demo
├── CMakeLists.txt
└── src
    └── main.cpp
```

cmake_demo/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(Demo)

add_executable(demo main.cpp)

find_package(Boost 1.67 REQUIRED program_options)

# Don't do the following:
include_directories(${Boost_INCLUDE_DIR})
target_link_libraries(demo ${Boost_LIBRARIES})
```

- Does it scale with multiple libraries?
- Dependencies of libraries?
- Dependencies of dependent libraries?
- Compiler flags/Minimum C++ version

Basic Example

```
cmake_demo
├── CMakeLists.txt
└── src
    └── main.cpp
```

- Does it scale with multiple libraries?
- Dependencies of libraries?
- Dependencies of dependent libraries?
- Compiler flags/Minimum C++ version

cmake_demo/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(Demo)

add_executable(demo main.cpp)

find_package(Boost 1.67 REQUIRED program_options)

# Don't do the following:
include_directories(${Boost_INCLUDE_DIR})
target_link_libraries(demo ${Boost_LIBRARIES})

find_package(OpenGL REQUIRED OpenGL EGL)

include_directories(
    ${OPENGL_INCLUDE_DIR}
    ${OPENGL_EGL_INCLUDE_DIRS}
)
target_link_libraries(demo ${OPENGL_LIBRARIES})
```

Targets

… are configured with properties

- **Target**
  - **Library, executable, external file**
  - **Properties**
    - **Source files**
    - **Compiler options**
    - **Include directories**
    - **Libraries to link**
  - **Requirements**
    - **INTERFACE**
      - **Use a target**
    - **PRIVATE**
      - **Build a target**
    - **PUBLIC**
      - **Both**

Example Library

```
libMyUtils
├── CMakeLists.txt
├── include
│   └── MyUtils
│       └── utils.h
└── src
    ├── private_header.h
    └── utils.cpp
```

libMyUtils/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(MyUtils)

set(PUBLIC_INCLUDE_DIR include/${PROJECT_NAME})
set(PUBLIC_HEADER_FILES
    ${PUBLIC_INCLUDE_DIR}/utils.h
)

add_library(${PROJECT_NAME} SHARED
    ${PUBLIC_HEADER_FILES}
    src/utils.cpp
    src/private_header.h
)
```

Target Properties

```
target_compile_definitions(<target> PUBLIC ...)
target_compile_features(   <target> PRIVATE ...)
target_compile_options(    <target> INTERFACE ... PRIVATE ...)
target_include_directories(<target> PRIVATE ...)
target_link_directories(   <target> PRIVATE ...)
target_link_libraries(     <target> PRIVATE ...)
target_link_options(       <target> PRIVATE ...)
target_sources(            <target> PRIVATE ...)
```

Example Library - Dependencies

libMyUtils/CMakeLists.txt

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        include
)
```

Example Library - Dependencies

libMyUtils/CMakeLists.txt

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        include
)
find_package(Boost 1.67 REQUIRED program_options system)
target_link_libraries(${PROJECT_NAME}
    PUBLIC
        Boost::system
    PRIVATE
        Boost::program_options
)
```

Example Library - Dependencies

libMyUtils/CMakeLists.txt

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        include
)
find_package(Boost 1.67 REQUIRED program_options system)
target_link_libraries(${PROJECT_NAME}
    PUBLIC
        Boost::system
    PRIVATE
        Boost::program_options
)
target_compile_definitions(${PROJECT_NAME}
    PUBLIC
        MY_UTILS_PRECISION=4
    PRIVATE
        -DUSE_FASTER_BUT_SLOWER_ALGORITHM=1
)
```

Example Library - Dependencies

libMyUtils/CMakeLists.txt

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        include
)
find_package(Boost 1.67 REQUIRED program_options system)
target_link_libraries(${PROJECT_NAME}
    PUBLIC
        Boost::system
    PRIVATE
        Boost::program_options
)
target_compile_definitions(${PROJECT_NAME}
    PUBLIC
        MY_UTILS_PRECISION=4
    PRIVATE
        -DUSE_FASTER_BUT_SLOWER_ALGORITHM=1
)
target_compile_features(${PROJECT_NAME}
    PUBLIC
        cxx_variadic_templates cxx_std_17
)
```

Example Library - Use it

```
cmake_demo
├── CMakeLists.txt
├── libMyUtils
│   ├── CMakeLists.txt
│   ├── highlight.html
│   ├── include
│   │   └── MyUtils
│   │       └── utils.h
│   └── src
│       ├── private_header.h
│       └── utils.cpp
└── src
    └── main.cpp
```

cmake_demo/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(Demo CXX)

add_subdirectory(libMyUtils)

add_executable(demo src/main.cpp)

target_compile_features(demo PUBLIC cxx_std_14)

# Get all PUBLIC and INTERFACE properties of MyUtils
target_link_libraries(demo PUBLIC MyUtils)
```

Example Library - Use it

```
cmake_demo
├── CMakeLists.txt
├── libMyUtils
│   ├── CMakeLists.txt
│   ├── highlight.html
│   ├── include
│   │   └── MyUtils
│   │       └── utils.h
│   └── src
│       ├── private_header.h
│       └── utils.cpp
└── src
    └── main.cpp
```

cmake_demo/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project(Demo CXX)

add_subdirectory(libMyUtils)

add_executable(demo src/main.cpp)

target_compile_features(demo PUBLIC cxx_std_14)

# Get all PUBLIC and INTERFACE properties of MyUtils
target_link_libraries(demo PUBLIC MyUtils)
```

MyUtils: INTERFACE_COMPILE_FEATURES=cxx_std_17

Demo: COMPILE_FEATURES=~~cxx_std_14~~ cxx_std_17

Inheritance Example

```
target_link_libraries(foo
    PUBLIC
        A
    INTERFACE
        B
    PRIVATE
        C
)
```

```
get_target_property(... foo ...)
```

```
INTERFACE_LINK_LIBRARIES: A;B
LINK_LIBRARIES: B;C
```

foo is linked with `LINK_LIBRARIES`

Inheritance Example

```
target_link_libraries(foo
    PUBLIC
        A
    INTERFACE
        B
    PRIVATE
        C
)
```

```
get_target_property(... foo ...)
```

```
INTERFACE_LINK_LIBRARIES: A;B
LINK_LIBRARIES: B;C
```

foo is linked with LINK_LIBRARIES

```
target_link_libraries(bar
    PUBLIC
        foo
    PRIVATE
        D
)
```

```
get_target_property(... bar ...)
```

```
INTERFACE_LINK_LIBRARIES: foo;A;B
LINK_LIBRARIES: D;foo;A;B
```

Inheritance Example

```
target_link_libraries(foo
    PUBLIC
        A
    INTERFACE
        B
    PRIVATE
        C
)
```

```
get_target_property(... foo ...)
```

```
INTERFACE_LINK_LIBRARIES: A;B
LINK_LIBRARIES: B;C
```

foo is linked with LINK_LIBRARIES

```
target_link_libraries(bar
    PUBLIC
        foo
    PRIVATE
        D
)
```

```
get_target_property(... bar ...)
```

```
INTERFACE_LINK_LIBRARIES: foo;A;B
LINK_LIBRARIES: D;foo;A;B
```

- `INTERFACE_LINK_LIBRARIES` of all `PRIVATE` and `PUBLIC` libraries are added to `LINK_LIBRARIES`
- `INTERFACE_LINK_LIBRARIES` of all `PUBLIC` and `INTERFACE` libraries are added to `INTERFACE_LINK_LIBRARIES`
- Same with other properties

Installing

... and using installed libraries

- Deploy files
    - Public headers, build artifacts
- Provide interface properties
    - Dependent libraries
    - Compile flags
    - `target_xxx(<INTERFACE|PUBLIC>)`
- Build configuration
- Version compatibility
- Find deployed files

Install files

libMyUtils/CMakeLists.txt

```
include(GNUInstallDirs)

# Install binary artifacts
install(TARGETS ${PROJECT_NAME}
    # Prepare export (target interface properties)
    EXPORT ${PROJECT_NAME}
    # Executables and DLLs
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR} # bin
    # Dynamic libraries (except DLLs)
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR} # lib, lib64, etc
    # Static libraries, Windows DLL import libraries
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
)

# Install public header files
install(DIRECTORY ${PUBLIC_INCLUDE_DIR} DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
```

Export Targets

- Create `<CamelCaseName>Targets.cmake` (name is arbitrary)

libMyUtils/CMakeLists.txt

```
include(GNUInstallDirs)

set(PROJECT_CMAKE_INSTALL_DIR "${CMAKE_INSTALL_LIBDIR}/cmake/${PROJECT_NAME}")

install(EXPORT ${PROJECT_NAME}
    FILE
        ${PROJECT_NAME}Targets.cmake
    NAMESPACE
        My::
    DESTINATION
        ${PROJECT_CMAKE_INSTALL_DIR}
)
```

Config.cmake

- Create `<lower-case-name>-config.cmake` or `<CamelCaseName>Config.cmake`
- No need to write FindXXX.cmake
  - Predefined search paths, or call `cmake` with `-DCMAKE_PREFIX_PATH=<path-to-install-libdir>`

libMyUtils/Config.cmake.in

```
@PACKAGE_INIT@

include(CMakeFindDependencyMacro)
find_dependency(Boost 1.67 REQUIRED system)

include("${CMAKE_CURRENT_LIST_DIR}/@PROJECT_NAME@Targets.cmake")
```

libMyUtils/CMakeLists.txt

```
include(CMakePackageConfigHelpers)
set(CONFIG_FILE "${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME}Config.cmake")

configure_package_config_file(Config.cmake.in ${CONFIG_FILE}
    INSTALL_DESTINATION ${PROJECT_CMAKE_INSTALL_DIR}
)

install(FILES ${CONFIG_FILE} DESTINATION ${PROJECT_CMAKE_INSTALL_DIR})
```

Version Check (optional)

- Create `<lower-case-name>-config-version.cmake` or `<CamelCaseName>ConfigVersion.cmake`
- Find library with version requirement

libMyUtils/CMakeLists.txt

```
project(MyUtils VERSION 0.4)
```

```
include(CMakePackageConfigHelpers)

set(VERSION_FILE_NAME ${PROJECT_NAME}ConfigVersion.cmake)

# Create version file in build directory:
write_basic_package_version_file(${VERSION_FILE_NAME}
    COMPATIBILITY SameMajorVersion
)

install(
    FILES ${CMAKE_CURRENT_BINARY_DIR}/${VERSION_FILE_NAME}
    DESTINATION ${PROJECT_CMAKE_INSTALL_DIR}
)
```

Installed Library

```
<CMAKE_INSTALL_PREFIX>
├── include
│   └── MyUtils
│       └── utils.h
└── lib
    ├── cmake
    │   └── MyUtils
    │       ├── MyUtilsConfig.cmake
    │       ├── MyUtilsConfigVersion.cmake
    │       ├── MyUtilsTargets.cmake
    │       ├── MyUtilsTargets-debug.cmake
    │       ├── MyUtilsTargets-noconfig.cmake
    │       └── MyUtilsTargets-release.cmake
    └── libMyUtils.so
```

Sequence diagram at: https://github.com/forexample/package-example#uml-sequence-diagram

Using the exported library

external_demo/CMakeLists.txt

```
project(DemoApp)

find_package(MyUtils 0.2 REQUIRED)

add_executable(demo main.cpp)
target_link_libraries(demo My::MyUtils)
```

```
cmake -DCMAKE_PREFIX_PATH=<library-CMAKE_INSTALL_PREFIX> <path-to-external_project>
```

Using the exported library

external_demo/CMakeLists.txt

```
project(DemoApp)

find_package(MyUtils 0.2 REQUIRED)

add_executable(demo main.cpp)
target_link_libraries(demo My::MyUtils)
```

```
cmake -DCMAKE_PREFIX_PATH=<library-CMAKE_INSTALL_PREFIX> <path-to-external_project>
```

Problem with include directory of the library
(It’s in the build directory!)

Quick dive into generator expressions

Generator Expressions

- Evaluated during build system generation (not configuration)
- Syntax `$<...>` (can be nested)
- Usable as arguments to eg. most `target_xxx` commands
- Examples
  - `$<condition:true_string>` (Evaluates to true_string if condition is 1, empty string otherwise)
  - `$<IF:condition,true_string,false_string>`
  - `$<COMPILE_FEATURES:features>` (Evaluates to 1 if all of the features are available, 0 otherwise)

Generator Expressions

- Evaluated during build system generation (not configuration)
- Syntax `$<...>` (can be nested)
- Usable as arguments to eg. most `target_xxx` commands
- Examples
  - `$<condition:true_string>` (Evaluates to `true_string` if `condition` is `1`, empty string otherwise)
  - `$<IF:condition,true_string,false_string>`
  - `$<COMPILE_FEATURES:features>` (Evaluates to `1` if all of the `features` are available, `0` otherwise)

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        $<IF:$<COMPILE_FEATURES:cxx_variadic_templates>,with_variadics,without_variadics>
)
```

Generator Expressions

- Evaluated during build system generation (not configuration)
- Syntax `$<...>` (can be nested)
- Usable as arguments to eg. most `target_xxx` commands
- Examples
    - `$<condition:true_string>` (Evaluates to true_string if condition is 1, empty string otherwise)
    - `$<IF:condition,true_string,false_string>`
    - `$<COMPILE_FEATURES:features>` (Evaluates to 1 if all of the features are available, 0 otherwise)

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        $<IF:$<COMPILE_FEATURES:cxx_variadic_templates>,with_variadics,without_variadics>
)
```

Tip: (message() does not work)

```
add_custom_target(genexdebug COMMAND ${CMAKE_COMMAND} -E echo "$<...>")
```

https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html#manual:cmake-generator-expressions(7)

Fixed Library Includepaths

- Different during build time and when installed
- `$<INSTALL_INTERFACE:<path>>`
    - `<path>` when installed, empty otherwise
- `$<BUILD_INTERFACE:<path>>`
    - `<path>` when in build directory, empty otherwise

Fixed Library Includepaths

- Different during build time and when installed
- `$<INSTALL_INTERFACE:<path>>`
  - `<path>` when installed, empty otherwise
- `$<BUILD_INTERFACE:<path>>`
  - `<path>` when in build directory, empty otherwise

libMyUtils/CMakeLists.txt

```
target_include_directories(${PROJECT_NAME}
    PUBLIC
        # Should be a relative path
        $<INSTALL_INTERFACE:include>

        # Has to be an absolute path
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)
```

Tips for local development

libMyUtils/CMakeLists.txt

```cmake
# Use without installation directly from the build tree (not relocatable)
export(EXPORT ${PROJECT_NAME}
    FILE
        ${PROJECT_NAME}TreeTargets.cmake
    NAMESPACE
        My::
)

# Optionally register to local package registry for local packages finding
each other
export(PACKAGE ${PROJECT_NAME})
```

DO

```
target_include_directories()
target_link_libraries()
target_compile_options()

target_compile_options(...)
target_compile_definitions(...)

target_compile_features(... cxx_std_*)
```

Always use `PRIVATE`, `PUBLIC` or `INTERFACE`

DO NOT!

```
include_directories()
link_libraries()
add_compile_options()

set(CMAKE_CXX_FLAGS …)

set(CMAKE_CXX_STANDARD …)
```

Abuse requirements

  Eg. `-Wall` as `PUBLIC`

Thanks!

Time for the quiz question

References

https://meetingcpp.com/mcpp/slides/2018/MoreModernCMake.pdf

CppCon2017 - Using Modern CMake Patterns to Enforce a Good Modular Design - Mathieu Ropert

C++Now 2017 - Effective CMake - Daniel Pfeifer

https://cliutils.gitlab.io/modern-cmake/

https://cmake.org/cmake/help/latest/

https://github.com/forexample/package-example