Lab 11

creating cross-platform C++ projects with CMake, vcpkg

22. 12. 2023

Based on slides by
Mr. Mejzlík and Mr. Klepl

Outline

1. Motivation to build and meta-build systems
2. CMake
3. Motivation for cross-platform dependency managers
4. vcpkg
5. Using CMake with vcpkg to have a project with dependencies

1) Motivation for build systems

Why you should use build systems

make
ninja
SCons

- Manual compilation is just not realistic with the real-world application
  - Too many flags, include dirs, link libs …
  - You would spend your whole life writing `g++` command

- Automation & less errors
  - Saves time
- Consistency
  - The build behaves the same for your colleagues
- Dependency management
  - Real-world projects use many third-party libraries
- Incremental build
  - Do not re-compile what is not necessary
- Distribution of your code

```
g++ -std=c++20 -O3 -g -Wall -Wextra -Werror -Wshadow -pedantic \
-Iinclude -Isrc -I/usr/local/include -Ithird_party/libA/include \
-Ithird_party/libB/include -DDEBUG -DUSE_SPECIAL_LIB \
-L/usr/local/lib -Lthird_party/libA/lib -Lthird_party/libB/lib \
-lmylibrary -lthird_party_libA -lthird_party_libB -lm -lpthread \
-o myapp \
src/main.cpp src/util.cpp src/logic.cpp src/algorithm.cpp
src/interface.cpp \
src/networking.cpp src/database.cpp src/compatibility.cpp
src/legacy.cpp \
src/new_feature.cpp src/security.cpp
src/performance_optimization.cpp \
src/third_party_integration.cpp \
-MMD -MP -MF build/dependencies.d \
-fdiagnostics-color=always -fPIC -fstack-protector-strong \
-fsanitize=address -fsanitize=undefined -flto
```

Why you should use meta build systems

CMake
Premake
GYP

- **Meta-build system**
  - It generates a temporary **project in a build system of choice**
  - Usually to ./build directory, you can delete that and generate a new one
  - It contains only references to the actual source files
- It makes your project **cross-platform**
  - E.g. make works fine across Linux platforms
  - On Windows, it's not that great

- We're going to use **CMake**
  - You have a CMake project and when you want to work on it you can generate temporary project for your favourite build system
  - Make
  - Visual Studio solution
  - Ninja
  - …

2) CMake

CMake is a meta build system

vcpkg
conan

- Cross-platform build system generator
- Widely used with C++ projects
- Target version 3.0, referred to as "Modern CMake"
  - Shift from "define flags and directories globally" to "define per targets"
  - You specify "things" for each target without affecting the other targets
- Configured per-folder by CMakeLists.txt

- Can generate projects in many build systems
  - make
  - ninja
  - Visual Studio

Old vs modern CMake

```
cmake_minimum_required(VERSION 2.8)
project(OldStyleProject)

# Global include directories for all targets
include_directories(include/)

# Global compiler flags
add_definitions(-DDEPRECATED_FLAG)

# Executable target
add_executable(old_app main.cpp)

# Linking the libraries globally
link_libraries(libA libB)
```

```
cmake_minimum_required(VERSION 3.0)
project(ModernStyleProject)

# Executable target
add_executable(new_app main.cpp)

# Specify include directories for this specific target
target_include_directories(new_app PRIVATE include/)

# Use target_compile_definitions for target-specific flags
target_compile_definitions(new_app PRIVATE -DUSE_MODERN_CMAKE)

# Use target_link_libraries for target-specific linking
target_link_libraries(new_app PRIVATE libA libB)
```

Using CMake

Min version of cmake requried

Name of the project, reference as ${PROJECT_NAME}

Define variable and set its value

Find dependency library installed in the system

Variable but only Boolean - ON/OFF | 1/0 | TRUE/FALSE

```
cmake_minimum_required(VERSION 3.20)
project(lab_06)

# Set the C++ standard
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find required packages
find_package(SFML COMPONENTS graphics REQUIRED)
find_package(Boost COMPONENTS system REQUIRED)

# Variables
set(MY_CACHE_VARIABLE "DEF VALUE" CACHE STRING "Description of the variable.")
option(TESTING "This is settable from the command line" OFF)

# ...
```

Again, variable… but this one is configurable from the outside using -D MY_CACHE_VARIABLE=hello

Using CMake

```
# ...
# Add executable
add_executable(${PROJECT_NAME} main.cpp)

# Include dirs
target_include_directories(${PROJECT_NAME} PRIVATE include/)

# Use target_compile_definitions for target-specific flags
target_compile_definitions(${PROJECT_NAME} PRIVATE -D OUR_FLAG)

# Link libraries
target_link_libraries(${PROJECT_NAME} PRIVATE sfml-graphics Boost::system)

# Include also other directories recursively
add_subdirectory(core)

# If statement
if(TESTING)
    add_subdirectory(tests)
endif()
```

Target name, use this for target_* commands

Define that executable will be produced from the given list of source files (not headers)

Include directories for this target, like -I in GCC

preprocessor defs that will be provided during compilation like #define OUR_FLAG

Link this target with these libraries, like -l in GCC

Recursively process this directory (must contain CMakeLists.txt)

If statement, true if TRUE, 1, ON

Using CMake – how to compile a program?

- CMake works in 2 steps:
  1. **Configure step**
     - CMakeLists.txt is processed and a build-system files are generated (make, ninja, VS)
     - Needs to be done just once after writing (or changing) CMakeLists.txt
  2. **Build step**
     - A generated build-system compiles the program
     - Needs to be done each time we change C++ files
- Visual Studio performs configure step automatically each time it is needed, clicking Build->BuildAll button performs the build step
- These steps can be also handled via command line:

```
1. cd project_folder          (navigate to the project folder, where CMakeLists.txt resides)
2. mkdir build && cd build    (create a directory, where the project will be built)
3. cmake ..                   (Configure step)
4. cmake --build .            (Build step)
```

3) Motivation for cross-platform dependency managers

There is no unified package manager across systems

- On Linux, you usually can get away with system package managers
  - apt, yum, dnf, …
  - pkgconfig
- On Windows, there is no such thing
- The solution is to use cross-platform package managers
  - **vcpkg**
    - https://vcpkg.io/en/
    - available for Windows, Linux, and Mac, open-source by Microsoft
  - conan
    - https://conan.io/
    - roughly the same, but from jFrog

4) vcpkg

vcpkg is cross-platform dependency manager for C++

https://vcpkg.io/en/getting-started

- List of many open-source libraries for C++ curated by Microsoft
- Libs are downloaded and locally compiled but are not installed outside of the vcpkg directory

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
.\bootstrap-vcpkg.bat
vcpkg install sfml:x64-windows boost-asio:x64-windows
```

5) CMake + vcpkg = ♥ ??

Nah, more like CMake + vckpg != total hell if you want to have your project working on both Windows and Linux

Putting it all together

- In CMake, find_package looks for dependencies in your system
- Or in paths explicitly provided by toolchain files

```
mkdir build
cd build
cmake .. -G "Visual Studio 17 2022" -A x64 \
    -DCMAKE_TOOLCHAIN_FILE=~/source/repos/vcpkg/scripts/buildsystems/vcpkg.cmake
cmake --build .
```

6) Example with Boost and Visual Studio

Create a default CMakeLists.txt

CMakeLists.txt*

```
1   cmake_minimum_required (VERSION 3.8)
2
3   project(my_project VERSION 0.1 LANGUAGES CXX)
4
5   set(CMAKE_CXX_STANDARD 20)
6   set(CMAKE_CXX_STANDARD_REQUIRED ON)
7
8   add_executable (project_app "main.cpp")
9
10  if(MSVC)
11      target_compile_options(project_app PRIVATE $<$<COMPILE_LANGUAGE:CXX>:/W4>)
12  else()
13      target_compile_options(project_app PRIVATE $<$<COMPILE_LANGUAGE:CXX>:-Wall -Wextra -pedantic>)
14  endif()
```

Set C++ standard

Add an executable name (project_app) and add all sources (cpp) it needs for compilation

Set warning flags for project_app

- If you created a VS *CMake project*, saving a CMakeLists.txt file should trigger Configure step
- If you add a new cpp file, it should be also added to the add_executable list so build system also knows about it

Add a library dependency 1

- We include a new header
  - But it is underlined – Build system can not find it
  - We need to modify CMakeLists.txt

main.cpp | CMakeLists.txt

project_app.exe - x64-Debug (Global S

```
1  #include <iostream>
2  #include <boost/asio.hpp>
3
4  int main()
5  {
6      boost::asio::io_context io;
7
8      boost::asio::steady_timer t(io, boost::asio::chrono::seconds(5));
9
10     t.wait();
11
12     std::cout << "Hello, world!" << std::endl;
13
14     return 0;
15 }
16
```

Add a library dependency 2

main.cpp CMakeLists.txt

```
1  cmake_minimum_required (VERSION 3.8)
2
3  project(my_project VERSION 0.1 LANGUAGES CXX)
4
5  set(CMAKE_CXX_STANDARD 20)
6  set(CMAKE_CXX_STANDARD_REQUIRED ON)
7
8  find_package(Boost REQUIRED)
9
10 add_executable (project_app "main.cpp")
11
12 target_include_directories(project_app PRIVATE ${Boost_INCLUDE_DIRS})
13 target_link_libraries(project_app ${Boost_LIBRARIES})
14
15 if(MSVC)
16     target_compile_options(project_app PRIVATE $<$<COMPILE_LANGUAGE:CXX>:/W4>)
17 else()
18     target_compile_options(project_app PRIVATE $<$<COMPILE_LANGUAGE:CXX>:-Wall -Wextra -pedantic>)
19 endif()
20
```

Find library

Include dirs and link libs are set for specific *target* (the project_app in our case)

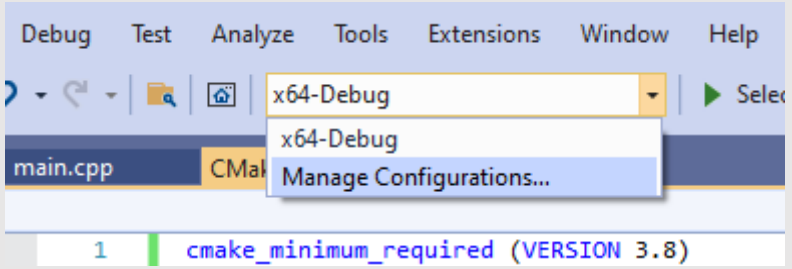
- First, we use `find_package` (line 8) – CMake will search for the lib in the system
- Then, we need to let the build-system known, where to look for boost headers (line 12) – `find_package` creates a new variable `package_INCLUDE_DIRS`
- Finally, we tell CMake to link our app with the library (line 13)

Vcpkg – what if we do not have Boost installed?

- We install boost (or other lib) using vcpkg:

```
C:\Libs\vcpkg>vcpkg install boost-asio:x64-windows
```

- We need to set CMAKE_TOOLCHAIN_FILE to point to `<vcpkg_dir>/scripts/buildsystems/vcpkg.cmake`
- We can set it in VS by clicking on Manage Configurations
- and then we set the CMake Command Argument



- After this configuration, Configure step should find all libraries installed by vcpkg

Lab 11 wrap up

- You should know
  - how to wrap a C++ project into CMake to make it cross-platform
  - how to add dependencies to the CMake project
  - how to install dependencies with vcpkg and use it with CMake

Next lab:

- Test Exam

Important date:

- **Technological Demo 19.1.**
  - Cross-platform skeleton with all specified third-party libs that is buildable
  - No actual logic required