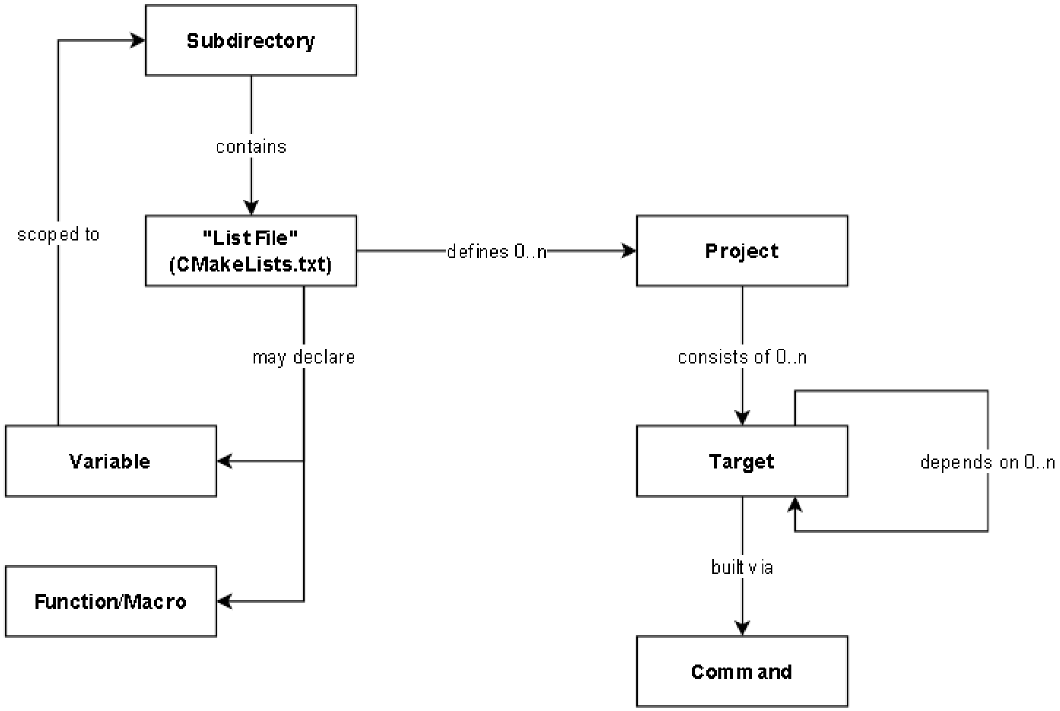
CMake

Feature Walkthrough
Instructive Examples
Lessons Learned
Best Practices for HBK

CONFIDENTIAL

CMake's "Object Model"



- Project – A conceptual grouping of targets; has a name, version number, and other metadata
- Target – A "thing" to be "built" which may depend on other "things"
- Command – A way to "build" a "target" from "sources"
- Subdirectory – A hierarchical tree of variable scopes (variables in one subdirectory are visible to all child subdirectories)
- "List File" – Fancy CMake term for the CMakeLists.txt file
- Variable – There are "normal" and "cache" variables (details later)
- Functions & Macros – Just what it sounds like; difference is a "function" introduces a new scope while a "macro" uses its parent's scope (just like a C function vs. preprocessor macro)

CMake – What is a “Target”?

- A target is “a thing to be built.”
- Targets are built via commands. Most target type / language combinations are built in, but custom targets and commands can be defined.
- Targets may depend on other targets.
- Targets may have **properties** which are **used by the build commands** to affect the build.
- Things like C/C++ include directories, compiler options, etc. are handled via properties.
- These properties have **scopes** which affect how they propagate to other dependent targets.

CMake – Property Scopes

- **PUBLIC** - The setting applies to **the target itself and** to **dependent targets.**
  - Public header include paths
  - Compile options which a target and its dependencies must agree on

- **PRIVATE** – The setting applies to **the target itself** but **not** to dependent targets.
  - Compile options such as warning/error levels which do not affect dependents
  - Statically-linked libraries **which are self-contained in the target** (no instances passed from a target to its dependencies/dependents)

- **INTERFACE** – The setting applies **only** to **dependents**
  - Very common use case: “fake” targets imported via find_package()
  - In some cases, public headers which a library itself doesn’t need, e.g. generated code

CMake – Variable Types

- **Normal** variables
  - Similar to variables in most programming languages
  - “Last setter wins”
  - Each of the following opens a new scope block:
    - A subdirectory
    - A function
    - **NOT** ~~a macro~~
  - Each scope block inherits normal variables from its parent, but NOT vice-versa
  - … unless the child uses set(… PARENT_SCOPE), which is like “export” in UNIX shells

- **Cache** variables
  - “First setter wins”
  - Value is stored between runs in the “cache” which can be modified with tools like ccmake
  - Global (no scopes)

CMake – Packages (1) Overview

- Packages can be “exported” so that find_package() can find them
- Exports typically result in the following being installed:
  - A “package configuration file” (libnameConfig.cmake)
    - Contains user-defined setup like find_dependency()
    - Can be generated automatically by CMake or via a template
    - Includes libnameTargets.cmake
  - A “package version file” (libnameConfigVersion.cmake) (optional, but highly recommended)
    - Checks if the installed version is “compatible” with the requested one
    - Can be generated automatically by CMake (according to a version compatibility policy) or generated manually by the user for complex version compatibility rules
  - A “targets” file (libnameTargets.cmake)
    - Generates the imported targets with appropriate interface properties for linking to parent targets
  - One or more “configuration” files (libnameTargets-noconfig.cmake, or -release, -debug, etc.)
    - Contains custom settings for a particular build configuration when side-by-side configurations are installed

CMake – Packages (2) Steps to Produce

- Install the library itself: install(TARGETS …)
  - If needed: install public headers: install(FILES …) or install(DIRECTORIES …)

- Generate the "targets" file: install(EXPORT …)

- Generate the "config" file
  - From a custom template: configure_package_config_file()

- Generate the "version" file
  - CMake-generated using a canned compatibility policy: write_basic_package_version_file()

- Install the targets, config, and version files: install(FILES …)

CMake – Packages (3) Steps to Consume

- It’s easy!!
  - find_package(pkgname 1.2.3 REQUIRED)

- Troubleshooting - Where does it look?
  - It needs to find:
    - pkgnameConfig.cmake (or pkgname-config.cmake)
    - pkgnameConfigVersion.cmake (or pkgname-config-version.cmake)
  - In:
    - A standard location
    - Locations set in CMAKE_PREFIX_PATH
    - Explicit location set in pkgname_DIR

CMake - FetchContent

- Fetches **and** loads packages from an external source
- Fetching logic inherited from ExternalProject

- **How does it work?**
  - The idea is simple:
    - 1. Download / check out the project in _deps (or a user-specified location)
    - 2. Load it with add_subdirectory()

- **What else does it do?**
  - It can "redirect" future calls to find_package() so that the fetched content satisfies the package dependency

CMake – FetchContent – Lessons Learned

- (Almost…) Always use the find-else-fetch pattern!
  - … unless it's GTest!
  - Or any other **development-only** dependency that **should not be installed locally** due to e.g. high probability of version conflict
  - If cmake >= 3.24 is available, use FIND_PACKAGE_ARGS to save a step!

- Don't use FetchContent_Populate() unless you **really** need to!
  - Use FetchContent_MakeAvailable() instead, and save a step!

- Don't encode authentication (e.g. GitHub PAT) in URLs if it's at all possible to avoid it!
  - For GitHub, use credential helpers (e.g. "gh" CLI tool) instead!

- Avoid SSH, prefer HTTPS wherever possible!
  - Authentication is usually easier
  - SSH is more often blocked than HTTPS

CMake – Other Tips and Lessons Learned

- Always use namespaces and namespaced aliases for targets!
  - This (mostly) avoids issues where you give target_link_libraries() the wrong library name and don’t know it!
- Never use global compile flags / include dirs / etc.!
  - Use target-scoped settings and correct PRIVATE/PUBLIC/INTERFACE scope.
- Keep it simple – as much as possible!
  - If you find yourself manually specifying include paths to a CMake-managed dependency, or drilling into that project’s internal CMake variables, or find_library()-style variables, **you’re probably doing it wrong!**

CMake – What We Didn’t Cover ☹ Maybe Next Time?

- Side-by-Side Configurations
  - Debug / Release
  - Shared / Static
  - Multithreaded / Single-Threaded
- Testing Framework
  - enable_testing()
  - add_test()
- Additional find_package() modes (Module Mode, etc.)

- Presets
- Package Components (e.g. Boost)
- Custom Targets / Commands
- Cross-compilation and Toolchains
- Policies
- Script Mode

CMake – Questions?

Thanks for your time!