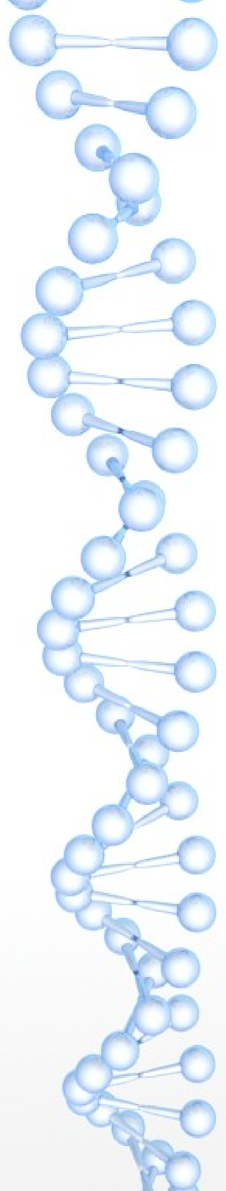
CMake

Zhivko Petrov

A guy that knows C/C++

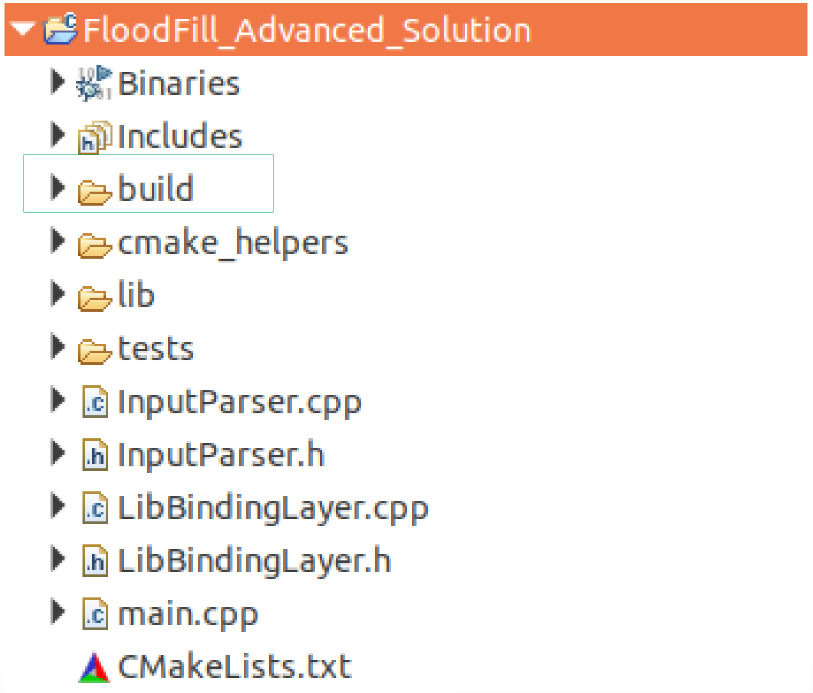


Overview

- CMake is a **cross-platform** tool that automates the building process of software C/C++ projects.

- Main **Pros**:
- Cross platform discovery of system libraries.
- Automatic discovery and configuration of the toolchain.
- Easier to compile your files into a shared library in a platform agnostic way, and in general easier to use than make.
- Out of source build

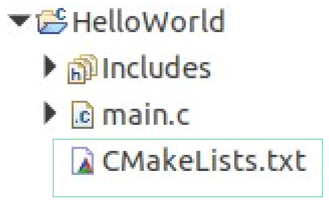
Out of source build

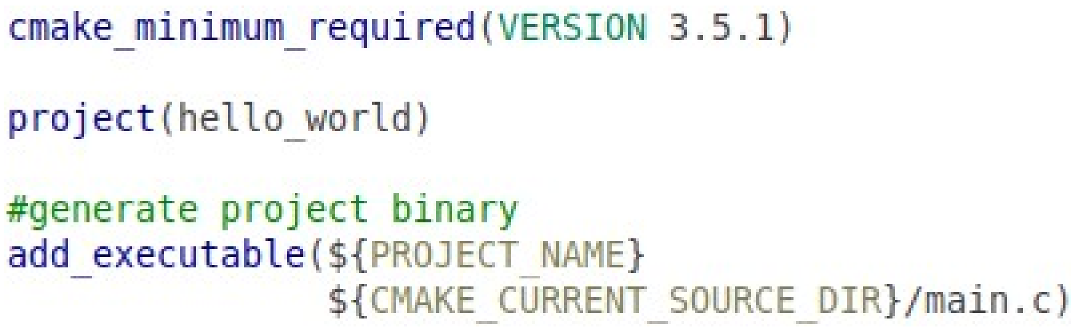


```
FloodFill_Advanced_Solution
  Binaries
  Includes
  build
  cmake_helpers
  lib
  tests
  InputParser.cpp
  InputParser.h
  LibBindingLayer.cpp
  LibBindingLayer.h
  main.cpp
  CMakeLists.txt
```

- **Bonus**: don’t need to write “make clean”
- Simply do “**rm -rf ***” in the **build** folder

Hello, World!





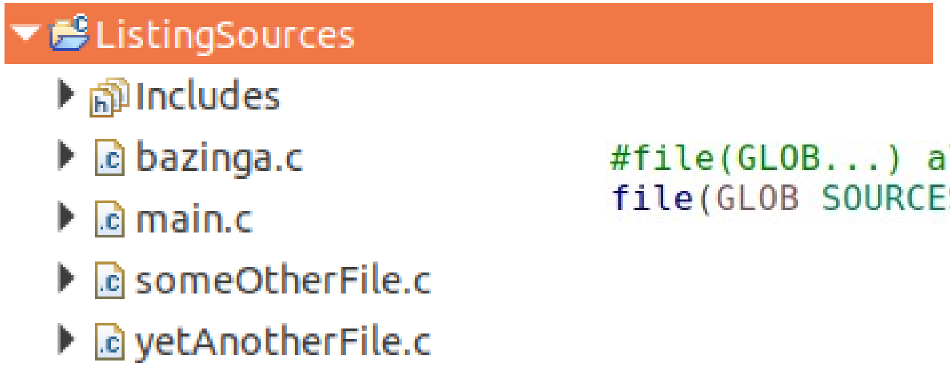
```
cmake_minimum_required(VERSION 3.5.1)

project(hello_world)

#generate project binary
add_executable(${PROJECT_NAME}
               ${CMAKE_CURRENT_SOURCE_DIR}/main.c)
```

- Every logical “level” of your file structure needs a “CMakeLists.txt”
- Run CMake from the **build**(external) folder

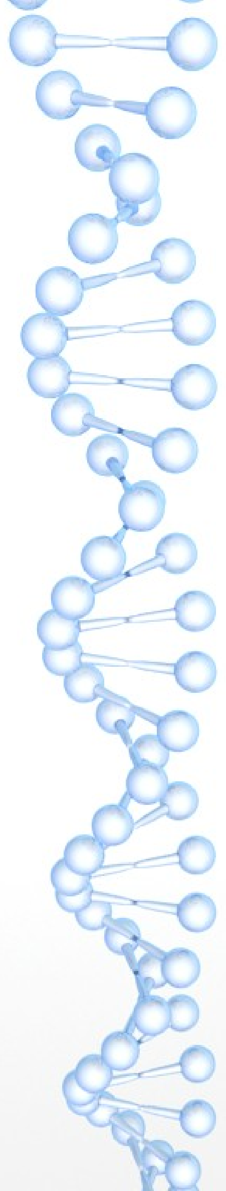
Listing your source files



```
#file(GLOB...) allows for wildcard additions:
file(GLOB SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/*c)
```

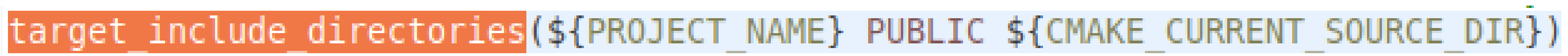
```
set(SRC_DIR ${CMAKE_CURRENT_SOURCE_DIR})

#generate project binary
add_executable(${PROJECT_NAME}
               ${SRC_DIR}/main.c
               ${SRC_DIR}/someOtherFile.c
               ${SRC_DIR}/yerAnotherFile.c
               ${SRC_DIR}/bazinga.c)
```



Including directories

- Each project can add **directories** to it’s include path



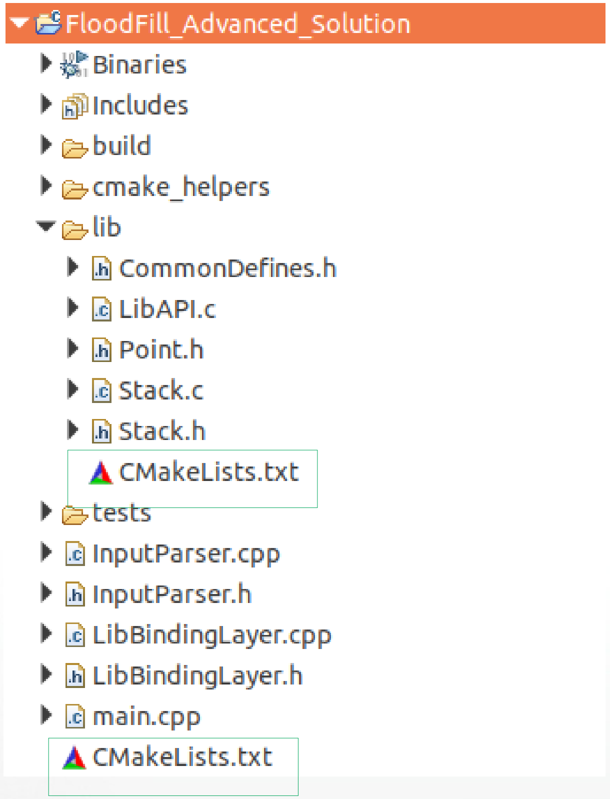
```
target_include_directories(${PROJECT_NAME} PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

- Access levels are:
- **PRIVATE** – only included for the project itself
- **PUBLIC** – included for itself and everyone,
  which links with that target (which “inherits it”)
- **INTERFACE** – not included for current target but only for those,
  which links against it

- *include_directories()* - can be used, but considered **bad** practice

Add subdirectory

- Each CMakeLists.txt file could invoke a child one (subproject)

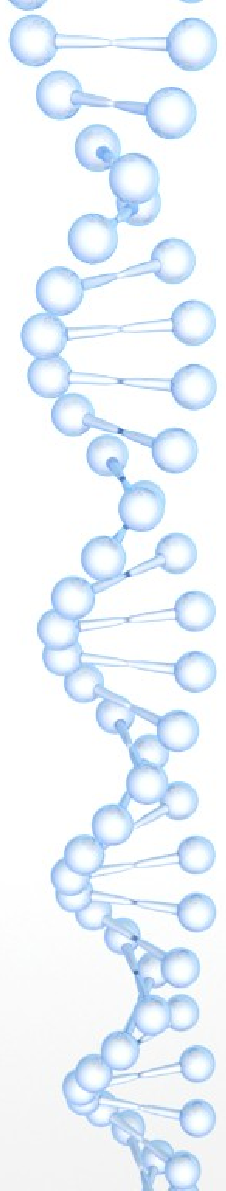


- FloodFill_Advanced_Solution
  - Binaries
  - Includes
  - build
  - cmake_helpers
  - lib
    - CommonDefines.h
    - LibAPI.c
    - Point.h
    - Stack.c
    - Stack.h
    - CMakeLists.txt
  - tests
  - InputParser.cpp
  - InputParser.h
  - LibBindingLayer.cpp
  - LibBindingLayer.h
  - main.cpp
  - CMakeLists.txt

```
#invoke child Cmake files
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/lib)
```

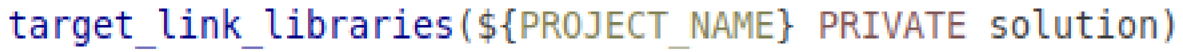
```
cmake_minimum_required(VERSION 3.5.1)

project(solution)
```



Linking with target

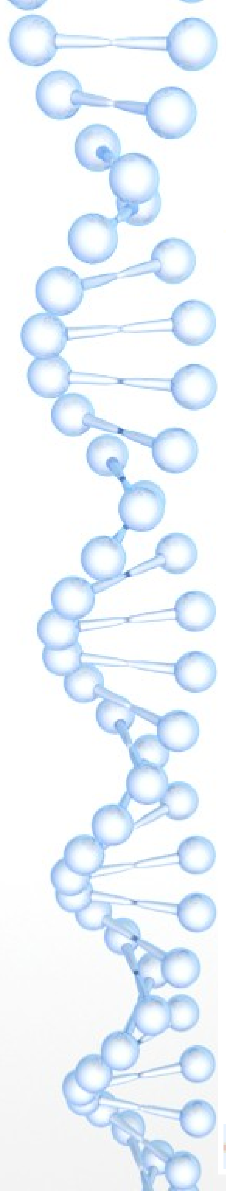
- Each project can link against other targets and directories



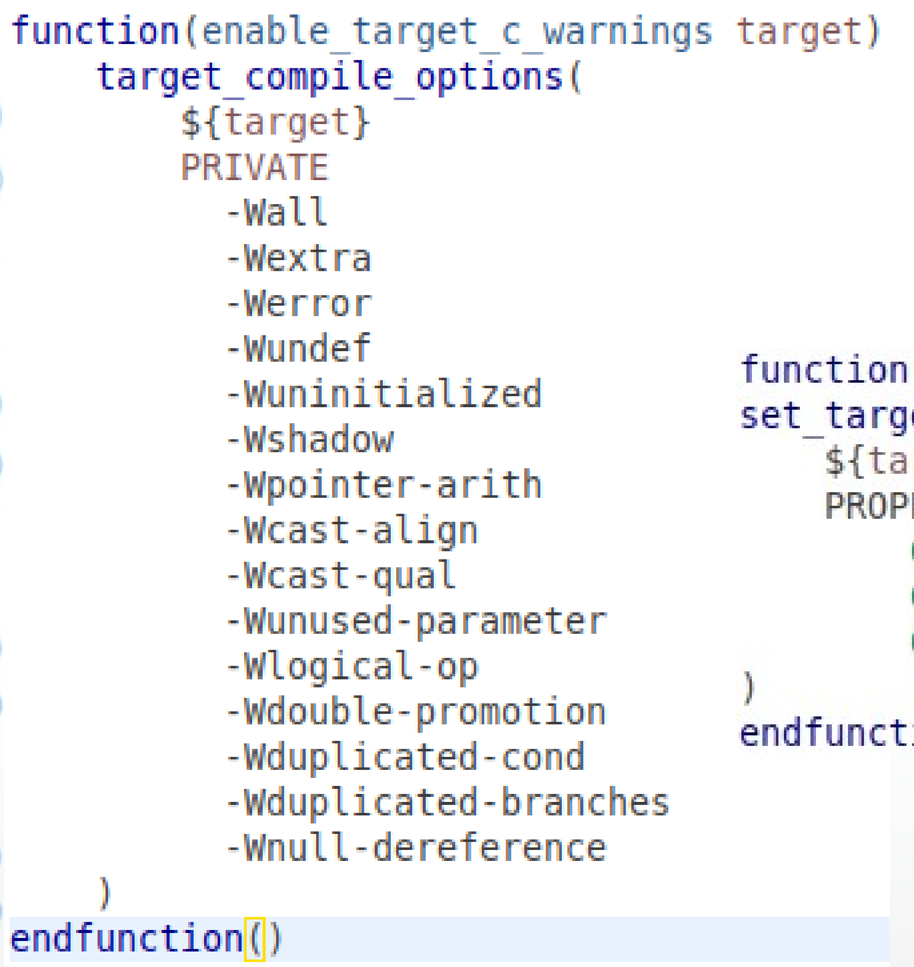
```
target_link_libraries(${PROJECT_NAME} PRIVATE solution)
```

- Access levels are the same: PRIVATE, PUBLIC and INTERFACE
- When a target links against other target:
- Cmake automatically handles the dependencies for you.
  solution will be build before ${PROJECT_NAME}
- The ${PROJECT_NAME} inherits all of “solution” PUBLIC/INTERFACE includes
- *link_directories()* - can be used, but considered **bad** practice

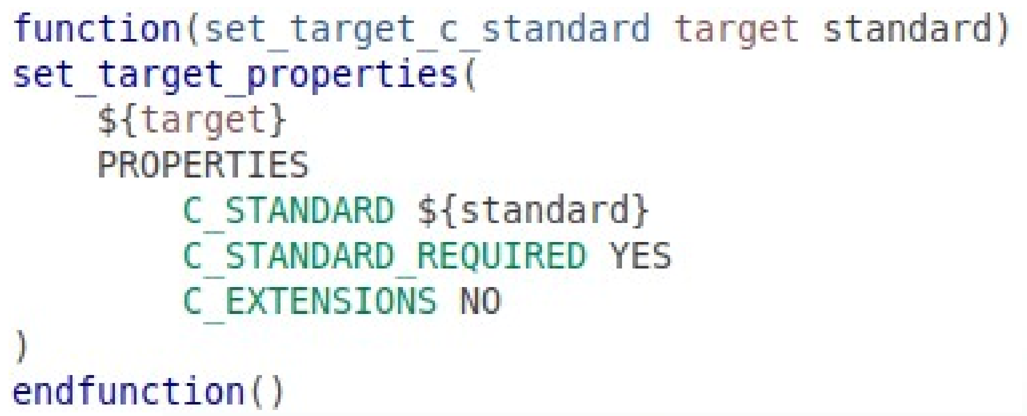
- NOTE: dependencies between target could be explicitly added
- *add_dependencies(${PROJECT_NAME} solution)*



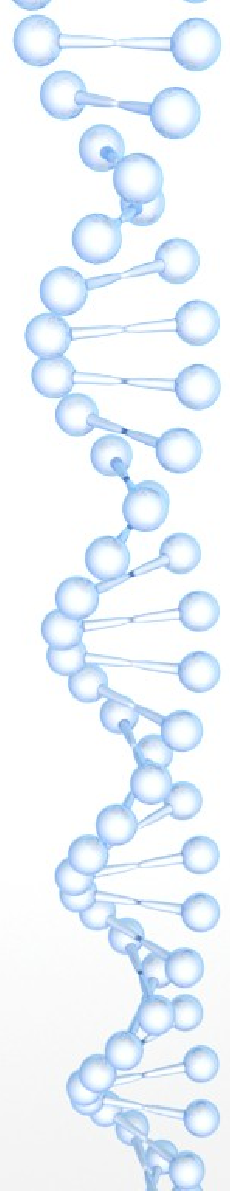
Functions



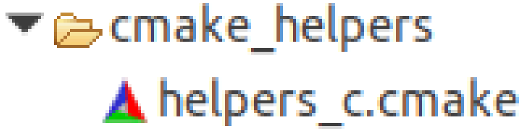
```
function(enable_target_c_warnings target)
    target_compile_options(
        ${target}
        PRIVATE
            -Wall
            -Wextra
            -Werror
            -Wundef
            -Wuninitialized
            -Wshadow
            -Wpointer-arith
            -Wcast-align
            -Wcast-qual
            -Wunused-parameter
            -Wlogical-op
            -Wdouble-promotion
            -Wduplicated-cond
            -Wduplicated-branches
            -Wnull-dereference
    )
endfunction()
```



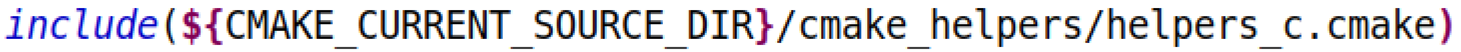
```
function(set_target_c_standard target standard)
set_target_properties(
    ${target}
    PROPERTIES
        C_STANDARD ${standard}
        C_STANDARD_REQUIRED YES
        C_EXTENSIONS NO
)
endfunction()
```



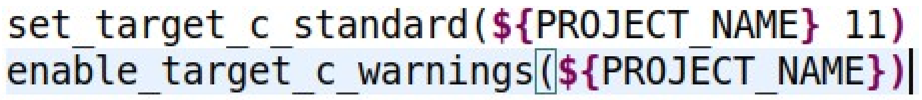
Helper files and includes



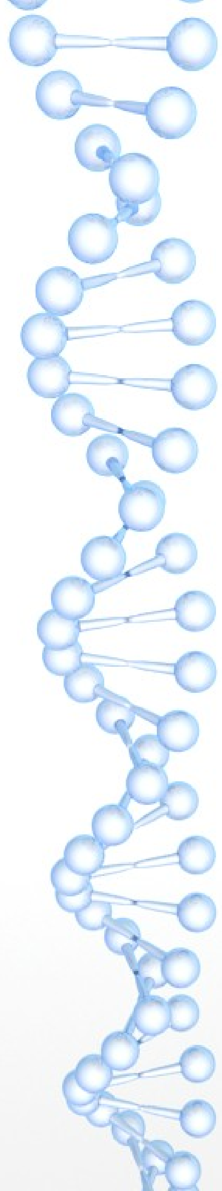
```
cmake_helpers
    helpers_c.cmake
```



```
include(${CMAKE_CURRENT_SOURCE_DIR}/cmake_helpers/helpers_c.cmake)
```

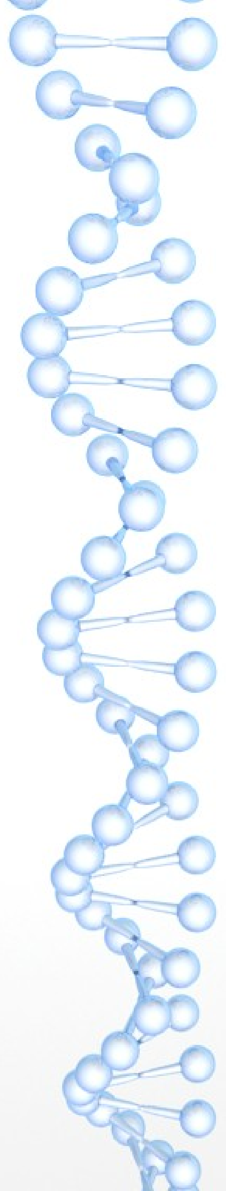


```
set_target_c_standard(${PROJECT_NAME} 11)
enable_target_c_warnings(${PROJECT_NAME})
```



Building targets

- All targets with their correct dependencies can be build with the command ***make*** (Unix Makefiles only)
- ***Make*** can be executed paralleled (example “make -j 8”)
- You can build standalone targets
- “make solution” - will build only solution library
- “make floodfill” - will build it’s dependencies first (the solution lib) and then the floodfill binary
- Note: there is a cross-platform make command “***cmake --build <dir_path>***”
- Sadly it does not use cache



Cmake invoke options

- The CMake build system can be invoked with different arguments
- One of the most common beginner friendly is the **build type**
- Can be:
- ***Debug*** (-g compiler flag)
- ***Release*** (-O3 compiler flag)
- ***RelWithDebInfo*** (-O2 -g compiler flags)
- ***MinSizeRel*** (-O2 with some flags disabled for minimum binary size)
- Invoked from the terminal:
  ”*cmake -DCMAKE_BUILD_TYPE=Release*” or
  ”*cmake -DCMAKE_BUILD_TYPE=Debug*”

thank you