https://simplifyCpp.org

DRAFT

Mastering CMake

The Fast Reference Guide

Prepared by Ayman Alheraki

First Edition

Mastering Modern CMake

The Fast Reference Guide

Prepared by Ayman Alheraki

simplifycpp.org

October 2025

Contents

Author's Preface

Dear Readers,
Following the release of my comprehensive book on CMake, which exceeded 600 pages, I noticed from reviews and feedback that many found it lengthy and challenging to navigate—especially for those seeking a quick, practical reference to the essential aspects of CMake.
For this reason, I decided to create this **Concise CMake Handbook**: a fast, focused, and practical reference for developers and C++ enthusiasts who want to leverage modern build environments efficiently. The objectives of this handbook are to provide:

- Core foundational and advanced concepts of CMake without overwhelming details.
- The latest features and best practices introduced post-CMake 2022, including **Presets, FetchContent, CI/CD integration, Unity Builds, and Link Time Optimization (LTO)**.
- Practical guidance for structuring projects, managing libraries, and handling dependencies effectively across multiple platforms.
- Clear, actionable tips to avoid common pitfalls and transition smoothly from legacy CMake to modern, target-oriented workflows.

This handbook is designed to serve as a **quick and reliable reference**, easily accessible during daily development tasks, while encompassing the latest methods and practices that every CMake professional should know.

I sincerely hope that this work proves beneficial to all readers, simplifying the learning and usage of CMake, whether you are a beginner or an experienced developer aiming to stay current with modern build techniques.
Wishing you success and efficiency in all your projects and software endeavors,

Stay Connected

For more discussions and valuable content about **Mastering Modern CMake: The Fast Reference Guide**, I invite you to follow me on **LinkedIn**:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Wishing everyone success and prosperity.

Ayman Alheraki

Chapter 1

Introduction to Modern CMake

1.1 What is CMake and Why It Matters Today

CMake is a cross-platform build system generator that automates the creation of native build environments. Unlike traditional build tools, CMake doesn't compile code directly; instead, it generates build files (such as Makefiles, Ninja files, or Visual Studio project files) tailored to the user's platform and toolchain. This abstraction allows developers to focus on specifying the structure and dependencies of their projects without being tied to a specific build system.

The significance of CMake in modern software development lies in its ability to handle complex, multi-platform projects efficiently. It supports a wide range of compilers and IDEs, making it a preferred choice for projects that aim for portability and scalability. Additionally, CMake's integration with continuous integration/continuous deployment (CI/CD) pipelines has streamlined automated testing and deployment processes.

1.2 Build System Generators: Ninja, Make, MSVC, Xcode, and Others

CMake supports various build system generators, each catering to different development environments:

- **Ninja**: A small, high-speed build system designed for parallel builds. It's particularly effective for incremental builds, reducing build times significantly.
- **Make**: A traditional build system that uses Makefiles. It's widely supported but can be slower for large projects due to its sequential nature.
- **MSVC (Microsoft Visual C++)**: Generates project files compatible with Microsoft's Visual Studio. It's essential for Windows-based development using Microsoft's toolchain.
- **Xcode**: Generates project files for Apple's Xcode IDE, facilitating development on macOS and iOS platforms.

Choosing the appropriate generator depends on the target platform and the development tools in use. CMake's flexibility allows developers to switch between these generators with minimal changes to the project configuration.

1.3 CMake vs. Make/Autotools

CMake offers several advantages over traditional build systems like Make and Autotools:

- **Cross-Platform Support**: CMake can generate build files for various platforms, whereas Make and Autotools are often tailored for Unix-like systems.
- **Ease of Use**: CMake's scripting language is designed to be more user-friendly and less error-prone compared to the shell scripting used in Autotools.

- **Integration with IDEs**: CMake seamlessly integrates with modern IDEs, providing features like code completion and debugging support, which are not inherently available in Make or Autotools.

While Autotools is still prevalent in many open-source projects, CMake's growing adoption reflects its advantages in modern development workflows.

1.4 Target-Based Approach vs. Global Variables

Modern CMake emphasizes a target-based approach, moving away from the traditional use of global variables. In this paradigm:

- **Targets**: Represent build artifacts like executables or libraries. Each target encapsulates its own properties, such as source files, include directories, and compiler options.
- **Properties**: Attributes associated with targets, allowing for fine-grained control over the build process.

This approach enhances modularity and maintainability, as changes to one target's properties don't inadvertently affect others. It also improves the clarity of build configurations, making them more intuitive and less error-prone.

1.5 Post-2022 Highlights: Presets, FetchContent, CI/CD, Unity Builds

Recent developments in CMake have introduced features that further streamline the build process:

- **Presets**: Introduced in CMake 3.21, presets allow developers to define and share common configuration settings across different environments. This feature simplifies the setup of CI/CD pipelines and ensures consistency across development setups.
- **FetchContent**: This module enables the downloading and building of external dependencies directly within the CMake project. It simplifies the management of third-party libraries, reducing the need for external package managers.
- **CI/CD Integration**: CMake's support for presets and its command-line interface make it well-suited for integration into CI/CD pipelines. This integration facilitates automated testing and deployment, enhancing the development workflow.
- **Unity Builds**: Unity builds involve grouping multiple source files into a single compilation unit to reduce compile times. CMake supports this technique, allowing developers to speed up the build process without altering the project's structure.

These advancements reflect CMake's ongoing evolution to meet the demands of modern software development, emphasizing efficiency, scalability, and ease of use.

Chapter 2

Installation and Setup

2.1 Installing CMake on Windows, Linux, and macOS

Windows:

- **Official Installer:** Download the latest CMake installer from the official website. The installer provides a straightforward setup process, allowing you to choose installation options such as adding CMake to the system PATH for all users.
- **Package Managers:** For users who prefer package managers, tools like Chocolatey or Winget can be used to install CMake. For example, using Winget:

```
winget install cmake
```

Linux:

- **Package Managers:** On Debian-based distributions like Ubuntu, you can install CMake using APT:

```
sudo apt update
sudo apt install cmake
```

For Red Hat-based distributions, use YUM or DNF:

```
sudo dnf install cmake
```

- **Manual Installation:** For the latest version, you can download the source code from the official website and compile it manually:

```
./bootstrap
make
sudo make install
```

macOS:

- **Homebrew:** The easiest way to install CMake on macOS is through Homebrew:

```
brew install cmake
```

- **MacPorts:** Alternatively, if you use MacPorts:

```
sudo port install cmake
```

2.2 Command Line, GUI, and VSCode Setup

Command Line:

- **Verify Installation:** After installation, verify that CMake is correctly installed by running:

```
cmake --version
```

This command should display the installed version of CMake.

- **Help Command:** To access the help documentation, use:

```
cmake --help
```

This provides information on available commands and usage.

GUI:

- **CMake GUI:** CMake provides a graphical user interface (GUI) that can be used for configuring and generating build files. The GUI allows you to set cache variables, choose the generator, and configure the project interactively.
  - **Windows:** The GUI is included with the Windows installer.
  - **Linux/macOS:** You can install the GUI separately if desired. For example, on Ubuntu:

    ```
    sudo apt install cmake-qt-gui
    ```

Visual Studio Code (VSCode):

- **Extensions:** To integrate CMake with VSCode, install the following extensions:
  - **C/C++:** Provides IntelliSense and debugging support.
  - **CMake Tools:** Offers CMake support, including configuring, building, and debugging CMake projects.
- **Setup:**
  - **Open Folder:** Open your CMake project folder in VSCode.
  - **Configure Project:** Use the Command Palette (Ctrl+Shift+P) and run `CMake: Configure`.
  - **Build Project:** After configuration, you can build the project using `CMake: Build`.
  - **Debug:** Set breakpoints and use the built-in debugger to debug your application.

2.3 Verifying Installation

After installing CMake, it's essential to verify that the installation was successful and that the tool is functioning correctly.

- **Check Version:** Run the following command to check the installed version of CMake:

```
cmake --version
```

This should output the version number of CMake, confirming that it's installed.

- **Access Help:** To view the help documentation and available commands, use:

```
cmake --help
```

This provides a list of commands and options you can use with CMake.

- **Test Configuration:** Create a simple `CMakeLists.txt` file in a new directory:

```
cmake_minimum_required(VERSION 3.10)
project(HelloWorld)
add_executable(hello main.cpp)
```

Then, run the following commands:

```
mkdir build
cd build
cmake ..
```

If CMake generates the build files without errors, the installation is successful.
By following these steps, you can ensure that CMake is correctly installed and set up on your system, ready for use in your development projects.

Chapter 3

Your First Project

3.1 Project Layout and Minimal `CMakeLists.txt`

A well-organized project structure is essential for scalability and maintainability. Here's a standard layout for a simple project:

```
<project_root>/
 CMakeLists.txt
 src/
    main.cpp
 build/
```

Minimal `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.15)
project(HelloWorld)

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

This configuration sets the minimum required CMake version, defines the project name, and specifies the executable target along with its source file.

3.2 Commands: `cmake_minimum_required`, `project`, `add_executable`

- **`cmake_minimum_required(VERSION <version>)`**: Specifies the minimum CMake version required to process the `CMakeLists.txt` file. This ensures compatibility and enables newer features.
- **`project(<name> [languages...])`**: Defines the project name and optionally the programming languages used. For example, `project(HelloWorld CXX)` sets the project name to `HelloWorld` and indicates that C++ is used.
- **`add_executable(<name> [sources...])`**: Creates an executable target from the specified source files. For instance, `add_executable(hello src/main.cpp)` defines an executable named `hello` built from `src/main.cpp`.

3.3 Configuring and Building

To configure and build the project, follow these steps:

1. **Create a Build Directory:**

```
mkdir build
cd build
```

2. **Configure the Project:**

```
cmake ..
```

   This command generates the necessary build files in the `build` directory.

3. **Build the Project:**

```
cmake --build .
```

This compiles the source code and creates the executable.

By following these steps, you can set up and build a basic CMake project.

3.4 Example of Configuration and Build Steps

Given the following project structure:

```
<project_root>/
 CMakeLists.txt
 src/
     main.cpp
```

Contents of `main.cpp`:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Steps to Build:

1. **Navigate to the Project Directory:**

   ```
   cd <project_root>
   ```

2. **Create a Build Directory:**

   ```
   mkdir build
   cd build
   ```

3. **Configure the Project:**

```
cmake ..
```

4. **Build the Project:**

```
cmake --build .
```

5. **Run the Executable:**

```
./hello
```

   This should output: `Hello, World!`

By following these steps, you can set up and build a basic CMake project.

Chapter 4

Anatomy of `CMakeLists.txt`

4.1 Structure and Modularization

Modern CMake encourages a modular approach to organizing projects. Instead of placing all configurations in a single `CMakeLists.txt` file, it's advisable to break down the project into multiple directories, each with its own `CMakeLists.txt`. This modularization enhances maintainability and scalability, especially for large projects.

Example Project Structure:

```
<project_root>/
 CMakeLists.txt
 src/
    main.cpp
 include/
     mylib.h
```

Top-Level `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.15)
```

```
project(MyProject)

add_subdirectory(src)
```

src/CMakeLists.txt:

```
add_executable(myapp main.cpp)
target_include_directories(myapp PRIVATE ../include)
```

This structure promotes clarity and separation of concerns.

4.2 Targets and Properties

In CMake, a target represents a build artifact, such as an executable or a library. Targets are created using commands like `add_executable()` or `add_library()`. Each target can have associated properties that define its behavior and characteristics.

Common Properties:

- **`INTERFACE_INCLUDE_DIRECTORIES`**: Specifies include directories for consumers of the target.
- **`PUBLIC`**: The property is used by both the target and its consumers.
- **`PRIVATE`**: The property is used only by the target itself.

Example:

```
add_library(mylib STATIC mylib.cpp)
target_include_directories(mylib PUBLIC ../include)
```

This configuration sets up a static library `mylib` and specifies that its consumers should include headers from the `../include` directory.

4.3 Common Patterns and Modern Best Practices

Modern CMake emphasizes clarity, maintainability, and efficiency. Adhering to best practices ensures that projects are robust and adaptable.

Best Practices:

- **Use Target-Based Commands:** Prefer commands like `target_include_directories()`, `target_compile_options()`, and `target_link_libraries()` over global variables to set properties. This approach encapsulates settings within targets, reducing side effects.
- **Avoid Global Variables:** Minimize the use of global variables. Instead, pass information through target properties or use generator expressions.
- **Use `target_compile_features()` for Language Standards:** Specify language standards using `target_compile_features()` rather than setting compiler flags directly. This ensures portability across different compilers.

Example:

```
add_executable(myapp main.cpp)
target_compile_features(myapp PUBLIC cxx_std_17)
```

This sets the C++ standard to C++17 for the `myapp` target.

4.4 CMake Variables: `CACHE`, `ENV`, and Local Scope

CMake provides various types of variables to control configuration and behavior. Understanding their scope and usage is crucial for effective project setup.

- **Local Variables:** Defined within a function or directory, these variables are not visible outside their scope.
- **Cache Variables (`CACHE`):** Stored in the `CMakeCache.txt` file, cache variables persist across CMake runs. They can be set using the `set()` command with the `CACHE` option.

  Example:

  ```
  set(MY_OPTION ON CACHE BOOL "Enable feature X")
  ```

  This defines a cache variable `MY_OPTION` with a default value of `ON`.
- **Environment Variables (`ENV`):** Accessed using the `$ENV{}` syntax, these variables reflect the environment in which CMake is run.

  Example:

  ```
  message(STATUS "PATH: $ENV{PATH}")
  ```

  This prints the system's `PATH` environment variable.

4.5 Use of `message()` for Debugging

Debugging is an essential part of the development process. CMake provides the `message()` function to output information during the configuration process.

Usage:

```
message(STATUS "MY_VARIABLE=${MY_VARIABLE}")
```

This command prints the value of `MY_VARIABLE` during the configuration process.

Advanced Debugging:

For more detailed debugging, CMake offers additional tools:

- **`cmake_print_variables()`**: This function prints the values of specified variables.

  Example:

  ```
  cmake_print_variables(MY_VARIABLE)
  ```

- **`cmake_print_properties()`**: This function prints the properties of specified targets.

  Example:

  ```
  cmake_print_properties(TARGETS my_target PROPERTIES
  ↪  POSITION_INDEPENDENT_CODE)
  ```

These functions provide a more structured way to inspect variables and properties during the configuration process.
By adhering to these principles and practices, you can create well-structured and maintainable CMake configurations that scale effectively with your project's complexity.

Chapter 5

Essential Commands Reference

5.1 `add_library()` (STATIC / SHARED / INTERFACE)

`add_library()` is used to define libraries in CMake, which can be of three main types:

- **STATIC:** Compiled into the final executable; the code is included directly.

```
add_library(mylib STATIC mylib.cpp)
```

- **SHARED:** Dynamic/shared library loaded at runtime; allows sharing between multiple executables.

```
add_library(mylib SHARED mylib.cpp)
```

- **INTERFACE:** Library that does not generate code but defines usage requirements for other targets.

```
add_library(mylib INTERFACE)
target_include_directories(mylib INTERFACE include/)
```

Modern CMake emphasizes target-based usage, so even INTERFACE libraries can propagate include directories, compile options, and linked libraries to dependent targets.

5.2 `target_link_libraries()` with PUBLIC / PRIVATE / INTERFACE

`target_link_libraries()` connects targets to libraries, controlling the propagation of usage requirements:

- **PRIVATE:** Dependencies are used only by the target itself.

```
target_link_libraries(myapp PRIVATE mylib)
```

- **PUBLIC:** Dependencies are used by the target and also propagate to consumers of the target.

```
target_link_libraries(myapp PUBLIC mylib)
```

- **INTERFACE:** Dependencies apply only to consumers of the target, not the target itself.

```
target_link_libraries(myapp INTERFACE mylib)
```

Using these keywords ensures proper encapsulation and avoids unnecessary propagation of build settings.

5.3 `target_include_directories()`

Specifies include directories for a target, with the same PUBLIC / PRIVATE / INTERFACE semantics:

```
target_include_directories(myapp
    PRIVATE src/
    PUBLIC include/
)
```

- **PRIVATE:** Used only by the target.
- **PUBLIC:** Used by the target and propagates to consumers.
- **INTERFACE:** Only for consumers of the target.

This command is central to modern CMake, replacing global include directories for modular projects.

5.4 `target_compile_definitions()`

Defines preprocessor macros for a target:

```
target_compile_definitions(myapp
    PRIVATE MYAPP_VERSION=1
    PUBLIC ENABLE_FEATURE_X
)
```

- **PRIVATE:** Macros used only during compilation of the target.
- **PUBLIC:** Propagated to consumers of the target.
- **INTERFACE:** Used only by consumers of the target.

5.5 `target_compile_options()`

Sets compiler-specific flags for a target:

```
target_compile_options(myapp
    PRIVATE -Wall -Wextra
    PUBLIC $<$<CXX_COMPILER_ID:MSVC>:/W4>
)
```

- Supports generator expressions for platform-specific or compiler-specific flags.
- Ensures modular and portable builds by avoiding global flags.

5.6 `set()`, `option()`, `list()`, `file()`

- **`set()`**: Assigns values to variables.

  ```
  set(MY_VAR "Hello")
  ```

  Can be **local** or **cache**:

  ```
  set(MY_OPTION ON CACHE BOOL "Enable feature")
  ```

- **`option()`**: Creates a boolean cache variable for user selection.

  ```
  option(USE_CUSTOM_ALLOCATOR "Enable custom memory allocator" OFF)
  ```

- **`list()`**: Manipulates lists (append, remove, insert, etc.).

  ```
  list(APPEND SOURCES main.cpp utils.cpp)
  ```

- **`file()`**: Handles file system operations (copy, glob, read/write).

  ```
  file(GLOB SOURCES "src/*.cpp")
  ```

5.7 `add_subdirectory()` and Project Organization

`add_subdirectory()` includes another directory containing a `CMakeLists.txt` file into the build:

```
add_subdirectory(lib/mylib)
add_subdirectory(src)
```

- Promotes **modular project structure**.
- Ensures **target isolation**, with properties and variables scoped appropriately.
- Can optionally pass **binary directory** and **CMake options**.

Example Structure:

```
<project_root>/
 CMakeLists.txt
 src/
    CMakeLists.txt
 lib/
     mylib/
         CMakeLists.txt
```

Top-level CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.15)
project(MyProject)

add_subdirectory(lib/mylib)
add_subdirectory(src)
```

`src/CMakeLists.txt`:

```
add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE mylib)
```

`lib/mylib/CMakeLists.txt`:

```
add_library(mylib STATIC mylib.cpp)
target_include_directories(mylib PUBLIC include/)
```

This pattern allows clean separation of modules and simplifies dependency management in large projects.

By mastering these essential commands, developers can write robust, modular, and maintainable CMake configurations, aligned with modern best practices and scalable project architectures.

Chapter 6

Building and Installing with CMake

6.1 Configure → Generate → Build Phases

CMake operates in three distinct phases:

- **Configure:** The `cmake` command is run in an empty build directory, pointing to the source directory. This step generates the necessary build system files (e.g., Makefiles, Visual Studio project files) based on the project's `CMakeLists.txt` configurations.
- **Generate:** CMake processes the configuration files and generates the appropriate build system files for the chosen generator. This step ensures that all dependencies and configurations are correctly set up.
- **Build:** The `cmake --build` command compiles the project using the generated build system files. This step produces the final executable or library.

This separation allows for a clean and organized build process, ensuring that configuration and generation are distinct from the actual compilation.

6.2 Build Types (`CMAKE_BUILD_TYPE`, Debug, Release, RelWithDebInfo)

The `CMAKE_BUILD_TYPE` variable specifies the build configuration for single-configuration generators like Makefiles or Ninja. Common values include:

- **Debug:** Enables debugging information and disables optimizations.
- **Release:** Enables optimizations and disables debugging information.
- **RelWithDebInfo:** Enables optimizations and includes debugging information.
- **MinSizeRel:** Enables optimizations and reduces binary size.

For multi-configuration generators like Visual Studio, the build configuration is selected during the build step using the `--config` option.

6.3 `cmake --build` and `cmake --install`

- **`cmake --build <dir>`:** Initiates the build process in the specified directory `<dir>`. It uses the generated build system files to compile the project.

  Example:

```
cmake --build build --config Release
```

- **`cmake --install <dir>`:** Installs the built project to the directory specified by `CMAKE_INSTALL_PREFIX`. This step copies the necessary files (executables, libraries, headers) to the appropriate locations.

  Example:

```
cmake --install build --prefix /path/to/install
```

These commands streamline the build and installation process, ensuring consistency and ease of use.

6.4 Installation Directories and Custom Install Targets

The `install()` command in CMake defines how and where files should be installed. Common usage includes:

- **Install Executables:**

```
install(TARGETS myapp DESTINATION bin)
```

- **Install Libraries:**

```
install(TARGETS mylib DESTINATION lib)
```

- **Install Headers:**

```
install(DIRECTORY include/ DESTINATION include)
```

- **Install Files:**

```
install(FILES myconfig.h DESTINATION include)
```

These commands ensure that the necessary files are placed in the correct directories during installation.

6.5 `CMAKE_INSTALL_PREFIX` and Packaging Introduction

The `CMAKE_INSTALL_PREFIX` variable specifies the root directory for installation. By default, it is set to `/usr/local` on Unix-like systems and `C:/Program Files` on Windows.
To customize the installation location, set the `CMAKE_INSTALL_PREFIX` variable:

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install ..
```

This allows for flexible installation paths, which is particularly useful for packaging and distributing software.
For packaging, CMake provides the `CPack` module, which can generate installer packages for various platforms. This module supports formats like ZIP, TGZ, RPM, and DEB, facilitating the distribution of software.
By understanding and utilizing these commands and variables, developers can effectively manage the build and installation process in CMake, leading to more organized and maintainable projects.

Chapter 7

Libraries and Dependencies

7.1 Static vs Shared Libraries

- **Static Libraries (`STATIC`)**
  - Compiled directly into the final executable.
  - No runtime dependency on external library files.
  - Advantages: simpler deployment, faster runtime performance.
  - Disadvantages: larger binary size, changes require recompilation.

```
add_library(mylib STATIC mylib.cpp)
```

- **Shared Libraries (`SHARED`)**
  - Compiled into separate dynamic libraries (.dll/.so/.dylib).
  - Executables link at runtime.
  - Advantages: smaller binaries, shared code across multiple executables, easier updates.

- Disadvantages: runtime dependency management, possible version conflicts.

```
add_library(mylib SHARED mylib.cpp)
```

7.2 Linking Internal and External Targets

- **Internal Targets:** Defined within the same project using `add_library()` or `add_executable()`.

```
target_link_libraries(myapp PRIVATE mylib)
```

- **External Targets:** Libraries outside the project, often found using `find_package()` or `find_library()`.

```
find_package(Boost REQUIRED COMPONENTS filesystem)
target_link_libraries(myapp PRIVATE Boost::filesystem)
```

7.3 `PUBLIC`, `PRIVATE`, `INTERFACE` Explained Clearly

- **PRIVATE:** Used only for the current target; does not propagate to dependents.
- **PUBLIC:** Applies to the target itself and propagates to consumers.
- **INTERFACE:** Used only by consumers, not the target itself.

```
target_link_libraries(myapp
    PRIVATE internal_lib
    PUBLIC mylib
    INTERFACE external_lib
)
```

- Helps manage dependencies cleanly, avoiding unnecessary propagation of include directories, compile options, or linked libraries.

7.4 Handling Transitive Dependencies

Modern CMake automatically propagates usage requirements via `PUBLIC` and `INTERFACE` keywords.

- A target linking a `PUBLIC` library will inherit all its `INTERFACE` properties.
- Reduces manual management of nested dependencies and ensures modular builds.

Example:

```
add_library(A INTERFACE)
target_include_directories(A INTERFACE include/)

add_library(B STATIC b.cpp)
target_link_libraries(B PUBLIC A)

add_executable(app main.cpp)
target_link_libraries(app PRIVATE B)  # app automatically sees A's include
↪  dirs
```

7.5 Using `find_package()` (CONFIG vs MODULE modes)

- **CONFIG Mode:** Uses pre-installed CMake configuration files (`<Package>Config.cmake`). Preferred for modern libraries with CMake support.
- **MODULE Mode:** Uses CMake's built-in `Find<Package>.cmake` scripts. Compatible with older libraries.

Example:

```
find_package(Qt6 REQUIRED COMPONENTS Core Widgets CONFIG)
```

- Provides imported targets like `Qt6::Core` for linking.

7.6 Writing Your Own `Config.cmake`

Custom libraries can expose CMake targets by creating `<Library>Config.cmake`.

- Provides imported targets (`add_library(... IMPORTED)`) with all necessary properties.
- Allows users to consume the library with `find_package(MyLib CONFIG REQUIRED)`.

Structure example:

```
# MyLibConfig.cmake
add_library(MyLib::MyLib STATIC IMPORTED)
set_target_properties(MyLib::MyLib PROPERTIES
    INTERFACE_INCLUDE_DIRECTORIES "${CMAKE_CURRENT_LIST_DIR}/include"
)
```

7.7 `pkg-config` and `find_library()`

- **`pkg-config`:** Standard tool to discover libraries and include paths in Unix systems.
- **`find_library()`:** CMake command to locate a library file manually.

Example:

```
find_library(MYLIB NAMES mylib PATHS /usr/local/lib)
```

- Useful for libraries without CMake configuration files.

7.8 FetchContent Module — Modern Dependency Fetching

- Enables downloading and building external dependencies during the CMake configuration phase.
- Integrates the dependency into the build tree, ensuring consistent versions and build settings.

Example:

```
include(FetchContent)
FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG release-1.12.1
)
FetchContent_MakeAvailable(googletest)
target_link_libraries(tests PRIVATE gtest_main)
```

- Eliminates manual cloning, building, or installing of dependencies.

7.9 `ExternalProject_Add` and Differences from FetchContent

- **ExternalProject_Add:** Downloads, configures, and builds a project as an external step; separate from the main build.
- **Differences from FetchContent:**
  - FetchContent integrates the dependency into the main build tree.

  - ExternalProject_Add builds dependencies independently, often in a separate build directory.

- Use `ExternalProject_Add` when you need to isolate complex dependencies or control build order strictly.

Example:

```
include(ExternalProject)
ExternalProject_Add(
    zlib
    URL https://zlib.net/zlib-1.2.13.tar.gz
)
```

7.10 Integration with Package Managers: vcpkg, Conan

- **vcpkg:** Microsoft-supported C++ package manager. CMake integrates automatically using toolchains.

```
cmake
↪  -DCMAKE_TOOLCHAIN_FILE=/path/to/vcpkg/scripts/buildsystems/vcpkg.cmake
↪  ..
```

- **Conan:** Popular C++ package manager supporting dependency versioning. Integration via `cmake` generator or `conan.cmake` script.

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
```

- Both package managers allow fetching and linking external libraries reliably, improving portability and reproducibility.

Mastering these commands and techniques ensures robust library management, correct dependency handling, and seamless integration of third-party packages, which is crucial for large-scale modern CMake projects.

Chapter 8

Multi-File and Large Projects

8.1 Organizing Large C++ Projects

Modern CMake encourages a structured approach for large-scale projects to ensure maintainability, scalability, and clean dependency management. Key organizational principles include:

- **Directory-based modularization:** Divide the project into logical directories (e.g., `core/`, `utils/`, `app/`) each with its own `CMakeLists.txt`.
- **Target encapsulation:** Define libraries and executables as targets with clearly scoped properties (`PRIVATE`, `PUBLIC`, `INTERFACE`) instead of relying on global variables.
- **Consistent naming:** Maintain a consistent naming convention for targets and source files to reduce confusion.

Example Layout:

```
<project_root>/
```

```
CMakeLists.txt
core/
   CMakeLists.txt
utils/
   CMakeLists.txt
app/
    CMakeLists.txt
```

8.2 Subprojects and Modular CMake Design

- **Subprojects**: Each module or library in a large project can be treated as a subproject with its own build rules.
- **Benefits:**
  - Independent compilation of modules.
  - Easier maintenance and testing.
  - Clear separation of responsibilities between different parts of the project.

Top-level `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.15)
project(LargeProject)

add_subdirectory(core)
add_subdirectory(utils)
add_subdirectory(app)
```

8.3 `add_subdirectory()` and `EXCLUDE_FROM_ALL`

- **`add_subdirectory(<dir> [binary_dir] [EXCLUDE_FROM_ALL])`**: Includes another directory with a `CMakeLists.txt` into the build.
  - `EXCLUDE_FROM_ALL` prevents building the subdirectory unless explicitly requested, which is useful for optional components, tests, or tools.

Example:

```
add_subdirectory(tests EXCLUDE_FROM_ALL)
```

- **Use case:** Large projects often include test suites, benchmarks, or experimental modules that should not be built by default.

8.4 Using Targets Instead of Global Variables

- Avoid global include paths, compile flags, and linked libraries.
- Encapsulate all configuration within targets using commands like:
  - `target_include_directories()`
  - `target_link_libraries()`
  - `target_compile_options()`
  - `target_compile_definitions()`

Example:

```
add_library(core STATIC core.cpp)
target_include_directories(core PUBLIC include/)
target_compile_features(core PUBLIC cxx_std_17)
```

- Benefits:
  - Reduces unintended side effects.
  - Makes dependencies and build requirements explicit.
  - Facilitates reuse in other projects or subprojects.

8.5 Reusability Patterns and Modular `Find<Lib>.cmake` Modules

- **Reusable CMake modules:** Create `Find<Library>.cmake` scripts for custom or third-party libraries not providing CMake configuration files.
- **Purpose:** Standardize how libraries are located, include paths are set, and linked targets are provided across multiple projects.
- **Example pattern for `FindMyLib.cmake`:**

```
find_path(MYLIB_INCLUDE_DIR mylib.h PATHS /usr/local/include)
find_library(MYLIB_LIBRARY NAMES mylib PATHS /usr/local/lib)

if(MYLIB_INCLUDE_DIR AND MYLIB_LIBRARY)
    add_library(MyLib::MyLib STATIC IMPORTED)
    set_target_properties(MyLib::MyLib PROPERTIES
        INTERFACE_INCLUDE_DIRECTORIES "${MYLIB_INCLUDE_DIR}"
        IMPORTED_LOCATION "${MYLIB_LIBRARY}"
    )
else()
    message(FATAL_ERROR "MyLib not found")
endif()
```

- **Modular design benefits:**

  - Easy integration into multiple projects.
  - Clear dependency declaration.
  - Simplifies testing and CI/CD setups.

8.6 Summary

By leveraging modular design, target encapsulation, `add_subdirectory()`, and reusable find modules, large C++ projects can achieve:

- Clear dependency management
- Improved maintainability
- Scalable build architecture
- Reusability of libraries and components

These patterns align with modern CMake practices, ensuring robust builds even in complex, multi-file projects.

Chapter 9

Tooling and IDE Integration

9.1 GUI vs CLI Usage

CMake supports both **Graphical User Interface (GUI)** and **Command Line Interface (CLI)** workflows:

- **CLI Usage:**
  - Preferred for automated builds, CI/CD pipelines, and scripting.
  - Example configuration command:

    ```
    cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
    ```

  - Building:

    ```
    cmake --build build --config Release
    ```

- **GUI Usage (CMake GUI):**
  - Provides a visual interface to configure, edit cache variables, and generate project files.

  - Especially useful for developers unfamiliar with CLI or when exploring options interactively.
  - Allows toggling variables, specifying compilers, and setting installation paths.

- **`ccmake` (Curses-based CLI GUI):**
  - Terminal-based interface for Unix/Linux systems.
  - Allows interactive editing of cache variables in a terminal without a full GUI.
  - Launch example:
    ```
    ccmake .
    ```

9.2 Cache Management

CMake uses `CMakeCache.txt` to store configuration variables:

- **Cache file location:** Build directory.
- **Purpose:** Stores compiler paths, build options, and external dependencies persistently across runs.
- **Manipulation:**
  - Edit manually in `CMakeCache.txt` (not recommended).
  - Use CLI:
    ```
    cmake -U <VAR> -B build
    ```
    to remove cache variables.
  - Resetting build:
    ```
    rm -rf build && cmake -S . -B build
    ```
- **Best practice:** Use cache for user-defined options (`option()` or `set(... CACHE ...)`) and let CMake manage generated variables automatically.

9.3 Integration with IDEs

CMake is compatible with major IDEs, providing seamless workflow for editing, building, and debugging:

- **Visual Studio:**
  - Supports CMake projects directly.
  - Recognizes targets, build configurations, and integrates with `CMakePresets.json`.
  - Debugging and IntelliSense work out-of-the-box.
- **CLion:**
  - Automatically detects CMake projects.
  - Uses CMake's generator to configure builds and runs.
  - Supports multiple build configurations and cross-platform targets.
- **Qt Creator:**
  - Deep integration for Qt projects.
  - Supports CMake's generator and build directories.
  - Provides code navigation, debugging, and profiling tools.
- **Xcode:**
  - Generates Xcode projects via the `-G Xcode` generator.
  - Supports multiple build configurations and schemes.

- **VS Code:**
  - Uses CMake Tools extension.
  - Supports configuring, building, and debugging with CMake build directories.
  - Can switch between presets, toolchains, and build configurations directly from the IDE.

9.4 How Presets Simplify Cross-Platform Builds

CMake Presets allow storing configuration and build settings in JSON files:

- **Benefits:**
  - Define common configurations for multiple developers or CI/CD environments.
  - Enable reproducible builds across platforms.
  - Reduce repetitive CLI commands and manual variable management.
- **Preset categories:**
  - **Configure presets:** Specify generator, build directory, cache variables, and toolchain.
  - **Build presets:** Specify configuration, targets, and build arguments.
  - **Test presets:** Define test configurations for `ctest`.

Example CLI usage with presets:

```
cmake --preset linux-release
cmake --build --preset linux-release
```

- Eliminates the need for repeated `-S` and `-B` options and manual cache configuration.

9.5 Modern Preset Files: `CMakePresets.json`

- Centralized JSON file stored in the project root.
- Example structure:

```
{
  "version": 3,
  "configurePresets": [
    {
      "name": "linux-release",
      "description": "Release build for Linux",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/linux-release",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release",
        "MY_OPTION": "ON"
      }
    }
  ],
  "buildPresets": [
    {
      "name": "linux-release",
      "configurePreset": "linux-release",
      "targets": ["all"]
    }
  ]
}
```

- **Key features:**
  - Version control friendly.
  - Easy to share across teams.
  - Supports conditional variables and environment-specific settings.
  - Works seamlessly with IDEs and CLI.

By combining GUI tools, CLI workflows, and modern presets, developers can manage complex CMake projects efficiently, streamline cross-platform builds, and simplify team collaboration.

Chapter 10

Testing with CTest and GoogleTest

10.1 `enable_testing()` and `add_test()` Basics

- **`enable_testing()`**
  - Activates CMake's testing capabilities.
  - Typically called once in the top-level `CMakeLists.txt`.

```
cmake_minimum_required(VERSION 3.15)
project(MyProject)
enable_testing()
```

- **`add_test()`**
  - Defines individual test cases that CTest can run.
  - Associates a test name with an executable or command.

```
add_executable(test_app test_main.cpp)
add_test(NAME MyTest COMMAND test_app)
```

- Tests are automatically registered with CTest and can be invoked from the command line or IDE.

10.2 Running Tests with `ctest`

- **Basic command:**

```
ctest
```

- **Advanced usage:**
  - Run specific tests:

    ```
    ctest -R MyTest
    ```

  - Run tests with verbose output:

    ```
    ctest -V
    ```

  - Run tests in parallel:

    ```
    ctest -j4
    ```

- **Integration:** Works with any CMake generator and is compatible with CI/CD pipelines.

10.3 Integrating GoogleTest

- GoogleTest is the most widely used C++ testing framework. Integration with CMake is straightforward:

Using FetchContent:

```
include(FetchContent)
FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG release-1.12.1
)
FetchContent_MakeAvailable(googletest)

enable_testing()
add_executable(unit_tests test_main.cpp)
target_link_libraries(unit_tests PRIVATE gtest_main)
add_test(NAME AllTests COMMAND unit_tests)
```

- **Benefits:**
  - Automatic target creation (`gtest`, `gtest_main`).
  - Seamless integration with CTest.
  - Supports parameterized tests and fixtures.

10.4 Test Result Reporting (XML, CI/CD Integration)

- CTest can generate XML output for CI/CD tools:

```
ctest --output-on-failure -T Test
ctest --output-on-failure --no-compress-output --test-output-junit
↪  results.xml
```

- **Usage in CI/CD:**
  - Tools like Jenkins, GitHub Actions, GitLab CI can parse XML results.
  - Facilitates automated test result reporting and build status checks.

- **Verbose reporting:**

```
ctest --verbose
```

Displays detailed information for failing tests, including stdout and stderr.

10.5 Parallel Testing and Timeouts

- **Parallel Testing:**
  - Run multiple tests concurrently to reduce build time.

```
ctest -jN  # N is number of parallel jobs
```

- **Timeouts:**
  - Set time limits per test to avoid hanging builds:

```
set_tests_properties(MyTest PROPERTIES TIMEOUT 30)
```

- **Best practice:** Combine parallel execution with timeouts in CI/CD environments to detect slow or stuck tests efficiently.

10.6 Common Troubleshooting

- **Test Not Running:**
  - Ensure `enable_testing()` is called.
  - Check that the test executable exists and is linked correctly.
- **Segmentation Faults / Crashes:**

  - Run test manually in debugger to identify issues.
  - Confirm correct linking of dependencies.

- **Environment Issues:**

  - Use `set_tests_properties()` to define environment variables.
  - Verify that dynamic libraries are accessible if using shared libraries.

- **Verbose Diagnostics:**

  ```
  ctest -VV
  ```

  Provides detailed logs including command execution and environment details.

By combining `enable_testing()`, `add_test()`, and modern frameworks like GoogleTest, developers can create robust automated tests, integrate them into CI/CD pipelines, and ensure high-quality C++ software. Proper use of parallel tests, timeouts, and structured reporting ensures scalable and maintainable test infrastructure.

Chapter 11

Packaging with CPack

11.1 `include(CPack)` Basics

- **CPack** is CMake's built-in packaging system for creating distributable packages of your project.
- **Basic usage:** Include CPack at the end of your top-level `CMakeLists.txt`:

```
include(CPack)
```

- Automatically reads standard project variables such as `PROJECT_NAME`, `PROJECT_VERSION`, and installation directories (`CMAKE_INSTALL_PREFIX`).
- Generates default packages without extensive configuration, suitable for simple projects.

11.2 Packaging Formats

CPack supports multiple package formats for different platforms:

- **Debian Packages (`.deb`)** – For Debian/Ubuntu systems.
- **RPM Packages (`.rpm`)** – For RedHat, Fedora, and CentOS.
- **Windows Installer (`.msi`)** – For Windows platforms.
- **Compressed Archives (`.tar.gz`, `.zip`)** – Cross-platform packaging without platform-specific installation.
- Example: Configure CPack to create a DEB package:

```
set(CPACK_GENERATOR "DEB")
set(CPACK_DEBIAN_PACKAGE_MAINTAINER "Your Name")
include(CPack)
```

11.3 Configuring `CPackConfig.cmake`

- **`CPackConfig.cmake`** allows detailed control over packaging behavior, metadata, and contents.
- Typical configurable properties:
  - Package name, version, vendor, license.
  - Installation files and directories.
  - Dependencies and system requirements.

Example configuration:

```
set(CPACK_PACKAGE_NAME "MyProject")
set(CPACK_PACKAGE_VERSION "1.2.3")
set(CPACK_PACKAGE_CONTACT "dev@example.com")
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "MyProject C++ Application")
```

```
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_SOURCE_DIR}/LICENSE.txt")
set(CPACK_INSTALL_CMAKE_PROJECTS "${CMAKE_BINARY_DIR};MyProject;ALL;/")
include(CPack)
```

- Supports multiple generators simultaneously:

```
set(CPACK_GENERATOR "DEB;RPM;ZIP")
```

11.4 Multi-OS Packaging and Distribution

- CPack allows generating packages for different operating systems from a single project configuration.
- Key considerations:
  - Platform-specific variables and installation paths.
  - Conditional packaging rules for Windows vs Linux.
  - Using platform-specific CPack options:

```
if(WIN32)
    set(CPACK_GENERATOR "WIX") # MSI installer
elseif(UNIX)
    set(CPACK_GENERATOR "DEB;RPM;TGZ")
endif()
```

- Cross-platform builds can leverage Docker, virtual machines, or CI/CD pipelines to produce consistent packages for multiple OS targets.

11.5 Tips for Versioning and Platform Support

- **Versioning:**

  - Maintain semantic versioning (`MAJOR.MINOR.PATCH`) via `PROJECT_VERSION` or `CPACK_PACKAGE_VERSION`.
  - Automatically increment package revisions in CI/CD builds using environment variables or scripts.

- **Platform Support:**

  - Test packaging on target OS platforms to validate installation paths, dependencies, and post-install scripts.
  - Include runtime dependencies in CPack configuration or rely on system package managers for installation.

- **Best Practices:**

  - Keep CPack configuration in the top-level `CMakeLists.txt` or a dedicated `CPackConfig.cmake` for clarity.
  - Combine with `CMakePresets.json` to automate builds and packaging across multiple environments.
  - Use `cpack --config CPackConfig.cmake` for custom builds and debugging package creation.

By leveraging CPack, developers can streamline the distribution of C++ projects across multiple platforms, ensuring consistent installation, proper versioning, and compatibility with system package managers. Proper configuration reduces manual packaging effort and integrates seamlessly with automated CI/CD pipelines.

Chapter 12

CI/CD and Cloud Integration

12.1 Using CMake in CI/CD Systems

Modern CMake integrates seamlessly with continuous integration and deployment (CI/CD) platforms such as:

- **GitHub Actions** – Native support for YAML workflows.
- **GitLab CI** – Supports `.gitlab-ci.yml` pipelines with multiple stages.
- **Jenkins** – Offers pipelines and scripted builds with CMake commands.

CMake's CLI commands (`cmake -S . -B build`, `cmake --build`, `ctest`, `cpack`) are fully compatible with automated pipelines, enabling consistent cross-platform builds and testing.

12.2 Example Workflow YAML for Cross-Platform Builds

GitHub Actions Example:

```
name: C++ CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        build_type: [Debug, Release]

    steps:
    - uses: actions/checkout@v3
    - name: Configure CMake
      run: cmake -S . -B build -DCMAKE_BUILD_TYPE=${{ matrix.build_type }}
    - name: Build
      run: cmake --build build --config ${{ matrix.build_type }} --
      ↪  -j$(nproc)
    - name: Run Tests
      run: ctest --output-on-failure --test-dir build
    - name: Package
      run: cpack --config build/CPackConfig.cmake
```

- **Matrix builds** allow testing multiple operating systems and configurations in parallel.
- The pipeline can be extended with custom steps for deployment or artifact management.

12.3 Automating Build, Test, and Packaging Pipelines

- **Build:** Automate compilation using `cmake --build` with appropriate configurations.
- **Test:** Use `ctest` for unit tests, GoogleTest integration, and parallel execution.
- **Package:** Integrate `cpack` to produce distributable artifacts automatically.
- **Best Practices:**
  - Separate build, test, and package stages for clarity.
  - Fail fast on compilation or test errors to save CI resources.
  - Leverage environment variables and presets for cross-platform consistency.

12.4 Artifacts and Caching Dependencies

- **Artifacts:** Store build outputs, test results, or packages in CI/CD storage for later retrieval.
  - Example in GitHub Actions:

```
- name: Upload Artifacts
  uses: actions/upload-artifact@v3
  with:
    name: MyProject-Build
    path: build/
```

- **Caching Dependencies:**
  - Reduce build time by caching external libraries, CMake downloads, or compiled object files.
  - GitHub Actions example:

```
- name: Cache CMake dependencies
  uses: actions/cache@v3
  with:
    path: build/_deps
    key: ${{ runner.os }}-cmake-deps-${{
     ↪  hashFiles('CMakeLists.txt') }}
```

- **Benefit:** Faster rebuilds and reduced network usage for external dependencies.

12.5 Cloud-Based Build Optimizations

- **Parallelization:** Execute matrix builds across multiple virtual machines or containers.
- **Remote Caching:** Use cloud storage to cache build artifacts between pipelines, minimizing compilation.
- **Prebuilt Dependencies:** Use precompiled packages via vcpkg, Conan, or FetchContent to reduce compilation time.
- **CI/CD Integration:** CMake presets simplify the setup of reproducible cross-platform builds in cloud environments.

Additional Tips:

- Use Docker containers for consistent build environments across developers and CI/CD runners.
- Enable verbose logging (`cmake --trace-expand`) for troubleshooting complex pipelines.
- Monitor cache hits and artifact sizes to optimize storage and build times.

By incorporating CMake into CI/CD workflows, teams can automate the entire lifecycle: configuration, building, testing, packaging, and distribution. Cloud-based optimization, artifact management, and caching ensure fast, reproducible, and scalable builds across multiple platforms.

Chapter 13

Performance Optimization

13.1 Build Caching with `ccache` and `distcc`

- **`ccache` (Compiler Cache):**
  - Speeds up recompilation by caching previous compilation results.
  - Works seamlessly with CMake by setting the C and C++ compilers to use `ccache`.

```
set(CMAKE_C_COMPILER_LAUNCHER ccache)
set(CMAKE_CXX_COMPILER_LAUNCHER ccache)
```

- **`distcc` (Distributed Compilation):**
  - Distributes compilation tasks across multiple machines.
  - Reduces build time for large projects on networks with multiple available cores.
- **Best practice:** Combine `ccache` and `distcc` for maximum efficiency: local cache avoids unnecessary compilation, and distributed compilation accelerates the remaining tasks.

13.2 Unity Builds

- **Definition:** Combine multiple source files into a single compilation unit.
- **Advantages:**
  - Reduces overhead from parsing headers repeatedly.
  - Often improves compile time in large projects.
- **Disadvantages:**
  - Can hide symbol conflicts or cause unexpected dependencies.
  - Harder incremental builds for frequently changing files.
- **CMake integration:**

```
set_target_properties(myapp PROPERTIES UNITY_BUILD ON)
```

- **Best use case:** Large projects with many small, stable source files and complex header dependencies.

13.3 Link Time Optimization (LTO)

- **Purpose:** Optimize the entire program at link time, allowing the compiler to inline functions across translation units and perform global optimization.
- **CMake configuration:**

```
set(CMAKE_INTERPROCEDURAL_OPTIMIZATION_RELEASE ON)
```

- Works with modern compilers (GCC, Clang, MSVC).

- **Considerations:**
  - Increases link time.
  - Best applied for release builds where runtime performance is critical.

13.4 Optimizing Compiler Flags

- **Release-specific flags:**
  - Set through `CMAKE_CXX_FLAGS_RELEASE` or `target_compile_options()` per target.
  - Examples:

```
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -DNDEBUG -march=native")
target_compile_options(myapp PRIVATE $<$<CONFIG:Release>:-O3
↪  -march=native>)
```

- **Key tips:**
  - Use `-O3` for aggressive optimization.
  - Enable architecture-specific tuning (`-march=native`).
  - Strip debug symbols in release builds unless needed for profiling.

13.5 Using Ninja for Fast Incremental Builds

- **Ninja:** A small, fast build system designed for incremental compilation.
- **Integration with CMake:**

```
cmake -G Ninja -S . -B build
cmake --build build
```

- **Benefits:**
  - Much faster dependency checking compared to Make.
  - Supports multi-threaded builds automatically (`ninja -j N`).
  - Works efficiently with CCache and LTO.
- **Best practices:**
  - Use Ninja as the default generator for large C++ projects.
  - Combine with CMake presets to ensure reproducible builds across CI/CD and developer machines.

By leveraging build caching, unity builds, LTO, optimized compiler flags, and fast generators like Ninja, CMake projects can achieve significant reductions in build time while improving runtime performance. Proper configuration ensures scalability and efficiency in both local and CI/CD environments.

Chapter 14

Writing Custom CMake Modules

14.1 Custom `Find<Lib>.cmake` Modules

- **Purpose:** Locate libraries, include directories, and binaries not providing a CMake configuration file.
- **Location:** Store in `cmake/Modules` or a similar directory and add to `CMAKE_MODULE_PATH`:

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
↪  "${CMAKE_SOURCE_DIR}/cmake/Modules/")
find_package(MyLib REQUIRED)
```

- **Structure of a `Find<Lib>.cmake` module:**
  - Locate include directories (`find_path`)
  - Locate library files (`find_library`)
  - Set imported targets (`add_library(... IMPORTED)`)

- Provide optional variables for versioning and required components

```
find_path(MYLIB_INCLUDE_DIR mylib.h PATHS /usr/local/include)
find_library(MYLIB_LIBRARY NAMES mylib PATHS /usr/local/lib)

if(MYLIB_INCLUDE_DIR AND MYLIB_LIBRARY)
    add_library(MyLib::MyLib STATIC IMPORTED)
    set_target_properties(MyLib::MyLib PROPERTIES
        INTERFACE_INCLUDE_DIRECTORIES "${MYLIB_INCLUDE_DIR}"
        IMPORTED_LOCATION "${MYLIB_LIBRARY}"
    )
else()
    message(FATAL_ERROR "MyLib not found")
endif()
```

14.2 Defining Your Own Commands (`function`, `macro`)

- **`function()`** – Defines reusable CMake functions with scoped variables:

```
function(add_my_executable target_name)
    add_executable(${target_name} ${ARGN})
    target_compile_features(${target_name} PRIVATE cxx_std_17)
endfunction()

add_my_executable(app main.cpp)
```

- **`macro()`** – Similar to function but variables are not scoped; affects parent scope.
- **Use cases:**
  - Automate repetitive target definitions
  - Centralize configuration for multiple executables or libraries
  - Simplify cross-platform conditional logic

14.3 Managing Environment Variables

- **Accessing environment variables:**

```
get_environment_variable(PATH_TO_TOOL ENV{TOOL_PATH})
```

- **Setting environment variables for targets or tests:**

```
set_tests_properties(MyTest PROPERTIES ENVIRONMENT "MY_VAR=value")
```

- **Best practices:**
  - Use environment variables for paths, credentials, or tool configurations in a reproducible manner.
  - Avoid hardcoding platform-specific paths; prefer variables or presets.

14.4 Code Reusability and Modularity

- **Goals:** Avoid duplication, simplify maintenance, and ensure consistency.
- **Techniques:**
  - Encapsulate common patterns in functions/macros.
  - Create reusable `Find<Lib>.cmake` scripts for third-party dependencies.
  - Organize custom modules in a `cmake/Modules` directory.

Example modular approach:

```
project_root/
 cmake/
    Modules/
```

```
      FindMyLib.cmake
   Functions/
       AddMyExecutable.cmake
src/
CMakeLists.txt
```

- Load modules in the top-level `CMakeLists.txt`:

```
include(cmake/Functions/AddMyExecutable.cmake)
add_my_executable(app main.cpp)
```

14.5 Sharing Reusable Modules

- **Internal sharing:**
  - Across multiple projects in a company by maintaining a centralized module repository.
  - Use `CMAKE_MODULE_PATH` to make modules available globally.
- **External sharing:**
  - Publish as part of a Git repository or package manager (vcpkg, Conan).
  - Standardize module names and targets to ensure compatibility.
  - Include documentation for variables, required dependencies, and expected behavior.
- **Best practices:**
  - Use clear naming conventions (`Find<Lib>.cmake`, `Add<MyTarget>.cmake`).
  - Avoid hardcoding platform-specific paths; leverage CMake variables and presets.

- Include version checks and error messages for missing dependencies.

By creating custom modules, defining reusable functions/macros, managing environment variables, and structuring code modularly, CMake projects achieve maintainability, scalability, and reusability. Sharing modules across projects or teams accelerates development and ensures consistency in complex C++ builds.

Chapter 15

Cross-Platform and Portability

15.1 Writing Platform-Independent `CMakeLists.txt`

- **Goal:** Ensure a single `CMakeLists.txt` works on multiple operating systems and compilers.
- **Key practices:**
  - Use CMake variables for paths, libraries, and compiler flags instead of hardcoding them.
  - Prefer `target_*` commands (`target_include_directories`, `target_compile_options`, `target_link_libraries`) for setting per-target properties.
  - Avoid global variables for include paths or compiler flags.

Example:

```
add_executable(MyApp main.cpp)
```

```
target_compile_features(MyApp PUBLIC cxx_std_17)
target_include_directories(MyApp PUBLIC ${PROJECT_SOURCE_DIR}/include)
```

15.2 Handling OS-Specific Logic and Toolchains

- **OS detection:** Use built-in CMake variables to conditionally apply settings:

```
if(WIN32)
    target_compile_definitions(MyApp PRIVATE WINDOWS_PLATFORM)
elseif(APPLE)
    target_compile_definitions(MyApp PRIVATE MACOS_PLATFORM)
elseif(UNIX)
    target_compile_definitions(MyApp PRIVATE LINUX_PLATFORM)
endif()
```

- **Toolchain files:** Specify compiler, linker, and platform-specific flags for reproducible builds:

```
cmake -S . -B build -DCMAKE_TOOLCHAIN_FILE=cmake/toolchains/arm.cmake
```

- **Use cases:** Embedded systems, cross-compiling to specialized hardware, or custom compiler environments.

15.3 Cross-Compiling (ARM, Android, iOS)

- **CMake Toolchains:** Central mechanism to define cross-compilation settings including compiler, sysroot, and target architecture.
- **ARM Example:**

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
```

```
set(CMAKE_C_COMPILER /usr/bin/arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER /usr/bin/arm-linux-gnueabihf-g++)
```

- **Android/iOS:**
  - Use CMake Android toolchain file for NDK builds: `-DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolcha`
  - Use `CMAKE_OSX_ARCHITECTURES` for iOS multi-architecture builds (`arm64;x86_64`).
- **Best practices:**
  - Use presets (`CMakePresets.json`) to manage cross-platform configurations.
  - Keep toolchain files reusable and version-controlled.

15.4 Environment Detection and Compiler Options

- **Compiler detection:** CMake automatically identifies compiler features, but explicit checks can be useful:

```
if(CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
    target_compile_options(MyApp PRIVATE -Wall -Wextra)
elseif(CMAKE_CXX_COMPILER_ID STREQUAL "Clang")
    target_compile_options(MyApp PRIVATE -Weverything)
endif()
```

- **Environment variables:** Influence compiler behavior, library paths, and custom tools:

```
if(DEFINED ENV{MY_LIB_PATH})
    target_include_directories(MyApp PRIVATE $ENV{MY_LIB_PATH})
endif()
```

- **Best practices:**
  - Detect platform, compiler, and environment dynamically.
  - Minimize assumptions about filesystem layout or available tools.
  - Combine with presets to maintain reproducibility across development, CI/CD, and deployment environments.

By applying platform-independent design, OS-specific logic handling, toolchain management, and cross-compilation strategies, CMake projects achieve portability across desktops, embedded systems, mobile platforms, and cloud environments, while maintaining maintainable and reproducible builds.

Appendices

Appendix A : Reference: Common Variables and Built-In Modules

- **Common CMake Variables:**
  - `CMAKE_SOURCE_DIR` / `PROJECT_SOURCE_DIR` – Root source directory.
  - `CMAKE_BINARY_DIR` / `PROJECT_BINARY_DIR` – Build directory.
  - `CMAKE_CXX_STANDARD` – Required C++ standard (e.g., 17, 20).
  - `CMAKE_BUILD_TYPE` – Build configuration (Debug, Release, RelWithDebInfo).
  - `CMAKE_INSTALL_PREFIX` – Default installation path.
- **Common Built-in Modules:**
  - `FindThreads.cmake` – Threading support.
  - `FindPackageHandleStandardArgs.cmake` – Simplifies `find_package` checks.
  - `CTest` – Testing framework integration.
  - `CPack` – Packaging support.

  - `FetchContent` – Fetch external dependencies at configure time.

- **Tips:** Use `message(STATUS "Variable: ${VAR}")` for debugging and exploring module variables.

Appendix B : Common Pitfalls and Anti-Patterns

- **Global variables instead of target properties:**
  - Avoid setting `INCLUDE_DIRECTORIES()` or `LINK_LIBRARIES()` globally; prefer `target_include_directories()` and `target_link_libraries()`.
- **Hardcoding paths:**
  - Avoid absolute paths; use variables, environment detection, or toolchains.
- **Mixing old and modern commands:**
  - Do not combine `add_definitions()` or `include_directories()` with target-based commands in the same project; it leads to unmaintainable builds.
- **Ignoring presets and reproducibility:**
  - Hardcoded compiler flags or directories reduce cross-platform portability.

Appendix C : Tips for Migrating from Legacy CMake

- **Move from global to target-based approach:**
  - Convert `add_definitions()` → `target_compile_definitions()`

  - Convert `include_directories()` → `target_include_directories()`

- **Replace old `find_package` logic:**

  - Use modern `Config` packages instead of custom `Find<Lib>.cmake` where possible.

- **Use modern caching and presets:**

  - Replace `set(... CACHE ...)` with well-defined options and `CMakePresets.json` for reproducibility.

- **Incremental modernization:**

  - Modernize gradually; keep legacy behavior working while refactoring targets to `target_*` commands.

Appendix C : Summary of Post-2022 Features

- **Presets (`CMakePresets.json`)**:

  - Standardizes configuration and build options across platforms, developers, and CI/CD.
  - Supports configure, build, and test presets for reproducible workflows.

- **FetchContent Module Enhancements:**

  - Simplifies integrating external dependencies without modifying system paths.

  - Supports `FETCHCONTENT_UPDATES_DISCONNECTED` to lock dependency versions.

- **CI/CD Integration:**

  - Simplifies GitHub Actions, GitLab CI, Jenkins, or cloud builds.
  - Supports artifacts, caching, and cross-platform builds via presets.

- **Unity Builds and LTO:**

  - Accelerates compilation and runtime performance for large projects.
  - Integrates seamlessly with target-based properties and modern CMake configurations.

This appendix consolidates the most important references, best practices, and modern CMake techniques. It provides a quick guide for migrating legacy projects, avoiding common mistakes, and fully leveraging post-2022 features such as presets, FetchContent, CI/CD pipelines, Unity builds, and LTO for efficient and maintainable builds.

References

Official CMake Documentation

- **CMake Reference Manual:** The primary source for commands, variables, targets, and modules.
- **CMake Command Documentation:** Detailed explanations for all commands, including `add_executable`, `add_library`, `target_*` commands, `find_package`, `include()`, `enable_testing()`, and `CPack`.
- **CMake Modules Reference:** Comprehensive list of built-in modules such as `CTest`, `CPack`, `FetchContent`, `Find<Lib>.cmake` templates.
- **CMake Presets:** Guidelines for `CMakePresets.json` including configure, build, and test presets.
- **CMake Toolchain Files:** Reference for cross-compiling, embedded targets, and platform-specific toolchains.

CMake Blog and Release Notes

- **CMake.org Blog:** Provides updates on post-2022 features including:

  - Presets improvements for CI/CD and cross-platform builds.
  - Enhancements to FetchContent and dependency management.
  - Unity builds and Link Time Optimization (LTO) support.

- **Release Notes:** Detailed information for each version including new commands, deprecations, and optimizations.

CTest and Testing Resources

- **CTest Manual:** Covers `enable_testing()`, `add_test()`, test properties, parallel execution, and XML reporting for CI/CD.
- **GoogleTest Integration Guides:** Shows best practices for integrating GoogleTest with CMake and generating CTest-compatible tests.

CPack and Packaging References

- **CPack Documentation:** Covers multi-format packaging (`.deb`, `.rpm`, `.msi`, `.tar.gz`) and multi-platform distribution.
- **CPackConfig.cmake Examples:** Guidelines for versioning, installation directories, and cross-platform packaging.

CI/CD and Cloud Integration Sources

- **GitHub Actions Documentation:** Workflow examples for cross-platform builds using CMake.
- **GitLab CI/CD Documentation:** Best practices for build, test, and packaging pipelines.

- **Jenkins Pipelines with CMake:** Guidance for automated builds and artifact management.
- **Cloud Build Optimization:** Strategies for caching, artifact storage, distributed builds, and containerized environments.

Performance Optimization Resources

- **Compiler Documentation:** GCC, Clang, and MSVC optimization flags, LTO, and multi-threaded build options.
- **Ninja Build System:** Reference for fast incremental builds, parallelization, and integration with CMake.
- **Build Caching Tools:** `ccache` and `distcc` usage in CMake projects for accelerated compilation.

Custom CMake Modules and Legacy Migration

- **Custom `Find<Lib>.cmake` Patterns:** Standard practices from CMake documentation and community modules.
- **Functions and Macros:** Best practices for reusability, modularity, and environment handling.
- **Migration Guides:** Community guidelines for transitioning legacy CMake scripts to target-based and modern CMake patterns.

Cross-Platform and Portability References

- **CMake Toolchain Documentation:** For ARM, Android, iOS, and embedded targets.

- **Platform Detection Variables:** Built-in variables like `WIN32`, `APPLE`, `UNIX` for OS-specific logic.
- **Cross-Compilation Guides:** Presets, environment detection, and compiler option handling for reproducible builds.

Modern Best Practices and Anti-Patterns

- **Target-Based Development:** Prefer `target_include_directories()`, `target_compile_definitions()`, `target_link_libraries()` over global variables.
- **Presets and Reproducibility:** Usage of `CMakePresets.json` for standardized configuration across teams.
- **FetchContent and Dependency Management:** Modern methods for handling external libraries without global path hacks.

Summary

This booklet consolidates knowledge from the official CMake documentation, release notes, community best practices, compiler manuals, and CI/CD platform guides. All modern features—including post-2022 enhancements such as presets, FetchContent, CI/CD integration, unity builds, and LTO—are based on authoritative sources, ensuring accurate, reliable, and up-to-date guidance for CMake users.