

**PEOPLE FIRST,
THEN
TECHNOLOGY.**



About this seminar.

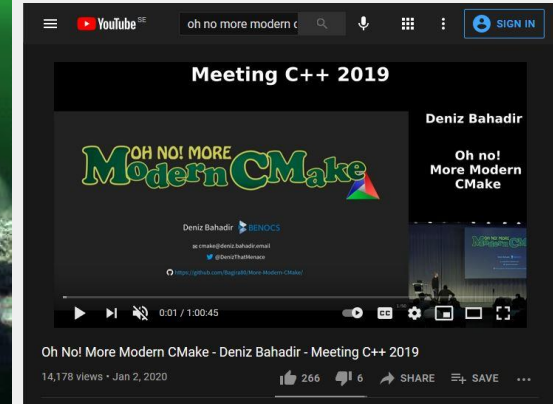
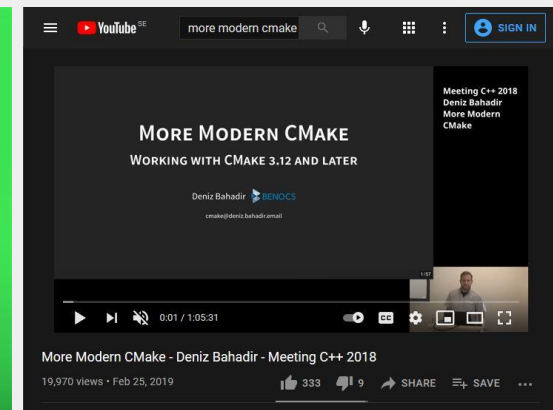
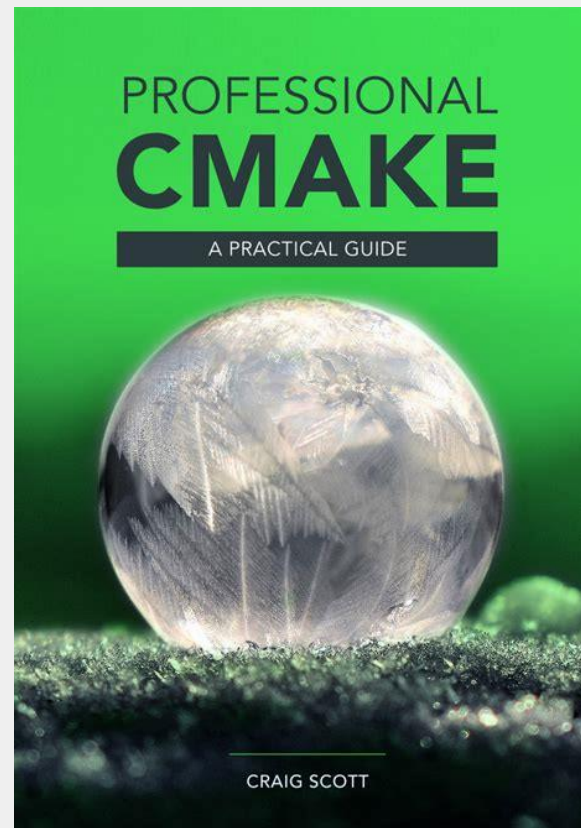


Luis Gisler

I am a cybersecurity and DevOps consultant with Semcon. I try to follow the 'keep it simple' and 'it just works' philosophies of Software development.

E-mail

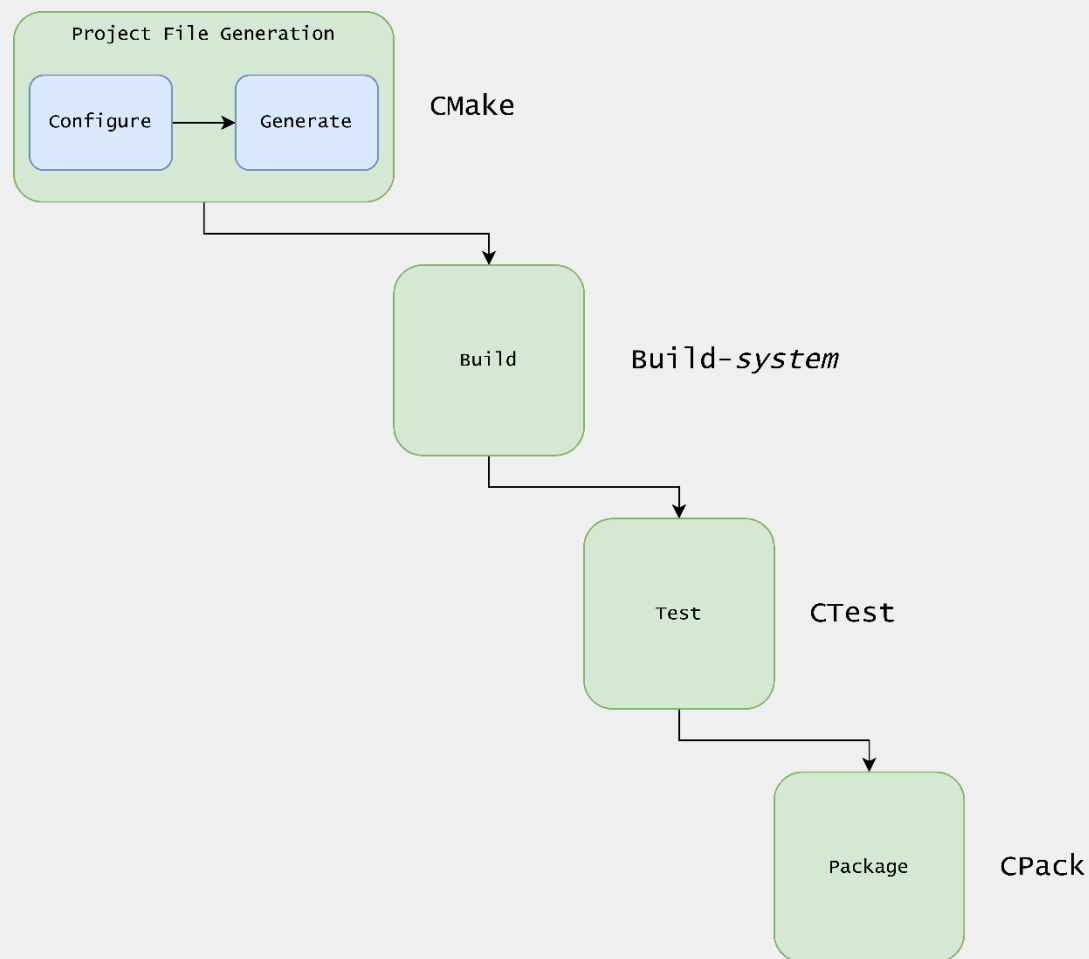
luis.gisler@semcon.com



Agenda.

- 1. Introduction**
- 2. What is CMake?**
- 3. Basic syntax and features.**
- 4. Setting up a project.**
- 5. Exporting, installing, and packaging projects.**
- 6. Live Demo.**

What is CMake?



- CMake is a suite of tools.
 - `cmake`, `ctest`, and `cpack`.
- CMake is a build-system *generator*.
 - Not a build-system!
 - generates *input files* for the build-system.
 - Supports: Make, Ninja, Visual Studio, XCode, ...
- It is cross-platform.
 - Supports *running on*: Linux, Windows, OSX, ...
 - Supports *building* cross-platform.

Modern CMake.

- CMake started around 1999/2000
 - Mainly focused on being '*directory orientated*'
 - referred to as 'traditional CMake'
- CMake is 'Modern' since around 3.0 [June 2014]
 - Introduced a new concept of being '*target orientated*'
- CMake became 'More Modern' since 3.12 [July 2018]
 - Completed a lot of requested 'Modern CMake' features
- What version should we use?
 - As new as possible, CMake has great backwards compatibility!

Recommended practices.

- Developers should install the newest available version.
 - By default Ubuntu [LTS] 18.04 has 3.10. Kitware has a APT repository with 3.20.
 - Windows current default version is 3.20.
 - Pip supports 3.18.
- Projects should require the latest version that will not stop people building the project.
 - The oldest supported Ubuntu LTS is a safe choice which is CMake 3.10.
- Ensure that the top level CMakeLists.txt file has a `cmake_minimum_required()` as the first line.
- Since CMake 3.12, we have the optional maximum value as well.
 - `cmake_minimum_required(VERSION 3.10...3.15)`

Variables.

```
# Local variable
# set(varName value ... [PARENT_SCOPE])

set(myVar a b c)      # myVar = "a;b;c"
set(myVar a;b;c)      # myVar = "a;b;c"
set(myVar "a b c")    # myVar = "a b c"
set(myVar a b;c)      # myVar = "a;b;c"
set(myVar a "b c")    # myVar = "a;b c"

# Cache variables
# set(varName value... CACHE type "docstring" [FORCE])

set(myVar foo)         # Local myVar = foo
set(result ${myVar})   # result = foo
set(myVar bar CACHE STRING "") # Cache myVar

set(result ${myVar})    # First run:      result = bar
                       # Subsequent runs: result = foo

set(myVar fred)
set(result ${myVar})    # result = fred

# Environment variables, RARELY USED
# set(ENV{varName} value)

set(ENV{PATH} "$ENV{PATH}:/opt/myDir")
```

- CMake treats all variables as strings.
- Normal variables have directory scope.
- Lists are just a single string with list items separated by semicolon.
- The `set()` command will only overwrite a cache variable if the `FORCE` keyword is present, unlike normal variables.
- A cache variable can be used so that the developer can override the value [from cli or gui] without having to edit the `CMakeLists.txt` file.

Recommended practices.

- Try to use cache variables to control optional parts of the build instead of encoding the logic in build scripts outside of CMake.
- Try to avoid relying on environment variables being defined, except for maybe PATH. Builds should be predictable, reliable and easy to set up.
- Establish a naming convention early. For cache variables consider grouping them with a common prefix. ie. PROJECTNAME_VARIABLE_NAME
- Avoid defining a variable with the same name as a cache variable.
- Try to use pre-defined CMake cache variables instead of custom cache variables.

Flow control.

```
if(expression1)
    # commands ...
elseif(expression2)
    # commands ...
else()
    # commands ...
endif()

foreach(loopVar arg1 arg2 ...)
    # ...
endforeach()

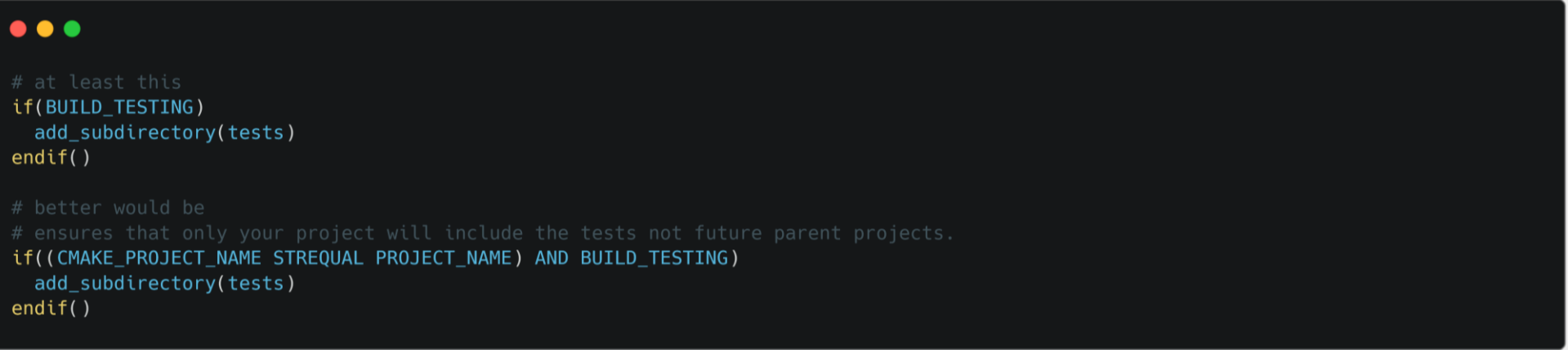
while(condition)
    # ...
endwhile()

# Common pattern, often used with variables defined with
# option() sets a BOOL CACHE variable
option(BUILD_MYLIB "Enable building the MyLib target")
if(BUILD_MYLIB)
    add_library(MyLib src1.cpp src2.cpp)
endif()
```

- CMake's logic for what is considered true or false is more complicated than most programming languages. In general.
- The following are the most common true values. case-insensitive.
 - TRUE, YES, ON, Y, or non-zero number.
- The following are false. case-insensitive.
 - FALSE, NO, OFF, N, IGNORE, NOTFOUND, "", *-NOTFOUND, or 0.
- Numbers are converted to BOOL following usual C rules.

Recommended practices.

- Minimize opportunities for strings to be unintentionally interpreted as variables.
 - Set minimum CMake version to above 3.1 to disable the old behaviour that allowed implicit conversions of a quoted string value to a variable name.
- Avoid variable expressions with quotes, use a specific string comparison instead.
- Use the predefined cache variable BUILD_TESTING to decide if you should build your tests.



```
# at least this
if(BUILD_TESTING)
    add_subdirectory(tests)
endif()

# better would be
# ensures that only your project will include the tests not future parent projects.
if((CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME) AND BUILD_TESTING)
    add_subdirectory(tests)
endif()
```

Macros and functions.

- Functions and macros behave very similar to their counterparts in C/C++.
- Functions introduce a new scope and the arguments become variables inside the body.
- Macros 'paste' the block of code in place and the arguments are expanded to strings.
- Function scope receives a *copy* of all the variables from the calling scope.
- In general, we should use functions instead of macros.

```
#
function(func arg)
  if(DEFINED arg)
    message("Function arg is a defined variable")
  else()
    message("Function arg is NOT a defined variable")
  endif()
endfunction()

func(foobar)
# output: Function arg is a defined variable

macro(macrg arg)
  if(DEFINED arg)
    message("Macro arg is a defined variable")
  else()
    message("Macro arg is NOT a defined variable")
  endif()
endmacro()

macrg(foobar)
# output: Macro arg is NOT a defined variable

function(func myArg)
  message("myArg = ${myArg}")
endfunction()

func(foobar)
# output: myArg = foobar

macro(macrg myArg)
  message("myArg = ${myArg}")
endmacro()

macrg(foobar)
# output: myArg = foobar
```

Recommended practices.

- In general, prefer using functions rather than macros, since the use of the function scope better isolates the function's effects on the calling scope.
- Macros should generally only be used when the contents of the macro *must* be executed in the calling scope.
- Prefer passing in required variables explicitly with the function arguments rather than relying on the *copy* variables that were implicitly added. It will be more robust to future changes and clearer and easier to maintain.
- It is *highly recommended* to use keyword-based handling provided by `cmake_parse_arguments()`.
 - in simple terms, parses the arguments of a function/macro and defines variables.
- Keep project functions in a top-level cmake directory and add these as XXX.cmake files.
 - This allows you to `include()` the cmake script files throughout your project.



```
cmake_minimum_required(VERSION 3.10)

project(Workshop VERSION 4.7.2 LANGUAGES CXX)

add_library(Toolbox)
add_library(Workshop::Toolbox ALIAS Toolbox)

target_sources(Toolbox
    PRIVATE
        "src/hammer.cc"
        "src/drill.cc"
        "src/saw.cc"
)

target_include_directories(Toolbox
    PRIVATE
        "src"
    INTERFACE
        "include"
)

target_compile_features(Toolbox
    PRIVATE
        "cxx_std_17"
)

add_executable(MainTool)

target_sources(MainTool
    PRIVATE
        "app/main.cc"
)

target_link_libraries(MainTool
    PRIVATE
        Workshop::Toolbox
)
```

What are CMake targets?

- Targets are *mainly* libraries and executables, but you can also add custom targets.
- Targets contain within themselves all their relevant information in the form of target properties.
- On the left we have:
 - target 1: library: Toolbox
 - target 2: alias library: Workshop::Toolbox
 - target 3: executable: MainTool
- Each target handles its own source files, include directories, linker dependencies and compiler features.
- We will discuss private and interface keywords further on.
 - PRIVATE is a build requirement
 - INTERFACE is a usage requirement.

Recommended practices.

- Target names do not need to be related to the project name.
 - Do *NOT* do this: `add_executable(${PROJECT_NAME})`
 - Choose a name that relates to what the target does not which project it is a part of.
- For library targets resist the temptation to add a 'lib' prefix or suffix to the name.
 - On most platforms CMake will add this automatically following appropriate conventions.
 - Linux: `libTargetName.a`
 - Windows: `TargetName.lib`
- Avoid defining `STATIC` or `SHARED` libraries in the `add_library()` call instead allow this to be set with the `BUILD_SHARED_LIBS` cache variable.
- *ALWAYS* [try to] use the `PRIVATE|INTERFACE|PUBLIC` keywords in the `target_link_libraries()` command.

Build requirements.



- Everything that is needed to [successfully] *build* that target.
 - source files.
 - include search paths.
 - pre-processor macros.
 - link-dependencies.
 - compiler/linker options.
 - compiler/linker features.
 - e.g. support for a C++ standard

Usage requirements.

- Everything that is needed to [successfully] *use* that target.
 - source files. [*But normally not!*]
 - include search paths.
 - pre-processor macros.
 - link-dependencies.
 - compiler/linker options.
 - compiler/linker features.
 - e.g. support for a C++ standard


Target orientated commands.

```



  //# Build requirements
  target_include_directories( <TargetName> PRIVATE <include-search-dir> ... )
  target_compile_definitions( <TargetName> PRIVATE <macro-definitions> ... )

```

```


  //# Both a build and a usage requirement
  target_include_directories( <TargetName> PUBLIC <include-search-dir> ... )
  target_compile_definitions( <TargetName> PUBLIC <macro-definitions> ... )
  target_compile_options( <TargetName> PUBLIC <compiler-options> ... )
  target_compile_features( <TargetName> PUBLIC <feature> ... )
  target_sources( <TargetName> PUBLIC <source-file> ... )
  target_link_libraries( <TargetName> PUBLIC <dependency> ... )
  target_link_options( <TargetName> PUBLIC <linker-options> ... )
  target_link_directories( <TargetName> PUBLIC <linker-search-dir> ... )

```

```

  target_compile_features( <TargetName> INTERFACE <feature> ... )
  target_sources( <TargetName> INTERFACE <source-file> ... )
  target_link_libraries( <TargetName> INTERFACE <dependency> ... )
  target_link_options( <TargetName> INTERFACE <linker-options> ... )
  target_link_directories( <TargetName> INTERFACE <linker-search-dir> ... )

```

Recommended practices.

- Projects should seek to define all dependencies with the `target_link_libraries()` command.
- Where possible, prefer to use the `target_...()` commands to describe relationships between targets and to modify compiler and linker behaviour.
- In general, prefer avoiding directory property commands. While they can be convenient in a few specific circumstances, consistent use of `target_...()` commands instead, will establish clear patterns for developers to follow.
- Avoid directly modifying the various `CMAKE_..._FLAGS` variables consider these for the developer to change locally if they wish.
- Developers should be familiar with the `PRIVATE|INTERFACE|PUBLIC` relationships and choose the correct one. They are crucial to the `target_...()` commands and to the install and packaging stages of projects.
 - Start with `PRIVATE` and only then set to `PUBLIC` if required.

Looking for dependencies.

- There are several methods for finding things in CMake with various find_ commands we will focus on find_package().
- There are two main ways of finding packages.
 - Find modules.
 - Config details.
- Both modules and config details provide the location of programs, libraries, flags to be used by consuming target.
- The variable CMAKE_PREFIX_PATH is often enough for finding the Find modules or config details.
- Packages with config details offer a richer, more robust way for projects to retrieve information about that package.

```
## Short form used mainly with FindPackage.cmake
## modules
find_package(packageName
  [version [EXACT] ]
  [QUIET] [REQUIRED]
  [ [COMPONENTS] component1 [component2...] ]
  [OPTIONAL_COMPONENTS component3 [component4...] ]
  [MODULE]
  [NO_POLICY_SCOPE]
)

## Example finding Qt5
find_package(Qt5 5.9 REQUIRED
  COMPONENTS Gui
  OPTIONAL_COMPONENTS DBus
)

## When using REQUIRED keyword COMPONENTS keyword
## is optional.
find_package(Qt5 5.9 REQUIRED Gui Widgets Network)

## Long form will skip the search for modules and
## continue to the config search. PackageConfig.cmake
find_package(packageName
  [version [EXACT] ]
  [QUIET | REQUIRED]
  [ [COMPONENTS] component1 [component2...] ]
  [OPTIONAL_COMPONENTS component3 [component4...] ]
  [NO_MODULE | CONFIG]
  [NO_POLICY_SCOPE]
  [NAMES name1 [name2 ...] ]
  [CONFIGS fileName1 [fileName2...] ]
  [HINTS path1 [path2 ...] ]
  [PATHS path1 [path2 ...] ]
  [PATH_SUFFIXES suffix1 [suffix2 ...] ]
  [CMAKE_FIND_ROOT_PATH_BOTH |
  ONLY_CMAKE_FIND_ROOT_PATH |
  NO_CMAKE_FIND_ROOT_PATH]
  [<skip-options>]
)
```

Recommended practices.

- Wherever possible, prefer using `find_package()` rather than finding individual things within the package.
- Ensure that *your* packages follow conventions so that users should only need to append to the `CMAKE_PREFIX_PATH` variable to find *your* package.
- The package registry feature [default behaviour in CMake 3.14 and below] should be approached with caution. Projects should avoid calls to `export(PACKAGE)` which uses this.
- Use `CMAKE_MODULE_PATH` when searching for `FindPackageName.cmake` files or when a module is brought in via `include()`. For everything else, including searching for `PackageNameConfig.cmake` files use `CMAKE_PREFIX_PATH`.
- Your packages *should* provide a `PackageNameConfig.cmake` file *NOT* a `FindPackageName.cmake` file. `FindPackageName.cmake` is for finding packages that do not supply config details.

Including external projects.

```
//requires CMake 3.11
include(FetchContent)
FetchContent_Declare(googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG release-1.10.0
)

//requires CMake 3.14
FetchContent_MakeAvailable(googletest)
```

```
cmake_minimum_required(VERSION 3.14)

include(FetchContent)
FetchContent_Declare(CompanyXToolchains
  GIT_REPOSITORY ...
  GIT_TAG ...
  SOURCE_DIR ${CMAKE_BINARY_DIR}/toolchains
)
FetchContent_MakeAvailable(CompanyXToolchains)

project(MyProj)
```

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchains/beta_cxx.cmake ..
```

- FetchContent is provided by CMake 3.11+ and allows you to include an external project into yours. Although CMake 3.14+ is recommended.
- FetchContent will download the external project sources and add their targets into your build so you can link towards them.
- FetchContent allows developers to work on multiple projects at once, making changes in both the main project and its dependencies.
- FetchContent can also be used before *your* project call. Allowing you to use remote configuration files in your cli call.

Recommended practices.

- FetchContent module is a good choice when other projects need to be added to the build in a way that allows them to be worked on at the same time. Allowing developers the freedom to work across projects and temporarily switch to local checkouts, change branches, and test with various release versions.
- Do not assume a project is the top level project. Use variables use `CMAKE_CURRENT_SOURCE_DIR` instead of `CMAKE_SOURCE_DIR` and `CMAKE_CURRENT_BINARY_DIR` instead of `CMAKE_BINARY_DIR`.
- Always use the namespace-aliased target name when linking [if available]
 - Prefer linking to `MyProj::foo` rather than `foo`
- Avoid forcibly setting cache variables, if you must change cache variables [to effect subprojects] considering using regular variables with the same name to shadow the cache variables.

CTest and GoogleTest.

```
cmake_minimum_required(VERSION 3.14)

project(MP3 VERSION 3.3.1 LANGUAGES CXX)

add_library(Compression)
add_library(MP3::Compression ALIAS Compression)

target_sources(Compression
    PRIVATE
        "src/Algorithm.cc"
)

target_include_directories(Compression
    PUBLIC
        "include"
)

include(CTest)

include(FetchContent)

FetchContent_Declare(googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG release-1.10.0
)

FetchContent_MakeAvailable(googletest)

// # Add test executable
add_executable(TestCompression "test/TestCompression.cc")

target_link_libraries(TestCompression
    PRIVATE
        MP3::Compression
        GTest::GTest
)

add_test(NAME Compression COMMAND TestCompression)
```

- CMake and ctest provide support for building, executing and determining pass/fail status of test. But the project is responsible for providing the test code itself and this is where testing frameworks like GoogleTest can be useful.
- Frameworks like GoogleTest facilitate the writing of clear, well-structure test cases that integrate well into CMake and ctest.
- Quickly and easily integrate the GoogleTest framework with ctest support. Seen on the left

Recommended practices.

- Make the test name short but specific to what is being tested.
- Avoid adding the word test to the name of the test. Note name of test and target name are different.
- Assume the project may one day be incorporated into a larger project with many other tests. Aim to use test names that are specific enough to reduce the chance of name clashes.
- Give parent projects the ability to include or exclude *your* tests. A suitable cache variable would be TEST_PROJECTNAME.
- Consider adding labels to the tests so that projects can group them.
 - eg. LongRunning if a test takes a long time to complete.

Exporting, installing and packaging.

- The final step of software development, delivery or deployment is often the most critical. The earlier in the project this is considered the easier it is to implement.
- Things to understand
 - Installation directory layout.
 - Base install location.
- Differentiate between install locations and build locations in the target properties.
- The preferred way for an installed project to make itself available for other CMake projects is to provide a config package file. eg. `PackageNameConfig.cmake`
- Always include a `PackageNameConfigVersion.cmake` file alongside your config file.

```
cmake_minimum_required(VERSION 3.14)

Project(Proj VERSION 1.2.3 LANGUAGES CXX)

include(GNUInstallDirs)

add_library(Foo)
target_sources(Foo
    PRIVATE
        "src/Foo.cc"
)

target_include_directories(Foo
    PRIVATE
        "src"
    INTERFACE
        $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}/somewhere>
        $<BUILD_INTERFACE:${Proj_BINARY_DIR}/anotherDir>
        $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
)

set_target_properties(Foo
    PROPERTIES
        SOVERSION ${PROJECT_VERSION_MAJOR}
        VERSION ${PROJECT_VERSION}
)

include(GNUInstallDirs)
install(TARGETS Foo
    EXPORT FooTargets
)

include(CMakePackageConfigHelpers)
write_basic_package_version_file(
    FooConfigVersion.cmake
    VERSION ${PROJECT_VERSION}
    COMPATIBILITY ExactVersion
)

install(
    FILES "${CMAKE_CURRENT_BINARY_DIR}/FooConfigVersion.cmake"
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Foo
)

install(
    DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/somewhere ${Proj_BINARY_DIR}/anotherDir
    DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/Foo
    COMPONENT headers
    FILES_MATCHING PATTERN "*.h"
)

install(EXPORT FooTargets
    FILE FooConfig.cmake
    NAMESPACE Foo::
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/Foo
)
```

Recommended practices.

- Projects should have a clear understanding of installation directory structures for all their supported platforms.
- Where possible projects should follow standard package layouts. This can be achieved with the GNUInstallDirs module. As of CMake 3.14 GNUInstallDirs added default locations based on file type.
- Strong consider making *your* packages relocatable.
- When defining target usage requirements use the generator-expression `$<BUILD_INTERFACE>:...` to correctly express the installed header search paths.
- For any library target, prefer to set the `INCLUDES_DESTINATION` instead of the `$<INSTALL_INTERFACE>:...` this is often more convenient and concise. In CMake 3.14 this is done automatically by `install[TARGETS ...]` command.

CPack

```
//# Bottom of the top-level CMakeLists.txt

set(CPACK_PACKAGE_NAME MyProj)
set(CPACK_PACKAGE_VENDOR MyCompany)
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "CPack example project")
set(CPACK_PACKAGE_INSTALL_DIRECTORY ${CPACK_PACKAGE_NAME})
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
set(CPACK_VERBATIM_VARIABLES YES)
set(CPACK_PACKAGE_DESCRIPTION_FILE ${CMAKE_CURRENT_LIST_DIR}/Description.txt)
set(CPACK_RESOURCE_FILE_WELCOME ${CMAKE_CURRENT_LIST_DIR}/Welcome.txt)
set(CPACK_RESOURCE_FILE_LICENSE ${CMAKE_CURRENT_LIST_DIR}/License.txt)
set(CPACK_RESOURCE_FILE_README ${CMAKE_CURRENT_LIST_DIR}/Readme.txt)

if(WIN32)
    set(CPACK_GENERATOR ZIP WIX)
elseif(APPLE)
    set(CPACK_GENERATOR TGZ productbuild)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Linux")
    set(CPACK_GENERATOR TGZ RPM)
else()
    set(CPACK_GENERATOR TGZ)
endif()

include(CPack)
```

- The easiest way to create a *cpack* input file is by including the CPack module, once for the whole project. Often the very last line of the top-level CMakeLists.txt.
- CPack will take everything that is to be installed and package it into different specified generators.
- CPack Generators:
 - Simple archives [.zip, .tar.xz, etc]
 - UI Installers [IFW, NSIS, WIX]
 - Non-UI packages [RPM, DEB packages]

Recommended practices.

- A good starting point is to consider providing at least one simple archive format and then one native format for each target platform.
- If a UI installer is appropriate for all platforms, consider using Qt installer framework IFW generator for a consistent UX across all platforms.
- Give consideration to whether end users should be able to install the product on a headless system. Meaning non-UI methods must be available.
- When defining component names, allow for the possibility that the project may be used as a child in some larger project hierarchy.
- Include the project name in the component name to avoid clashes between projects.
- The component name shown to users in UI installers, package file names, etc can be set to something different than the internal CMake component names.

Superbuild structure

- When the dependencies do not use CMake this is generally preferred.
- Each dependency is its own separate build with the main project directing the overall sequence and how details are passed between them.
- Mainly uses ExternalProject module.
- Will avoid target name clashes.
- Strongly helps if the dependencies are mature and well defined.

Non-superbuild structure

- If a project has no dependencies or if dependencies are being brought into the main build using FetchContent or gitsubmodules.
- Usually requires 'good' CMake hygiene in all dependencies to avoid name clashes and directory properties being set.
- Usually the most flexible, allows for development across multiple projects simultaneously.

Live demo.

`https://github.com/lgisler/brown-fox.git`