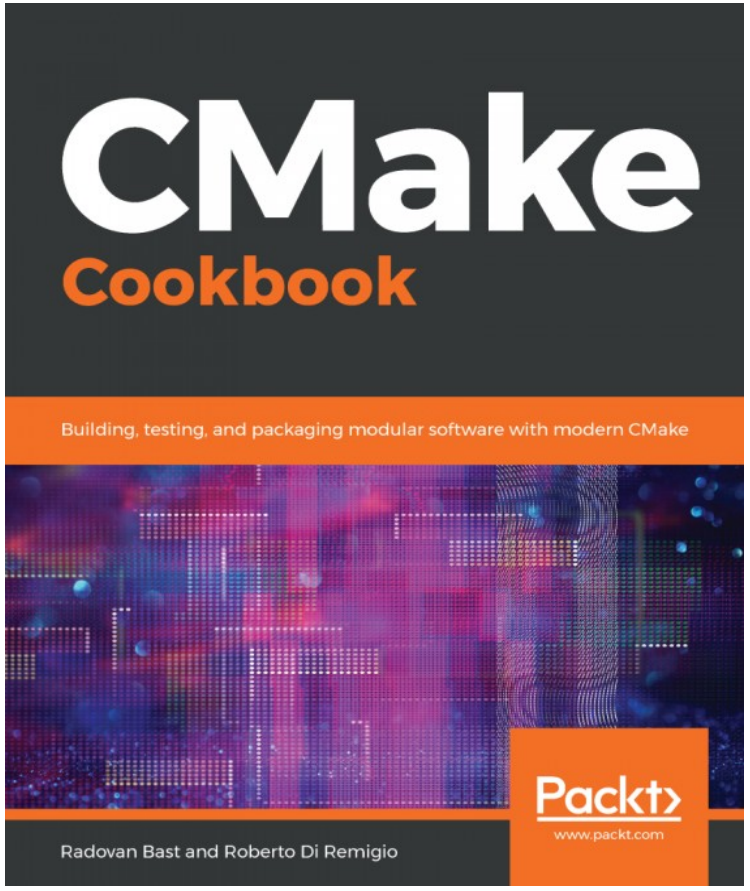
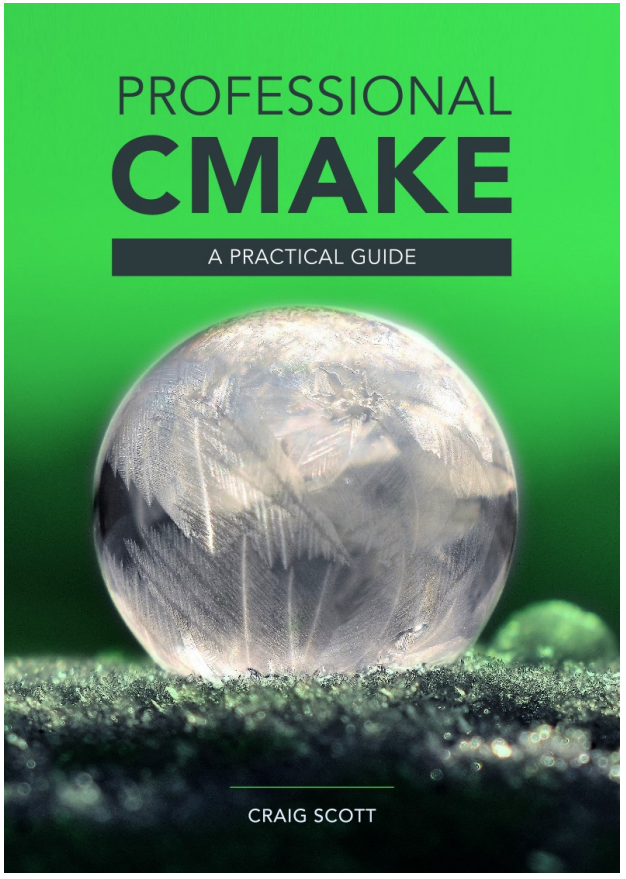
CMAKE

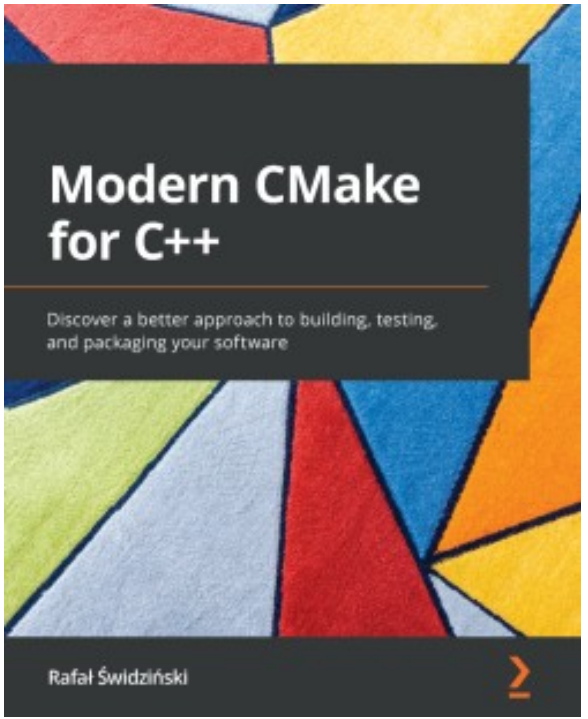
Lieven de Cock
www.codeblocks.org

Literature


CMake
Cookbook
Building, testing, and packaging modular software with modern CMake
Packt
www.packt.com
Radovan Bast and Roberto Di Remigio


PROFESSIONAL
CMAKE
A PRACTICAL GUIDE
CRAIG SCOTT

Literature


Modern CMake
for C++
Discover a better approach to building, testing,
and packaging your software
Rafał Świdziński


CMake
Best Practices
Discover proven techniques for creating and
maintaining programming projects with CMake
Dominik Berner | Mustafa Kemal Gilor

Cmake is NOT

- A build system
- No it is not !!!

Cmake is

- A build system GENERATOR

CMake : several stages

- **Configuration**
- **Generation**
- **Building**
- Testing (CTest)
- Reporting (CDash)
- Install
- Packaging (CPack)
- Package Install

Configuration

- Done by the developer
- Via CMakeLists.txt files
- What to build
- How to build
- Done while invoking CMake
- Targets : executables, libraries, custom targets

Generation

- Done while invoking CMake, after successful configuration stage
- Generates the Build System
- Many Generators (-G option)
  - Makefiles
  - Ninja
  - Visual Studio Workspaces/Solutions
  - Code::Blocks Workspaces/Projects
  - ...

Building

- Invoke the native build tools
- Or via ‘cmake --build’, a platform independent build invoke wrapper

Languages

- C
- C++ (CXX)
- Fortran
- …
-
- Variables per language: CMAKE_<LANG>_…
- Eg. : CMAKE_CXX_COMPILER, CMAKE_CXX_FLAGS

Build Types: CMAKE_BUILD_TYPE

- Debug
- Release
- RelWithDebInfo
- MinSizeRel
- Extendable : create your own

- Not specifying it => none of the above !

Modern CMake

- CMake 2.x : drop it, ditch it, …
- Don’t use variables (yourself)
- Don’t GLOB
- Usage specifications (aka how to consume)
- Out of Source builds

Out of source builds

- Source Tree
- Build system is generated in a different location outside of the source tree → binary directory, build directory, binary tree, build tree, ...
- No ignores for git/svn/… needed
- Source tree remains clean
- Entire Source tree directory structure is mimicked in the binary directory
- Multiple binary trees can exist for 1 source tree (Debug/Release/cross compilation/...)

Let’s roll : always required

- Minimal cmake version we require
- Minimum 1 project definition (can just be something at the top of our source tree), specify which languages are supported (by default ; C, CXX)

```
cmake_minimum_required(VERSION 3.20)

project (MyLittleProject)
```

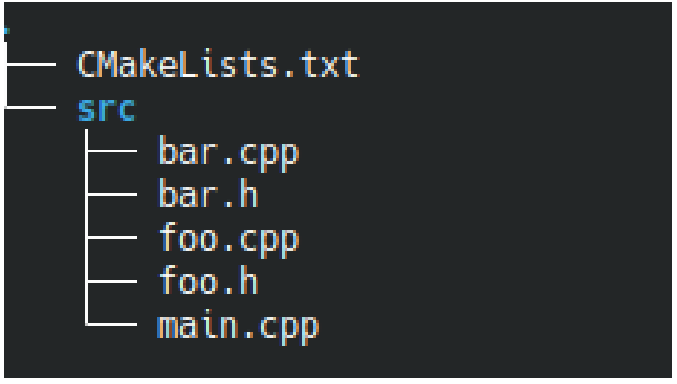
Build the source tree

- Add the next level, aka subdirectories through the CMakeLists.txt of their parent directory
- ParentDir
  - Subdir1
  - Subdir2

```
add_subdirectory(Subdir1)
add_subdirectory(Subdir2)
```

Executable

- No dependencies
- Source files (cpp and h)
- No need to enumerate headers
- add_executable




```
add_executable(CMakeExe1NoDeps
    ./src/main.cpp
    ./src/foo.cpp
    ./src/bar.cpp
)
```

Usage Specifications

- How do YOU use ME ?
- How do I use MYSELF ?
- How do YOU use ME, and I do NOT use MYSELF ?
- Applies to:
  - Include paths
  - Compile definitions
  - Compile options
  - (Linker) dependencies

Usage Specifications

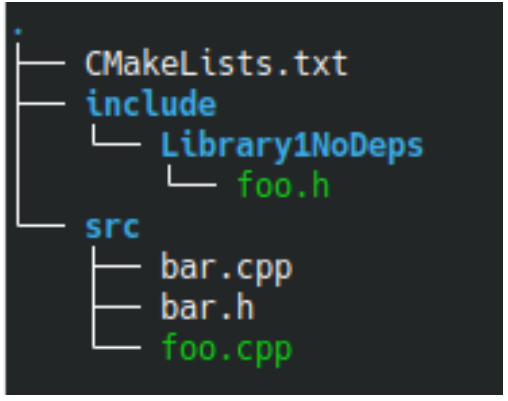
- (only) YOU ===> INTERFACE
- (only) ME ===> PRIVATE
- YOU AND ME ===> PUBLIC

(Static) Library : no dependencies

- Own (internal) headers
- Exported headers
- Users need to know (at minimum) the include path
- DRY : you do NOT want to specify this for every user
- User should just say, I will use (depend on) that library

(Static) Library

- add_library
- target_include_directories
- ${CMAKE_CURRENT_LIST_DIR}



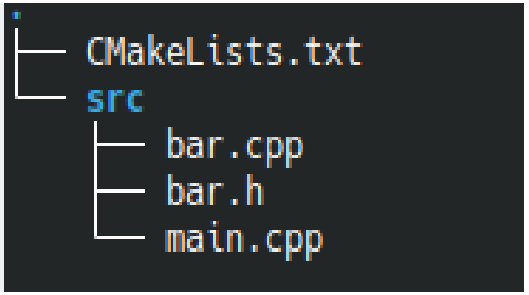

```
add_library(Library1NoDeps STATIC
    ./src/foo.cpp
    ./src/bar.cpp
)
target_include_directories(Library1NoDeps
    PUBLIC ${CMAKE_CURRENT_LIST_DIR}/include)
```

Executable using our library

- ==> dependency on our library
- Recompile when included headers change
- Link with library
- Relink, when implementation of library changes
- And first recompile the library when it changes

Executable using our library

- Just specify that we depend (PRIVATE) on the library, nothing more
- target_link_libraries



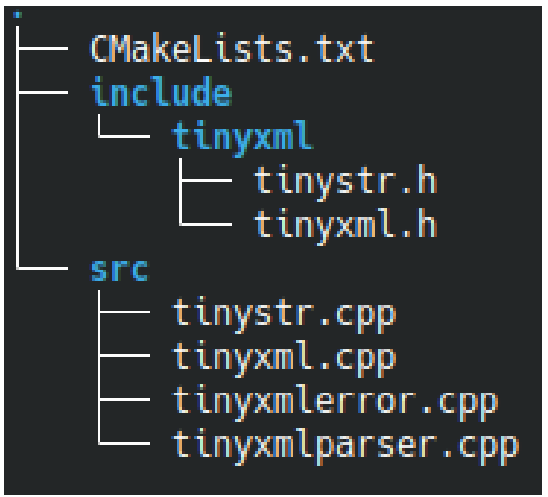

```
add_executable(Exectable2WithDependency
    ./src/main.cpp
    ./src/bar.cpp
)

target_link_libraries(Exectable2WithDependency
    PRIVATE Library1NoDeps
)
```

(Static) Library : compile definitions

- Tinyxml
- Either std::string or its own string class
- Choice determines the API
- Done by a define : TIXML_USE_STL
- Needs to be in sync for YOU AND ME => PUBLIC
- Say we always want std::string (aka stl)
- DRY : specify once and is applied to every user of the library
- User just says : depend on Tinyxml library

(Static) Library : compile definitions

- target_compile_definitions




```
add_library(tinyxml STATIC
    local/tinystr.cpp
    local/tinyxmlerror.cpp
    local/tinyxmlparser.cpp
    local/tinyxml.cpp
)

target_include_directories(tinyxml
    PUBLIC ${CMAKE_CURRENT_LIST_DIR}/include)

target_compile_definitions(tinyxml   PUBLIC TIXML_USE_STL)
```

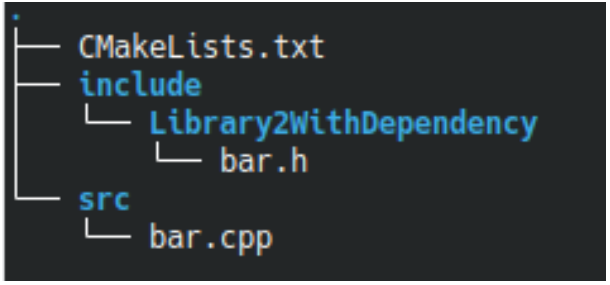
(Static) Library : with a (PRIVATE) dependency

- Library depends on other library
- PRIVATE : pure implementation detail
- As such not visible via exported headers
- DRY : specify once and is applied to every user of the library
- Obviously users of our library should in the end link with the library we are depending on (and build it first)
- And for that matter if that one depends itself on other libraries, and so on … (CMake takes care of the dependency tree)

(Static) Library : with a (PRIVATE) dependency

-

```
add_library(Library2WithDependency STATIC
    ./src/bar.cpp
)
```

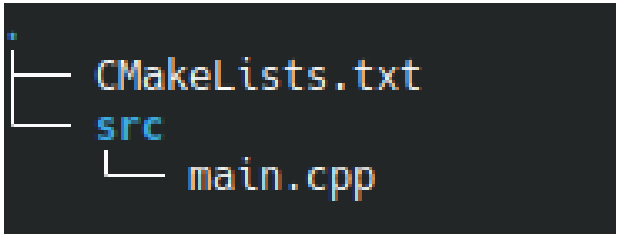



```
target_include_directories(Library2WithDependency
    PUBLIC ${CMAKE_CURRENT_LIST_DIR}/export)
```

target_link_libraries(Library2WithDependency **PRIVATE** Library1NoDeps)

Executable using our library (with its own PRIVATE dependency)

- Just specify that we depend (PRIVATE) on the library, nothing more
- We don’t see, nor care that the library we use has its own dependencies



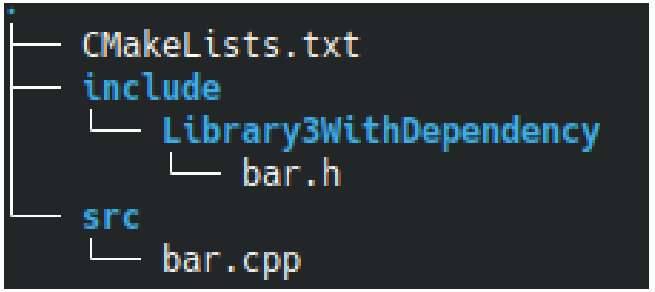

```
add_executable(Executable3WithDependency
    ./src/main.cpp
)

target_link_libraries(Executable3WithDependency
    PRIVATE Library2WithDependency
)
```

(Static) Library : with a (PUBLIC) dependency

- Library depends on other library
- **PUBLIC** : visible implementation detail
- **Visible** via **exported headers**
- So when user includes our header, the compiler should not only find our header being included, but also the headers we are including from the library we depend on
- **TRANSITIVY**
- DRY : specify once and is applied to every user of the library
- Basically at the user point we do NOT want to specify the include path (or other stuff) of that other library
- **GOOD NEWS : DO NOTHING** ==> cmake takes care of this, via the usage specification, transitivity percolates up

(Static) Library : with a (PUBLIC) dependency

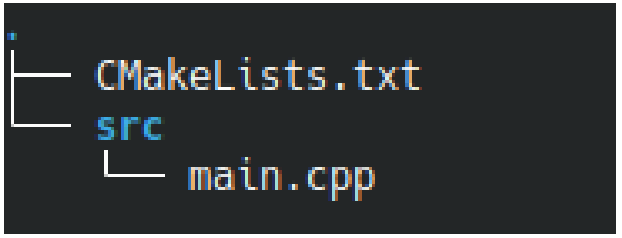
- Target link libraries
- We specify we **PUBLIC** depend on the other library, aka YOU and ME
- The YOU part is the magic key




```
add_library(Library3WithPublicDependency STATIC
    ./src/bar.cpp
)

target_include_directories(Library3WithPublicDependency
    PUBLIC ${CMAKE_CURRENT_LIST_DIR}/export)

target_link_libraries(Library3WithPublicDependency PUBLIC
Library1NoDeps)
```

Executable using our library (with its own PUBLIC dependency)

- Just specify that we depend (PRIVATE) on the library, nothing more
- We don’t see, nor care that the library we use has its own dependencies (public nor private, though public affects us)



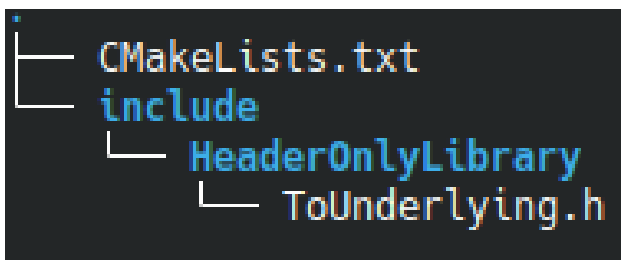

```
add_executable(Executable4WithDependency
    ./src/main.cpp
)

target_link_libraries(Executable4WithDependency
    PRIVATE Library3WithDependency
)
```

HEADER ONLY (INTERFACE) Library

- Library can still depend on other libraries
- Library can have compile definitions, compile options , ….
- There are **no source files**
- Only exported headers
- So no ME in the build story, only YOU ===> **INTERFACE**
- From users perspective, just like any other library, who cares about its special nature
- Examples:
  - Library with type declarations/definitions
  - Template library

HEADER ONLY (INTERFACE) Library

- ...



add_library(HeaderOnlyLibrary **INTERFACE**)

target_include_directories(HeaderOnlyLibrary **INTERFACE** ${CMAKE_CURRENT_LIST_DIR}/include)

Some target : compile options

- target_compile_options
- Eg for warnings suppression or other compiler options

target_compile_options(SomeTarget PUBLIC **"-Wno-unused-parameter" "-Wno-sign-compare"**)

Using 3rd party libraries

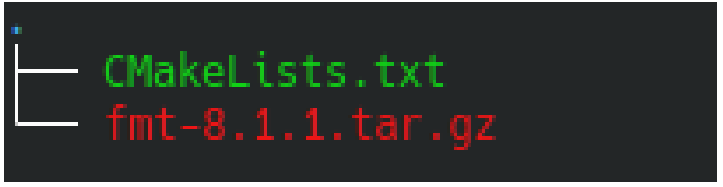
- We need to get them in our source tree
- 2 ways
- 1) FetchContent
  - For cmake based (and good behaving)
- 2) ExternalProject
  - Non cmake based
  - Cmake based but bad behaving
- Retrieve from internet (tar/zip/git/svn/…) or retrieve locally
- In the examples that follow we retrieve locally (aka we downloaded upfront and added the tar/zip manually in our repository)
- Extracted sources end up in the BINARY/BUILD directory
- Patches can be applied

GOOD behaving cmake

- Be reusable
- Be humble (serve but not rule)
- Don’t decide on language version or other compiler options (at best on your target)
- No global variables or manipulations
- Avoid to use findpackage

FetchContent

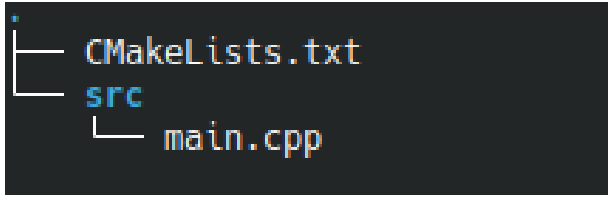
- Example : fmt library
- We will get the target (and others) : fmt::fmt-header-only
- That is an (namespaced) **ALIAS** for some internal name we would like to avoid to use and don’t care about




```
include(FetchContent)

FetchContent_Declare(fmt
    URL file://${CMAKE_CURRENT_LIST_DIR}/fmt-8.1.1.tar.gz
    )

FetchContent_MakeAvailable(fmt)
```

Executable using fmt

- Use it just like any other library (but we use the alias)




```
add_executable(ExecutableUsingFmt
    ./src/main.cpp
)

target_link_libraries(ExecutableUsingFmt
    PRIVATE fmt::fmt-header-only
)
```

ALIAS Library

- New scoped name for an existing **target**
- Target must have been found during configuration step in the source tree
- Is interpreted as a target, not just a library name

```
target_link_libraries(SomeOtherTarget
    PRIVATE Target1 #===> if no target builds this or exists it will be assumed to be the name of a library for the linker to use (and to be found by it)
    PRIVATE Foo::Target2  #target not found => configuration error
)
```

ALIAS Library : avoid name conflicts

- When you provide libraries and other targets for consumption by others, use the following convention
- Say our library would be called ‘Foo’
- Target : Foo_Foo (the real internal name)
- Alias : Foo::Foo (the name for the user)

ExternalProject_Add

- Example : libxml2 library
- Need to wrap an other build system
- Need to pass flags accordingly
- Powerful but can be complex
- We will create an INTERFACE library wrapping the outcome and making it consumable by regular CMake

ExternalProject_Add

- Example below contains some stuff from our way to allow cross compilation

include(ExternalProject)

include(ProcessorCount)
ProcessorCount(NPROCS)

ExternalProject_Add(libxml2_EP
  URL file://${CMAKE_CURRENT_LIST_DIR}/libxml2-2.9.0.tar.gz
  CONFIGURE_COMMAND PATH=${TOOLCHAIN_LOC}:$(PATH) <SOURCE_DIR>/configure --prefix=$
{CMAKE_CURRENT_BINARY_DIR} --without-python --without-zlib --without-lzma --libdir=${CMAKE_CURRENT_BINARY_DIR}/lib
$<$<BOOL:${TOOLCHAIN}>:--host=${TOOLCHAIN}> CFLAGS=-O2
  BUILD_COMMAND PATH=${TOOLCHAIN_LOC}:$(PATH) make -j${NPROCS}
  INSTALL_COMMAND PATH=${TOOLCHAIN_LOC}:$(PATH) make install
)

add_library(LibXml2_libxml2 INTERFACE)
target_include_directories(LibXml2_libxml2 INTERFACE ${CMAKE_CURRENT_BINARY_DIR}/include/libxml2)
target_link_libraries(LibXml2_libxml2 INTERFACE ${CMAKE_CURRENT_BINARY_DIR}/lib/libxml2.a)

add_dependencies(LibXml2_libxml2 libxml2_EP)

add_library(LibXml2::libxml2 ALIAS LibXml2_libxml2)

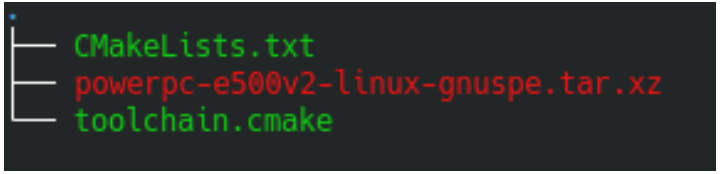
Cross compilation

- Define your cross compiler
- Incorporate it BEFORE the project()

```
cmake_minimum_required(VERSION 3.15 FATAL_ERROR)

add_subdirectory(powerpc-e500v2-linux-gnuspe)

project(Foo)
...
```

Cross compilation : powerpc example

- CMakeLists.txt → extract and variable for name of compiler




```
get_filename_component(TOOLCHAIN ${CMAKE_CURRENT_LIST_DIR} NAME)

include(FetchContent)

FetchContent_Declare(powerpc-e500v2-linux-gnuspe
    URL file://${CMAKE_CURRENT_LIST_DIR}/powerpc-e500v2-linux-gnuspe.tar.xz
    SOURCE_DIR ${CMAKE_BINARY_DIR}/${TOOLCHAIN}
    )

FetchContent_MakeAvailable(powerpc-e500v2-linux-gnuspe)
```

Cross compilation : powerpc example

- toolchain.cmake → compiler definition
- This file will be specified during command line option to cmake invocation to generate the build system

```
get_filename_component(TOOLCHAIN ${CMAKE_CURRENT_LIST_DIR} NAME)

set (TOOLCHAIN_LOC ${CMAKE_CURRENT_BINARY_DIR}/${TOOLCHAIN}/bin/)
set (CMAKE_SYSTEM_NAME Linux)  ######### this means to cmake we are cross compiling
set (CMAKE_C_COMPILER   ${TOOLCHAIN_LOC}/${TOOLCHAIN}-gcc)
set (CMAKE_CXX_COMPILER ${TOOLCHAIN_LOC}/${TOOLCHAIN}-g++)

set (CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set (CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set (CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
```

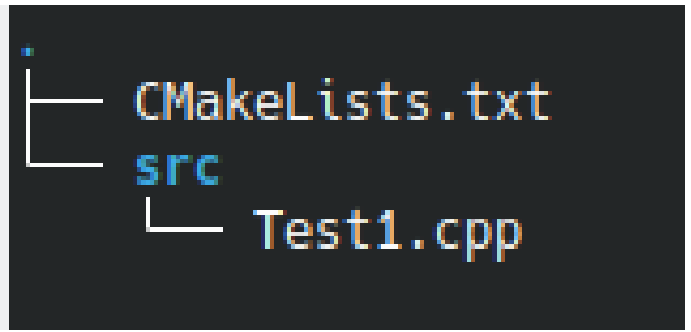
Cross compilation : powerpc example

- CMake invocaton:
- cmake
  -D CMAKE_BUILD_TYPE=Release
  -D CMAKE_TOOLCHAIN_FILE=powerpc-e500v2-linux-gnuspe/toolchain.cmake -S . -B build/powerpcRelease
- CMake 3.21 > :
  --toolchain=powerpc-e500v2-linux-gnuspe/toolchain.cmake
- CMAKE_SYSROOT : path to the sysroot

ctest

- Test is typically an executable implementing some tests
- Could be a script, …
- CMake knows several test frameworks (or they know CMake)
- Can run in parallel (-j)
- We will use catch2 as an example
- We will have 3 tests (each test (executable) can contain several tests of the testing framework)
- Test1 will pass, Test2 consists out of 2 tests and the first one will fail, Test3 will pass, but Address Sanitizer will not like it
- **include(ctest)** at top level before add_subdirectory() calls

ctest




```
add_executable(Test1
    src/Test1.cpp
)
target_link_libraries(Test1
    PRIVATE Catch2::Catch2WithMain)
add_test(
    NAME Test1
    COMMAND $<TARGET_FILE:Test1>)
```

```
#include <catch2/catch_test_macros.hpp>
namespace
{
int sum(int a, int b)
{
    return a + b;
}
}
TEST_CASE("Test2PositiveNumbers")
{
    REQUIRE(10 == sum(7, 3));
}
TEST_CASE("Test2NegativeNumbers")
{
    REQUIRE(-10 == sum(-7, -3));
}
```

ctest

```
ldco@localhost:~/Projects/Teaching/NewCmake/build/Debug> ctest
Test project /home/ldco/Projects/Teaching/NewCmake/build/Debug
    Start 1: Test1
1/3 Test #1: Test1 .............................   Passed    0.01 sec
    Start 2: Test2
2/3 Test #2: Test2 .............................***Failed    0.01 sec
    Start 3: Test3
3/3 Test #3: Test3 .............................   Passed    0.01 sec

67% tests passed, 1 tests failed out of 3

Total Test time (real) =   0.05 sec

The following tests FAILED:
          2 - Test2 (Failed)
Errors while running CTest
Output from these tests are in: /home/ldco/Projects/Teaching/NewCmake/build/Debug/Testing/Temporary/LastTest.log
Use "--rerun-failed --output-on-failure" to re-run the failed cases verbosely.
```

ctest

```
ldco@localhost:~/Projects/Teaching/NewCmake/build/Debug> ctest --output-on-failure
Test project /home/ldco/Projects/Teaching/NewCmake/build/Debug
    Start 1: Test1
1/3 Test #1: Test1 ............................   Passed    0.01 sec
    Start 2: Test2
2/3 Test #2: Test2 ............................***Failed    0.01 sec
Randomness seeded to: 3850387311

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Test2 is a Catch2 v3.0.1 host application.
Run with -? for options

-------------------------------------------------------------------------------
Test2PositiveNumbers
-------------------------------------------------------------------------------
/home/ldco/Projects/Teaching/NewCmake/Test2/src/Test2.cpp:13
...............................................................................

/home/ldco/Projects/Teaching/NewCmake/Test2/src/Test2.cpp:15: FAILED:
  REQUIRE( 22 == multiply(7, 3) )
with expansion:
  22 == 21

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed


    Start 3: Test3
3/3 Test #3: Test3 ............................   Passed    0.01 sec

67% tests passed, 1 tests failed out of 3

Total Test time (real) =   0.05 sec

The following tests FAILED:
          2 - Test2 (Failed)
Errors while running CTest
```

Custom buildtype

- Let’s create a custom build type, which will pass extra options during compilation/linking so we activate the Address Sanitizer
- Let’s call it “DebugWithAddressSanitizer”
- CMake invocation:

cmake -D **CMAKE_BUILD_TYPE=DebugWithAddressSanitizer** -S. -B build/DebugAsan

Custom buildtype : definition

```
get_property(isMultiConfig GLOBAL PROPERTY GENERATOR_IS_MULTI_CONFIG)
if(isMultiConfig)
    if(NOT "DebugWithAddressSanitizer" IN_LIST CMAKE_CONFIGURATION_TYPES)
        list(APPEND CMAKE_CONFIGURATION_TYPES DebugWithAddressSanitizer)
    endif()
else()
    set(allowableBuildTypes Debug Release RelWithDebInfo MinSizeRel   DebugWithAddressSanitizer)
    set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS "${allowableBuildTypes}")
    if(NOT CMAKE_BUILD_TYPE)
        if(NOT CMAKE_CROSSCOMPILING)
            set(CMAKE_BUILD_TYPE Debug CACHE STRING "" FORCE)
        elseif()
            set(CMAKE_BUILD_TYPE Release CACHE STRING "" FORCE)
        endif()
    elseif(NOT CMAKE_BUILD_TYPE IN_LIST allowableBuildTypes)
        message(FATAL_ERROR "Invalid build type: ${CMAKE_BUILD_TYPE}")
    endif()
endif()

if(CMAKE_CXX_COMPILER_ID MATCHES Clang)
    set(STATIC_LIBASAN "-static-libsan")
    set(STATIC_LIBTSAN "-static-libsan")
else()
    set(STATIC_LIBASAN "-static-libasan")
    set(STATIC_LIBTSAN "-static-libtsan")
endif()

# DebugWithAddressSanitizer
set(CMAKE_C_FLAGS_DEBUGWITHADDRESSSANITIZER "${CMAKE_C_FLAGS_DEBUG}" CACHE STRING "" FORCE)
set(CMAKE_CXX_FLAGS_DEBUGWITHADDRESSSANITIZER "${CMAKE_CXX_FLAGS_DEBUG} -fsanitize=address ${STATIC_LIBASAN}" CACHE STRING "" FORCE)
set(CMAKE_EXE_LINKER_FLAGS_DEBUGWITHADDRESSSANITIZER "${CMAKE_EXE_LINKER_FLAGS_DEBUG} -fsanitize=address ${STATIC_LIBASAN}" CACHE STRING ""
FORCE)
set(CMAKE_SHARED_LINKER_FLAGS_DEBUGWITHADDRESSSANITIZER "${CMAKE_SHARED_LINKER_FLAGS_DEBUG} -fsanitize=address ${STATIC_LIBASAN}" CACHE STRING
"" FORCE)
set(CMAKE_STATIC_LINKER_FLAGS_DEBUGWITHADDRESSSANITIZER "${CMAKE_STATIC_LINKER_FLAGS_DEBUG}" CACHE STRING "" FORCE)
set(CMAKE_MODULE_LINKER_FLAGS_DEBUGWITHADDRESSSANITIZER "${CMAKE_MODULE_LINKER_FLAGS_DEBUG}" CACHE STRING "" FORCE)

```

Ctest - asan

```cpp
TEST_CASE("TestViolateAsan")
{
    int* x = new int();
    (void)x;
    REQUIRE(21 == multiply(7, 3));
}
```

```
ldco@localhost:~/Projects/Teaching/NewCmake/build/DebugAsan> ctest -R Test3 --output-on-failure
Test project /home/ldco/Projects/Teaching/NewCmake/build/DebugAsan
    Start 3: Test3
1/1 Test #3: Test3 .............................***Failed    0.12 sec
Randomness seeded to: 413574477
===============================================================================
All tests passed (1 assertion in 1 test case)


=================================================================
==17508==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x4a3bc8 in operator new(unsigned long) ../../../../libsanitizer/asan/asan_new_delete.cpp:95
    #1 0x4eafb9 in CATCH2_INTERNAL_TEST_0() (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x4eafb9)
    #2 0x5b4618 in Catch::TestInvokerAsFunction::invoke() const (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x5b4618)
    #3 0x56de65 in Catch::TestCaseHandle::invoke() const (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x56de65)
    #4 0x56b9f4 in Catch::RunContext::invokeActiveTestCase() (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x56b9f4)
    #5 0x56b2b2 in Catch::RunContext::runCurrentTest(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&, std::__cxx11::basic_string<char, std::char_traits<char
>, std::allocator<char> >&) (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x56b2b2)
    #6 0x567798 in Catch::RunContext::runTest(Catch::TestCaseHandle const&) (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x567798)
    #7 0x57d16d in Catch::(anonymous namespace)::TestGroup::execute() (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x57d16d)
    #8 0x57fd7d in Catch::Session::runInternal() (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x57fd7d)
    #9 0x57f315 in Catch::Session::run() (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x57f315)
    #10 0x4ecc9b in int Catch::Session::run<char>(int, char const* const*) (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x4ecc9b)
    #11 0x4ecad6 in main (/home/ldco/Projects/Teaching/NewCmake/build/DebugAsan/Test3/Test3+0x4ecad6)
    #12 0x7f3aa9c9d2bc in __libc_start_main (/lib64/libc.so.6+0x352bc)

SUMMARY: AddressSanitizer: 4 byte(s) leaked in 1 allocation(s).


0% tests passed, 1 tests failed out of 1

Total Test time (real) =   0.13 sec

The following tests FAILED:
          3 - Test3 (Failed)
Errors while running CTest
```

QUESTIONS