Effective Modern CMake project

Sławomir Grabowski — 27.10.2021

×

Who I am?

Sławomir Grabowski

10 years of experience mostly in:

— modern C++
— Qt
— 3D

Other:

— co-founder & developer in Quick Turn Studio
— CMake, Qt, modern C++ trainer
— game development after hours

Presentation scope

- project structure
- compiler flags
- integration with tools
- external dependencies management approaches

Classic CMake project

Classic CMake project

- `include_directories()`
- `add_definitions()`
- `link_directories()`
- subdirectories, but only for sorting headers and sources

Modern CMake approach

Idea

- every folder can be CMake subproject, of course containing own `CMakeLists.txt`
- static library is first candidate for subproject output
- subproject is added to main project from parent directory by using `add_subdirectory(<directory_name>)`
- generated CMake files in build directory has same structure as source directory

Advantages

- Module (specially static library) can be reused in many times in project, eg:
  - parent executable or library
  - tests of that module
- Easier navigation between files - easier to find entire logic for given feature instead of "GUI" directory
- Faster compilation - after change one file we recompile one module instead of entire project

Advantages

- Dividing on parts forces to think about dividing on modules and specifying interfaces
- Less dependencies per subproject, so it is easier to replace components
- Less connections between files, specially from different directories
- Possibility of having different rules for every module, eg. compile flags, Clang-Tidy rules

Modern CMake approach

- Target based instead of variable based
- includes, linked libraries are propagated by properties of linked target

Visualizing project dependencies

- use flag `--graphviz=filename.dot` to CMake command to export targets relations to graphviz format that can be visualized
- put `CMakeGraphVizOptions.cmake` in source root or build directory root file with options in case you would like to change default export settings

Dynamic libraries and MSVC

- during compilation msvc needs to know which classes and functions will be exported - we need to add `__declspec(dllexport)` for each of them
- when linking to dynamic library msvc needs to know which function and classes are exported - for each of them we need to add `__declspec(dllimport)`

```cpp
// when function is build in dynamic library
__declspec(dllexport) bool function();

// when class is build in dynamic library
class __declspec(dllexport) MyClass
{
};

// ------------------------------------------------

// when the same function is linked to another target
__declspec(dllimport) bool function();

// when class is build in dynamic library
class __declspec(dllimport) MyClass
{
};
```

Providing sources to target

Many ways, sill no one ideal solution, war is ongoing

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

set(SRC_FILES
        main.cpp
        FileWriter.cpp
        FileLoader.cpp
        TerminalLogger.cpp
        FilePrinter.cpp
        )

add_executable(${APP_NAME} ${SRC_FILES})
```

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

# during loading project CMake scans given directory looking for sources
# like *.c, *.cpp, *.cxx etc. and appends the list to SRC_FILES VARIABLE

aux_source_directory(${CMAKE_CURRENT_SOURCE_DIR} SRC_FILES)

add_executable(${APP_NAME} ${SRC_FILES})
```

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

# during loading project CMake scans given directory looking for sources
# like *.c, *.cpp, *.cxx etc. and appends the list to SRC_FILES VARIABLE

aux_source_directory(. SRC_FILES)

add_executable(${APP_NAME} ${SRC_FILES})
```

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

# during loading project CMake scans given directory looking for files
# in given directory that match pattern SRC_FILES VARIABLE

file(GLOB SRC_FILES ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)

add_executable(${APP_NAME} ${SRC_FILES})
```

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

# CONFIGURE_DEPENDS make checking not during loading but before every build

file(GLOB SRC_FILES CONFIGURE_DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)

add_executable(${APP_NAME} ${SRC_FILES})
```

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

# CONFIGURE_DEPENDS make checking not during loading but before every build

file(GLOB SRC_FILES CONFIGURE_DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)

add_executable(${APP_NAME} ${SRC_FILES})
```

Note from CMake documentation

Note: *We do not recommend using* `GLOB` *to collect a list of source files from your source tree. If no CMakeLists.txt file changes when a source is added or removed then the generated build system cannot know when to ask CMake to regenerate.* ***The*** `CONFIGURE_DEPENDS` ***flag may not work reliably on all generators****, or if a new generator is added in the future that cannot support it, projects using it will be stuck. Even if* `CONFIGURE_DEPENDS` *works reliably, there is still a cost to perform the check on every rebuild.*

```
set(APP_NAME FilePrinter)

project(${APP_NAME})

# GLOB_RECURSE looks for files in given directory and it's subdirectories

file(GLOB_RECURSE SRC_FILES
     CONFIGURE_DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)

add_executable(${APP_NAME} ${SRC_FILES})
```

Setting compiler flags

-Wall -Werror for all = error

Compile flags

- Optimization flags, eg. `-O2`
- Code check flags, eg. `-Wall`
- Sanitizer flags, eg. `-fsanitize=address`
- Code coverage flags
- Other

Two ways for set

- setting flags per target by using `target_compile_options()`
- variables like `CMAKE_CXX_FLAGS`

target_compile_options()

- preferred way of setting compile flags
- remember about:
  - programming language
  - compiler
- use `$<COMPILE_LANG_AND_ID: ..., ...>`
- or `$<CXX_COMPILER_ID: >` if you have one programming language in use
- at beginning seems to be copy pasted when are set per target...
- ... but in result it helps to prevent turn of good compiler flag cause one external library

```
set(APP_NAME Application)

project(${APP_NAME})

aux_source_directory(. SRC_FILES)

add_executable(${APP_NAME} ${SRC_FILES})

target_compile_options(
    ${APP_NAME} PRIVATE $<$<COMPILE_LANG_AND_ID:CXX,Clang,AppleClang>:-Wall -Werror -O3>
                PRIVATE $<$<COMPILE_LANG_AND_ID:CXX,GNU>:-Wall -Werror -O3>
                PRIVATE $<$<COMPILE_LANG_AND_ID:CXX,MSVC>: /O2>)
```

Variables

- convention is `CMAKE_<language>_FLAGS`, eg. `CMAKE_CXX_FLAGS`
- setting value means adding flags to every target, so do not use for it flags like `-Wall -Werror`
- use it for configuration specific compile flags, like code coverage, sanitization
- always use existing flags when we set value of flags, because CMake add some of it by default, eg. debug flags

Variables

- we have also linker flags in same pattern, eg. `CMAKE_LINKER_FLAGS`
- we can specify flags for given configuration basing on `CMAKE_BUILD_TYPE` by using pattern `CMAKE_CXX_FLAGS_<build-type>`, eg.
  - `CMAKE_CXX_FLAGS_DEBUG`
  - `CMAKE_CXX_FLAGS_RELEASE`
  - `CMAKE_CXX_FLAGS_MY_OWN_BUILD_TYPE`

Linking static lib to dynamic lib

- in some compilers we need to add proper flag to make compilation successful
- eg. for GCC we need to add `-fPIC` (Position Independent Code) flag

Clang-Tidy setup

Please, do this at beginning of project

What can Clang-Tidy do?

- Cpp Core Guidelines checks
- Coding standard check, eg. google standard
- Naming convention checks
- Modern C++ checks
- Readability checks
- many other

Clang-Tidy setup

- setup main rules for root
- add/remove specific checks for given submodule if needed
- prefer using checks as `WarningsAsErrors`
- use one given version of Clang-Tidy, eg. 13

Managing dependencies

What your solutions are?

Before package managers...

- git submodules
- external libraries installers
- downloading binary releases

Submodules

- easy to download - just `git submodule update --init`
- noe easy mechanism to apply custom changes, patching
- still can have other dependencies
- compiled into build directory, so many times recompiled

Using downloaded binaries

- fast product compilation
- complicated environment setup
- potential conflict with other projects
- hard to determinate the same versions on team members setups
- for some cases we can use Docker

Conan

- software that helps to manage dependencies
- created as answer for lack of package managers in C++
- uses recipes written in python, that build dependency with given generator
- support artifactory, so ready binaries can be downloaded

What can be a dependency?

- external 3rd party library
- framework, eg. cross compiler
- separate library of my project?

External package
or not external package?

How to avoid making conan your build system

Dependency

- thing that is used by our component
- dependency has own API (interface) that we use

External dependency, but why?

- from architectural perspective separated component
- it seems to be library that can be used in other projects in the future
- team members involved in developing specific component
- one of components has complex environment setup

Before making your code external dependency

- what is the frequency of changes in that external dependency candidate?
- what is the frequency of changes in API?
- how many tasks/user stories/features requires the changes in both components?
- what is the advantage of that solution over keeping it still in the same repository?

Use your component as subproject

- create that component as subproject by creating next CMakeLists.txt
- in parent component just use `add_subdirectory()`
- onced built is not affecting main component compile time
- all versioning is done by control version system
- no factitious version releases and upgrades
- easy to use build system

Real external dependency

- 3rd party project
- component that is really used in many projects in many configurations
- stable implementation
- stable API

Build system & CI

Targets and flags as build system interface

- CMake flags
  - prefer parametrization of build system by using CMake flags over implementing many strategies inside build system
  - name flags with your project specific prefix
- Targets
  - prefer creating tasks by using CMake targets over creating additional scripts
  - create task even for CMake embedded targets like test, install, package

Thanks!

Sławomir Grabowski

grabowski@quickturnstudio.com
https://linkedin.com/in/slawomirgrabowski/
https://github.com/grabusr
https://github.com/Quick-Turn-Studio
www.sevenide.com