https://www.simplifycpp.org

CMake

The Comprehensive Guide to Managing and Building C++ Projects

From Basics to Mastery

Prepared by: Ayman Alheraki

First Edition

CMake:
The Comprehensive Guide to Managing and Building C++ Projects
(From Basics to Mastery)

Prepared by Ayman Alheraki
simplifycpp.org

February 2025

Contents

Author's Introduction

One of the most challenging aspects of C++ programming is the compilation process. Unlike many modern languages that come with built-in package managers and streamlined build systems, C++ requires developers to have a deep understanding of compilation, linking, dependency management, and platform-specific configurations. This complexity often discourages students and even experienced developers, leading them to abandon C++ in favor of languages with simpler build processes. However, while C++ may have a steep learning curve in this regard, mastering the right tools can dramatically improve the development experience and unlock the full potential of the language.

This is where CMake comes in. CMake is not just another build system; it is a powerful meta-build tool that simplifies and standardizes the configuration, compilation, and linking processes across multiple platforms and compilers. In large-scale projects, particularly those targeting multiple operating systems (Windows, macOS, Linux) or architectures (x86, ARM, embedded systems), managing build files manually can be overwhelming. CMake provides an elegant solution by allowing developers to define their build processes in a platform-independent manner, generating appropriate build scripts for a variety of compilers and environments.

Many C++ programmers hesitate to learn CMake, thinking of it as an additional layer of complexity rather than a solution. However, once you understand its workflow, CMake becomes an indispensable tool that simplifies project setup, dependency

management, and integration with external libraries. Whether you are working on small projects or large-scale software systems, using CMake can save you countless hours of manual configuration and troubleshooting.

In this book, I will guide you through the fundamentals of CMake, from basic setup to advanced configurations. You will learn how to efficiently manage source files, handle third-party dependencies, optimize compilation settings, and create robust cross-platform builds. By the end of this journey, you will have a solid grasp of CMake, allowing you to focus more on writing great C++ code rather than struggling with build issues.

I strongly encourage every C++ developer to invest time in mastering CMake. It is not just a tool—it is an essential skill that will make your C++ development process more efficient, scalable, and enjoyable.

Stay Connected

For more discussions and valuable content about Modern C++ Pointers, I invite you to follow me on LinkedIn:

https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Chapter 1

Introduction to CMake

1.1 Why Do You Need CMake?

1.1.1 Introduction

Software development, particularly in languages like C++, involves multiple stages, including writing source code, compiling it into object files, linking those files to create an executable, and finally deploying the application. As projects grow in complexity, managing these tasks efficiently becomes increasingly difficult. While simple programs with a few source files can be compiled manually using compiler commands, larger projects require automated build systems to handle dependencies, multiple files, libraries, and different build configurations.

Historically, developers have relied on manual Makefiles, Autotools, and other platform-specific build scripts to manage the compilation process. However, these traditional methods come with significant limitations, particularly when it comes to cross-platform compatibility, maintainability, and scalability.

CMake is a modern, flexible, and cross-platform build system generator that simplifies

the process of compiling, linking, and managing C++ projects. Instead of manually writing complex and platform-dependent Makefiles or build scripts, developers define their project's structure in a simple and declarative manner using CMakeLists.txt, and CMake generates the appropriate build system for the target platform.
This section explores the need for CMake by identifying the challenges associated with traditional build systems and demonstrating how CMake provides a powerful solution.

1.1.2 The Challenges of Traditional Build Systems

1. Platform-Specific Build Scripts

   One of the biggest challenges in software development is ensuring that a project can be compiled and executed on multiple operating systems. Many projects need to support Windows, Linux, macOS, and even embedded platforms.

   When using traditional build methods, developers often have to write separate build scripts for each platform:

   - Windows: Batch scripts (.bat), PowerShell scripts (.ps1), or Visual Studio project files
   - Linux/macOS: GNU Makefiles (Makefile), shell scripts (.sh), or Autotools configurations

   Each of these build scripts is tailored to the specific operating system and compiler used. A Makefile that works on Linux with GCC might not work on Windows without modifications, and a Visual Studio project file cannot be easily ported to Linux. This fragmentation leads to increased maintenance overhead and makes cross-platform development cumbersome.

2. Complex Dependency Management

Modern C++ projects often rely on multiple external libraries, such as Boost, OpenCV, Qt, Eigen, GLFW, and SQLite. Managing these dependencies manually presents several challenges:

- Locating the library: Developers must specify the correct paths for headers and compiled binaries.
- Handling different versions: Different systems may have different versions of a library installed, leading to potential compatibility issues.
- Static vs. dynamic linking: Some projects require static linking, while others need shared libraries, leading to different linking options.
- Managing transitive dependencies: A library may depend on other libraries, complicating the linking process.

Traditional methods require developers to write complex shell scripts or pkg-config configurations to handle these dependencies. Errors in locating or linking a dependency can lead to frustrating build failures.

3. Lack of Maintainability and Scalability

   As projects grow, managing a build system manually becomes increasingly difficult. Adding new source files to a project means modifying existing Makefiles or build scripts, which increases the risk of introducing errors.

   For example, a manually written Makefile may require:

```
OBJS = main.o module1.o module2.o
CC = g++
CFLAGS = -Wall -O2

app: $(OBJS)
^^I$(CC) $(CFLAGS) -o app $(OBJS)
```

Every time a new source file is added, it must be explicitly listed in OBJS, making maintenance error-prone. Large projects with hundreds of source files require a more dynamic and automated approach to handling build configurations.

4. Difficulty in Multi-Platform Builds

   A project developed on Linux using Makefiles and GCC may not compile on Windows without modification. Differences in compilers, library locations, and system APIs require additional effort to ensure cross-platform compatibility.

   Maintaining multiple separate build configurations for different operating systems increases complexity and maintenance overhead. This issue is particularly relevant for open-source projects where contributors may use different operating systems, requiring a flexible and portable build system.

1.1.3 How CMake Solves These Challenges

CMake provides a high-level abstraction for defining build configurations, allowing developers to describe what should be built rather than specifying how to build it. CMake then generates the appropriate build system for the target platform, handling platform-specific details automatically.

1. Cross-Platform Compatibility

   CMake is designed to be platform-independent. A single CMakeLists.txt file can be used to generate build configurations for multiple operating systems. CMake supports various build generators, including:

   - Makefiles (for Linux and macOS)
   - Ninja (for fast parallel builds)
   - Visual Studio project files (for Windows development)

- Xcode project files (for macOS development)
- MSBuild, NMake, and others

This allows developers to write once and build anywhere, eliminating the need for maintaining multiple build scripts for different platforms.

2. Simplified Dependency Management

   CMake provides built-in functionality for locating and integrating third-party libraries. Instead of manually specifying library paths, developers can use:

   - find_package() – Automatically locates installed libraries such as OpenGL, Boost, and Qt.
   - FetchContent – Fetches external libraries and integrates them at build time.
   - ExternalProject_Add – Fetches and compiles external projects.

   This streamlines dependency management and reduces the risk of version mismatches or missing dependencies.

3. Automatic System Detection

   CMake detects the compiler, available system libraries, and hardware capabilities automatically. This allows developers to write portable build configurations without worrying about platform-specific details.

   For example, CMake can check for the presence of certain libraries and enable features accordingly:

```
find_package(OpenGL REQUIRED)
find_package(Boost 1.71 REQUIRED)
```

If the required libraries are not found, CMake can provide meaningful error messages, guiding users to install the necessary dependencies.

4. Scalability for Large Projects

   CMake supports modular project structures, allowing large projects to be divided into multiple subdirectories, each with its own CMakeLists.txt. This improves maintainability and organization.

   CMake also supports out-of-source builds, preventing source directories from being cluttered with build artifacts.

1.1.4 Conclusion

CMake addresses the limitations of traditional build systems by providing a cross-platform, maintainable, and scalable approach to building C++ projects. It simplifies dependency management, multi-platform support, and automatic configuration detection, making it the preferred choice for modern C++ development. With CMake, developers can focus on writing code instead of dealing with the intricacies of manually managing builds, dependencies, and platform-specific configurations.

1.2 CMake vs. Traditional Build Systems (Make, Autotools, Ninja, etc.)

1.2.1 Introduction

The process of building software from source code involves compiling, linking, and organizing dependencies to create an executable or library. As software projects grow in complexity, managing the build process manually becomes inefficient and error-prone. Traditionally, developers relied on build systems such as Make, Autotools, and Ninja to automate the compilation process. However, these systems come with limitations in cross-platform support, maintainability, and flexibility.

CMake was developed to overcome these limitations by providing a higher-level build system generator that abstracts platform-specific complexities. Unlike traditional build systems that require developers to write platform-specific scripts, CMake allows them to define their projects once and generate the appropriate build files for multiple platforms and compilers.

This section provides an in-depth comparison between CMake and traditional build systems, highlighting their strengths, weaknesses, and use cases.

1.2.2 Traditional Build Systems: Overview and Limitations

Before comparing CMake with other build systems, it is important to understand how traditional build systems work and where they fall short.

1. Make (GNU Make)

   Make is one of the earliest and most widely used build systems. It is primarily used in Unix-like operating systems and relies on Makefiles to define build rules and dependencies.

How Make Works

Make processes a Makefile, which specifies how to compile and link a program. The Makefile contains:

- Targets: The files to be built (e.g., object files, executables).
- Dependencies: The source files required to build a target.
- Commands: The compilation and linking commands to execute.

Example Makefile for a Simple C++ Project

```
CC = g++
CFLAGS = -Wall -O2
OBJ = main.o module.o

app: $(OBJ)
^^I$(CC) $(CFLAGS) -o app $(OBJ)

%.o: %.cpp
^^I$(CC) $(CFLAGS) -c $< -o $@
```

To build the project, the user runs:

```
make
```

This compiles the source files and links them into an executable.

Advantages of Make

- Simple and widely available: Make is installed by default on most Unix-like systems.

- Parallel execution: Supports multi-threaded builds using make -j.
- Customizability: Allows developers to define their own build rules.

Disadvantages of Make

- Platform-dependent: Make is Unix-centric and requires modifications to work on Windows.
- Manual dependency tracking: Developers must explicitly list source files and dependencies.
- Difficult to maintain: Large projects require complex Makefiles, making maintenance difficult.

2. Autotools (GNU Autoconf, Automake, Libtool)

   Autotools is a suite of tools designed to enhance portability across Unix-like systems by automatically generating Makefiles based on system configuration.

   How Autotools Works

   Autotools follows a three-step process:

   1. Autoconf (configure.ac) – Creates a configure script to detect system-specific settings.
   2. Automake (Makefile.am) – Generates platform-specific Makefiles.
   3. Libtool – Handles shared and static library creation across different platforms.

   To build a project using Autotools, the user runs:

```
./configure
make
make install
```

Advantages of Autotools

- Better portability: Generates system-specific Makefiles.
- Automatic feature detection: Checks compiler settings, libraries, and dependencies.
- Standardized build process: Commonly used in open-source projects.

Disadvantages of Autotools

- Complex configuration: Requires multiple scripts and configuration files.
- Slow build process: Running ./configure can be time-consuming.
- Difficult Windows support: Primarily designed for Unix; requires additional tools like MinGW or Cygwin for Windows compatibility.

3. Ninja

   Ninja is a build system optimized for speed and efficiency. Unlike Make and Autotools, which handle dependency resolution and build configuration, Ninja is designed purely for executing build tasks as quickly as possible.

   How Ninja Works

   Ninja relies on a build.ninja file, which describes how source files should be compiled and linked. However, developers do not write these files manually—they are typically generated by higher-level tools like CMake or Meson.

Example Ninja Build File

```
rule compile
  command = g++ -c $in -o $out
build main.o: compile main.cpp
```

Advantages of Ninja

- Extremely fast: Optimized for incremental builds.
- Parallel execution: Utilizes multiple CPU cores efficiently.
- Widely used in large projects: Used by Chromium and LLVM.

Disadvantages of Ninja

- Not standalone: Requires an external build generator like CMake.
- Less human-readable: Ninja build files are machine-generated and difficult to modify manually.

1.2.3 How CMake Compares to Traditional Build Systems

CMake is fundamentally different from traditional build systems because it is a build system generator rather than a build system itself. Instead of managing the build process directly, CMake generates platform-specific build files for Make, Ninja, Visual Studio, and others.

1. Key Advantages of CMake

Feature	Make	Autotools	Ninja	CMake
Cross-platform support	Limited	Unix-focused	Limited	Full support (Windows, Linux, macOS)
Dependency management	Manual	Checks for libraries	None	Built-in (find_package(), FetchContent)
Multi-platform build generation	No	No	No	Supports Make, Ninja, Visual Studio, Xcode
Automatic system detection	No	Yes	No	Detects compiler, OS, libraries
Ease of use	Medium	Complex	Requires generator	Simple CMakeLists.txt
Scalability for large projects	Hard	Hard	Fast but manual	Supports modularization (add_subdirectory())
Build speed	Slow	Slow	Very fast	Fast, supports Ninja

2. Example of a CMake Build System

   A simple CMakeLists.txt replaces complex Makefiles:

```
cmake_minimum_required(VERSION 3.16)
project(MyApp)

add_executable(MyApp main.cpp)
```

   To build the project:

```
cmake -S . -B build
cmake --build build
```

CMake automatically detects the compiler, generates the appropriate build system (Makefiles, Ninja, Visual Studio), and compiles the project efficiently.

1.2.4 Conclusion

While traditional build systems like Make, Autotools, and Ninja have been widely used, they come with limitations in portability, dependency management, and maintainability. CMake provides a modern, flexible, and cross-platform solution that simplifies the build process by generating native build files for different systems. With its ability to handle dependencies, detect system configurations, and support multiple build backends, CMake has become the industry standard for managing C++ projects.

1.3 Installing CMake on Different Operating Systems (Windows, Linux, macOS)

1.3.1 Introduction

CMake is a powerful and flexible build system generator that plays a crucial role in simplifying the process of building C++ projects across different platforms. The installation process is straightforward, but due to the variety of operating systems and user preferences, there are multiple methods to install it. This section will provide a detailed guide on how to install CMake on the most widely used operating systems: Windows, Linux, and macOS.

The installation process for CMake can involve precompiled binary installers, package managers, or even manual compilation from source. The method chosen depends on the user's specific needs, such as ensuring the latest version, ease of use, or whether the user prefers a command-line or graphical interface for installation.

Regardless of the installation method, once CMake is installed, it will allow you to generate build files for various platforms, manage dependencies, and enable a seamless integration with a variety of development tools. This section will walk you through each installation method for different operating systems, and provide verification steps to ensure CMake is installed and functioning correctly.

1.3.2 Installing CMake on Windows

Windows provides several methods for installing CMake, including using an official installer, package managers such as Chocolatey, or manual methods via Scoop. Here's a breakdown of each method.

1. Using the Official CMake Installer

One of the easiest and most common methods for installing CMake on Windows is by using the official CMake installer provided by Kitware, the creators of CMake. This method ensures that you have the latest stable version of CMake, and it allows for an easy installation process with a graphical user interface (GUI).

- Step 1: Download the Installer
  1. Go to the official CMake website: https://cmake.org/download/
  2. Download the Windows installer for your system architecture (either x64 or x86). Generally, x64 is the most common for modern systems.
- Step 2: Run the Installer
  1. After the download is complete, double-click the .msi file to launch the CMake installation wizard.
  2. During the installation, you will be presented with different options. The key option is to add CMake to the system PATH. This is a critical step because it allows you to run CMake from the command line (Command Prompt or PowerShell) without needing to specify its full path. You can select the option "Add CMake to the system PATH for all users".
  3. Proceed with the default installation settings and click Next through the installation wizard until the installation is complete.
- Step 3: Verify the Installation

  Once the installation is complete, it is important to verify that CMake has been installed correctly.
  1. Open Command Prompt or PowerShell.
  2. Type the following command to check the CMake version:

```
cmake --version
```

If CMake has been installed correctly, you should see the version of CMake displayed in the terminal, similar to:

```
cmake version 3.x.x
CMake suite maintained and supported by Kitware (https://cmake.org).
```

If you see this output, CMake has been installed and is ready to use.

- Installing CMake via Chocolatey (Alternative Method)

  Chocolatey is a package manager for Windows, and it provides a simple way to install software through the command line. If you have Chocolatey installed, you can install CMake using the following steps.

- Step 1: Install Chocolatey

  If you don't have Chocolatey installed yet, open PowerShell as Administrator and run the following command to install Chocolatey:

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
↪ [System.Net.ServicePointManager]::SecurityProtocol =
↪ [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
↪ System.Net.WebClient).DownloadString('https://community.chocolatey.org/install.ps1'))
```

- Step 2: Install CMake Using Chocolatey

  Once Chocolatey is installed, you can install CMake with a single command:

```
choco install cmake --installargs 'ADD_CMAKE_TO_PATH=System'
```

This will automatically install CMake and add it to your system's PATH so that you can use it from the command line.

- Step 3: Verify the Installation

  After the installation completes, open Command Prompt or PowerShell and run:

  ```
  cmake --version
  ```

  You should see the version of CMake that was installed.

- Installing CMake via Scoop (Alternative Method)

  Scoop is another command-line-based package manager for Windows that allows users to install software easily. If you prefer using Scoop, follow these steps.

- Step 1: Install Scoop

  Open PowerShell and run the following command to install Scoop:

  ```
  iwr -useb get.scoop.sh | iex
  ```

- Step 2: Install CMake Using Scoop

  Once Scoop is installed, you can install CMake by running:

  ```
  scoop install cmake
  ```

- Step 3: Verify the Installation

  After the installation is complete, verify it by running:

  ```
  cmake --version
  ```

1.3.3 Installing CMake on Linux

Linux distributions offer different ways to install CMake. You can use package managers for quick installations, or if you want to ensure you're getting the latest version, you can compile CMake from source.

1. Installing CMake via Package Managers

    - Ubuntu/Debian-based Distributions

      If you're using a Debian-based distribution such as Ubuntu, CMake can easily be installed using the APT package manager:

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

      Once the installation completes, you can verify it by running:

      ```
      cmake --version
      ```

    - Fedora-based Distributions

      For Fedora or similar distributions, the DNF package manager is used:

      ```
      sudo dnf install cmake
      ```

      You can check the installation with:

      ```
      cmake --version
      ```

    - Arch Linux (Manjaro, EndeavourOS, etc.)

If you're using Arch Linux or an Arch-based distribution like Manjaro, you can install CMake using the Pacman package manager:

```
sudo pacman -S cmake
```

Verify it using:

```
cmake --version
```

- Installing CMake from Source (For All Linux Distributions)

  Package managers often provide older versions of CMake. If you need the latest version, you can compile CMake from source.

  - Step 1: Download CMake Source
    Visit the official CMake GitHub repository or the CMake website to download the latest source code. Use the following command to download the source:

    ```
    wget
    ↪ https://github.com/Kitware/CMake/releases/latest/download/cmake-3.x.x.tar.gz
    ```

    Extract the downloaded file:

    ```
    tar -xvzf cmake-3.x.x.tar.gz
    cd cmake-3.x.x
    ```

  - Step 2: Compile and Install CMake
    First, you need to prepare the environment:

```
./bootstrap
```

Next, compile the source code:

```
make -j$(nproc)
```

Finally, install CMake on your system:

```
sudo make install
```

– Step 3: Verify Installation
Once the installation is complete, check that CMake was installed successfully:

```
cmake --version
```

1.3.4 Installing CMake on macOS

macOS offers multiple methods for installing CMake, including Homebrew, MacPorts, or manual installation via a graphical installer.

1. Installing CMake via Homebrew (Recommended)

   Homebrew is the most popular package manager for macOS, and it simplifies software installation.

   - Step 1: Install Homebrew

     If Homebrew is not already installed, you can install it by running the following command in Terminal:

```
/bin/bash -c "$(curl -fsSL
↪ https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- Step 2: Install CMake Using Homebrew

  After Homebrew is installed, you can install CMake with the following command:

```
brew install cmake
```

- Step 3: Verify the Installation

  Once installed, check the version of CMake:

```
cmake --version
```

- Installing CMake via MacPorts

  MacPorts is an alternative package manager for macOS that can also be used to install CMake.

- Step 1: Install MacPorts

  Download and install MacPorts from the official website: https://www.macports.org/install.php

- Step 2: Install CMake Using MacPorts

  After installing MacPorts, use the following command to install CMake:

```
sudo port install cmake
```

- Step 3: Verify the Installation

  You can check the installation by running:

```
cmake --version
```

- Installing CMake via the Official macOS Installer

  If you prefer a GUI-based approach, CMake also offers an official .dmg installer for macOS.

- Step 1: Download the Installer

  1. Visit the official CMake website: https://cmake.org/download/
  2. Download the macOS Universal Binary (.dmg) file.

- Step 2: Install CMake

  1. Open the .dmg file and drag the CMake.app into the /Applications folder.
  2. To enable command-line usage, open CMake.app, go to Tools > How to Install For Command Line Use, and follow the steps provided.

- Step 3: Verify the Installation

  After completing the steps, verify the installation by running:

```
cmake --version
```

1.3.5 Conclusion

The installation of CMake varies depending on the operating system being used, but regardless of the method, CMake provides the necessary tools to streamline and automate the build process for C++ projects. Whether you use an installer, a package manager, or build from source, the goal remains the same: to ensure that CMake is set

up properly so that you can manage and configure your project builds across various platforms.

Once CMake is installed successfully, you can begin using it to generate platform-specific build files, configure your project, and manage complex builds in a consistent and efficient manner. In the next section, we will explore CMake's basic commands and structure, which will lay the groundwork for mastering CMake in the context of real-world projects.

1.4 Verifying CMake Installation and Running It

1.4.1 Introduction

After you have installed CMake on your system, it is essential to ensure that the installation was successful and that CMake is functioning properly. This process is vital because any issues with the installation or configuration of CMake could lead to problems when building C++ projects. Verifying CMake's installation helps to confirm that the required binaries, environment variables, and necessary configuration files have been set up correctly.
CMake is a powerful build system generator, and if installed and configured properly, it should work seamlessly across various platforms like Windows, Linux, and macOS. This section walks you through how to verify your CMake installation and offers guidance on how to run it for the first time.

1.4.2 Verifying CMake Installation

After installation, you need to confirm that CMake has been correctly installed and can be accessed from the command line interface (CLI). The easiest way to verify the installation is by checking its version. This ensures that CMake is available in your system's PATH and that the correct version is installed. Let's go over the steps to check for CMake's version across different operating systems.

1. Checking the Version of CMake

    1. On Windows:

        - Open the Command Prompt (cmd) or PowerShell window.
        - Type the following command:

```
cmake --version
```

- If CMake has been installed successfully, you will see the version number of CMake. For example:

```
cmake version 3.x.x
CMake suite maintained and supported by Kitware (https://cmake.org).
```

This indicates that CMake is installed and ready to use. If you encounter an error message such as "command not found" or "CMake is not recognized as an internal or external command," this suggests that either the installation has failed or the system's PATH environment variable is not set correctly.

2. On Linux/macOS:
   - Open a Terminal window.
   - Run the following command:

```
cmake --version
```

   - If CMake is correctly installed, you should see an output similar to the one on Windows:

```
cmake version 3.x.x
```

   If you see an error indicating that cmake is not found, you may need to recheck the installation process and ensure that the cmake binary is properly linked to your system's PATH.

2. Troubleshooting CMake Installation

If you receive an error message indicating that the cmake command cannot be found, here are some troubleshooting steps:

- Ensure CMake is Added to PATH: When you install CMake, it is important to add the CMake executable to your system's PATH environment variable. If this step was missed during installation, you can manually add CMake to your PATH:
  - On Windows: You can add CMake to the PATH through the Environment Variables settings. To do this, go to System Properties > Advanced > Environment Variables, then edit the System PATH and add the directory where CMake is installed (e.g., C:\Program Files\CMake\bin).
  - On Linux/macOS: Open the shell configuration file (.bashrc or .zshrc, depending on your shell) and add the following line to include CMake in your PATH:

    ```
    export PATH="/path/to/cmake/bin:$PATH"
    ```

    After editing the file, run source ~/.bashrc or source ~/.zshrc to apply the changes.
- Verify Installation Location: Ensure that CMake was installed in the correct directory. If you installed it via a package manager, it may have been installed in a non-standard directory, especially on Linux or macOS. Verify that the installation path is valid and contains the cmake binary.
- Reinstall CMake: If none of the above solutions work, you may need to reinstall CMake. Be sure to follow the installation instructions carefully to avoid errors, and make sure to include the option to add CMake to the system PATH during the installation.

1.4.3 Running CMake for the First Time

Once you have confirmed that CMake is properly installed and the version is correctly displayed, the next step is to test CMake by running it on a simple project. This will help you verify that CMake is functioning as expected and that it can generate build files for your C++ projects.

1. Setting Up a Simple C++ Project for Testing

   Before you can run CMake, it is best to set up a basic C++ project. This allows you to test CMake's functionality by creating a minimal project and using CMake to generate the necessary build files. The following example demonstrates a very simple C++ program and how to use CMake with it.

   1. Create a Project Directory: First, create a directory to house the project files. Open a terminal or command prompt and create a new directory:

      ```
      mkdir MyTestProject
      cd MyTestProject
      ```

   2. Create a Simple C++ Source File: Inside the MyTestProject directory, create a simple C++ source file named main.cpp:

      ```
      // main.cpp
      #include <iostream>

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

3. Create the CMakeLists.txt Configuration File: Now, create the CMakeLists.txt file in the same directory, which will tell CMake how to build your project. Create a file called CMakeLists.txt with the following content:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.0)

project(MyTestProject)

add_executable(MyTestProject main.cpp)
```

In this example:

- cmake_minimum_required(VERSION 3.0) specifies that the project requires at least version 3.0 of CMake.
- project(MyTestProject) defines the project name as MyTestProject.
- add_executable(MyTestProject main.cpp) tells CMake to generate an executable named MyTestProject from the main.cpp source file.

This configuration file is very basic, but it covers the essential elements of a typical CMake project.

2. Run CMake to Generate Build Files

Now that you have a basic project and a CMakeLists.txt file, it's time to run CMake to generate the build system files (such as Makefiles or Visual Studio project files).

1. Create a Build Directory: To keep the build files separate from the source code, it is common practice to create a separate build directory. Create a new directory inside your project folder called build:

```
mkdir build
cd build
```

2. Run CMake to Configure the Project: Inside the build directory, run the following CMake command to configure your project:

```
cmake ..
```

This command tells CMake to look for the CMakeLists.txt file in the parent directory (..) and generate the build system files based on the configuration in that file. If everything is set up correctly, CMake will display output showing the configuration process, where it detects the system's compilers, checks for required tools, and prepares the necessary build files.

The output will look something like:

```
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/gcc
-- Check for working CXX compiler: /usr/bin/g++
-- Configuring done
-- Generating done
-- Build files have been written to: /path/to/MyTestProject/build
```

If there are any errors during this process, CMake will output detailed messages that can help you diagnose the issue. Common issues include missing dependencies, incorrect file paths, or unsupported compilers.

3. Building the Project

Once the configuration step is complete, you can now build the project. CMake has generated the necessary build files, so you can use the appropriate build tool to compile the code.

1. On Linux/macOS (Using Makefiles): If CMake generated Makefiles for your system, use the following command to compile the project:

   ```
   make
   ```

   This command will invoke the build system (Make) to compile the main.cpp file and generate the executable MyTestProject.

2. On Windows (Using Visual Studio Project Files): If you are using Windows and CMake generated Visual Studio project files, you can open the .sln file generated by CMake and build the project directly in Visual Studio. Alternatively, you can use the MSBuild command to build from the command line:

   ```
   MSBuild MyTestProject.sln
   ```

4. Running the Executable

   Once the build completes successfully, you can run the executable generated by CMake.

   - On Linux/macOS: In the terminal, run:

     ```
     ./MyTestProject
     ```

     The program should output:

```
Hello, CMake!
```

- On Windows: If you are using Visual Studio or the command line, you can simply run the MyTestProject.exe executable.

1.4.4 Troubleshooting Common Issues

While running CMake for the first time, there are a few common issues that you might encounter:

- Missing or Incorrect Compiler: CMake requires a working C++ compiler to build your project. If CMake cannot detect a valid compiler, it will show an error. Make sure that a C++ compiler (like GCC, Clang, or MSVC) is properly installed and accessible. On Linux/macOS, you can check the installed compiler version using gcc --version or clang --version. On Windows, make sure the Visual Studio build tools are installed correctly.

- Permissions Issues: On Linux and macOS, you may encounter permission-related errors if you do not have write access to certain directories. Ensure that you are running CMake with the appropriate permissions or try running with sudo if necessary.

- CMake Cache Conflicts: CMake caches configuration data to avoid reprocessing the same information multiple times. However, if you make changes to the project structure or the CMakeLists.txt file, the cached configuration may cause issues. You can delete the CMakeCache.txt file in your build directory and rerun the CMake command to clear the cache.

1.4.5 Conclusion

Verifying CMake installation and running it for the first time is a crucial step in setting up your development environment. By checking the CMake version and running it on a simple C++ project, you can ensure that everything is working as expected. If you encounter issues, troubleshooting steps such as checking the system PATH, verifying the compiler installation, or clearing the CMake cache can help resolve common problems. Once CMake is verified and working, you can proceed to more advanced topics, such as configuring larger projects and utilizing CMake's advanced features to improve your build system and project management workflow.

1.5 Your First CMake Project

1.5.1 Introduction

The previous sections provided an understanding of the importance of CMake, how it differs from traditional build systems, and how to install it on various operating systems. Now, it's time to take the next step and create your very first CMake project. This hands-on guide will walk you through the entire process, from setting up a simple C++ program to configuring the necessary build files, and ultimately compiling and running the project. By the end of this section, you will be comfortable with the fundamental aspects of working with CMake, which will form the foundation for tackling more advanced CMake features in subsequent sections.
Creating a project with CMake is straightforward and involves defining how your source code is compiled and linked, specifying compiler options, and ensuring that CMake generates the correct build system files for your chosen platform. With CMake, the complexity of build system generation is abstracted away, which saves time and reduces the possibility of errors in managing project configurations.

1.5.2 Setting Up a Simple C++ Project

To begin, you will create a minimal C++ project using CMake. This will include:

1. A basic C++ source file.
2. A CMakeLists.txt configuration file.
3. Building the project using CMake-generated files.

1. Project Structure

The basic structure for your project will include the source code and a CMake configuration file. Start by creating the project directory on your local machine. The directory should look like this:

```
MyFirstCMakeProject/
    CMakeLists.txt
    main.cpp
```

Here's what each part of this structure represents:

- CMakeLists.txt: This file is the heart of CMake. It contains instructions that CMake uses to configure the build process. It defines things like which source files to compile, which compiler options to use, and how to organize the output.
- main.cpp: This is your source code, containing the C++ code that will be compiled into an executable.

2. Writing the C++ Code (main.cpp)

Let's start by writing the code for the C++ program. Create the main.cpp file inside the MyFirstCMakeProject directory. Here is a simple "Hello World" program to begin with:

```
// main.cpp
#include <iostream>

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

This is a basic C++ program that prints "Hello, CMake!" to the console. It contains the minimal code needed to ensure the program compiles successfully and serves as a simple test case for understanding how CMake is used to build a C++ project.

1.5.3 Creating the CMakeLists.txt File

Next, you need to create the CMakeLists.txt file. This is the configuration file that CMake will use to configure the project. It tells CMake how to process the source code and generate the build files for compiling and linking the project.

1. Create a file named CMakeLists.txt in the root of your project directory.
2. Inside CMakeLists.txt, add the following code:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.10)

# Define the project
project(MyFirstCMakeProject)

# Add the executable target with the main.cpp file
add_executable(MyFirstCMakeProject main.cpp)
```

Let's break down each part of this CMakeLists.txt file:

- cmake_minimum_required(VERSION 3.10): This line specifies the minimum version of CMake that is required to configure and build the project. It's important to use an appropriate version of CMake to ensure compatibility with all CMake features.

- project(MyFirstCMakeProject): This defines the name of the project, which will also be used to name the executable created by CMake.

- add_executable(MyFirstCMakeProject main.cpp): This is the crucial instruction that tells CMake to generate an executable named MyFirstCMakeProject from the source file main.cpp. This line specifies that main.cpp is the source code to be compiled into the executable.

This is the simplest form of a CMakeLists.txt file, but as your projects become more complex, this file will grow to include other configurations, such as libraries, dependencies, compiler flags, etc.

1.5.4 Configuring the Build System with CMake

Now that you have your project set up, it's time to configure the build system. This involves using CMake to generate the build files needed for compiling the project. To keep things organized, it is best practice to create a separate build directory. This prevents the project directory from getting cluttered with generated files.

1. Creating a Separate Build Directory

    1. Navigate to your project's root directory (where CMakeLists.txt is located).
    2. Inside your project directory, create a new directory called

        ```
        build
        ```

        :

```
mkdir build
cd build
```

This will be the directory where all the build-related files will be placed, such as Makefiles or Visual Studio project files.

2. Running CMake to Generate Build Files

   Now that you have the build directory, run the following command from within the build directory to configure the project:

```
cmake ..
```

   This command tells CMake to look for the CMakeLists.txt file in the parent directory (..) and configure the project according to the specifications in that file.

   Upon running the command, CMake will inspect the environment and attempt to detect which compiler and tools to use for building the project. After processing the CMakeLists.txt file, it will generate the necessary files for the build system. On a typical system, this could include Makefiles, Visual Studio project files, or Ninja files, depending on your platform.

   Example output from the command might look like:

```
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/gcc
-- Check for working CXX compiler: /usr/bin/g++
-- Configuring done
-- Generating done
-- Build files have been written to: /path/to/MyFirstCMakeProject/build
```

At this point, CMake has successfully configured the project and written the necessary build files in the build directory. These build files describe the process CMake will use to compile your project.

1.5.5 Building the Project

With the build files generated, you can now compile your project using the generated build system.

1. Building on Linux/macOS with Make

   If the build system is configured to use Make, which is common on Linux and macOS systems, you can compile the project by running the following command:

   ```
   make
   ```

   This will invoke the make utility, which reads the generated Makefile and begins the process of compiling your project. Make will compile the main.cpp source file into an object file and link it to create the final executable.

   The output will indicate the progress of the build process, and once the build is complete, you should see something like:

   ```
   [100%] Built target MyFirstCMakeProject
   ```

   This means the build was successful, and the MyFirstCMakeProject executable has been created.

2. Building on Windows with Visual Studio

If you're on a Windows system and CMake generated Visual Studio project files, you have the option to either open the generated .sln solution file in Visual Studio and build the project through the IDE or use the command line.

To build the project from the command line, use MSBuild as follows:

```
MSBuild MyFirstCMakeProject.sln
```

This command tells MSBuild to use the Visual Studio build system to compile and link the project. After the build completes, you will have an executable ready to run.

1.5.6 Running the Executable

Once the build completes successfully, it's time to run the executable and see the output.

1. Running on Linux/macOS

   On Linux and macOS, the executable is typically located in the build directory. To run the executable, use the following command:

   ```
   ./MyFirstCMakeProject
   ```

   This will execute the program, and you should see the output:

   ```
   Hello, CMake!
   ```

   This confirms that the build was successful, and your program has run as expected.

2. Running on Windows

If you are using Visual Studio, you can run the executable directly from the IDE by pressing the "Start" button. Alternatively, after building the project, you can navigate to the Debug or Release folder (depending on your build configuration) and double-click the executable MyFirstCMakeProject.exe to run it.

1.5.7 Understanding the Build Process

To understand how the whole process fits together, let's review the steps involved when you run CMake and build the project.

1. CMake Configuration: When you run cmake .., CMake reads the CMakeLists.txt file in the parent directory. It checks for the system's environment, such as the available compiler and toolchain, and configures the project according to the options specified in the CMakeLists.txt file.

2. Build Generation: CMake generates build files that describe how to compile and link the project. This could be Makefiles, Visual Studio project files, or Ninja files, depending on your platform and configuration.

3. Compilation: When you run make or MSBuild, the build system compiles your source code files into object files and links them to form the final executable.

4. Execution: Once the executable is built, you can run it and see the output. If all the steps are followed correctly, you should see your program's output displayed on the terminal or IDE.

1.5.8 Conclusion

In this section, you learned how to create a basic CMake project, write the necessary C++ code, configure the project using the CMakeLists.txt file, and then build and run

the project. You should now understand the basic workflow involved in working with CMake and be able to create simple projects.

This foundational knowledge will serve as a springboard for diving into more advanced CMake features, such as managing external libraries, building multi-target projects, handling dependencies, and customizing build options. Understanding the basics of how CMake configures and generates build files is crucial to unlocking the full potential of CMake in larger, more complex projects.

Chapter 2

Fundamentals of CMakeLists.txt

2.1 Understanding the Structure of CMakeLists.txt

The CMakeLists.txt file is the cornerstone of every CMake-based project. It serves as the configuration script that CMake uses to generate build files, such as Makefiles or project files for IDEs like Visual Studio, Xcode, or others. Understanding the structure of this file is critical to mastering CMake and efficiently managing your build process. This section will explore the various components of the CMakeLists.txt file, including its organization, commands, and how different sections work together to define the build process for a project. We will go over everything from simple declarations to advanced usage in multi-directory projects, allowing you to use CMake to its full potential.

2.1.1 Introduction to CMakeLists.txt

The CMakeLists.txt file contains a series of CMake commands that specify how the project should be built. These commands define everything from the minimum required version of CMake to the actual files that will be compiled and linked into executables

and libraries. The beauty of CMake lies in its flexibility and portability; once a project is set up correctly, the same CMakeLists.txt file can generate build files for different platforms without any modification.
A simple project might only have one CMakeLists.txt file located at the root of the project directory. However, for larger projects with multiple modules or libraries, each directory might have its own CMakeLists.txt file that CMake will read recursively.
Here is an example of the simplest CMakeLists.txt file, which declares a minimum version of CMake, defines a project, and specifies an executable target:

```
cmake_minimum_required(VERSION 3.10)

project(MyProject)

add_executable(MyExecutable main.cpp)
```

This minimal setup creates a project called MyProject and an executable named MyExecutable, built from the source file main.cpp.

2.1.2 Key Sections of a CMakeLists.txt File

A well-structured CMakeLists.txt file usually follows a consistent pattern, starting with some basic setup, followed by project-specific configuration, and ending with target definitions and external dependencies. We will now go over the different sections you may encounter in a typical CMakeLists.txt file.

1. Minimum CMake Version Declaration

   Every CMakeLists.txt file begins with the declaration of the minimum version of CMake required to process it. This is essential because different versions of CMake may support different sets of features, syntax, or functionality. By specifying the

minimum version, you ensure that the build system will only be configured using a version of CMake that is compatible with your project's requirements.

For example:

```
cmake_minimum_required(VERSION 3.10)
```

This line tells CMake that the project requires at least version 3.10 of CMake to work correctly. If the user tries to configure the project with an older version of CMake, an error will be generated.

2. Project Declaration

   The project() command is another foundational element of a CMakeLists.txt file. This command declares the project name, version, and optionally, the programming languages that the project uses. It is often one of the first commands that appear after the cmake_minimum_required() declaration.

   For instance, the following line declares a project named MyProject that uses the C++ language:

```
project(MyProject VERSION 1.0 LANGUAGES CXX)
```

   In this case:

   - MyProject is the project name.
   - VERSION 1.0 declares the project version (although version numbers are often omitted for smaller projects).
   - LANGUAGES CXX specifies that the project uses the C++ language. This is optional in CMake 3.0 and later, as CMake automatically assumes C and C++.

3. Build Configuration

   This section defines various settings and configuration variables that affect the overall build process. The settings in this section can include the programming language standards, compiler flags, and whether to use debug or release builds.

   For example:

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_BUILD_TYPE Debug)
```

   - set(CMAKE_CXX_STANDARD 17) ensures that the C++17 standard is used for compiling C++ code. This is equivalent to passing the -std=c++17 flag to the compiler.
   - set(CMAKE_BUILD_TYPE Debug) specifies that the project should be built in the Debug configuration, which will include debugging information and disable optimizations.

   Additionally, other build configuration settings can include enabling/disabling certain features, adjusting optimizations, or controlling platform-specific settings.

4. Defining Executables and Libraries

   One of the most important parts of a CMakeLists.txt file is the definition of executables and libraries. These are the actual targets that will be compiled and linked by the build system.

   - Executables: To define an executable, the add_executable() command is used. This command tells CMake to create an executable target and specifies the source files needed to compile it.

For example:

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

In this case, MyExecutable is the name of the executable, and src/main.cpp is the source file that will be compiled into it.

- Libraries: Similarly, to create a library, the add_library() command is used. You can create both static and shared libraries. By default, the add_library() command creates a static library, but you can specify SHARED or MODULE for dynamic/shared libraries.

For example:

```
add_library(MyLibrary SHARED src/my_library.cpp)
```

Here, MyLibrary is a shared library built from the src/my_library.cpp file.

5. Target Properties and Dependencies

   Once the targets (executables or libraries) are defined, you can modify their properties, link them with other libraries, and define include directories. CMake provides commands like target_include_directories(), target_link_libraries(), and target_compile_options() to achieve this.

   - Include Directories: To specify additional directories where the compiler should look for header files, use the target_include_directories() command.

   Example:

```
target_include_directories(MyExecutable PRIVATE ${PROJECT_SOURCE_DIR}/include)
```

This command tells CMake to include the include directory, which is located at the root of the project (${PROJECT_SOURCE_DIR} is a CMake variable that holds the root project directory).

- Linking Libraries: The target_link_libraries() command links the target with other libraries. In this case, we are linking MyExecutable with MyLibrary.

Example:

```
target_link_libraries(MyExecutable PRIVATE MyLibrary)
```

This makes sure that MyExecutable will be linked with MyLibrary when it is built.

- Compiler Options: You can also set compiler-specific options for individual targets using target_compile_options().

Example:

```
target_compile_options(MyExecutable PRIVATE -Wall)
```

This would enable all compiler warnings for MyExecutable.

6. Handling External Dependencies

   CMake makes it easy to manage external dependencies, such as third-party libraries or tools. You can use commands like find_package() to search for pre-installed libraries or ExternalProject to fetch and build external projects during the configuration process.

For example, to find and link the Boost library, you would use the find_package() command as follows:

```
find_package(Boost REQUIRED)
target_link_libraries(MyExecutable Boost::Boost)
```

In this example, find_package(Boost REQUIRED) searches for the Boost library and ensures it is found. If Boost is not found, CMake will stop with an error. The target_link_libraries() command then links Boost::Boost to MyExecutable.

7. Handling Multiple Directories and Projects

   For large projects, you may want to break the project into smaller, manageable submodules. CMake allows you to include other CMakeLists.txt files from subdirectories by using the add_subdirectory() command.

   Example:

```
add_subdirectory(lib)
add_subdirectory(app)
```

   In this example, CMake will process the CMakeLists.txt files in the lib and app subdirectories. Each of these directories can have its own targets and build configuration, making it easier to modularize the build process. The main CMakeLists.txt file remains clean and high-level, delegating the detailed configuration to these subdirectories.

8. Comments and Documentation

   While not a functional part of the build process, comments are extremely important for documenting the CMakeLists.txt file. CMake allows single-line

comments using the # symbol, and multiline comments can be handled by using an if() block with endif().

For example:

```
# This is a simple comment in CMake
```

For more complex explanations:

```
if(FALSE)
  # This block is not executed, but useful for documentation
endif()
```

Comments can clarify the purpose of certain sections or describe why specific options are used, helping future developers (or yourself) understand the rationale behind the configuration.

2.1.3 Best Practices for Organizing CMakeLists.txt

- Minimal Root CMakeLists.txt: Keep the root CMakeLists.txt minimal and high-level. Focus on defining the project and calling add_subdirectory() for modules or libraries.
- Avoid Hardcoding Paths: Instead of hardcoding paths, use variables and CMake's built-in path handling functions to make your project portable. This ensures your build configuration works across different environments and operating systems.
- Use Variables Wisely: Define and use variables to store file paths, flags, and other project-specific settings. This makes the configuration more flexible and easier to maintain.

- Modularize Large Projects: For large projects, break them into smaller subprojects and use add_subdirectory() to include these submodules in the build process. This keeps your CMakeLists.txt files clean and modular.

- Write Clear and Descriptive Comments: It's important to explain complex sections of the CMakeLists.txt file. A well-commented file makes it easier to understand the build process, especially when dealing with large or complex projects.

2.1.4 Conclusion

The CMakeLists.txt file is an essential part of every CMake-based project. Understanding its structure and commands gives you the power to manage and customize the build process for your C++ projects. By organizing the file into well-defined sections—such as setting up the minimum CMake version, declaring the project, defining targets, handling dependencies, and configuring the build—you ensure that your project is flexible, maintainable, and portable.
With the knowledge from this section, you should be able to write and understand the basic structure of a CMakeLists.txt file, setting you on the path toward becoming proficient in CMake and mastering your C++ build system.

2.2 Defining the Minimal CMake Project

In this section, we will go through the essential steps required to define a minimal CMake project. A minimal CMake project is the simplest form of a project that can be built using CMake. It is useful as an introductory example for beginners and as a baseline for more complex projects as you begin to incorporate additional CMake functionality. Understanding this minimal structure will give you a solid foundation for working with more advanced features in later sections.

2.2.1 What Makes a CMake Project "Minimal"?

A minimal CMake project contains only the essential components required for building a basic executable. It avoids unnecessary complexities and focuses on the core elements needed to configure a CMake-based build system. The components include:

1. Minimum required version of CMake.
2. Project declaration (defining the project's name and version).
3. Defining an executable target.
4. Specifying source files.
5. Basic configuration (such as setting the C++ standard).

This structure is sufficient to compile and link a simple C++ program. Once these basic elements are understood, you can gradually extend the configuration to handle more complex tasks like linking external libraries, defining multiple targets, or creating shared/static libraries.
Let's begin by examining the key components and how to define them in a minimal CMake project.

2.2.2 CMake File Structure

A minimal CMake project typically has the following folder structure:

```
/MyMinimalProject
   CMakeLists.txt
   main.cpp
```

- CMakeLists.txt: The configuration file used by CMake to define the build instructions.
- main.cpp: A simple source file that will be compiled into an executable.

2.2.3 CMakeLists.txt File for the Minimal Project

The CMakeLists.txt file in the minimal project is simple but contains all the necessary components to create a working CMake-based build system. Below is an example of the file:

```
cmake_minimum_required(VERSION 3.10)

# Define the project
project(MyMinimalProject VERSION 1.0 LANGUAGES CXX)

# Set the C++ standard
set(CMAKE_CXX_STANDARD 17)

# Add the executable
add_executable(MyMinimalExecutable main.cpp)
```

Let's break this down step by step:

1. cmake_minimum_required()

```
cmake_minimum_required(VERSION 3.10)
```

   This command sets the minimum version of CMake required to process the project. It ensures that the CMake version being used is at least 3.10, which is necessary for certain features that might be used in the configuration. This line is essential because it guarantees that your CMake file will not break or behave unexpectedly on older versions of CMake that do not support newer commands or features.

   The minimum required version should be chosen carefully based on the features your project needs. It is a good practice to specify a version that is compatible with the features you plan to use, but also widely available across different environments.

2. project()

```
project(MyMinimalProject VERSION 1.0 LANGUAGES CXX)
```

   The project() command defines the name, version, and language of the project. In this case, MyMinimalProject is the name of the project, and 1.0 is the version number. The LANGUAGES CXX argument specifies that the project is written in C++ (CMake defaults to C and C++ if no languages are specified, but explicitly stating it can prevent potential confusion).

   This command also sets some project-wide variables, such as PROJECT_NAME (which holds the name of the project) and PROJECT_VERSION (which holds the version number). These variables can be used later in the build process or in the project documentation.

3. set(CMAKE_CXX_STANDARD 17)

```
set(CMAKE_CXX_STANDARD 17)
```

The set() command is used here to define the C++ standard version for the project. In this case, we are specifying that the project should be compiled using the C++17 standard. This is equivalent to passing the -std=c++17 flag to the C++ compiler. By setting this in the CMake configuration, we ensure that the C++17 features are enabled across the project.

You can change this value to 11, 14, 20, etc., depending on the version of C++ you wish to use in your project. This is an important setting because CMake will automatically propagate the standard across all targets in the project, reducing the need for repetitive compiler flags.

4. add_executable()

```
add_executable(MyMinimalExecutable main.cpp)
```

The add_executable() command defines an executable target that will be built from the provided source files. In this case, we are creating an executable named MyMinimalExecutable from the main.cpp source file.

This is the key command that ties together your source files and defines the primary output of the build process—an executable program.

- MyMinimalExecutable: This is the name of the executable that will be generated after building the project. This name can be any valid name for an executable file.

- main.cpp: This is the source file that CMake will compile and link to generate the executable. CMake automatically determines the dependencies between source files and includes them in the build process.

5. Building the Project

   Once you have the CMakeLists.txt file set up and the source files in place, you can now build your project using CMake. Here are the steps to build a minimal CMake project from the command line:

   1. Create a Build Directory: It is a best practice to create a separate build directory outside the source directory. This keeps your source directory clean and allows for out-of-source builds.

      ```
      mkdir build
      cd build
      ```

   2. Run CMake: Run CMake from the build directory, specifying the path to the root directory of the project (where the CMakeLists.txt file is located):

      ```
      cmake ..
      ```

      This command will generate the necessary build system files (such as Makefiles or Visual Studio project files) based on the configuration in the CMakeLists.txt file.

   3. Build the Project: Once the build files have been generated, you can build the project using the appropriate build tool. If you're using Makefiles, for example, you can use the make command:

```
make
```

4. Run the Executable: After the build process completes, you can run the executable:

```
./MyMinimalExecutable
```

This should output the result of your main.cpp program, which, in this case, might simply be a "Hello, World!" message or any other code you include in the main.cpp file.

2.2.4 Understanding the Minimal CMake Project

Let's take a moment to reflect on why this setup is considered minimal, and why it works for a basic project:

- Simplicity: The project is small and simple, containing only one executable target and one source file. This minimal structure is useful for learning and testing the most basic functionality of CMake.
- Automatic Dependency Management: By using add_executable() and the CMAKE_CXX_STANDARD variable, CMake takes care of the underlying complexity of finding dependencies and ensuring that the correct compiler flags are applied for compiling and linking the project.
- Portability: Once you have a minimal CMake project set up, it is portable. By adjusting only the CMakeLists.txt file, you can generate build files for different platforms, such as Linux, Windows, and macOS, without having to modify your source code or project structure.

- Extensibility: Although this project is minimal, it serves as a foundation for extending the project as you add more features. For instance, you can later add more source files, link external libraries, define multiple targets, or introduce more advanced CMake features like custom build commands or conditional logic.

2.2.5 Next Steps and Expansion

While the minimal CMake project provides the basic building blocks for a CMake-based build system, there are several directions in which you can expand the project:

1. Adding More Source Files: If your project grows and you need to organize your code into multiple files, you can simply add more source files to the add_executable() command.

   Example:

   ```
   add_executable(MyMinimalExecutable main.cpp utils.cpp)
   ```

2. Handling External Dependencies: As you add external libraries or dependencies, CMake provides powerful commands like find_package() to locate and link these libraries.

3. Building Libraries: If your project requires shared or static libraries, you can use the add_library() command to define libraries in addition to executables.

4. Organizing Source Files: For larger projects, consider organizing source files into directories. You can then use add_subdirectory() to manage different parts of the project.

2.2.6 Conclusion

In this section, we defined the minimal CMake project, which includes the essential components necessary to build a simple C++ program. By creating a CMakeLists.txt file with the minimum required CMake version, project declaration, executable definition, and compiler settings, you have established a basic build system that is portable, extensible, and easy to maintain.
Understanding this minimal structure serves as a foundation for more complex projects. Once you have mastered this, you can start adding features like multiple targets, external dependencies, custom build commands, and other advanced CMake functionality. This is just the first step toward building more sophisticated and scalable CMake projects.

2.3 Essential CMake Commands (cmake_minimum_required, project, add_executable)

In this section, we will focus on three essential CMake commands that are foundational to every CMake project. These commands—cmake_minimum_required, project, and add_executable—are crucial to setting up and configuring a CMake project. Understanding their purpose, syntax, and how to use them correctly is key to creating functional and efficient CMake build systems.
These commands are the building blocks that help define the minimum requirements for your project, specify the project's name and version, and define executable targets. Let's explore each of these in detail.

2.3.1 cmake_minimum_required

1. Purpose

   The cmake_minimum_required command is used to specify the minimum required version of CMake that is needed to process a CMakeLists.txt file. This command ensures that your CMake configuration will only run on a version of CMake that supports the features and syntax your project requires.

   This command is especially important for maintaining compatibility with newer CMake features, as older versions of CMake may not support all the commands and options available in the latest versions. By explicitly defining the minimum version, you can avoid running into issues when your project is being configured on systems with older versions of CMake.

2. Syntax

```
cmake_minimum_required(VERSION <version>)
```

- VERSION <version>: Specifies the minimum version of CMake that is required. Replace <version> with the desired CMake version (for example, 3.10 or 3.15).

3. Example

```
cmake_minimum_required(VERSION 3.10)
```

In this example, the project will require at least CMake version 3.10. If CMake is run with an older version, an error will occur, and the build process will not proceed. This is important to ensure that your CMakeLists.txt file uses only features and commands that are supported by the specified version or later.

4. Why is cmake_minimum_required Important?

- Compatibility: It guarantees that your CMakeLists.txt file will run on systems with a version of CMake that supports all the commands used in the script. If the minimum version is not specified, CMake will assume that any version of CMake is valid, which could lead to compatibility issues.
- Error Prevention: By specifying the minimum required version, you prevent unexpected errors related to incompatible features and behaviors that may be introduced in future versions of CMake.
- Clarity: It provides clear documentation about the version of CMake needed to build the project, which helps anyone working with the project (especially in a team or open-source context) understand which version of CMake is compatible with the project.

2.3.2 project

1. Purpose

   The project command is used to define the project's name, version, and the programming languages that the project uses. This command essentially declares the project's identity and tells CMake how to configure the build process accordingly. It is typically one of the first commands in a CMakeLists.txt file after the cmake_minimum_required command.

2. Syntax

```
project(<name> [<language1> <language2> ...] [VERSION <version>] [DESCRIPTION
↪   <description>])
```

   - <name>: The name of the project. This is the primary identifier for the project and is often used to define the output executable or library names.
   - <language1> <language2> ...: A list of programming languages used in the project (e.g., CXX for C++, C for C). If this is omitted, CMake assumes the project uses both C and C++ by default.
   - VERSION <version>: Optionally defines the project version (e.g., 1.0).
   - DESCRIPTION <description>: Optionally provides a brief description of the project.

3. Example

```
project(MyProject VERSION 1.0 LANGUAGES CXX)
```

In this example, we define a project named MyProject, with a version of 1.0, and we specify that the project uses the C++ programming language (CXX).

4. Key Features of the project Command

   - Project Name: The name specified in the project command is stored in the PROJECT_NAME variable, and this name is used throughout the project configuration process. For example, the output executables and libraries will often take the project name as part of their default names.
   - Project Version: The version is stored in the PROJECT_VERSION variable and can be used for version-specific logic in the CMakeLists.txt file, such as selecting different compiler flags, dependencies, or features based on the version of the project.
   - Language Declaration: By specifying LANGUAGES, you make it clear to CMake which compilers to use for the project. For example, LANGUAGES CXX tells CMake to use a C++ compiler. If not specified, CMake assumes C and C++ by default. The list of languages allows you to include other programming languages (like Fortran, CUDA, or Python) depending on the needs of your project.

2.3.3 add_executable

1. Purpose

   The add_executable command is used to define an executable target for your project. This is the primary command for specifying the compilation of a source file or set of source files into an executable program. It ties together the source code and tells CMake to generate the corresponding binary after compilation.

This command can be thought of as the key step in creating an application or a runnable program. It is typically followed by additional configuration to link libraries or specify custom compile options.

2. Syntax

```
add_executable(<name> [source1] [source2] ...)
```

- <name>: The name of the executable that will be generated. This name will be the resulting file's name (on Linux or macOS, the executable will not have an extension, but on Windows, it will have a .exe extension).
- [source1] [source2] ...: A list of source files that will be compiled into the executable. These can be C++ source files (.cpp), header files (.h), or other files needed for the build process.

3. Example

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

In this example, CMake will compile the source files main.cpp and utils.cpp into an executable called MyApp. After running cmake and make (or an equivalent build tool), an executable file named MyApp will be generated.

4. Key Considerations When Using add_executable

- Target Name: The <name> parameter in add_executable defines the name of the output executable. It is best to use a name that clearly represents the program's purpose.

- Source Files: The source files listed in add_executable are compiled together to produce the final binary. It is important to ensure that the correct set of files is included for the project to build successfully.
- Multiple Source Files: You can list multiple source files within the add_executable command, and CMake will handle their compilation. For larger projects, it is also common to organize source files into directories and use CMake variables or file(GLOB ...) to automatically collect source files.
- Dependencies: After defining an executable, you will often link it to other libraries or dependencies using the target_link_libraries() command. This ensures that the executable has access to the necessary functionality provided by external libraries.

2.3.4 Putting It All Together: A Simple Example

Let's take a look at a simple CMakeLists.txt that incorporates all three commands: cmake_minimum_required, project, and add_executable.

```
cmake_minimum_required(VERSION 3.10)

project(SimpleApp VERSION 1.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)

add_executable(SimpleApp main.cpp utils.cpp)
```

Explanation:

1. cmake_minimum_required(VERSION 3.10): Specifies that the project requires at least CMake version 3.10.

2. project(SimpleApp VERSION 1.0 LANGUAGES CXX): Declares the project with the name SimpleApp, sets its version to 1.0, and specifies that it uses C++.

3. set(CMAKE_CXX_STANDARD 17): Specifies that C++17 should be used for compiling the project.

4. add_executable(SimpleApp main.cpp utils.cpp): Compiles main.cpp and utils.cpp into an executable named SimpleApp.

This project can be built by creating a separate build directory, running CMake, and then building the executable.

2.3.5 Conclusion

In this section, we examined three essential CMake commands that form the backbone of a basic CMake project: cmake_minimum_required, project, and add_executable.

- cmake_minimum_required ensures compatibility with the appropriate CMake version.

- project defines the project's name, version, and programming language(s), establishing the identity of the project.

- add_executable links source files together to create an executable target.

Mastering these commands is critical for any developer working with CMake. They help structure your project and ensure that the build process is compatible with different CMake versions and setups, making your project easier to manage and maintain. These commands form the foundation on which more advanced CMake features—such as library management, external dependencies, and complex configurations—are built. By understanding these fundamental commands, you can begin creating simple CMake

projects, and as your needs grow, you can expand the configuration to accommodate more complex requirements.

2.4 CMake Variable Types (CACHE, ENV, LOCAL)

In CMake, variables play an essential role in controlling the configuration and build process. However, not all variables in CMake behave the same way. Understanding the different types of variables in CMake—CACHE, ENV, and LOCAL—is crucial for managing the scope and behavior of these variables, especially as your project grows and becomes more complex.

Each variable type has different rules for its scope, persistence, and how it interacts with other parts of the build system. This section explores the three primary types of variables you will encounter when working with CMake, along with practical examples of their usage.

2.4.1 CACHE Variables

1. Purpose

   CACHE variables are used to store values that should persist across multiple runs of CMake. These variables are typically used for configuration settings that need to be set once, either by the user or during an initial setup, and then remain consistent throughout the project lifecycle.

   A key characteristic of CACHE variables is that they are stored in the CMake cache file (CMakeCache.txt). This file can be inspected or modified between CMake runs, and it allows users to control values without having to modify the CMakeLists.txt file directly.

2. Syntax

```
set(<variable> <value> CACHE <type> <docstring> [FORCE])
```

- <variable>: The name of the variable.
- <value>: The value to assign to the variable.
- CACHE: Indicates that the variable is a cache variable.
- <type>: The type of the variable (e.g., STRING, PATH, BOOL, FILEPATH).
- <docstring>: A description of the variable, which is helpful for documentation purposes and will be displayed in the CMake GUI or when using cmake-gui or ccmake.
- [FORCE]: Optional. Forces the variable to be set even if it has already been defined in the cache.

3. Example

```
set(MY_PROJECT_PATH "/path/to/my/project" CACHE PATH "Path to the main project
↪  directory")
```

In this example, MY_PROJECT_PATH is a CACHE variable. The value /path/to/my/project is set for the variable, and the type is PATH (which indicates that it is a file or directory path). The docstring provides additional information about the variable for users to understand its purpose.

4. Use Cases for CACHE Variables

- User-defined configurations: If you want to allow the user to set certain variables during configuration (e.g., through the CMake GUI or via the command line), you can use CACHE variables. These are ideal for settings that are configurable, such as the installation directory, path to external dependencies, or build options.

- Persistent settings: Cache variables persist between different runs of CMake, making them ideal for configuration settings that don't change frequently, such as the path to installed libraries or version numbers.
- Controlling build options: You can use CACHE variables to allow users to toggle features (e.g., whether to enable a particular module or build type) during the CMake configuration phase.

5. Modifying CACHE Variables

   To modify a CACHE variable, you can either:

   1. Use the cmake-gui or ccmake interface to modify the variable.
   2. Set it directly from the command line by passing the variable to CMake:

      ```
      cmake -DMY_PROJECT_PATH="/new/path/to/project" ..
      ```

   3. If you want to force a value to be set for a cache variable, even if it has already been set, you can use the FORCE option:

      ```
      set(MY_PROJECT_PATH "/new/path" CACHE PATH "Updated project path"
      ↪  FORCE)
      ```

2.4.2 ENV Variables

1. Purpose

   ENV variables are used to access environment variables within CMake. These variables provide a way for your build system to interact with the host operating system's environment and are often used to pass system-level settings or configuration details to the CMake build process.

Environment variables are not set by CMake but are inherited from the operating system's environment or shell. You can use ENV variables to read environment settings, such as paths to compilers, library directories, or system configuration details.

2. Syntax

```
set(<variable> $ENV{<env_variable>})
```

- <variable>: The name of the CMake variable you want to assign.
- $ENV{<env_variable>}: Accesses the environment variable <env_variable>. This is a special syntax that retrieves the value of the environment variable.

3. Example

```
set(MY_LIBRARY_PATH $ENV{LIBRARY_PATH})
```

In this example, the CMake variable MY_LIBRARY_PATH will be set to the value of the environment variable LIBRARY_PATH. The value of LIBRARY_PATH is typically set by the system and contains directories where libraries are located.

4. Use Cases for ENV Variables

- System-level configuration: If you want your CMake configuration to automatically read certain system-level environment variables, you can use ENV variables. For instance, this is useful when dealing with tools or compilers that are set by the operating system or environment.

- Accessing environment-specific settings: When building on different systems or environments (e.g., development, staging, production), you may want to access different paths, settings, or credentials based on environment variables that change between systems.
- Portable builds: ENV variables help make your build system more portable across different machines by automatically picking up paths and settings defined in the environment.

5. Common Environment Variables

   - PATH: Contains directories for executable binaries. You can use this to find tools like compilers.
   - LD_LIBRARY_PATH or DYLD_LIBRARY_PATH: Specifies directories for shared libraries on Linux and macOS.
   - CXX and CC: Used to specify the C++ and C compilers.
   - HOME: Represents the user's home directory and can be used for paths to configuration files, data directories, etc.

2.4.3 LOCAL Variables

1. Purpose

   LOCAL variables are used to define variables that exist only within the scope of the current directory or block (such as a function() or macro()). These variables are not visible outside of the scope in which they are defined, ensuring that they do not interfere with other parts of the build configuration.

   LOCAL variables are the default type of variable in CMake. When you use the set() command without explicitly specifying a variable type, it creates a LOCAL

variable. These variables are temporary and are discarded once the scope in which they are defined ends.

2. Syntax

```
set(<variable> <value>)
```

   - <variable>: The name of the variable.
   - <value>: The value to assign to the variable.

3. Example

```
set(MY_LOCAL_VAR "This is a local variable")
```

   In this example, MY_LOCAL_VAR is a LOCAL variable. It is available only within the current scope (e.g., within the CMakeLists.txt file or a specific function or block) and cannot be accessed outside of it.

4. Use Cases for LOCAL Variables

   - Temporary values: Use LOCAL variables when you need to store temporary values that will only be used within a specific part of the CMakeLists.txt file, such as within a loop or function.
   - Avoiding conflicts: Since LOCAL variables are confined to the scope in which they are created, they help avoid conflicts with other variables defined in different parts of the project. This is especially useful when writing functions or macros that need to operate without altering the global state.
   - Scope control: LOCAL variables provide better control over where a variable is accessible. They don't "leak" into other parts of the project, which can help keep the build configuration clean and organized.

2.4.4 Summary of Variable Types

Variable Type	Scope	Persistence	Typical Use Cases
CACHE	Global (across all directories)	Persistent	User-defined configurations, settings, options
ENV	Global (system environment)	Persistent	Access system environment settings, system paths
LOCAL	Local to the current scope	Temporary	Temporary, non-persistent values within a block

- CACHE variables are used for persistent values that need to be available across multiple CMake runs and can be modified by the user.
- ENV variables are used to access system or environment variables from the host operating system.
- LOCAL variables are temporary and only exist within the scope in which they are defined, making them ideal for internal values that don't need to persist beyond the current configuration step.

2.4.5 Conclusion

Understanding the different variable types in CMake—CACHE, ENV, and LOCAL—is essential for effective project configuration and build management. By selecting the right variable type for the job, you can control the scope, persistence, and visibility of configuration values in a way that helps keep your build system clean and maintainable.

- Use CACHE for user-configurable settings that should persist across multiple CMake runs.

- Use ENV for environment-specific values that need to be accessed during the build process.
- Use LOCAL for temporary values that are needed only in a specific scope.

By mastering these variable types, you can ensure that your CMake configuration is both flexible and efficient.

2.5 Using message() for Debugging and Output

In CMake, managing the configuration of your project can become complex, especially as the number of variables, dependencies, and build options increases. In such scenarios, debugging and providing feedback during the CMake configuration process is essential. One of the most useful tools for this task is the message() command.
The message() command in CMake is a simple yet powerful way to output information during the configuration and build process. It can be used to display debugging information, warnings, errors, or general status updates. By understanding how to use message() effectively, you can gain better insight into your project's build configuration, detect issues early, and track the flow of the build process.
This section will cover the basics of the message() command, its various usage options, and how to use it for debugging and providing meaningful output during the configuration process.

2.5.1 Purpose of the message() Command

The primary purpose of the message() command is to allow you to display messages to the user during the CMake configuration phase. These messages are typically used for:

- Debugging: Printing variable values, checking paths, or verifying conditions during the configuration phase.
- Status Updates: Informing users or developers about the progress or state of the build configuration.
- Warnings and Errors: Alerting users to issues that need attention or stopping the configuration process if critical errors occur.

The message() command can output messages at different levels of severity, allowing you to categorize the output based on its importance.

2.5.2 Syntax of message()

The basic syntax of the message() command is as follows:

```
message([<mode>] <message>)
```

- <mode>

  : Optional. Defines the severity level of the message. It can be one of the following:

  - STATUS: Default. Prints a regular informational message.
  - WARNING: Prints a warning message. It shows in yellow and can indicate a non-fatal issue.
  - AUTHOR_WARNING: A warning message intended only for the author (developer), not the user. It is similar to WARNING but can be filtered out by users in a non-interactive setup.
  - SEND_ERROR: Prints an error message and halts the configuration process. This is used to indicate a critical issue that prevents further configuration.
  - FATAL_ERROR: Similar to SEND_ERROR, but it immediately stops the configuration process, making it impossible to continue. This is used when a critical error prevents any further progress.
  - DEPRECATION: Used to warn the user about the use of deprecated features or practices in the CMake configuration.

- <message>: The text or string that you want to print. This can include variables, paths, or any other information you want to display.

2.5.3 Basic Examples of message()

1. Simple Message

```
message("This is a simple message")
```

   This will print the message "This is a simple message" in the standard output during the configuration phase.

2. Using STATUS for Informational Messages

   By default, message() uses the STATUS mode, which is suitable for informational messages that are not critical to the build process.

```
message(STATUS "Configuring project...")
```

   This will display the message "Configuring project..." in the output, typically with a green color to denote that it's informational.

3. Using WARNING for Warnings

   You can use WARNING to display a warning message. This is helpful when you want to inform users of a potential issue that does not block the build process but might require attention.

```
message(WARNING "Warning: The path to the library is not set correctly!")
```

   This will print a yellow-colored warning message that informs users about a possible issue, but it won't stop the build configuration.

4. Using SEND_ERROR for Errors

   If you encounter a situation that must be addressed before proceeding with the configuration, you can use SEND_ERROR to display an error message. This will not stop the configuration immediately but will mark the build as having an error, preventing the generation of makefiles or build files.

```
message(SEND_ERROR "Error: Missing required dependency!")
```

   This will print the error message and continue with the configuration process, but the error will be recorded, and no build files will be generated.

5. Using FATAL_ERROR for Critical Errors

   If the configuration cannot proceed due to a critical error, you can use FATAL_ERROR. This will immediately stop the configuration process and prevent any further steps.

```
message(FATAL_ERROR "Critical error: Cannot find the required C++ compiler!")
```

   When this message is encountered, CMake will stop immediately, and no build files will be generated. This is useful for situations where proceeding without resolving the error would lead to a broken or incomplete build.

6. Using DEPRECATION for Deprecated Features

   If you are working with deprecated features or commands in your CMakeLists.txt file, you can use DEPRECATION to alert the user about the deprecated feature.

```
message(DEPRECATION "Warning: The `add_custom_command` is deprecated, consider
↪   using `add_custom_target`.")
```

This will display a message indicating that a certain feature is deprecated, helping guide users or developers toward better practices.

2.5.4 Using Variables in message()

A powerful feature of the message() command is its ability to print variable values. By including CMake variables in the message string, you can dynamically generate output based on the current configuration state.

1. Displaying Variable Values

```
set(MY_VAR "Hello, CMake!")
message(STATUS "The value of MY_VAR is: ${MY_VAR}")
```

This will output:

```
The value of MY_VAR is: Hello, CMake!
```

Using ${} allows you to reference the value of a variable and incorporate it into the message.

2. Debugging with Variables

You can use message() to debug variable values during the configuration process. This is particularly useful when you want to track the values of important variables at different points in the CMakeLists.txt file.

```
set(MY_VAR "Some value")
message(STATUS "Before: MY_VAR = ${MY_VAR}")
# Modify the variable
```

```
set(MY_VAR "New value")
message(STATUS "After: MY_VAR = ${MY_VAR}")
```

This will output:

```
Before: MY_VAR = Some value
After: MY_VAR = New value
```

This helps in tracking changes to variables as the CMake configuration progresses.

2.5.5 Controlling Output Visibility

Sometimes, you may want to control the visibility of the messages during the configuration process. For example, you might want to suppress some messages unless you explicitly enable debug output.

1. Controlling Verbosity with CMAKE_VERBOSE_MAKEFILE

   One way to control verbosity is by setting the CMAKE_VERBOSE_MAKEFILE variable. When set to TRUE, it enables more detailed output during the build process, which can be helpful for debugging the build steps themselves. However, this does not directly control message() output, but it can help control the level of detail you get from the build process.

   ```
   set(CMAKE_VERBOSE_MAKEFILE TRUE)
   ```

2. Controlling Debug Output with CMAKE_MESSAGE_LOG_LEVEL

   Another way to control output visibility is by setting the CMAKE_MESSAGE_LOG_LEVEL variable. This determines the threshold of

message severity that is displayed. You can choose to show only errors, warnings, or detailed status messages.

```
set(CMAKE_MESSAGE_LOG_LEVEL "WARNING")
```

This would display only warnings and errors, suppressing informational messages.

Chapter 3

Building Projects with CMake

3.1 Understanding "Configure," "Generate," and "Build" Steps

CMake is a powerful tool that automates the process of building and managing complex projects. However, before the actual build process takes place, CMake goes through a few preliminary steps: configure, generate, and build. These three distinct phases are essential to the CMake workflow and understanding their roles is crucial for effectively using CMake to manage your C++ projects. In this section, we will delve into each of these steps and explore what they involve, how they relate to each other, and why they are important.

3.1.1 Overview of the CMake Workflow

The CMake build process typically involves three primary steps:

1. Configure: This is where CMake inspects your environment, reads the CMakeLists.txt files, and generates necessary configuration files that are tailored to your system and project. This phase is about defining the build environment, checking dependencies, and setting up necessary flags and options for the build process.
2. Generate: After the configuration step, CMake generates the build system files. These files are specific to the generator you selected (such as Makefiles, Visual Studio project files, or Xcode project files). The generated files are used by the build tools to carry out the actual compilation and linking of the project.
3. Build: This step is where the actual compilation and linking of your project take place. It involves invoking a build tool (like make, ninja, or the native build system for IDEs such as Visual Studio or Xcode) to perform the build based on the files generated in the previous step.

3.1.2 The "Configure" Step

The configure step is the initial phase of working with CMake, and it is where CMake sets up everything needed to generate the build files. During configuration, CMake performs the following tasks:

- Reads CMakeLists.txt Files: The CMakeLists.txt file contains the project's build instructions. CMake processes these files to understand what needs to be built, which libraries are required, and what dependencies need to be resolved. If there are any find_package() or find_program() calls, CMake will attempt to locate these packages and executables on your system.
- Checks the Environment: CMake inspects your system environment to determine the necessary tools and libraries for building the project. It checks

for compilers (e.g., gcc, clang, or MSVC), system libraries, required tools, and other software dependencies. If a required dependency is missing, CMake will either notify you or attempt to download or build it.

- Sets Configuration Variables: Configuration variables (like compiler flags, paths to libraries, and options for features like multi-threading or debugging) are set during this step. You can provide values for these variables via the CMake command line, environment variables, or by editing the CMakeLists.txt file.
- Generates Cache Variables: The configuration step creates a CMakeCache.txt file in the build directory. This file contains key configuration information and variable values, which persist across CMake runs. If you change settings or modify paths, they can be reflected in the cache and used in subsequent builds.

1. Example of Running the configure Step

   The configure step can be initiated using the CMake command line interface (CLI) as follows:

   ```
   cmake <path-to-source>
   ```

   For example:

   ```
   cmake ../my_project
   ```

   This will trigger CMake to process the CMakeLists.txt files in the specified directory and configure the project for the current system. Once complete, CMake will have generated the necessary build system files for the next step.

2. Common CMake Configuration Options

Some commonly used configuration options include:

- -DCMAKE_BUILD_TYPE=Release: Specifies the build type, such as Release, Debug, or RelWithDebInfo.
- -DCMAKE_INSTALL_PREFIX=<path>: Sets the installation directory.
- -DUSE_FOO=ON: Enables or disables specific features or packages.

3.1.3 The ”Generate” Step

Once the configuration step is complete, the next phase is the generate step. In this phase, CMake generates the files required by the build system. The generation process is determined by the generator you select, which could be a build tool or IDE-specific file format.

CMake supports several types of generators, such as:

- Makefiles: This is the most common generator for Linux and macOS environments. It produces a Makefile that can be used with the make tool to compile and link the project.
- Ninja: A small, fast build system that is an alternative to make. If you specify -G Ninja, CMake will generate build.ninja files for use with the ninja build tool.
- IDE-Specific Generators: These are used to generate project files for various IDEs like Visual Studio, Xcode, or CodeBlocks. For example, on Windows, running cmake -G ”Visual Studio 16 2019” .. will generate Visual Studio project files that you can open directly in the IDE.
- Unix Makefiles: These are the default generator on many Unix-like systems, producing a set of Makefile scripts that can be used to invoke make.

Example of Running the generate Step

The generation step is invoked automatically as part of the configuration phase when you run the CMake command. For example:

```
cmake -G "Unix Makefiles" ../my_project
```

This command will configure and then generate Makefile build files in the build directory.

3.1.4 The "Build" Step

After the configuration and generation phases are complete, you move to the build step. This is the phase in which the actual compilation, linking, and final build of your project occur.

During this step, the build tool (such as make, ninja, or Visual Studio) uses the files generated in the previous phase to build the project. The build tool will execute the instructions specified in the generated files to compile the source code, link the object files into executables, and create libraries as defined in the CMakeLists.txt file.

1. Running the Build Step
    - With Makefiles: If you used cmake -G "Unix Makefiles", you can build the project by running make in the build directory:

      ```
      make
      ```

    - With Ninja: If you used the Ninja generator, you would use the ninja tool to build the project:

```
ninja
```

- With IDEs (e.g., Visual Studio): If you generated project files for an IDE like Visual Studio, you can build the project directly from within the IDE interface or use the command line:

  ```
  msbuild MyProject.sln
  ```

2. Build Targets and Customization

   You can also build specific targets or configure additional steps in your build process. For example:

   ```
   make install
   ```

   This will install the project if you have set up the installation rules in your CMakeLists.txt file using commands like install().

3.1.5 Relationship Between Configure, Generate, and Build

While the configure, generate, and build steps are distinct, they are interdependent and occur in sequence:

1. Configure: Set up the project, inspect the system, define variables, and check dependencies.
2. Generate: Create the necessary build files (such as Makefile, ninja, or IDE-specific files) based on the configuration.
3. Build: Use the generated build files to compile and link the project into executables or libraries.

It's important to note that the configure step often only needs to be run once unless you make changes to the configuration (such as adding new source files, changing build options, or modifying dependencies). The generate step is run after configuration to generate the appropriate build system files, and the build step can be run multiple times during the development cycle, especially when making incremental changes to the project.

3.1.6 Re-running the Configuration Steps

If you need to modify your build configuration (for example, to change compiler flags or enable/disable features), you can re-run the configure and generate steps. When this happens, CMake will read the configuration files again, update the cache, and regenerate the build system files.

Sometimes, changes to the CMakeLists.txt files or other source files will require cleaning the build directory before re-running the configuration. CMake supports incremental builds, but certain changes might require a fresh configuration.

Example of Re-running CMake

```
cmake ../my_project
```

This will reconfigure the project. If the configuration or generator has changed, CMake will regenerate the necessary files.

3.1.7 Summary of the Configure, Generate, and Build Phases

Step	Description	When to Run
Configure	CMake inspects the system and prepares the build configuration files.	Whenever you change project settings, dependencies, or configurations.
Generate	CMake generates the build system files (Makefiles, Visual Studio project files, etc.).	After configuration, whenever build system files need to be generated or regenerated.
Build	The actual compilation and linking process occurs using the generated build system files.	Repeatedly during development as changes are made to the project code.

3.1.8 Conclusion

Understanding the three key steps of the CMake workflow—configure, generate, and build—is critical for efficiently managing and building C++ projects with CMake. These steps are interdependent and serve distinct purposes in the overall process:

- Configure: Set up the project environment and check dependencies.
- Generate: Create the appropriate build system files for your platform.
- Build: Compile the project and create the desired outputs.

By following this workflow, you can efficiently manage complex builds, handle dependencies, and customize your project setup according to your system and development environment.

3.2 Running cmake with Different Generators (Ninja, Makefile, Visual Studio)

CMake supports a variety of generators that allow you to configure and generate build files for different platforms, build systems, and Integrated Development Environments (IDEs). These generators define the type of build system that CMake will create for your project. Understanding how to run cmake with different generators—such as Ninja, Makefile, and Visual Studio—is essential for customizing your build process to suit your development environment.

In this section, we will explore how to use CMake with different generators and how they affect the project setup and build process. We'll walk through the specifics of working with each of these popular build systems, explaining their strengths and providing practical examples.

3.2.1 Overview of CMake Generators

When you run CMake, one of the key options you specify is the generator. The generator determines what kind of build files CMake will produce. Each generator corresponds to a specific build system or IDE, and selecting the right one ensures that CMake can interact seamlessly with your development environment.

Some of the most common generators include:

- Ninja: A fast, small, and efficient build system.
- Makefile: The traditional Unix-based build system that uses make to build the project.
- Visual Studio: A set of generators that produce project files for various versions of Microsoft Visual Studio.

These generators offer flexibility in terms of performance, platform compatibility, and user preference. Let's dive into how to configure CMake to use each of these generators and the scenarios in which they are most useful.

3.2.2 Running cmake with the Ninja Generator

Ninja is a small, fast build system with a focus on performance. It is often used in environments where speed is important and works especially well for large projects. Ninja operates by processing small build files that contain just enough information to trigger the necessary build steps, making it significantly faster than traditional build systems in many cases.

1. Why Choose Ninja?
    - Fast: Ninja is known for its speed in incremental builds. It minimizes the work done by only rebuilding parts of the project that have changed.
    - Minimalistic: Unlike other build systems that generate large build files, Ninja generates concise and efficient build files, resulting in reduced I/O and faster execution.
    - Cross-Platform: Ninja can be used on multiple platforms (Linux, macOS, and Windows), making it a great choice for cross-platform projects.
2. Using Ninja with CMake

    To use Ninja as your build system, you need to first install Ninja (if it's not already installed on your system). Once installed, you can specify Ninja as the generator when running the cmake command:

    ```
    cmake -G Ninja <path-to-source>
    ```

This command tells CMake to generate Ninja build files in the build directory, based on the source code in the specified directory. After configuration, you can then use Ninja to build the project:

```
ninja
```

3. Example: Using Ninja with a Project

   Consider a project located in ~/projects/my_app. To configure and build this project using Ninja, you would run the following commands:

```
mkdir build
cd build
cmake -G Ninja ../my_app
ninja
```

   This sequence of commands will configure the project, generate the Ninja build files, and then execute the build process.

3.2.3 Running cmake with the Makefile Generator

Make is a well-known build system in Unix-like environments. It uses Makefiles to determine how to build and link the project. The Makefile generator is the most common choice for Linux and macOS systems, as make is a standard tool in these environments.

1. Why Choose Makefile?

   - Standard: Make is widely supported on Unix-based systems and is the default build system for many projects.

- Flexibility: Makefiles are extremely flexible and customizable, offering extensive control over the build process.
- Toolchain Integration: Make integrates well with a variety of compilers and build tools, making it easy to work with complex toolchains and custom configurations.

2. Using Makefile with CMake

   To use make as the generator, specify the Unix Makefiles generator in the cmake command:

```
cmake -G "Unix Makefiles" <path-to-source>
```

   After CMake generates the necessary Makefiles, you can build the project using the make command:

```
make
```

3. Example: Using Makefiles with a Project

   Let's say you have a project at ~/projects/my_app. To configure and build this project using Makefiles, you would run:

```
mkdir build
cd build
cmake -G "Unix Makefiles" ../my_app
make
```

   This will configure the project, generate the Makefile, and compile the project using the make tool.

3.2.4 Running cmake with the Visual Studio Generator

1. Visual Studio is one of the most widely used IDEs for C++ development, particularly on Windows. CMake provides generators for multiple versions of Visual Studio, which allow you to create Visual Studio project files (e.g., .sln, .vcxproj) for your project. This is particularly useful for developers who prefer the Visual Studio environment for building and debugging C++ projects.
2. Why Choose Visual Studio?
   - IDE Support: Visual Studio is a powerful IDE that provides an extensive set of features such as debugging, profiling, and an intuitive graphical interface for project management.
   - Native Windows Development: Visual Studio is the standard development environment for C++ on Windows, making it ideal for targeting Windows-specific APIs and libraries.
   - Advanced Features: Visual Studio provides features like IntelliSense, a visual debugger, and integrated testing tools that improve productivity.
3. Using Visual Studio with CMake

   To generate Visual Studio project files with CMake, specify the appropriate version of Visual Studio as the generator. For example, to generate project files for Visual Studio 2019, you would run:

   ```
   cmake -G "Visual Studio 16 2019" <path-to-source>
   ```

   This will create .sln files that can be opened directly in Visual Studio. After the project is generated, you can open the .sln file in Visual Studio and build the project from the IDE.

4. Example: Using Visual Studio with a Project

   Let's say you have a project located in C:\projects\my_app. To configure and generate Visual Studio project files, use:

   ```
   mkdir build
   cd build
   cmake -G "Visual Studio 16 2019" C:\projects\my_app
   ```

   This will generate a Visual Studio solution file (e.g., my_app.sln) in the build directory. You can then open this .sln file in Visual Studio and build the project.

3.2.5 Choosing the Right Generator for Your Project

Choosing the correct generator depends on your specific requirements and development environment. Here are some guidelines to help you decide which generator to use:

- Ninja: Ideal for fast, efficient builds, especially in large projects. It's a good choice if you prioritize build speed and want a cross-platform solution.
- Makefile: Best for traditional Unix-based systems (Linux, macOS). It's the default for many open-source projects and is widely supported.
- Visual Studio: Perfect for Windows-based development using the Visual Studio IDE. If you need to work in a Microsoft-centric development environment, generating Visual Studio project files is the way to go.

Additionally, consider the complexity of your project. For simple, small projects, any generator will work fine. For large projects with complex dependencies or

custom build steps, you might prefer Ninja or Makefile due to their simplicity and speed. Visual Studio is best suited for projects that benefit from deep IDE integration, such as debugging or visual design tools.

3.2.6 Conclusion

Understanding how to run cmake with different generators—Ninja, Makefile, and Visual Studio—is essential for tailoring the build process to your development environment. Each generator has its advantages and is best suited for different use cases:

- Ninja offers fast and efficient builds, making it ideal for large projects.
- Makefile is the traditional choice for Unix-based systems and offers flexibility and control over the build process.
- Visual Studio is a powerful IDE for Windows development and integrates seamlessly with CMake for generating project files.

By selecting the appropriate generator, you can streamline the build process and integrate CMake more effectively into your existing workflow. Whether you are developing on a Linux, macOS, or Windows platform, CMake provides the tools you need to manage and build your C++ projects efficiently.

3.3 Managing Source Files and Output Executables

When working with CMake, understanding how to effectively manage source files and define output executables is a critical part of setting up your build process. CMake simplifies this task by providing clear, intuitive mechanisms to specify the organization of source files and control where the final executables and libraries are placed. This section will explain how to manage source files and control output executables in a CMake project, covering basic commands like add_executable(), add_library(), and the use of source groups.

3.3.1 Overview of Source Files in CMake

Source files are the building blocks of your project—they contain the code that will be compiled into the final executable or library. CMake offers a variety of ways to manage these files, from defining them explicitly to leveraging wildcard patterns and automatic file discovery. Whether your project has a simple structure or a more complex, multi-directory setup, CMake provides tools to help organize and include source files efficiently.

CMake supports different types of source files for various build targets:

- C++ Source Files (.cpp, .cc, .cxx)
- Header Files (.h, .hpp)
- Resource Files (e.g., .rc on Windows, .qml files in Qt-based projects)

The key here is to specify the source files correctly to ensure that they are compiled into the appropriate output.

3.3.2 Defining Source Files for Executables

The most fundamental operation in CMake is defining the source files that will be compiled into an executable. This is achieved using the add_executable() command. The general syntax is:

```
add_executable(<name> <source1> <source2> ... <sourceN>)
```

- <name>: The name of the executable you want to create.
- <source1>, <source2>, ... <sourceN>: The list of source files (e.g., .cpp files) that will be compiled to create the executable.

1. Example: Simple Executable Definition

   Consider a project where you have the following C++ files:

   ```
   main.cpp
   foo.cpp
   foo.h
   ```

   To define an executable named my_app using main.cpp and foo.cpp, you would write:

   ```
   add_executable(my_app main.cpp foo.cpp)
   ```

   This tells CMake to create an executable named my_app from the source files main.cpp and foo.cpp. After running the build process, the output will be an executable file named my_app (or my_app.exe on Windows).

2. Handling Header Files

   Header files are typically included in source files and don't require direct specification in add_executable(). However, it's a good practice to group them into logical directories for better organization. CMake automatically handles headers as long as they are included in the relevant source files.

3.3.3 Organizing Source Files Using file() and aux_source_directory()

For larger projects with multiple directories, you might want to automate the process of collecting source files. CMake provides the file() and aux_source_directory() commands to facilitate this.

1. Using aux_source_directory()

   The aux_source_directory() command scans a directory and adds all source files within that directory to a variable. It can be particularly useful for large projects with many source files located in subdirectories.

   Example:

   ```
   aux_source_directory(src SOURCES)
   add_executable(my_app ${SOURCES})
   ```

   This command will search the src directory for all .cpp files and add them to the SOURCES variable. The executable my_app will then be built from all the source files found in the src directory.

2. Using file(GLOB ...)

   Alternatively, you can use file(GLOB ...) to collect all files matching a specific pattern, such as all .cpp files in a given directory:

```
file(GLOB SOURCES "src/*.cpp")
add_executable(my_app ${SOURCES})
```

The file(GLOB ...) command is useful when you have files in a directory that follow a specific naming pattern. However, it's generally recommended to avoid overusing this command, as it can make the build system harder to maintain when files are added or removed.

3.3.4 Defining Output Executables and Directories

Once you have specified the source files for your project, you need to define where the resulting executable should be placed. CMake provides various commands and options to control this.

1. Setting the Output Directory for Executables

   By default, CMake places the output executables in the CMAKE_BINARY_DIR directory (which is usually the build directory). However, you can customize the location of the executable with the RUNTIME_OUTPUT_DIRECTORY property:

   ```
   set_target_properties(my_app PROPERTIES RUNTIME_OUTPUT_DIRECTORY
   ↪  ${CMAKE_BINARY_DIR}/bin)
   ```

   This sets the output directory for the executable my_app to a subdirectory called bin inside the build directory.

2. Example: Organizing Executables in a bin Directory

   For example, if you want all your executables to be placed in a bin directory within the build folder, you can add the following to your CMakeLists.txt:

```
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

add_executable(my_app main.cpp foo.cpp)
add_executable(my_app2 main2.cpp bar.cpp)
```

This will result in my_app and my_app2 being placed in the bin directory inside your build folder, making it easier to manage the outputs of your project.

3.3.5 Managing Multiple Executables and Targets

Many projects may require the creation of multiple executables from different sets of source files. In CMake, each executable or library you define is treated as a target, and you can manage them separately.

1. Adding Multiple Executables

   To add another executable, simply call add_executable() with a different target name and source files:

   ```
   add_executable(my_app main.cpp foo.cpp)
   add_executable(my_app2 main2.cpp bar.cpp)
   ```

   Each target is built independently, and CMake will ensure that all source files are compiled correctly and linked into the appropriate executables.

2. Organizing Executables into Directories

   If you have a large number of executables, it can be helpful to organize them into subdirectories. You can use add_subdirectory() to create logical groupings in your project:

```
my_project/
    CMakeLists.txt
    bin/
        CMakeLists.txt
        my_app.cpp
        my_app2.cpp
    lib/
        libfoo.cpp
```

In the top-level CMakeLists.txt, you could add:

```
add_subdirectory(bin)
```

In the bin/CMakeLists.txt, you would then define the executables:

```
add_executable(my_app my_app.cpp)
add_executable(my_app2 my_app2.cpp)
```

This modular structure helps keep the build process organized, especially when dealing with large projects.

3.3.6 Defining Libraries for Reusability

In addition to executables, CMake also makes it easy to define libraries. Libraries are reusable collections of code that can be linked with other projects or executables.

1. Creating a Static Library

   To create a static library, use the add_library() command with the STATIC option:

```
add_library(my_lib STATIC foo.cpp bar.cpp)
```

This creates a static library named my_lib.a (on Unix-like systems) or my_lib.lib (on Windows), which can be linked to other executables or libraries.

2. Creating a Shared Library

   To create a shared library (dynamic link library, DLL, or .so), use the SHARED option with add_library():

   ```
   add_library(my_lib SHARED foo.cpp bar.cpp)
   ```

   This creates a shared library that can be dynamically loaded by executables at runtime.

3. Linking Libraries to Executables

   Once a library is created, it can be linked to an executable or another library using the target_link_libraries() command:

   ```
   target_link_libraries(my_app my_lib)
   ```

   This links the static or shared library my_lib to the executable my_app.

3.3.7 Conclusion

Managing source files and output executables in CMake is an essential part of setting up a successful build system. By using commands like add_executable(), add_library(), and target_link_libraries(), you can define which source files to compile, where to place your output executables, and how to manage libraries

within your project. Additionally, CMake offers powerful tools like file() and aux_source_directory() to help automate the discovery and organization of source files, especially in larger projects. By mastering these techniques, you'll be able to build robust, organized, and efficient CMake projects that scale with the complexity of your C++ codebase.

3.4 Running cmake --build and cmake --install

Once you've configured your CMake project and generated the appropriate build files, the next step is to actually build and install your project. These tasks are essential to the process of turning your source code into a usable application or library. CMake simplifies the build and install process with the cmake --build and cmake --install commands, respectively.

In this section, we'll delve into how to run these commands, what they do, and how they fit into the broader CMake workflow.

3.4.1 Running cmake --build

The cmake --build command is used to compile and link the project, essentially performing the build step. This command simplifies the process by abstracting away the complexities of interacting directly with the build system (such as make, ninja, or MSBuild), and it ensures consistency across different platforms and environments.

1. Overview of cmake --build

   After running the initial cmake command to configure your project, you'll typically be left with build files (such as Makefiles, Ninja build files, or Visual Studio solution files). The cmake --build command is then used to invoke the appropriate tool (like make or ninja) to perform the actual compilation and linking process.

   The basic syntax for cmake --build is:

```
cmake --build <build-directory> [options]
```

- <build-directory>: This is the path to the build directory where CMake has generated the build files (e.g., build/).
- [options]: These are optional flags that you can pass to modify the build process (such as building a specific target or specifying the number of parallel jobs).

2. Running cmake --build Without Arguments

   In its simplest form, you can run cmake --build without any additional arguments to build the default target (typically the project's primary executable or library):

   ```
   cmake --build build/
   ```

   This command will invoke the correct build system for your platform (e.g., make, ninja, or MSBuild) and will build the default target. If you are in the build directory, you can simply run:

   ```
   cmake --build .
   ```

   CMake will handle determining the correct tool and invoking it with the necessary arguments to perform the build.

3. Specifying Build Targets

   You can specify which target you want to build by using the --target option. This is useful if you want to build a specific target, such as a particular executable or library, rather than the default target.

   For example, to build a specific executable (e.g., my_app), you would run:

```
cmake --build build/ --target my_app
```

This command will only build the my_app target, skipping the other targets in the project. This is particularly useful in larger projects with multiple components, as it allows you to build only the necessary parts of the project.

4. Building with Parallel Jobs

   To speed up the build process, you can specify the number of parallel jobs to use with the -- -j option (this is passed to the underlying build tool, like make or ninja). This can dramatically reduce build times, especially in large projects with many files to compile.

   For example, to use 4 parallel jobs, you can run:

   ```
   cmake --build build/ -- -j4
   ```

   This tells the build system to use 4 processors to compile the project in parallel.

5. Cleaning the Build

   If you want to clean up intermediate build files, CMake provides the --target clean option. This is equivalent to running make clean or the relevant clean command for the build tool you're using.

   ```
   cmake --build build/ --target clean
   ```

   This removes the object files and other intermediate files, but leaves the CMake-generated files (such as Makefiles or Visual Studio project files) intact.

3.4.2 Running cmake --install

After building the project, the next step is typically to install it. The cmake --install command allows you to copy the built files (executables, libraries, headers, etc.) to their final locations on the system, making them ready for use or distribution. This step is crucial when you want to deploy your project or make it available for other software to link to.

1. Overview of cmake --install

   The cmake --install command copies the built project to a specific installation directory. The installation process uses paths defined by CMake variables such as CMAKE_INSTALL_PREFIX, which specifies where the files will be installed. If you have not customized this variable, the default installation path is /usr/local on Unix-like systems (Linux/macOS) and C:\Program Files on Windows.

   The basic syntax for cmake --install is:

   ```
   cmake --install <build-directory> [options]
   ```

   - <build-directory>: The path to the build directory where the project was built.
   - [options]: Optional flags to customize the installation process.

2. Specifying the Installation Directory

   You can specify a custom installation directory by setting the CMAKE_INSTALL_PREFIX variable. This can be done either in the CMakeLists.txt file or via the command line.

   To set the installation directory during configuration (before building), use the following command:

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install/directory <path-to-source>
```

Alternatively, you can specify the installation directory during the cmake --install command itself:

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

3. Running cmake --install

   Once the build process is complete, you can install the project with:

   ```
   cmake --install build/
   ```

   This will copy the necessary files (such as executables, libraries, and headers) from the build directory to the installation directory defined by CMAKE_INSTALL_PREFIX.

4. Installing Specific Targets

   If your project has multiple installable targets (e.g., both an executable and a library), you can specify which target to install by using the --target option:

   ```
   cmake --install build/ --target my_app
   ```

   This installs only the my_app executable, not any other components.

5. Installing with Multiple Configurations (For Multi-Configuration Generators)

   When working with multi-configuration generators like Visual Studio or Xcode, you can specify which build configuration (such as Debug or Release) to install using the --config option:

```
cmake --install build/ --config Release
```

This command installs the project built in the Release configuration.

3.4.3 Customizing the Installation Process

CMake allows you to define custom installation rules for specific files or directories using the install() command within your CMakeLists.txt. This command provides flexibility in deciding what gets installed and where.

For example, if you want to install an executable and a library, you can define the following in your CMakeLists.txt:

```
install(TARGETS my_app DESTINATION bin)
install(TARGETS my_lib DESTINATION lib)
install(DIRECTORY include/ DESTINATION include)
```

This would install:

- my_app to the bin directory.
- my_lib to the lib directory.
- The contents of the include/ directory to the include directory.

By customizing the install() commands, you can control the installation of executables, libraries, headers, and other project files to the appropriate locations on the system.

3.4.4 Conclusion

The cmake --build and cmake --install commands are key components of the CMake build process.

- cmake --build compiles and links the project, invoking the underlying build system (such as make, ninja, or MSBuild) to produce the final executables, libraries, or other targets.
- cmake --install copies the built project files to a specified installation directory, making them ready for use or distribution.

By understanding how these commands work and how to configure them for your needs, you can effectively manage the build and installation of your CMake-based projects. Whether you're working on a small application or a large library, CMake provides a flexible and consistent way to build and install your software across multiple platforms.

3.5 Controlling Build Options via CMAKE_BUILD_TYPE

CMake provides a powerful mechanism for controlling the behavior of the build process through various configuration options, one of the most crucial being CMAKE_BUILD_TYPE. This variable determines the type of build configuration you want to generate, such as a debug build, release build, or a custom build type. This section will explore how to effectively control build options using CMAKE_BUILD_TYPE, the impact of different build types, and how to customize and fine-tune the configuration of your builds.

3.5.1 Overview of CMAKE_BUILD_TYPE

The CMAKE_BUILD_TYPE variable is one of the primary ways to control how your project is built. It specifies the type of build configuration that CMake should use when generating the build system. Typically, this is a setting you configure before you generate your build files, and it can affect several important aspects of the build, such as optimization levels, debugging information, and compiler flags.

The CMAKE_BUILD_TYPE variable is primarily used with single-configuration generators, such as Makefiles, Ninja, and Unix-style build systems. For multi-configuration generators like Visual Studio or Xcode, the build type is typically specified as part of the build process rather than the configuration process, and CMAKE_BUILD_TYPE does not have the same effect.

Typical Build Types

CMake supports several common build types, each of which comes with different optimizations and debugging settings. The most commonly used build types are:

- Debug: This build type generates debugging information and disables optimizations to help with debugging. It's suitable when you need to inspect your code in a debugger or need detailed information about your program's state during execution.
  - Compiler flags: -g (GCC/Clang), /Zi (MSVC)
  - No optimizations enabled
- Release: This build type is optimized for performance. It enables compiler optimizations and disables debugging information. This is the default build type for production-ready code.
  - Compiler flags: -O2 or -O3 (GCC/Clang), /O2 (MSVC)
  - No debugging symbols included
- RelWithDebInfo: This build type strikes a balance between performance and debugging. It enables optimizations but also includes debugging information. It's a good choice if you need performance but still want to debug the program if necessary.
  - Compiler flags: -O2 -g (GCC/Clang), /O2 /Zi (MSVC)
- MinSizeRel: This build type is focused on minimizing the size of the compiled binary while still providing optimizations. It's often used for embedded systems or other scenarios where small binaries are essential.
  - Compiler flags: -Os (GCC/Clang), /O1 (MSVC)

3.5.2 Setting CMAKE_BUILD_TYPE

To set the build type in CMake, you can define CMAKE_BUILD_TYPE during the configuration phase. This is typically done from the command line when you run the cmake command to generate the build files.

1. Basic Example

   For example, to configure the build for a Debug build type, you can run:

   ```
   cmake -DCMAKE_BUILD_TYPE=Debug /path/to/source
   ```

   This command tells CMake to configure the project for debugging, meaning it will generate the appropriate build system with debugging flags and without optimizations.

   Similarly, to generate a Release build type, you would use:

   ```
   cmake -DCMAKE_BUILD_TYPE=Release /path/to/source
   ```

   This will configure the project for release, ensuring that compiler optimizations are enabled and debugging symbols are removed.

2. Other Build Types

   You can set other build types by using CMAKE_BUILD_TYPE with the following commands:

   ```
   cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo /path/to/source
   cmake -DCMAKE_BUILD_TYPE=MinSizeRel /path/to/source
   ```

   Each of these will configure CMake to use the corresponding set of compiler flags and options.

3.5.3 Effect of CMAKE_BUILD_TYPE on the Build Process

Setting the CMAKE_BUILD_TYPE variable not only determines the compiler flags for optimization and debugging but also influences several other aspects of the build:

- Compiler Options: The compiler flags vary between build types, such as optimizations (-O3), debugging symbols (-g), and additional runtime checks (e.g., -fsanitize=address for sanitizers).
- Linker Flags: Depending on the build type, CMake may add different linker options. For example, a release build might include additional flags to strip debugging information or reduce the size of the final executable.
- Debugging Symbols: In the Debug build type, CMake ensures that debugging symbols are included, which are necessary for debugging tools like gdb or lldb. In contrast, in Release builds, debugging symbols are typically omitted to improve performance and reduce the size of the binary.
- Optimization: Release builds enable higher levels of optimization, leading to faster execution times, while debug builds avoid optimization to make stepping through code easier in a debugger. RelWithDebInfo provides a middle ground by optimizing the code while still including some debug information.
- Conditional Code: Some code in the project might only be included in specific build types. For example, CMake allows conditional inclusion of certain code depending on the build type, using constructs like if(CMAKE_BUILD_TYPE MATCHES ”Debug”).

3.5.4 Multi-Configuration Generators and CMAKE_BUILD_TYPE

As mentioned earlier, CMAKE_BUILD_TYPE is mainly used with single-configuration generators like Makefiles or Ninja. For multi-configuration generators such as Visual Studio or Xcode, the build type is usually determined during the actual build process, not during the configuration phase.

For example, with Visual Studio, you can select the build configuration (such as Debug, Release, or others) when opening the project in the Visual Studio IDE. You don't need to specify the build type during the configuration step with Visual Studio because the IDE handles it dynamically.

Example for Visual Studio

When generating build files for Visual Studio, you do not need to specify the build type at the configuration step:

```
cmake -G "Visual Studio 16 2019" /path/to/source
```

After running this command, you can open the generated .sln file in Visual Studio and select the build configuration (Debug, Release, etc.) from the IDE's build settings.

Similarly, with Xcode, you can specify the configuration directly within Xcode once the project has been generated.

3.5.5 Customizing Build Types

CMake also allows you to customize build types by defining your own custom configurations. For example, you may want to create a special configuration that combines optimizations and additional warnings or debugging checks for a particular use case.

To do this, you can define custom build types by adding to the CMAKE_CXX_FLAGS variable in your CMakeLists.txt. For example:

```
set(CMAKE_CXX_FLAGS_CUSTOM "-O2 -Wall -DDEBUG")
```

Then, when running the configuration, you can specify this custom type:

```
set(CMAKE_CXX_FLAGS_CUSTOM "-O2 -Wall -DDEBUG")
```

This approach provides flexibility, especially in complex projects or when dealing with specialized requirements like performance profiling or testing.

3.5.6 Advanced Control of Build Options

CMake provides additional mechanisms to control build behavior beyond just CMAKE_BUILD_TYPE. Here are some of the most commonly used variables that work alongside CMAKE_BUILD_TYPE:

- CMAKE_CXX_FLAGS_DEBUG: This variable allows you to add or modify flags specifically for the Debug build type.
- CMAKE_CXX_FLAGS_RELEASE: This variable lets you adjust flags for the Release build type.
- CMAKE_CXX_FLAGS_RELWITHDEBINFO: Adjusts flags for the RelWithDebInfo build type.
- CMAKE_CXX_FLAGS_MINSIZEREL: Used to modify flags for the MinSizeRel build type.

For example, you might want to modify the compiler flags to add more strict debugging or security checks for the Debug build:

```
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG}
↪  -fsanitize=address")
```

This will enable AddressSanitizer in the debug configuration, helping to catch memory errors.

3.5.7 Conclusion

CMAKE_BUILD_TYPE is a powerful tool in CMake for controlling the type of build you generate, whether you're working on a development/debugging phase or preparing for a production release. By setting CMAKE_BUILD_TYPE, you can adjust compiler and linker flags, optimization levels, and debugging information to match the needs of your project.

For single-configuration generators, setting the CMAKE_BUILD_TYPE is crucial for customizing the build behavior. For multi-configuration generators like Visual Studio and Xcode, this step is less critical, as these IDEs let you select the build configuration during the actual build process. By combining CMAKE_BUILD_TYPE with other CMake features like custom flags and multi-stage builds, you can fine-tune your project's build process and make it more efficient and easier to maintain.

Chapter 4

Working with Libraries in CMake (Static & Shared)

4.1 Difference Between Static and Shared Libraries

In CMake, when building C++ projects, libraries play an essential role in modularizing the code, making it reusable, and simplifying the build process. These libraries can be either static libraries or shared libraries, and understanding the differences between them is key to managing dependencies and ensuring the correct build setup for your project. This section dives into the key differences between static and shared libraries, their use cases, and how they affect the build process.

4.1.1 Definition

- Static Libraries: A static library, also known as an archive in C++, is a collection of object files that are bundled into a single file. During the build process, the linker copies the object code from the static library directly into the executable. As a result, the executable does not need the static library at runtime because all the necessary code is included within the executable itself.
- Shared Libraries: Also known as dynamic libraries or DLLs (Dynamic Link Libraries) on Windows, shared libraries contain code that is linked at runtime rather than during the compile time. The executable or other libraries that use the shared library will dynamically load it during execution. This means the executable depends on the shared library being present in the system at runtime to function correctly.

4.1.2 Build Process

- Static Libraries:
  - The static library is compiled from the source code into object files. These object files are then bundled together into a single library file, typically with a .a extension on Unix-like systems and .lib on Windows.
  - During linking, the entire library is copied into the executable. This process results in larger executable files but does not require the library to be available during runtime.
  - Static libraries are commonly used for applications that require all dependencies to be packaged into a single executable.
- Shared Libraries:

- A shared library is compiled and linked in a way that only the symbol information (function names, data structures, etc.) is placed in the executable. The actual code for these functions is placed in the shared library itself.
- The executable is not directly linked to the code but to the dynamic library file. The linking to the library happens at runtime when the program is executed.
- Shared libraries are typically used in systems where multiple applications or components can benefit from the same code base, reducing the size of the executables and enabling the sharing of common functionality.

4.1.3 Key Differences Between Static and Shared Libraries

Aspect	Static Libraries	Shared Libraries
Linking Time	Linked during compile time.	Linked during runtime.
Dependency at Runtime	No dependency at runtime (code is embedded).	Requires the shared library to be available at runtime.
File Size	Larger executable size (includes the library code).	Smaller executables (library code is external).
Performance	Faster startup time (no need to load libraries).	Slightly slower startup time (requires loading the library at runtime).
Memory Usage	Higher memory usage, as each process has its own copy of the library code.	Lower memory usage, as the same shared library is used across multiple processes.

Aspect	Static Libraries	Shared Libraries
Updates/ Versioning	Requires recompilation of the executable if the library changes.	Easy to update; only the shared library file needs to be updated.
Platform Dependency	Platform-specific (a static library is specific to a platform).	Also platform-specific, but can be shared across different applications.
Portability	Less portable; each executable contains its own copy of the library.	More portable; can be shared across multiple systems.

4.1.4 Advantages and Disadvantages

1. Static Libraries

   Advantages:

   - Standalone Executable: The executable is independent and contains all the necessary code. This makes it easier to distribute because there are no external dependencies.
   - Faster Execution: Since the linking is done at compile time, the application may have slightly faster execution times compared to shared libraries, which require runtime linking.
   - No Versioning Issues: With static libraries, the version of the library used during compilation is always the one included in the executable, avoiding potential issues with incompatible versions of shared libraries.

   Disadvantages:

   - Larger Executables: The final executable file size is larger because it includes all the code from the static libraries.

- No Shared Memory: Each running instance of the application gets its own copy of the static library code, leading to higher memory usage when the program is running.
- Updates Require Rebuilding: If the library code is updated, all executables that use it must be recompiled and redistributed.

2. Shared Libraries

   Advantages:

   - Smaller Executables: The executable file is smaller because it doesn't contain the code from the shared libraries; it only contains references to them.
   - Shared Memory: Multiple running instances of a program can share the same loaded version of a shared library, which reduces overall memory usage.
   - Easier Updates: When a shared library is updated, all executables that depend on it will benefit from the update immediately without needing recompilation.

   Disadvantages:

   - Dependency Management: Shared libraries must be available at runtime. This can lead to dependency hell when different applications require different versions of the same library.
   - Slower Startup: There is a slight overhead when loading shared libraries at runtime.
   - Potential Compatibility Issues: If an application expects a particular version of a shared library, updating the library may cause compatibility issues unless proper versioning is handled.

4.1.5 Use Cases

- Static Libraries:
  - Ideal for standalone applications that do not need to rely on external libraries at runtime.
  - Useful in embedded systems, where minimizing external dependencies is crucial.
  - Used when you want to distribute a single file containing all necessary components, such as in a proprietary application or for performance-sensitive applications.
- Shared Libraries:
  - Perfect for applications that share common functionality. This allows multiple programs to link to the same library, reducing redundancy and making updates easier.
  - Common in large-scale enterprise applications where different components need to use the same underlying functionality.
  - Beneficial in systems where memory usage and disk space need to be minimized, as shared libraries can be loaded once and used by multiple processes.

4.1.6 CMake Configuration for Static and Shared Libraries

In CMake, you can specify whether to create a static or shared library using the add_library() command. Here's how you would do that:

- Static Library:

```
add_library(my_library STATIC src/my_library.cpp)
```

- Shared Library:

```
add_library(my_library SHARED src/my_library.cpp)
```

You can also specify different build types (Release, Debug) and link libraries conditionally based on the type of build being performed.

4.1.7 Conclusion

Choosing between static and shared libraries depends on the specific requirements of your project. Static libraries are useful for reducing external dependencies, making the build simpler, and ensuring that the application is completely self-contained. Shared libraries, on the other hand, are more efficient in terms of memory and disk space, especially when the same code is used across multiple applications or processes.

In CMake, it is straightforward to configure either type of library, but it is important to understand the implications of your choice to make informed decisions about the architecture of your project.

4.2 Creating a Static Library (add_library(MyLib STATIC))

In this section, we will explore how to create a static library in CMake. A static library is a collection of object files bundled together into a single file. This file can be linked into executables at compile-time, resulting in larger executables but ensuring that no external dependencies are required at runtime.

Creating a static library in CMake is simple, but understanding the process and configuration is crucial to ensure it integrates seamlessly into your build system. This section will cover the add_library() command in CMake, the steps to create a static library, and the typical workflow involved in using static libraries within a C++ project.

4.2.1 Understanding Static Libraries in CMake

A static library is a collection of precompiled object files (typically .o or .obj files) packed into a single archive file (usually .a on Unix-like systems or .lib on Windows). The code in a static library is copied into the final executable at compile time, making the executable self-contained.

In CMake, the add_library() command is used to create libraries, and the STATIC keyword specifies that the library will be a static library.

4.2.2 Basic Syntax of add_library()

The basic syntax for creating a static library in CMake is:

```
add_library(<library_name> STATIC <source_files>)
```

Where:

- <library_name>: The name you want to give to your library (e.g., MyLib).
- STATIC: Specifies that the library is a static library.
- <source_files>: The list of source files to include in the library, such as .cpp files.

For example, to create a static library MyLib from the source files my_lib.cpp and my_lib_utils.cpp, you would write:

```
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)
```

4.2.3 Detailed Example: Creating a Static Library

Let's walk through an example to create a static library MyLib in a CMake project:

1. Create the directory structure:

```
MyProject/
    CMakeLists.txt
    src/
        CMakeLists.txt
        my_lib.cpp
        my_lib_utils.cpp
    main.cpp
```

2. Top-level CMakeLists.txt: At the top level, you'll specify the minimum required version of CMake, the project name, and include the src directory for the actual library creation.

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

add_subdirectory(src)  # Include the 'src' directory to build the library
```

3. src/CMakeLists.txt: In the src/CMakeLists.txt, you create the static library from the source files my_lib.cpp and my_lib_utils.cpp:

```
# Create a static library 'MyLib' from the source files
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)

# Specify the include directories if needed
target_include_directories(MyLib PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

4. src/my_lib.cpp: This is the main source file for the static library.

```
// my_lib.cpp
#include "my_lib.h"

void MyLib::doSomething() {
    // Implement some functionality here
}
```

5. src/my_lib_utils.cpp: Another source file that adds utility functions to the library.

```
// my_lib_utils.cpp
#include "my_lib_utils.h"

int MyLibUtils::add(int a, int b) {
    return a + b;
}
```

6. main.cpp: The main executable that links to the static library MyLib.

```
#include <iostream>
#include "my_lib.h"
#include "my_lib_utils.h"

int main() {
    MyLib lib;
    lib.doSomething();

    MyLibUtils utils;
    std::cout << "Sum: " << utils.add(3, 4) << std::endl;

    return 0;
}
```

7. Building the Project: After configuring CMake, you can build the project:

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

This process will compile the static library and link it to the main executable.

4.2.4 Linking the Static Library

Once the static library is created, you need to link it to the executable that depends on it. In CMake, this is done using the target_link_libraries() command.

For example, in the CMakeLists.txt file for the main project:

```
add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE MyLib)  # Link the static library
```

This will ensure that MyApp links against the static library MyLib during the linking phase, incorporating the object files from MyLib into the final executable.

4.2.5 Using target_include_directories()

When creating a static library, you often want to make sure that the header files used by the library are accessible to other parts of the project. The target_include_directories() command is used to specify the include directories for the library.

In the src/CMakeLists.txt file, you can add:

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

- PUBLIC: This keyword makes the include directory available both to the static library and to any target that links against the library (like MyApp).
- PRIVATE: If the include directory is only needed internally by the static library, you can use PRIVATE instead.

4.2.6 Handling Dependencies in Static Libraries

If your static library has dependencies on other libraries (e.g., third-party libraries), you need to link those libraries within your CMake configuration.

For example, if MyLib depends on another library OtherLib, you would modify the CMake configuration as follows:

```
# Create a static library 'MyLib'
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)

# Link the static library 'OtherLib' to 'MyLib'
target_link_libraries(MyLib PRIVATE OtherLib)
```

This ensures that when MyLib is linked to an executable or another library, the OtherLib will also be included as a dependency.

4.2.7 Using CMake Variables for Source Files

Instead of hardcoding the list of source files, you can use CMake variables to collect them, especially if you have a large number of files or want to avoid manual updates.

For example:

```
set(SOURCES
    my_lib.cpp
    my_lib_utils.cpp
)

add_library(MyLib STATIC ${SOURCES})
```

This approach is useful when organizing larger projects with multiple source files.

4.2.8 Installing the Static Library

If you want to install the static library for use outside the current project, you can use the install() command. This is often done to create a CMake package for others to use.

For example:

```
install(TARGETS MyLib DESTINATION lib)
install(FILES my_lib.h my_lib_utils.h DESTINATION include)
```

This will install the MyLib static library to the lib directory and the headers to the include directory.

4.2.9 Advantages of Static Libraries

- Self-contained: The executable does not require external dependencies at runtime, making distribution easier.
- Performance: Static linking can result in faster program startup time because all code is embedded into the executable.
- No Versioning Issues: The specific version of the library used during compilation is included in the executable, avoiding potential issues caused by mismatched versions at runtime.

4.2.10 Disadvantages of Static Libraries

- Larger Executables: The executable becomes larger since it includes all the necessary object files from the static library.
- No Shared Memory: Each running instance of the program gets its own copy of the library code, leading to higher memory consumption.
- Rebuild Required: If the static library is updated, you need to rebuild and redistribute all executables that depend on it.

4.2.11 Conclusion

Creating a static library in CMake is a straightforward process that involves using the add_library() command with the STATIC keyword. Static libraries are a great choice when you want self-contained executables and don't mind the larger file sizes. They are particularly useful for smaller applications or when you want to avoid runtime dependencies. By following the steps in this section, you'll be able to create, link, and manage static libraries effectively in your C++ projects with CMake.

4.3 Creating a Shared Library (add_library(MyLib SHARED))

In this section, we will explore the process of creating a shared library in CMake, which is also referred to as a dynamic library. Shared libraries differ from static libraries in that they are linked at runtime, rather than being statically included within the executable. This allows for smaller executables, easier updates, and shared code across different applications, but it also introduces some challenges, such as dependency management at runtime.

This section will guide you through the creation of shared libraries using CMake, explain the key differences between static and shared libraries, and provide best practices for managing and linking shared libraries in CMake projects.

4.3.1 Understanding Shared Libraries in CMake

A shared library (also called a dynamic library) is a collection of object files that are linked at runtime. When an executable or another shared library is linked to a shared library, the actual code is not included in the executable. Instead, a reference is created, and the code from the shared library is loaded when the application is run.

- On Unix-like systems (Linux, macOS), shared libraries typically have .so (Shared Object) extensions.
- On Windows, they are usually referred to as .dll (Dynamic Link Libraries).

The key advantage of using shared libraries is that they can be shared between multiple programs or processes, which helps reduce memory usage and the overall

size of executables. Additionally, shared libraries can be updated independently without requiring the applications that use them to be recompiled, as long as the interface remains compatible.

In CMake, shared libraries are created using the add_library() command with the SHARED keyword.

4.3.2 Basic Syntax of add_library() for Shared Libraries

The basic syntax to create a shared library in CMake is:

```
add_library(<library_name> SHARED <source_files>)
```

Where:

- <library_name>: The name of the shared library you want to create (e.g., MyLib).
- SHARED: Specifies that the library will be a shared library.
- <source_files>: A list of source files, typically .cpp files, to be compiled into the library.

For example, to create a shared library MyLib from the source files my_lib.cpp and my_lib_utils.cpp, you would use:

```
add_library(MyLib SHARED my_lib.cpp my_lib_utils.cpp)
```

4.3.3 Detailed Example: Creating a Shared Library

Let's walk through a detailed example of creating a shared library MyLib in a CMake-based project. We'll also link this shared library to an executable and explore the necessary configurations.

1. Create the directory structure:

```
MyProject/
    CMakeLists.txt
    src/
        CMakeLists.txt
        my_lib.cpp
        my_lib_utils.cpp
    main.cpp
```

2. Top-level CMakeLists.txt: At the top level, specify the minimum required version of CMake, the project name, and include the src directory to build the shared library.

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

add_subdirectory(src)  # Include the 'src' directory to build the library
```

3. src/CMakeLists.txt: In the src/CMakeLists.txt, you create the shared library from the source files my_lib.cpp and my_lib_utils.cpp.

```
# Create a shared library 'MyLib' from the source files
add_library(MyLib SHARED my_lib.cpp my_lib_utils.cpp)

# Specify the include directories if needed
target_include_directories(MyLib PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

4. src/my_lib.cpp: This is the main source file for the shared library.

```
// my_lib.cpp
#include "my_lib.h"

void MyLib::doSomething() {
    // Implement some functionality here
}
```

5. src/my_lib_utils.cpp: Another source file that adds utility functions to the library.

```
// my_lib_utils.cpp
#include "my_lib_utils.h"

int MyLibUtils::add(int a, int b) {
    return a + b;
}
```

6. main.cpp: The main executable that will link to the shared library MyLib.

```
#include <iostream>
#include "my_lib.h"
#include "my_lib_utils.h"
```

```
int main() {
    MyLib lib;
    lib.doSomething();

    MyLibUtils utils;
    std::cout << "Sum: " << utils.add(3, 4) << std::endl;

    return 0;
}
```

7. Building the Project: After configuring the CMake project, you can build it using the following commands:

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

This process will compile the shared library MyLib, create the corresponding shared object (.so or .dll), and link it to the executable MyApp.

4.3.4 Linking the Shared Library

Once the shared library is created, you need to link it to the executable that depends on it. This is done using the target_link_libraries() command in CMake.

For example, in the CMakeLists.txt file for the main project:

```
add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE MyLib)  # Link the shared library
```

This command tells CMake to link the MyApp executable with the MyLib shared library during the linking phase.

4.3.5 Handling RPATH and Shared Library Location

Since shared libraries are loaded at runtime, you must ensure that the shared library is available to the executable when it runs. This is typically managed using RPATH (runtime library search path), which tells the system where to look for shared libraries.

You can configure RPATH in CMake with the following commands:

```
set(CMAKE_INSTALL_RPATH "$ORIGIN")  # Set RPATH relative to the executable's
↪  location
```

- $ORIGIN means that the library will be searched for in the directory where the executable is located.
- Alternatively, you can specify an absolute path or relative path to the shared library.

When installing the shared library, CMake can also set the proper install paths for the shared library and the executable:

```
install(TARGETS MyLib DESTINATION lib)
install(TARGETS MyApp DESTINATION bin)
```

This ensures that the shared library is placed in the correct lib directory, and the executable is placed in the bin directory.

4.3.6 Using target_include_directories()

As with static libraries, you can use target_include_directories() to specify the include directories for the shared library:

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

The PUBLIC keyword indicates that this include directory is necessary not only for the library itself but also for any executable or library that links to it.

4.3.7 Versioning Shared Libraries

Shared libraries often require versioning to ensure that applications can link to a specific version of the library. To manage versioned shared libraries in CMake, you can use the following syntax:

```
add_library(MyLib SHARED my_lib.cpp my_lib_utils.cpp)

set_target_properties(MyLib PROPERTIES
    VERSION 1.0.0
    SOVERSION 1
)
```

- VERSION: Specifies the full version of the shared library.
- SOVERSION: Specifies the API version of the shared library. This version is used to ensure compatibility between the library and the applications that use it.

When building the shared library, CMake will automatically append the version and SOVERSION to the library filename (e.g., libMyLib.so.1.0.0 or libMyLib.so.1), which can help avoid version conflicts.

4.3.8 Advantages of Shared Libraries

- Smaller Executables: The executable remains small because the shared library code is not embedded within the executable.
- Memory Efficiency: Multiple running applications can share the same instance of the shared library in memory, reducing memory usage.
- Easier Updates: You can update the shared library independently of the applications that use it, as long as the interface remains backward compatible.
- Modularity: Shared libraries promote modularity and code reuse, as different applications can use the same shared library.

4.3.9 Disadvantages of Shared Libraries

- Dependency Management: The main disadvantage of shared libraries is that the application depends on the shared library being available at runtime. If the shared library is missing, the application will fail to run.
- Versioning Issues: When a shared library is updated, it can potentially break applications that rely on an older version. Proper versioning and backward compatibility are essential.
- Slightly Slower Startup: Shared libraries are loaded at runtime, which can result in a slight delay in application startup.

4.3.10 Conclusion

Creating a shared library in CMake is a powerful way to modularize your C++ projects, reduce the size of executables, and promote code reuse across multiple applications. By using the add_library(MyLib SHARED) command, you can easily create shared libraries, link them to executables, and manage dependencies efficiently.

However, as with any dynamic linking, shared libraries introduce challenges such as dependency management and versioning, which need to be carefully managed to ensure the stability of your application. With proper setup and configuration, shared libraries can significantly enhance the maintainability and scalability of your C++ projects.

4.4 Linking Libraries (target_link_libraries)

Linking libraries is a fundamental concept when building C++ projects using CMake. The target_link_libraries() command in CMake is used to specify which libraries (static or shared) an executable or another library should be linked with. Proper linking ensures that all necessary code from external libraries (whether static or shared) is incorporated into the final executable or library.

In this section, we will explore the target_link_libraries() command in detail, how it works with static and shared libraries, and provide practical examples to demonstrate how to manage and link libraries effectively in your CMake project.

4.4.1 Understanding the Purpose of target_link_libraries()

When you create a library or executable in CMake, it typically relies on external libraries to provide functionality that isn't part of the C++ standard library. These external libraries could be static libraries (which are compiled into the executable at compile time) or shared libraries (which are linked at runtime). The target_link_libraries() command is used to link an executable or library to one or more of these external libraries.

The basic syntax of the target_link_libraries() command is as follows:

```
target_link_libraries(<target> <libraries>)
```

Where:

- <target>: The name of the target (either an executable or another library) that you want to link the libraries to.

- <libraries>: The libraries that the target should be linked with. This can be a single library or a list of libraries.

For example:

```
add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE MyLib)
```

This command tells CMake to link the MyApp executable with the library MyLib.

4.4.2 Linking Static Libraries

When linking static libraries, CMake will ensure that the object files from the static library are copied into the final executable during the linking phase. Static libraries are included in the executable at compile-time, making the executable self-contained.

Here's how to link a static library to an executable using target_link_libraries():

```
# Create a static library
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)

# Create an executable
add_executable(MyApp main.cpp)

# Link the static library 'MyLib' to the executable 'MyApp'
target_link_libraries(MyApp PRIVATE MyLib)
```

In this case, MyApp depends on MyLib, and the MyLib library will be included directly into the final executable. The PRIVATE keyword indicates that MyLib is

needed only for MyApp and does not need to be propagated to other targets that depend on MyApp.

4.4.3 Linking Shared Libraries

When linking shared libraries, the executable or library doesn't include the shared library's object files at compile time. Instead, a reference to the shared library is included, and the actual code is loaded into memory when the program runs (at runtime). This means the shared library must be available when the program starts.

Here's how to link a shared library to an executable:

```
# Create a shared library
add_library(MyLib SHARED my_lib.cpp my_lib_utils.cpp)

# Create an executable
add_executable(MyApp main.cpp)

# Link the shared library 'MyLib' to the executable 'MyApp'
target_link_libraries(MyApp PRIVATE MyLib)
```

In this case, MyApp will not contain the code from MyLib directly, but it will rely on the shared library at runtime. The shared library (MyLib.so, .dll, or .dylib) must be found in the system's library search paths when the executable is run.

4.4.4 Specifying Link Dependencies

The target_link_libraries() command can also be used to specify a target's dependencies on other libraries. These dependencies can either be private, public,

or interface:

- PRIVATE: The library is only required for the target itself. Other targets that link to this target do not need to know about this library.
- PUBLIC: The library is required for the target, and any other target that links to this target will also need to link to this library.
- INTERFACE: The library is not required for the target itself, but any target that links to this target will need to link to the library.

Here's an example with each type of dependency:

```
# Create a static library
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)

# Create a shared library
add_library(OtherLib SHARED other_lib.cpp)

# Create an executable
add_executable(MyApp main.cpp)

# Link libraries with different visibility
target_link_libraries(MyApp PRIVATE MyLib)      # Only MyApp needs MyLib
target_link_libraries(MyApp PUBLIC OtherLib)    # MyApp and any other target linking
↪  MyApp will need OtherLib
```

- MyLib is linked only for MyApp and won't propagate to any target that depends on MyApp (via the PRIVATE keyword).
- OtherLib is required both for MyApp and for any target that links against MyApp (via the PUBLIC keyword).

4.4.5 Linking Multiple Libraries

You can link multiple libraries to a target at once. In this case, simply list the libraries separated by spaces:

```
# Create an executable
add_executable(MyApp main.cpp)

# Link multiple libraries
target_link_libraries(MyApp PRIVATE MyLib OtherLib AnotherLib)
```

In this example, MyApp is linked with three libraries: MyLib, OtherLib, and AnotherLib.

4.4.6 Linking System Libraries

In addition to linking libraries that are part of your project, you may need to link to system libraries or third-party libraries installed on your system (such as pthread, zlib, or boost). These libraries can be linked in the same way:

```
# Link to system libraries like pthread and zlib
target_link_libraries(MyApp PRIVATE pthread zlib)
```

You can also use find_package() to find installed libraries, such as Boost, and then link them to your target:

```
find_package(Boost REQUIRED)

add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE Boost::Boost)
```

In this case, CMake will search for the Boost library on your system and link it to MyApp.

4.4.7 Handling Transitive Dependencies

When a target is linked to another target that itself has dependencies, CMake will automatically propagate those dependencies, provided the correct visibility is specified. For instance, if a shared library OtherLib is linked to MyLib, and MyLib is linked to MyApp, the dependencies of MyLib will be propagated to MyApp if the PUBLIC or INTERFACE keyword is used:

```
add_library(MyLib SHARED my_lib.cpp)
add_library(OtherLib STATIC other_lib.cpp)

target_link_libraries(MyLib PUBLIC OtherLib)  # MyLib depends on OtherLib

add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE MyLib)  # MyApp will also depend on OtherLib
```

Here, since MyLib is linked to OtherLib using PUBLIC, MyApp will also implicitly depend on OtherLib when it links to MyLib.

4.4.8 Linking Libraries with Custom Paths

In some cases, your libraries may not be located in standard directories (such as /usr/lib or /lib). You can specify custom library paths using the link_directories() command:

```
link_directories(/path/to/custom/libs)

add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE MyLib)
```

This command tells CMake to look for libraries in the specified directory when linking. However, it's usually better practice to use find_package() for third-party libraries or to specify the full path in target_link_libraries() to avoid issues with finding libraries across different systems.

4.4.9 Best Practices for Linking Libraries

Here are some best practices to keep in mind when linking libraries with target_link_libraries():

- Be explicit: Always specify whether the library should be linked as PRIVATE, PUBLIC, or INTERFACE. This makes the dependencies clear and avoids unnecessary linking.
- Avoid link_directories() when possible: Rather than relying on link_directories(), prefer to specify full library paths or use find_package() for external libraries. This makes your project more portable.
- Group related libraries: If you have a large number of dependencies, consider grouping them logically (e.g., utility libraries, third-party libraries) and documenting them for easier maintenance.
- Use find_package() for system libraries: For commonly used external libraries (such as Boost, OpenSSL, or zlib), use CMake's find_package() functionality to automatically locate and link these libraries.

4.4.10 Conclusion

The target_link_libraries() command is a critical part of CMake's functionality for managing dependencies between libraries and executables. By properly linking your targets, you ensure that all necessary code is included during the linking phase, whether you are using static or shared libraries. Understanding the PRIVATE, PUBLIC, and INTERFACE keywords allows you to manage dependencies efficiently, ensuring your project is modular and maintainable.

With the knowledge gained in this section, you can confidently link multiple libraries, manage dependencies across targets, and ensure your CMake projects are properly structured and portable across different systems.

4.5 Controlling Symbol Visibility (PUBLIC, PRIVATE, INTERFACE)

4.5.1 Introduction to Symbol Visibility in CMake

When working with libraries in CMake, controlling symbol visibility is crucial for managing dependencies, improving build efficiency, and ensuring modularity. The PUBLIC, PRIVATE, and INTERFACE keywords help define how include directories, compile options, and linking dependencies are propagated between different targets.

Understanding these keywords allows developers to:

- Control which dependencies are exposed to consumers of a library.
- Optimize compilation by limiting unnecessary propagation of dependencies.
- Improve maintainability by keeping libraries self-contained.

This section explains the role of these keywords in CMake, how they affect compilation and linking, and provides real-world examples.

4.5.2 Understanding PUBLIC, PRIVATE, and INTERFACE

The PUBLIC, PRIVATE, and INTERFACE keywords in CMake define how compile options, include directories, and library dependencies are applied to a target and its consumers.

These keywords are primarily used with:

- target_link_libraries() – Specifies library dependencies.

- target_include_directories() – Specifies include directories.
- target_compile_options() – Specifies compilation flags or options.

- PRIVATE
  - The setting applies only to the target itself.
  - It does not propagate to other targets that depend on this target.
- PUBLIC
  - The setting applies to the target itself and also to any targets that link to it.
  - Useful for dependencies that must be available both during compilation of the library and when used by consumers.
- INTERFACE
  - The setting applies only to targets that link to this library.
  - The target itself does not use the setting.
  - Useful for header-only libraries or when exposing dependencies to consumers without affecting the library itself.

The following table summarizes how these keywords behave:

Keyword	Affects the Library Itself?	Affects Consumers of the Library?
PRIVATE	Yes	No
PUBLIC	Yes	Yes
INTERFACE	No	Yes

4.5.3 Controlling Include Directories with target_include_directories()

The target_include_directories() command specifies where the compiler should look for header files. The PUBLIC, PRIVATE, and INTERFACE keywords define how include directories are applied to the library and its consumers.

- Example: Include Directories with Visibility Control

  Consider the following CMake project structure:

```
MyProject/
    CMakeLists.txt
    src/
        CMakeLists.txt
        include/
            MyLib.h
        my_lib.cpp
        my_lib_utils.cpp
        my_lib_utils.h
    main.cpp
```

- Using PRIVATE Include Directories

```
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)

# This include directory is only used internally by MyLib
target_include_directories(MyLib PRIVATE
↪   ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

  - MyLib uses headers from include/, but these headers are not exposed to consumers.
  - If another target links against MyLib, it won't see this include directory.

- Using PUBLIC Include Directories

```
target_include_directories(MyLib PUBLIC
↪  ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

  - MyLib and any target linking to MyLib will use the include/ directory.
  - If MyLib.h is part of the public API, it must be accessible to consumers.

- Using INTERFACE Include Directories

```
target_include_directories(MyLib INTERFACE
↪  ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

  - MyLib itself does not use this include directory.
  - However, any target that links against MyLib will see include/.
  - Useful for header-only libraries where no compilation is needed.

4.5.4 Controlling Linking with target_link_libraries()

When linking a library to an executable or another library, PUBLIC, PRIVATE, and INTERFACE determine how dependencies are propagated.

Example: Linking Libraries with Visibility Control

```
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)
add_library(OtherLib SHARED other_lib.cpp)

# Different visibility levels
target_link_libraries(MyLib PRIVATE OtherLib)   # MyLib depends on OtherLib, but
↪  consumers don't
target_link_libraries(MyLib PUBLIC OtherLib)    # MyLib and its consumers require
↪  OtherLib
target_link_libraries(MyLib INTERFACE OtherLib) # Only consumers of MyLib need
↪  OtherLib
```

Visibility	Effect
PRIVATE	MyLib needs OtherLib, but consumers of MyLib don't need it.
PUBLIC	MyLib and all targets linking to MyLib also need OtherLib.
INTERFACE	MyLib itself doesn't use OtherLib, but consumers must link to it.

4.5.5 Controlling Compile Options with target_compile_options()

The target_compile_options() command applies compiler flags to a target and can use visibility keywords.

Example: Compiler Options with Visibility Control

```
add_library(MyLib STATIC my_lib.cpp my_lib_utils.cpp)

# Only MyLib is compiled with -Wall
target_compile_options(MyLib PRIVATE -Wall)
```

```
# MyLib and any target linking to MyLib will be compiled with -DDEBUG
target_compile_options(MyLib PUBLIC -DDEBUG)

# Consumers of MyLib get -O2, but MyLib itself does not
target_compile_options(MyLib INTERFACE -O2)
```

Visibility	Effect
PRIVATE	-Wall applies only to MyLib.
PUBLIC	-DDEBUG applies to MyLib and its consumers.
INTERFACE	-O2 applies only to consumers of MyLib.

4.5.6 Choosing the Right Visibility for Your Library

Here are guidelines to help choose the correct visibility:

- Use PRIVATE for internal dependencies that should not be exposed.
- Use PUBLIC when the dependency is required by both the target and its consumers (e.g., a core utility library).
- Use INTERFACE when only consumers need the dependency, such as a header-only library.

Example Use Cases

Scenario	Visibility	Why?
Library needs a private helper library	PRIVATE	The helper library should not be exposed to consumers.

Scenario	Visibility	Why?
A framework library exposes a public API	PUBLIC	Both the library and its users require the headers and linked dependencies.
A header-only library	INTERFACE	The library itself doesn't compile, but consumers need access to its headers.

4.5.7 Conclusion

Understanding PUBLIC, PRIVATE, and INTERFACE in CMake is essential for controlling include directories, linked libraries, and compiler options. Proper use of these keywords ensures modularity, optimizes dependency management, and keeps libraries self-contained while exposing only necessary parts to consumers.

By applying these principles, you can build scalable, maintainable, and efficient C++ projects using CMake.

Chapter 5

Organizing Large-Scale Projects with CMake

5.1 Understanding Multi-File Project Structure

5.1.1 Introduction to Multi-File Project Structure

As C++ projects grow in complexity, they often transition from a single-file structure to a multi-file structure. While small projects may have a single main.cpp file containing all logic, larger projects require modular organization to improve maintainability, reusability, and scalability.

CMake provides robust mechanisms to structure large-scale projects effectively. A well-organized project:

- Encourages code modularity by separating concerns into different files.
- Facilitates compilation efficiency by enabling incremental builds.

- Simplifies dependency management by clearly defining relationships between components.
- Enhances collaboration by making it easier for multiple developers to work on different parts of the codebase.

This section explores how to design and implement a multi-file project structure, including best practices and an example project setup using CMake.

5.1.2 Evolution of Project Structure

- Single-File Projects (Simple Programs)

  In the early stages of development, a C++ project may consist of just one or two source files.

  Example (single_file_project/):

```
single_file_project/
    CMakeLists.txt
    main.cpp
```

  main.cpp:

```cpp
#include <iostream>

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

  This structure is suitable for small prototypes or simple scripts, but as the codebase grows, this monolithic file becomes difficult to manage.

- Multi-File Projects (Modular Approach)

  To manage complexity, larger projects are broken into multiple files, often structured into header (.h) files, source (.cpp) files, and separate libraries.

  A typical multi-file project includes:

  1. A main executable (main.cpp) – Entry point of the application.
  2. A src/ directory – Contains implementation files (.cpp).
  3. An include/ directory – Contains header files (.h) for function/class declarations.
  4. A lib/ directory – Stores external or custom libraries.
  5. A CMakeLists.txt for each directory – Defines how each component is built.

  Example (multi_file_project/):

```
multi_file_project/
    CMakeLists.txt      # Root CMake file
    src/                # Source code directory
        CMakeLists.txt  # CMake file for source files
        main.cpp        # Main application entry
        MyLibrary.cpp   # Implementation of MyLibrary
        MyLibrary.h     # Header for MyLibrary
        Utilities.cpp   # Additional source file
    include/            # Header files
        MyLibrary.h
        Utilities.h
    lib/                # External or custom libraries
        ThirdPartyLib/
            CMakeLists.txt
            third_party.cpp
            third_party.h
```

```
build/              # Build directory (generated by CMake)
```

5.1.3 Setting Up a Multi-File Project with CMake

1. Root CMakeLists.txt

   At the root of the project, CMakeLists.txt sets up the project name, minimum required CMake version, and subdirectories for modular components.

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Add subdirectories
add_subdirectory(src)
add_subdirectory(lib/ThirdPartyLib)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

2. src/CMakeLists.txt (Managing Source Files)

   The src/ directory contains the main application and related source files. Here, we define an executable target and specify dependencies.

```
# Add the executable
add_executable(MyApp main.cpp MyLibrary.cpp Utilities.cpp)

# Include the "include/" directory so headers can be found
target_include_directories(MyApp PRIVATE ${PROJECT_SOURCE_DIR}/include)
```

```
# Link with third-party libraries if needed
target_link_libraries(MyApp ThirdPartyLib)
```

Explanation:

- add_executable(MyApp ...) – Creates an executable called MyApp from main.cpp, MyLibrary.cpp, and Utilities.cpp.
- target_include_directories() – Ensures that headers from include/ are available.
- target_link_libraries() – Links MyApp with an external library (ThirdPartyLib).

3. lib/ThirdPartyLib/CMakeLists.txt (External Libraries)

   The lib/ThirdPartyLib/ directory may contain third-party libraries or custom-built libraries. To build it separately:

```
add_library(ThirdPartyLib STATIC third_party.cpp)

target_include_directories(ThirdPartyLib PUBLIC
↪  ${CMAKE_CURRENT_SOURCE_DIR})
```

   Here, ThirdPartyLib is compiled as a static library, which can be linked to other targets.

5.1.4 Benefits of a Multi-File Project Structure

1. Improved Readability and Maintainability
   - Each file focuses on a single responsibility, making the project easier to understand.

2. Faster Compilation
    - CMake compiles only changed files, reducing build times.
3. Code Reusability
    - Libraries (lib/) can be reused across multiple projects.
4. Better Dependency Management
    - Using target_link_libraries(), dependencies are clearly defined.
5. Easier Collaboration
    - Multiple developers can work on different parts of the codebase simultaneously.

5.1.5 Best Practices for Structuring Large-Scale Projects

1. Follow a Modular Approach
    - Group related code into separate directories (src/, include/, lib/).
    - Use CMake subdirectories (add_subdirectory()) to organize components.
2. Keep Headers and Source Files Separate
    - Store .h files in include/ and .cpp files in src/.
    - Example:

```
include/MyLibrary.h  # Declarations
src/MyLibrary.cpp    # Implementations
```

3. Use Libraries for Code Reuse
    - Convert reusable components into static or shared libraries.

- Example:

```
add_library(MyLib STATIC my_lib.cpp)
target_include_directories(MyLib PUBLIC
↪  ${PROJECT_SOURCE_DIR}/include)
```

4. Clearly Define Dependencies
   - Use target_link_libraries() to specify dependencies explicitly.
5. Use CMAKE_SOURCE_DIR and CMAKE_BINARY_DIR Properly
   - Use CMAKE_SOURCE_DIR to reference source files.
   - Use CMAKE_BINARY_DIR for output locations.
6. Encapsulate Configuration in CMakeLists.txt Files
   - Each subdirectory should have its own CMakeLists.txt to keep configurations modular.

5.1.6 Conclusion

As projects grow, adopting a multi-file project structure becomes essential for maintainability and efficiency. CMake simplifies this process by allowing developers to organize files into logical components, define dependencies clearly, and manage builds effectively.

By following best practices such as modular organization, separate header/source files, and proper use of CMake's subdirectory and linking mechanisms, developers can create scalable and maintainable C++ projects.

With this foundation, we can now explore creating and linking static/shared libraries in the next sections of this chapter.

5.2 Creating Subprojects (add_subdirectory)

5.2.1 Introduction to Subprojects in CMake

As C++ projects grow, breaking them into subprojects (or modules) improves modularity, maintainability, and scalability. Instead of managing a large monolithic codebase, CMake allows projects to be structured into smaller, independent subprojects, each with its own CMake configuration.

CMake achieves this using the add_subdirectory() command, which enables:

- Hierarchical project organization by treating different components as independent subprojects.
- Incremental compilation by allowing selective rebuilding of modified subprojects.
- Code reuse by defining libraries in separate subprojects and linking them to the main project.
- Team collaboration by allowing developers to work on different subprojects independently.

This section explores how to structure and build multi-module C++ projects using add_subdirectory() in CMake.

5.2.2 Understanding add_subdirectory()

The add_subdirectory() command in CMake allows you to include another directory as a subproject.

Syntax:

```
add_subdirectory(<source_dir> [<binary_dir>] [EXCLUDE_FROM_ALL])
```

- <source_dir>: The path to the subproject's source directory.
- <binary_dir> (optional): The path where the build files for this subdirectory should be placed.
- EXCLUDE_FROM_ALL (optional): If specified, the subproject is not included in the default build target (useful for optional components).

Example Usage

```
add_subdirectory(lib/MyLibrary)
```

- This includes lib/MyLibrary/CMakeLists.txt, which defines a library or module.
- The main project can link to this subproject like any other CMake target.

5.2.3 Structuring a Project with Subprojects

Typical Multi-Module Project Structure

A large-scale project using subprojects may have the following structure:

```
MyProject/
  CMakeLists.txt        # Root CMake file
  src/                  # Main application source
    CMakeLists.txt
    main.cpp
    App.cpp
    App.h
  lib/                  # Libraries (subprojects)
    MyLibrary/
      CMakeLists.txt
      MyLibrary.cpp
      MyLibrary.h
    Utils/
      CMakeLists.txt
      Utils.cpp
      Utils.h
  build/                # Build directory (created by CMake)
```

- The src/ directory contains the main application.
- The lib/ directory contains two subprojects (MyLibrary and Utils).
- Each subproject has its own CMakeLists.txt, allowing independent compilation.

5.2.4 Defining Subprojects in CMake

1. Root CMakeLists.txt

   The main CMakeLists.txt should include all subprojects using add_subdirectory().

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Set C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Add subprojects
add_subdirectory(lib/MyLibrary)
add_subdirectory(lib/Utils)
add_subdirectory(src)

# Define the main executable
add_executable(MyApp src/main.cpp)

# Link subproject libraries
target_link_libraries(MyApp PRIVATE MyLibrary Utils)
```

Key Points:

- add_subdirectory(lib/MyLibrary) includes MyLibrary as a subproject.
- add_subdirectory(lib/Utils) includes Utils as a subproject.
- target_link_libraries(MyApp PRIVATE MyLibrary Utils) links these subprojects to the main executable.

2. lib/MyLibrary/CMakeLists.txt (Defining a Subproject)

   Each subproject should define its own library and specify include directories.

```
add_library(MyLibrary STATIC MyLibrary.cpp)

# Specify include directories
target_include_directories(MyLibrary PUBLIC
↪  ${CMAKE_CURRENT_SOURCE_DIR})
```

- MyLibrary is compiled as a static library.
- target_include_directories() makes its headers available to other subprojects.

3. lib/Utils/CMakeLists.txt (Another Subproject)

```
add_library(Utils STATIC Utils.cpp)

# Make include directory available
target_include_directories(Utils PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

4. src/CMakeLists.txt (Main Application Subproject)

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

# Include headers
target_include_directories(MyApp PRIVATE
↪  ${PROJECT_SOURCE_DIR}/lib/MyLibrary)
target_include_directories(MyApp PRIVATE ${PROJECT_SOURCE_DIR}/lib/Utils)

# Link libraries
target_link_libraries(MyApp PRIVATE MyLibrary Utils)
```

5.2.5 Benefits of Using add_subdirectory() for Subprojects

1. Improved Code Organization
   - Each component/library has its own directory and CMake configuration, keeping the root project clean.

2. Modular Compilation
    - Only modified subprojects are recompiled during incremental builds.
3. Easier Dependency Management
    - Subprojects can be linked only where needed using target_link_libraries().
4. Better Team Collaboration
    - Teams can work on separate subprojects independently without affecting the main build.
5. Reusability
    - Libraries created in lib/ can be used across multiple projects.

5.2.6 Using EXCLUDE_FROM_ALL for Optional Subprojects

If a subproject is optional, you can exclude it from the default build using EXCLUDE_FROM_ALL.

```
add_subdirectory(lib/ExperimentalFeature EXCLUDE_FROM_ALL)
```

- This prevents ExperimentalFeature from being built unless explicitly requested.

To enable it, use:

```
cmake -DBUILD_EXPERIMENTAL=ON ..
```

5.2.7 Best Practices for Managing Subprojects in CMake

1. Use add_subdirectory() for Modular Organization
    - Place independent components in their own directories.
2. Keep Each Subproject Self-Contained
    - Each subproject should have its own CMakeLists.txt and include directories.
3. Use PUBLIC, PRIVATE, and INTERFACE for Proper Dependency Control
    - Use target_include_directories() and target_link_libraries() properly.
4. Minimize Cross-Dependencies Between Subprojects
    - Avoid unnecessary dependencies between libraries to reduce coupling.
5. Use EXCLUDE_FROM_ALL for Optional Components
    - Keep optional features out of the default build.

5.2.8 Conclusion

The add_subdirectory() command is a powerful tool in CMake for organizing large-scale C++ projects into modular subprojects. By structuring a project into multiple smaller, independently compiled components, development becomes more scalable, maintainable, and efficient.

By following best practices for subproject organization, dependency management, and modular compilation, you can build robust, scalable, and reusable C++ applications using CMake.

5.3 Using find_package() to Locate External Libraries

5.3.1 Introduction to find_package()

As C++ projects grow in size and complexity, they often rely on external libraries for additional functionality. Instead of manually managing and compiling dependencies, CMake provides the find_package() command, which enables projects to automatically locate and use prebuilt libraries.

Using find_package(), CMake can:

- Search for installed libraries on the system.
- Retrieve library paths and configuration details automatically.
- Simplify dependency management by reducing the need for manual configuration.
- Improve cross-platform compatibility by adapting to different system environments.

This section explores how to use find_package() to locate external libraries, configure dependencies, and integrate them into CMake projects effectively.

5.3.2 Understanding find_package()

Basic Syntax

```
find_package(<PackageName> [version] [REQUIRED] [QUIET] [MODULE|CONFIG])
```

- <PackageName> – The name of the external library (e.g., Boost, OpenSSL, Eigen).
- version (optional) – Specifies a required version (e.g., 3.2.1).
- REQUIRED (optional) – Fails the configuration if the library is not found.
- QUIET (optional) – Suppresses warnings if the package is missing.
- MODULE or CONFIG (optional) – Specifies whether to look for a module or a config package.

Example Usage

```
find_package(OpenSSL REQUIRED)
```

- This searches for OpenSSL on the system.
- If OpenSSL is found, CMake sets up necessary variables (e.g., OPENSSL_INCLUDE_DIR, OPENSSL_LIBRARIES).
- If not found, CMake produces an error due to REQUIRED.

5.3.3 Finding and Linking External Libraries

Example 1: Finding OpenSSL and Linking It

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Find OpenSSL
find_package(OpenSSL REQUIRED)
```

```
# Create an executable
add_executable(MyApp main.cpp)

# Link OpenSSL
target_link_libraries(MyApp PRIVATE OpenSSL::SSL OpenSSL::Crypto)
```

- find_package(OpenSSL REQUIRED) locates OpenSSL.
- target_link_libraries(MyApp PRIVATE OpenSSL::SSL OpenSSL::Crypto) links OpenSSL libraries.
- This ensures that OpenSSL functions are available for MyApp.

Example 2: Finding and Using Boost

Boost is a widely used C++ library with many modules. Using find_package(Boost), we can include Boost components in a project.

```
find_package(Boost REQUIRED COMPONENTS filesystem system)

add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE Boost::filesystem Boost::system)
```

- COMPONENTS filesystem system ensures only required parts of Boost are found.
- Boost::filesystem and Boost::system are linked to MyApp.

5.3.4 Understanding find_package() Search Mechanism

How CMake Searches for Packages

When calling find_package(), CMake looks in:

1. CMake package registry (~/.cmake/packages/) – Stores paths of installed CMake packages.
2. System package managers (e.g., apt, brew, vcpkg, conan).
3. Environment variables (CMAKE_PREFIX_PATH, CMAKE_MODULE_PATH).
4. Standard system locations (/usr/lib/, /usr/local/lib/).

If a package isn't found, install the missing library using:

- Linux: sudo apt install libboost-dev
- macOS: brew install boost
- Windows: vcpkg install boost

5.3.5 Using CONFIG and MODULE Modes

CMake uses two modes to locate packages:

1. Config Mode (CONFIG)

   Modern libraries provide a CMake configuration file (<Package>Config.cmake) that tells CMake where to find headers and libraries.

```
find_package(OpenSSL CONFIG REQUIRED)
```

If OpenSSL is installed via vcpkg or conan, CMake finds it using OpenSSLConfig.cmake.

2. Module Mode (MODULE)

   If a package does not provide a Config.cmake file, CMake looks for a FindModule file (Find<Package>.cmake).

   ```
   find_package(OpenSSL MODULE REQUIRED)
   ```

   If FindOpenSSL.cmake exists, CMake uses it to locate OpenSSL.

5.3.6 Handling Missing Dependencies Gracefully

If a dependency is optional, use QUIET to suppress errors:

```
find_package(OpenSSL QUIET)
```

To provide a custom error message:

```
if(NOT OpenSSL_FOUND)
    message(FATAL_ERROR "OpenSSL not found! Please install it.")
endif()
```

5.3.7 Best Practices for Using find_package()

1. Always use REQUIRED for critical dependencies

```
find_package(OpenSSL REQUIRED)
```

2. Specify only needed components

```
find_package(Boost REQUIRED COMPONENTS filesystem system)
```

3. Use QUIET for optional dependencies

```
find_package(OpenSSL QUIET)
```

4. Provide clear error messages if dependencies are missing

```
if(NOT OpenSSL_FOUND)
    message(FATAL_ERROR "OpenSSL not found! Please install it.")
endif()
```

5. Set CMAKE_PREFIX_PATH for custom install locations

```
cmake -DCMAKE_PREFIX_PATH=/path/to/custom/install ..
```

5.3.8 Conclusion

The find_package() command is an essential tool in CMake for managing external dependencies efficiently. By using it, projects can automatically detect installed libraries, link dependencies correctly, and ensure compatibility across platforms.

By following best practices, CMake projects can integrate external libraries seamlessly, improve modularity, and reduce manual configuration overhead.

5.4 Using FetchContent to Download Dependencies at Build Time

5.4.1 Introduction to FetchContent

In modern C++ projects, managing external dependencies is a crucial aspect of project organization. Traditionally, dependencies were handled by manually installing libraries, using package managers, or relying on find_package(). However, these approaches often require preinstalled dependencies and additional configuration.

To address this challenge, CMake provides FetchContent, a powerful module that allows projects to download, configure, and build dependencies automatically at build time. This eliminates the need for external package managers and simplifies dependency management by ensuring that required libraries are fetched dynamically.

Advantages of FetchContent

- No need for preinstalled dependencies – Downloads and builds libraries as part of the project.
- Improved portability – Ensures dependencies are available regardless of system configuration.
- Simplifies build setup – Avoids the need for users to manually install libraries.
- Direct integration with CMake targets – Enables easy linking of fetched libraries.

This section explores how to use FetchContent to fetch, configure, and integrate dependencies in a CMake project.

5.4.2 Understanding FetchContent

The FetchContent module provides a mechanism to download source code from external repositories (e.g., GitHub) and build it within the current project.

Including the FetchContent Module

To use FetchContent, first include the module in CMakeLists.txt:

```
include(FetchContent)
```

This makes FetchContent available for downloading and managing dependencies.

Basic Syntax

```
FetchContent_Declare(
    <DependencyName>
    GIT_REPOSITORY <URL>
    GIT_TAG <TagOrBranch>
)
FetchContent_MakeAvailable(<DependencyName>)
```

- FetchContent_Declare defines an external dependency.
- GIT_REPOSITORY specifies the repository URL.
- GIT_TAG sets the commit, branch, or tag to use.
- FetchContent_MakeAvailable downloads and integrates the dependency.

5.4.3 Example: Fetching and Using an External Library

Using FetchContent to Download and Build fmt

fmt is a popular C++ formatting library. To include fmt in a CMake project dynamically, use FetchContent:

```
cmake_minimum_required(VERSION 3.14)
project(MyProject)

include(FetchContent)

# Fetch fmt library from GitHub
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 10.0.0 # Specify version
)

# Make fmt available for use
FetchContent_MakeAvailable(fmt)

# Define an executable
add_executable(MyApp main.cpp)

# Link fmt to the executable
target_link_libraries(MyApp PRIVATE fmt)
```

Explanation

1. FetchContent_Declare(fmt ...) specifies that fmt should be fetched from

GitHub.

2. FetchContent_MakeAvailable(fmt) downloads and integrates fmt into the build.
3. target_link_libraries(MyApp PRIVATE fmt) links the fmt library to the executable.

This approach ensures that fmt is automatically downloaded and built if not already available, eliminating the need for manual installation.

5.4.4 Using FetchContent with CMake Packages

If an external library provides a CMake package (i.e., a Config.cmake file), FetchContent can integrate it seamlessly.

Example: Fetching and Using googletest (GoogleTest)

GoogleTest (GTest) is a popular testing framework for C++. The following example shows how to integrate it using FetchContent:

```
include(FetchContent)

FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG v1.13.0
)

# Disable GoogleTest installation to avoid conflicts
set(INSTALL_GTEST OFF CACHE BOOL "Disable installation of GTest" FORCE)
```

```
FetchContent_MakeAvailable(googletest)

add_executable(MyTests test.cpp)

# Link GoogleTest to the test executable
target_link_libraries(MyTests PRIVATE gtest gtest_main)
```

Key Points

- FetchContent_Declare(googletest ...) specifies GoogleTest as a dependency.
- set(INSTALL_GTEST OFF ...) prevents GoogleTest from installing globally.
- FetchContent_MakeAvailable(googletest) fetches and builds the library.
- target_link_libraries(MyTests PRIVATE gtest gtest_main) links the test executable with GoogleTest.

This ensures that GoogleTest is always available for unit testing, without requiring users to install it manually.

5.4.5 Handling Already Installed Dependencies

A key advantage of FetchContent is that it avoids redundant downloads if a dependency is already available.

To check whether a package is already found via find_package() before downloading it:

```
find_package(fmt QUIET)
```

```
if(NOT fmt_FOUND)
    include(FetchContent)
    FetchContent_Declare(
        fmt
        GIT_REPOSITORY https://github.com/fmtlib/fmt.git
        GIT_TAG 10.0.0
    )
    FetchContent_MakeAvailable(fmt)
endif()
```

How This Works

- First, attempt to find the package (find_package(fmt QUIET)).
- If fmt is not found, download it using FetchContent.

This approach allows the project to use a system-installed version of the library if available, reducing unnecessary downloads.

5.4.6 Using FetchContent with Non-Git Sources

While FetchContent is commonly used with Git repositories, it also supports:

Fetching Tarball Archives

```
FetchContent_Declare(
    mylib
    URL https://example.com/mylib.tar.gz
    URL_HASH SHA256=abcdef1234567890
)
FetchContent_MakeAvailable(mylib)
```

This downloads a tarball, verifies its integrity with SHA256, and extracts it.

Fetching from Local Directories

```
FetchContent_Declare(
    mylib
    SOURCE_DIR ${CMAKE_SOURCE_DIR}/third_party/mylib
)
FetchContent_MakeAvailable(mylib)
```

This allows bundling dependencies within the project instead of downloading them from external sources.

5.4.7 Best Practices for Using FetchContent

1. Use find_package() before FetchContent
    - Avoid redundant downloads by checking for preinstalled versions.
2. Pin Specific Versions (GIT_TAG)
    - Always specify a commit/tag to ensure reproducibility.
3. Minimize External Dependencies
    - Only use FetchContent for essential libraries.
4. Use FetchContent_MakeAvailable() for Seamless Integration
    - Automatically configures dependencies within the project.
5. Disable Installation for FetchContent Dependencies
    - Prevent unnecessary installation of fetched libraries (set(INSTALL_GTEST OFF)).

5.4.8 Conclusion

FetchContent provides an efficient way to manage external dependencies at build time. By dynamically downloading, configuring, and building libraries, it simplifies project setup and ensures that dependencies are always available.

By following best practices, developers can integrate third-party libraries seamlessly, improve build automation, and eliminate manual dependency installation, making projects more maintainable and portable.

5.5 Using ExternalProject_Add for External Project Integration

5.5.1 Introduction to ExternalProject_Add

When working on large-scale C++ projects, it is common to depend on external libraries or third-party projects. These dependencies may need to be downloaded, configured, built, and installed as part of the project's build process. While find_package() and FetchContent are useful for integrating prebuilt or source-based dependencies, they do not provide full control over the build process of external projects.

CMake's ExternalProject_Add module addresses this by:

- Allowing external projects to be downloaded, configured, and built independently.
- Supporting various source types (Git, tarballs, local directories).
- Providing custom build commands for fine-grained control over dependencies.
- Ensuring that external dependencies are built before the main project.

This section explores how to use ExternalProject_Add to integrate and manage external dependencies effectively.

5.5.2 UnderstandingExternalProject_Add

The ExternalProject_Add command is part of CMake's ExternalProject module, which provides a way to build an external library separately from the main project.

Unlike FetchContent, which integrates dependencies into the main build tree, ExternalProject_Add treats dependencies as completely separate projects.

Including the ExternalProject Module

To use ExternalProject_Add, include the module in CMakeLists.txt:

```
include(ExternalProject)
```

This makes ExternalProject_Add available for managing external dependencies.

Basic Syntax

```
ExternalProject_Add(
    <ProjectName>
    GIT_REPOSITORY <URL>
    GIT_TAG <TagOrBranch>
    PREFIX <Directory>
    CMAKE_ARGS <Arguments>
    BUILD_COMMAND <BuildCommand>
    INSTALL_COMMAND <InstallCommand>
)
```

- <ProjectName> – The name of the external project.
- GIT_REPOSITORY – URL of the Git repository (or URL for tarballs).
- GIT_TAG – Commit hash, tag, or branch to checkout.
- PREFIX – The directory where the project will be downloaded and built.
- CMAKE_ARGS – Additional arguments to pass to CMake when configuring the external project.

- BUILD_COMMAND – Custom build command (default: make).
- INSTALL_COMMAND – Custom install command (default: make install).

5.5.3 Example: Building an External Library with ExternalProject_Add

Using ExternalProject_Add to Integrate fmt

```
cmake_minimum_required(VERSION 3.14)
project(MyProject)

include(ExternalProject)

# Define an external project for fmt
ExternalProject_Add(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 10.0.0
    PREFIX ${CMAKE_BINARY_DIR}/external/fmt
    CMAKE_ARGS
    ↪  -DCMAKE_INSTALL_PREFIX=${CMAKE_BINARY_DIR}/external/fmt/install
)

# Include fmt headers
include_directories(${CMAKE_BINARY_DIR}/external/fmt/install/include)

# Define an executable
add_executable(MyApp main.cpp)

# Link fmt library manually
target_link_libraries(MyApp PRIVATE
↪  ${CMAKE_BINARY_DIR}/external/fmt/install/lib/libfmt.a)
```

Explanation

1. ExternalProject_Add(fmt …) – Downloads fmt from GitHub and builds it.
2. CMAKE_INSTALL_PREFIX – Specifies where fmt should be installed.
3. include_directories(...) – Ensures that the fmt headers are available.
4. target_link_libraries(...) – Manually links the built fmt library.

5.5.4 Building and Installing External Projects

ExternalProject_Add provides full control over the build and installation process. If a library needs specific configuration options, pass them using CMAKE_ARGS.

Example: Custom Build and Install Commands

```
ExternalProject_Add(
    mylib
    GIT_REPOSITORY https://github.com/example/mylib.git
    GIT_TAG master
    PREFIX ${CMAKE_BINARY_DIR}/mylib
    CONFIGURE_COMMAND ./configure --prefix=${CMAKE_BINARY_DIR}/mylib/install
    BUILD_COMMAND make -j4
    INSTALL_COMMAND make install
)
```

Key Differences

- CONFIGURE_COMMAND – Uses ./configure instead of CMake.

- BUILD_COMMAND – Uses make -j4 for parallel builds.
- INSTALL_COMMAND – Runs make install to install the library.

This flexibility makes ExternalProject_Add ideal for integrating projects that do not use CMake.

5.5.5 Handling Dependencies Between External Projects

Ensuring One Project Builds Before Another

If multiple external projects depend on each other, use DEPENDS to ensure correct build order.

```
ExternalProject_Add(
    mylib
    GIT_REPOSITORY https://github.com/example/mylib.git
    PREFIX ${CMAKE_BINARY_DIR}/mylib
)

ExternalProject_Add(
    myapp
    GIT_REPOSITORY https://github.com/example/myapp.git
    PREFIX ${CMAKE_BINARY_DIR}/myapp
    DEPENDS mylib
)
```

- mylib is built before myapp to satisfy dependencies.

5.5.6 Using ExternalProject_Add with CMake Targets

Unlike FetchContent, ExternalProject_Add does not automatically create CMake targets. To link an external project as a target, create an imported library:

Example: Creating an Imported Target

```
add_library(fmt STATIC IMPORTED)
set_target_properties(fmt PROPERTIES
    IMPORTED_LOCATION ${CMAKE_BINARY_DIR}/external/fmt/install/lib/libfmt.a
    INTERFACE_INCLUDE_DIRECTORIES
    ↪  ${CMAKE_BINARY_DIR}/external/fmt/install/include
)

target_link_libraries(MyApp PRIVATE fmt)
```

Why Use Imported Targets?

- Avoids manually specifying library paths.
- Makes target_link_libraries(MyApp PRIVATE fmt) more readable.
- Ensures correct dependency management.

5.5.7 Differences Between ExternalProject_Add and FetchContent

Feature	FetchContent	ExternalProject_Add
When to Use	When sources should be part of the main build	When the external project should be built separately

Feature	FetchContent	ExternalProject_Add
Dependency Integration	Integrated into the main CMake project	Built as an independent project
Supports Non-CMake Projects	No	Yes
Requires Preinstalled Dependencies	No	No
Builds Dependencies Separately	No	Yes
Use Case	Header-only libraries or small dependencies	Large projects, external dependencies with complex build steps

5.5.8 Best Practices for Using ExternalProject_Add

1. Use DEPENDS to specify build order
   - Ensures that dependencies are built before the main project.
2. Use CMAKE_ARGS to pass options to external builds
   - Example:

```
ExternalProject_Add(myproj CMAKE_ARGS
↪  -DCMAKE_BUILD_TYPE=Release)
```

3. Use INSTALL_COMMAND to control how external projects are installed
   - Avoid unnecessary global installations.

4. Create Imported Targets
    - Allows easier linking to the external project.
5. Prefer FetchContent for header-only or small dependencies
    - FetchContent is simpler when integration within the main project is needed.

5.5.9 Conclusion

The ExternalProject_Add module is a powerful tool for integrating external projects into a CMake build system. Unlike FetchContent, it treats dependencies as fully independent builds, making it ideal for large, complex dependencies that require separate configuration, compilation, and installation.

By understanding how to fetch, build, and integrate external projects, developers can improve project organization, simplify dependency management, and ensure robust cross-platform builds.

Chapter 6

Dependency Management in CMake

6.1 Using find_package() to Locate Installed Libraries

6.1.1 Introduction

When developing a C++ project with CMake, it's common to rely on external libraries to extend functionality and avoid reinventing the wheel. Managing these dependencies efficiently is crucial for maintainability and portability. CMake provides the find_package() command as a robust mechanism to locate and integrate installed libraries on a system. This section delves into how find_package() works, its usage patterns, and best practices.

6.1.2 Understanding find_package()

The find_package() command in CMake is used to locate external libraries or packages installed on a system. It determines the necessary include directories,

library paths, and compile definitions required to use the library in a C++ project.

Basic Syntax

```
find_package(<PackageName> [version] [REQUIRED] [QUIET] [COMPONENTS <comp1>
↪  <comp2> ...] [CONFIG|MODULE])
```

Arguments Explanation

- <PackageName> – The name of the package to find (e.g., Boost, Eigen3, OpenCV).
- version – (Optional) The minimum required version of the package.
- REQUIRED – If specified, CMake will terminate with an error if the package is not found.
- QUIET – Suppresses messages when the package is not found.
- COMPONENTS – Specifies particular submodules of the package to locate.
- CONFIG or MODULE – Specifies whether to look for a package using a Config-file or a Module-file approach (explained later).

6.1.3 Locating a Library Using find_package()

Example 1: Finding a Library Without Version Specification

Suppose we want to use Eigen3, a popular C++ linear algebra library, in our CMake project.

```
find_package(Eigen3 REQUIRED)
```

- This command searches for Eigen3 and ensures it is found before proceeding.
- If Eigen3 is not installed, CMake generates an error and stops the configuration process.

Example 2: Finding a Library With a Version Requirement

If our project needs Eigen3 version 3.3 or later, we specify it like this:

```
find_package(Eigen3 3.3 REQUIRED)
```

- If Eigen3 version 3.3 or higher is found, it proceeds normally.
- If an older version or no installation is found, CMake stops with an error.

Example 3: Using COMPONENTS to Find Specific Parts of a Library

Some libraries provide multiple components. Boost is a common example. If we only need the filesystem and system components:

```
find_package(Boost REQUIRED COMPONENTS filesystem system)
```

- This ensures that both Boost::filesystem and Boost::system are found.
- If any required component is missing, the configuration fails.

6.1.4 Config-Mode vs. Module-Mode in find_package()

CMake supports two methods to find packages: Config-Mode and Module-Mode.

Config-Mode (CONFIG)

- Used when the library provides its own CMake configuration files (<PackageName>Config.cmake or <PackageName>-config.cmake).
- Typically found in installation directories like /usr/lib/cmake/ or custom locations.
- More reliable than Module-Mode because it contains package-specific settings.

Example:

```
find_package(OpenCV CONFIG REQUIRED)
```

- Looks for OpenCVConfig.cmake in standard or user-specified locations.

If OpenCV is found, it provides imported targets such as OpenCV::Core and OpenCV::ImgProc, which can be linked in a CMake project:

```
target_link_libraries(my_project PRIVATE OpenCV::Core OpenCV::ImgProc)
```

Module-Mode (MODULE)

- CMake ships with built-in Find Modules (Find<PackageName>.cmake) for common libraries.

- These modules contain logic to locate library headers and binaries.

Example:

```
find_package(OpenGL REQUIRED)
```

- CMake will look for FindOpenGL.cmake in its module path (usually inside CMake's installation directory).

While this method works for many libraries, it is less preferred than Config-Mode because it may not always align with the latest versions of the library.

6.1.5 Handling Missing Dependencies Gracefully

If a package is not found, the find_package() command typically sets a <PackageName>_FOUND variable to FALSE. Developers can check for this and provide alternative actions:

```
find_package(Eigen3 QUIET)

if (NOT Eigen3_FOUND)
    message(FATAL_ERROR "Eigen3 was not found! Please install it.")
endif()
```

Alternatively, an optional dependency can be handled gracefully:

```
find_package(SDL2 QUIET)

if (SDL2_FOUND)
```

```
    message(STATUS "SDL2 found: ${SDL2_INCLUDE_DIRS}")
else()
    message(WARNING "SDL2 not found, disabling SDL2 support.")
endif()
```

6.1.6 Specifying Custom Paths for Dependencies

If a library is installed in a non-standard location, find_package() may fail to locate it. We can specify additional search paths using:

```
set(CMAKE_PREFIX_PATH "/custom/install/path" CACHE STRING "Custom search path
↪   for dependencies")
find_package(Eigen3 REQUIRED)
```

Alternatively, users can provide the path at the CMake command line:

```
cmake -DCMAKE_PREFIX_PATH=/custom/install/path ..
```

6.1.7 Using find_package() in a Complete Project Example

Project Structure

```
/my_project
    CMakeLists.txt
    main.cpp
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Locate Eigen3
find_package(Eigen3 REQUIRED)

add_executable(my_project main.cpp)

# Link Eigen3 to the project
target_link_libraries(my_project PRIVATE Eigen3::Eigen)
```

main.cpp

```
#include <Eigen/Dense>
#include <iostream>

int main() {
    Eigen::Matrix2d mat;
    mat << 1, 2,
           3, 4;
    std::cout << "Matrix:\n" << mat << std::endl;
    return 0;
}
```

Build Instructions

```
cmake -B build
cmake --build build
```

6.1.8 Best Practices for Using find_package()

- Prefer Config-Mode (CONFIG) over Module-Mode whenever possible.
- Use REQUIRED only when the package is essential.
- Use QUIET for optional dependencies.
- Check <PackageName>_FOUND before using the package.
- Allow users to specify custom paths using CMAKE_PREFIX_PATH.
- Provide clear error messages when dependencies are missing.

6.1.9 Conclusion

The find_package() command is a fundamental tool in CMake for managing dependencies. It simplifies integration with external libraries while maintaining portability. By understanding its syntax, Config/Module modes, and best practices, developers can create more maintainable and flexible build systems for C++ projects.

6.2 Creating Config.cmake Files for Custom Libraries

6.2.1 Introduction

When working with CMake, managing custom libraries or dependencies within your own projects becomes more seamless if you create and use Config.cmake files. These files provide a standardized way to define how your library should be located, linked, and configured within other CMake-based projects. In this section, we'll discuss how to create Config.cmake files for custom libraries, which simplifies the integration of your library into other projects.

6.2.2 What is a Config.cmake File?

A Config.cmake file is a CMake script used to configure and locate a library or package in a CMake project. It contains necessary information, such as include directories, library paths, and targets, that other CMake projects can use to link with your library. It enables a clean, reusable, and easy-to-maintain interface for your library, allowing other developers to seamlessly find and use it.

In contrast to the module-based approach (via Find<PackageName>.cmake), the Config approach is more modern and preferable for custom libraries. It explicitly defines the package, making it more flexible and robust. Typically, Config.cmake files are used when distributing libraries that are CMake-aware.

6.2.3 Structure of a Config.cmake File

A typical Config.cmake file has the following structure:

1. Set Include Directories
   Specify the headers or directories to include in other projects that use your library.
2. Set Library Directories
   Provide paths to the library files (e.g., .so, .a, .dll).
3. Define Targets
   Create imported targets to represent the library, so users can link to it easily.
4. Set Variables
   Define variables that CMake users can refer to in their own CMakeLists.txt.
5. Package-Specific Settings
   Define specific options that may be needed to configure the package (e.g., build settings, compile options).

Basic Example of a Config.cmake File

Let's consider a simple C++ library called MyLib. Below is an example of what the MyLibConfig.cmake file might look like.

```
# MyLibConfig.cmake

# Ensure the package is being found correctly
set(MYLIB_VERSION "1.0.0" CACHE STRING "Version of MyLib")

# Specify the installation directory for the headers and libraries
set(MYLIB_INCLUDE_DIR "${CMAKE_INSTALL_PREFIX}/include")
set(MYLIB_LIBRARIES "${CMAKE_INSTALL_PREFIX}/lib/libmylib.a")

# Define an imported target for easy usage
add_library(MyLib::MyLib STATIC IMPORTED)
```

```
set_target_properties(MyLib::MyLib PROPERTIES
    IMPORTED_LOCATION "${MYLIB_LIBRARIES}"
    INTERFACE_INCLUDE_DIRECTORIES "${MYLIB_INCLUDE_DIR}"
)

# Optionally, define additional variables or configuration settings
set(MYLIB_FOUND TRUE CACHE BOOL "Indicates if MyLib is found")
```

Explanation of the File Components

- Set Variables: MYLIB_INCLUDE_DIR and MYLIB_LIBRARIES define the paths to the library's headers and compiled libraries, respectively.
- Define Targets: The add_library() command creates an imported target MyLib::MyLib, which can be used by other projects to link with your library.
- Set Properties: set_target_properties() assigns properties like the location of the library and the include directories to the target.
- Version and Metadata: The version and MYLIB_FOUND variable provide metadata that can be checked by users.

6.2.4 Where Should the Config.cmake File Be Installed?

For other CMake projects to find your library, the Config.cmake file should be installed into a known CMake search directory. Common locations include:

- System-wide directories (e.g., /usr/local/lib/cmake/mylib/)
- User-defined directories (e.g., /home/user/mylib/cmake/)

The CMAKE_PREFIX_PATH variable is often used to specify the path where CMake should search for the Config.cmake files.

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

If you are distributing the library, you might want to install the Config.cmake file into the following location in your library's installation process:

```
install(
    FILES MyLibConfig.cmake
    DESTINATION lib/cmake/MyLib
)
```

This installation ensures that when another project uses find_package(MyLib REQUIRED), CMake can locate and use the MyLibConfig.cmake file.

6.2.5 Using find_package() to Find a Custom Library

Once the Config.cmake file is created and installed, other projects can use the find_package() command to locate and link your library. For example, in another project, you can include the following in your CMakeLists.txt:

```
find_package(MyLib REQUIRED)

add_executable(my_project main.cpp)

# Link the imported target for MyLib
target_link_libraries(my_project PRIVATE MyLib::MyLib)
```

When find_package() is called, CMake will search for the MyLibConfig.cmake file. If the library is found, it will configure the project to use MyLib. This includes setting include paths, linking to the correct library, and creating an imported target for the library.

6.2.6 Handling Multiple Versions of a Custom Library

If your library is evolving and you want to support multiple versions, you can specify the version in the Config.cmake file. Additionally, you can enforce version checks by specifying a version requirement in the find_package() call.

Example:

```
# In MyLibConfig.cmake
set(MYLIB_VERSION "1.0.0")

# If MyLib is part of a versioned library
set(MYLIB_VERSION "2.0.0" CACHE STRING "Version of MyLib")
```

Then, in the consuming project:

```
find_package(MyLib 2.0.0 REQUIRED)
```

This approach ensures that only the correct version of the library is linked with the consuming project.

6.2.7 Supporting Optional Features and Components

Libraries often have optional components or features that can be enabled or disabled. You can add configuration options in the Config.cmake file to manage

these features.

For example, suppose MyLib has an optional feature for SSL support. You could add an option in the Config.cmake file like this:

```
# In MyLibConfig.cmake
option(MYLIB_USE_SSL "Enable SSL support" ON)

if(MYLIB_USE_SSL)
    target_compile_definitions(MyLib::MyLib INTERFACE USE_SSL)
endif()
```

When using the library, consumers can enable or disable SSL support by setting the option:

```
find_package(MyLib REQUIRED)

# Optionally, configure SSL
option(MYLIB_USE_SSL "Enable SSL support" OFF)
```

This way, the Config.cmake file provides flexibility in how the library is configured.

6.2.8 Best Practices for Creating Config.cmake Files

Here are some best practices for creating and maintaining Config.cmake files for custom libraries:

- Consistency in naming: Use consistent naming conventions for variables, targets, and CMake properties. For example, always use the prefix MYLIB_ for library-specific variables.

- Versioning: Make sure to include version information in the Config.cmake file, and support version checks when appropriate.
- Target properties: Use imported targets (add_library(... IMPORTED)) whenever possible to maintain a clean and modern CMake interface.
- Support multiple components: If your library provides optional components, use find_package()'s COMPONENTS option and allow consumers to specify which components they need.
- Clear error handling: Always provide meaningful error messages if the package is not found or if required components are missing.

6.2.9 Conclusion

Creating Config.cmake files is an essential skill for developers managing custom C++ libraries. These files provide a robust and flexible interface for users of your library, helping them integrate it seamlessly into their own CMake-based projects. By following the outlined best practices, you can ensure that your library is easy to use, versioned correctly, and well-suited for complex dependency management.

6.3 Using FetchContent to Fetch Dependencies Dynamically

6.3.1 Introduction

Managing external dependencies is a crucial part of any C++ project, especially when building modular and scalable applications. While methods like find_package() or manually linking libraries have been widely used, CMake offers an elegant solution for dynamically fetching and managing dependencies during the build process: FetchContent. This feature allows you to download, configure, and use external dependencies directly from the source at the time of the build, without the need for pre-installed packages or system-wide configurations. In this section, we will explore how to use FetchContent effectively for fetching dependencies dynamically.

6.3.2 What is FetchContent?

FetchContent is a CMake module introduced to simplify dependency management by automatically downloading content (libraries, files, or other resources) from external repositories and making them available within your project. Unlike traditional methods where dependencies need to be pre-installed, FetchContent enables you to fetch dependencies during the build process directly from a specified URL or version tag.

This is particularly useful when you:

- Don't want users to manually install dependencies.
- Need to work with specific versions of a library or tool.
- Want to keep all dependencies under version control for reproducibility.

FetchContent ensures that the required dependencies are downloaded, configured, and built as part of the project, making the integration process seamless.

6.3.3 Basic Syntax of FetchContent

The core command provided by CMake to use this feature is FetchContent_Declare(). This command declares a content to be fetched, while other commands are used to ensure the content is actually downloaded and available for use.

```
FetchContent_Declare(
    <content_name>
    GIT_REPOSITORY <repository_url>
    GIT_TAG <commit_or_branch_or_tag>  # Optional: specify a version
    DOWNLOAD_NO_EXTRACT <ON|OFF>      # Optional: whether to extract content
)

# Make content available and add it to the project
FetchContent_MakeAvailable(<content_name>)
```

Parameters of FetchContent_Declare():

- <content_name>: The name of the content being fetched.
- GIT_REPOSITORY: The URL of the Git repository from which the content will be fetched. Alternatively, URL can be used if fetching from a different source (like a zip file or tarball).
- GIT_TAG: The specific version (commit hash, tag, or branch) of the repository to checkout. You can specify master, a version number, or a commit hash.

- DOWNLOAD_NO_EXTRACT: If set to ON, CMake will only download the content but will not extract it. Useful when dealing with archives that don't need to be extracted.

6.3.4 Example: Using FetchContent to Fetch a Dependency

Let's take an example where we want to fetch a library called fmt, a popular C++ formatting library. Here is how we can declare and fetch it dynamically using FetchContent.

CMakeLists.txt Example:

```
cmake_minimum_required(VERSION 3.14)
project(MyProject)

# Declare the 'fmt' library as content to be fetched from GitHub
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 8.0.1  # Fetch a specific version
)

# Make the 'fmt' content available for the project
FetchContent_MakeAvailable(fmt)

# Use fmt in your target
add_executable(my_project main.cpp)
target_link_libraries(my_project PRIVATE fmt::fmt)
```

Explanation:

- FetchContent_Declare(fmt ...): We declare that we want to fetch the fmt library from the GitHub repository. The GIT_TAG specifies the exact version (in this case, 8.0.1).
- FetchContent_MakeAvailable(fmt): This command ensures that the library is fetched and added to the build process. It makes the library available as if it were part of the project.
- target_link_libraries(): After the dependency is available, you can link it as you would any other library. The fmt::fmt target is automatically created by the FetchContent module.

6.3.5 Fetching Dependencies from Different Sources

While Git repositories are commonly used with FetchContent, CMake also supports fetching from other sources like tarballs or zip files. The general idea remains the same, but the parameters change slightly.

Example 1: Fetching from a Tarball:

```
FetchContent_Declare(
    my_library
    URL https://example.com/my_library.tar.gz
)

FetchContent_MakeAvailable(my_library)
```

- URL: Specifies the location of the tarball or zip file to be downloaded and extracted.

Example 2: Fetching from a Subdirectory (Local Files):

If your project has a local subdirectory containing source files for an external dependency, you can use FetchContent to manage it similarly.

```
FetchContent_Declare(
    my_local_lib
    URL_FILEPATH ${CMAKE_CURRENT_SOURCE_DIR}/libs/my_local_lib.tar.gz
)

FetchContent_MakeAvailable(my_local_lib)
```

Here, URL_FILEPATH allows you to refer to a local path.

6.3.6 Managing Dependency Versions

When using FetchContent, it's important to manage the versions of dependencies being fetched. CMake provides the ability to specify exact versions through the GIT_TAG, GIT_SHA1, or GIT_HASH options for repositories. This ensures that your project builds against a specific, known version of a dependency.

Example:

```
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 8.0.1  # Fetch a specific release version
)
```

Alternatively, you can specify a commit hash directly:

```
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 94e57d7a098ae3289e6a658801c5b5487ef0981a  # Specific commit hash
)
```

This ensures that the content will always be retrieved at that exact state, guaranteeing reproducibility.

6.3.7 Using FetchContent for Multiple Dependencies

You can fetch multiple dependencies in the same CMakeLists.txt file by calling FetchContent_Declare() multiple times and using FetchContent_MakeAvailable() for each one.

Example: Fetching Multiple Dependencies:

```
cmake_minimum_required(VERSION 3.14)
project(MyProject)

# Fetch fmt
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 8.0.1
)
FetchContent_MakeAvailable(fmt)

# Fetch spdlog
FetchContent_Declare(
```

```
    spdlog
    GIT_REPOSITORY https://github.com/gabime/spdlog.git
    GIT_TAG v1.9.2
)
FetchContent_MakeAvailable(spdlog)

# Create executable and link dependencies
add_executable(my_project main.cpp)
target_link_libraries(my_project PRIVATE fmt::fmt spdlog::spdlog)
```

In this example, both fmt and spdlog are fetched dynamically, and we link them to our project.

6.3.8 Benefits and Limitations of FetchContent

Benefits:

- No Need for Pre-Installation: Dependencies are downloaded and built automatically, eliminating the need for manual installation or pre-configured systems.
- Version Control: You can specify exact versions of dependencies, ensuring consistency across different builds.
- Simplified Dependency Management: All dependencies are handled within the CMake build process, reducing configuration overhead for developers.

Limitations:

- Network Dependency: The build process relies on an active internet connection to fetch the dependencies.
- Build Time: Fetching and building dependencies during the configuration process can increase the overall build time, especially if the dependencies are large or numerous.
- Complexity: While FetchContent simplifies dependency management, it can introduce complexity in terms of version control and potential conflicts between dependencies.

6.3.9 Best Practices for Using FetchContent

- Use for Dependencies with Less Frequent Updates: Since FetchContent downloads the dependency every time the project is built, it's better suited for stable, well-maintained libraries that don't update too frequently.
- Use Version Control: Always specify a version (tag or commit hash) to ensure reproducibility. Avoid using branches like master unless necessary.
- Minimize Overuse: For large or numerous dependencies, consider other methods such as find_package() for common system-wide libraries to avoid downloading and building everything.
- Check CMake Version: Ensure that you're using a version of CMake that supports FetchContent (CMake 3.14 or newer).

6.3.10 Conclusion

FetchContent is an incredibly powerful feature in CMake that simplifies dependency management, making it easier to work with external libraries without

worrying about pre-installation. It offers flexibility, control over versions, and a seamless integration process. By understanding how to use FetchContent, you can ensure that your project remains portable and easy to build on any system, regardless of whether the dependencies are pre-installed or not.

6.4 Integrating pkg-config and find_library()

6.4.1 Introduction

When managing dependencies in a CMake-based project, you often encounter scenarios where external libraries are installed via package management systems on Linux-based systems. Many of these libraries provide a tool called pkg-config, which is commonly used to retrieve metadata about installed libraries. CMake, being a cross-platform build system, can easily integrate with pkg-config to fetch information about installed libraries and their locations.

In this section, we'll explore how to integrate pkg-config with CMake and use find_library() to locate libraries dynamically, enabling you to link external dependencies seamlessly into your project. This integration is especially useful when working with libraries that provide pkg-config files, such as GTK, OpenSSL, or others commonly used in the open-source ecosystem.

6.4.2 What is pkg-config?

pkg-config is a tool used on Unix-like systems (Linux, macOS, etc.) to provide information about installed libraries. It gives details like:

- Compiler flags (-I, -L, -D)
- Library locations
- Required dependencies of a library
- Version information

pkg-config works by reading .pc files, which are installed by the package manager when a library is installed. These files contain the necessary information about the library and are typically located in directories like /usr/lib/pkgconfig/ or /usr/local/lib/pkgconfig/.

Example: Using pkg-config

For example, if you want to link with the libpng library, you can use pkg-config as follows:

```
pkg-config --cflags --libs libpng
```

This command would output something like:

```
-I/usr/include/libpng -L/usr/lib -lpng
```

You can pass these flags to the compiler and linker manually, or automate the process with CMake.

6.4.3 Integrating pkg-config with CMake

To integrate pkg-config with CMake, you use the find_package() or pkg_check_modules() commands in your CMakeLists.txt. This allows CMake to use pkg-config to retrieve information about external libraries and automatically configure the build.

Using find_package() with pkg-config

If the library you are working with supports pkg-config and CMake's find_package() is designed to work with it, you can use the following approach:

1. Enable pkg-config: First, ensure that CMake knows to look for pkg-config by enabling the PkgConfig module.
2. Use find_package(): You can then call find_package() to find the required library, relying on pkg-config to handle the search.

Example:

Let's say we want to use the libpng library in our project. CMake has support for pkg-config through its FindPkgConfig module, and we can use it like this:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.10)
project(MyProject)

# Enable pkg-config
find_package(PkgConfig REQUIRED)

# Use pkg-config to locate the libpng library
pkg_check_modules(LIBPNG REQUIRED libpng)

# Print the library's flags
message(STATUS "LIBPNG_CFLAGS: ${LIBPNG_CFLAGS}")
message(STATUS "LIBPNG_LIBRARIES: ${LIBPNG_LIBRARIES}")

# Add your executable and link the found library
add_executable(my_project main.cpp)
target_include_directories(my_project PRIVATE ${LIBPNG_INCLUDE_DIRS})
target_link_libraries(my_project PRIVATE ${LIBPNG_LIBRARIES})
```

Explanation:

- find_package(PkgConfig REQUIRED): This ensures that pkg-config is available.
- pkg_check_modules(LIBPNG REQUIRED libpng): This calls pkg-config to find the libpng library and sets variables like LIBPNG_INCLUDE_DIRS, LIBPNG_LIBRARIES, and LIBPNG_CFLAGS. These variables store the necessary information to configure the build.
- target_include_directories() and target_link_libraries(): We link the required flags and libraries to our project using the variables set by pkg_check_modules().

6.4.4 Using find_library() to Find Libraries

While pkg-config is helpful for retrieving metadata, CMake also has the built-in find_library() function for locating libraries in specified directories. This command searches for a library by name and returns the full path to the library file. It's particularly useful when the library is installed but doesn't have a .pc file for pkg-config to find.

The syntax for find_library() is as follows:

```
find_library(<VAR> name [path1 path2 ...])
```

Where:

- <VAR>: The variable that will store the path to the found library.
- name: The name of the library (without any prefixes or extensions, e.g., m for libm.so).
- [path1 path2 ...]: Optional search directories to look for the library.

Example: Finding libm (Mathematics Library)

Here's how you can use find_library() to locate the mathematics library libm:

```
find_library(MATH_LIB m)
if(MATH_LIB)
    message(STATUS "Found libm at ${MATH_LIB}")
else()
    message(STATUS "libm not found")
endif()

# Add executable and link the found library
add_executable(my_project main.cpp)
target_link_libraries(my_project PRIVATE ${MATH_LIB})
```

In this example:

- find_library(MATH_LIB m): Searches for libm (the math library).
- target_link_libraries(my_project PRIVATE ${MATH_LIB}): If libm is found, it is linked with the project.

6.4.5 Combining pkg-config and find_library()

In some cases, you might want to use both pkg-config and find_library() together. This is common when a library is available through pkg-config, but its dependencies are not declared or need to be manually located.

Here's an example of using both pkg-config and find_library() for a project that depends on the libpng library (via pkg-config) and the libm math library (using find_library()):

```
# Enable pkg-config
find_package(PkgConfig REQUIRED)

# Find libpng using pkg-config
pkg_check_modules(LIBPNG REQUIRED libpng)

# Find libm using find_library()
find_library(MATH_LIB m)

# Create executable
add_executable(my_project main.cpp)

# Include directories for libpng
target_include_directories(my_project PRIVATE ${LIBPNG_INCLUDE_DIRS})

# Link libraries
target_link_libraries(my_project PRIVATE ${LIBPNG_LIBRARIES} ${MATH_LIB})
```

Explanation:

- pkg_check_modules(LIBPNG REQUIRED libpng): Locates libpng using pkg-config.
- find_library(MATH_LIB m): Locates libm using the find_library() function.
- target_link_libraries(): Links both libpng and libm to the project.

This approach is useful for combining the strengths of both methods: pkg-config for easily finding well-supported libraries and find_library() for libraries that don't provide pkg-config support or when you need to locate additional dependencies manually.

6.4.6 Best Practices for Using pkg-config and find_library()

- Use find_package() with PkgConfig when available: CMake has built-in support for pkg-config via the PkgConfig module, so use find_package(PkgConfig REQUIRED) whenever possible to handle external dependencies more elegantly.
- Set CMAKE_PREFIX_PATH for Custom Locations: If your libraries are installed in non-standard locations, you can set the CMAKE_PREFIX_PATH to help pkg-config or find_library() locate the libraries.

```
cmake -DCMAKE_PREFIX_PATH=/path/to/custom/libs ..
```

- Check for Dependency Availability: Always verify that the required dependencies are found before proceeding with the build. You can use find_package() or check the results of find_library() to ensure everything is in place.

```
if(NOT LIBPNG_FOUND)
    message(FATAL_ERROR "libpng not found!")
endif()
```

- Use find_package() with explicit version requirements: When relying on external dependencies, always specify the minimum version to ensure compatibility.

```
find_package(PkgConfig REQUIRED)
pkg_check_modules(LIBPNG REQUIRED libpng>=1.6)
```

6.4.7 Conclusion

Integrating pkg-config with CMake using find_package() and pkg_check_modules() provides a convenient way to manage external dependencies that are commonly available on Linux and other Unix-like systems. It allows CMake to automatically retrieve the necessary information about external libraries, making it easier to link them into your project. By combining pkg-config and find_library(), you gain flexibility in managing both well-known libraries with .pc files and more generic libraries that don't provide such support.

6.5 Working with vcpkg and Conan for Package Management

6.5.1 Introduction

Managing external dependencies efficiently is an essential part of any C++ project, particularly when working with larger, more complex applications. While find_package(), pkg-config, and FetchContent provide robust solutions for handling dependencies, tools like vcpkg and Conan bring a new level of convenience and integration when dealing with third-party libraries. These modern package managers simplify the installation, management, and integration of C++ libraries into your CMake projects.

In this section, we'll dive deep into how to integrate vcpkg and Conan with CMake, providing you with powerful tools for dependency management, versioning, and cross-platform builds. We will cover each tool's installation, configuration, and usage within CMake.

6.5.2 What are vcpkg and Conan?

vcpkg

vcpkg is a cross-platform C++ package manager that simplifies library management by providing an easy-to-use interface for installing and integrating third-party libraries into C++ projects. It works on Windows, Linux, and macOS, making it a versatile tool for managing dependencies in CMake-based projects.

Key features of vcpkg include:

- Cross-platform: It supports Windows, macOS, and Linux, ensuring that your dependencies can be handled uniformly across platforms.

- Precompiled Binaries: Many libraries in vcpkg come with precompiled binaries, reducing the need for building dependencies from source.
- CMake Integration: vcpkg integrates seamlessly with CMake, allowing it to automatically manage dependencies for your project.

Conan

Conan is another popular C++ package manager that focuses on providing a decentralized, user-centric way of managing dependencies. Unlike vcpkg, which uses a central repository, Conan supports multiple repositories and gives developers the flexibility to host and manage their own packages.

Key features of Conan include:

- Decentralized: Conan allows developers to share and distribute packages across various repositories, providing flexibility.
- Package Versioning: Conan helps manage dependencies with version control, ensuring consistency and reproducibility in your builds.
- CMake Integration: Conan also integrates well with CMake, automating the process of finding and linking packages.

6.5.3 Integrating vcpkg with CMake

vcpkg simplifies the process of installing libraries and managing them for CMake-based projects. Below is a detailed guide on how to use vcpkg in your CMake workflow.

Installing vcpkg

1. Clone the vcpkg repository: First, clone the vcpkg repository from GitHub:

```
git clone https://github.com/microsoft/vcpkg.git
```

2. Bootstrap the vcpkg build system: Inside the vcpkg directory, run the bootstrap script to build the package manager:

   On Windows:

```
.\bootstrap-vcpkg.bat
```

   On Linux/macOS:

```
./bootstrap-vcpkg.sh
```

3. Install packages with vcpkg: Use vcpkg to install C++ libraries. For example, to install the fmt library:

```
./vcpkg install fmt
```

Integrating vcpkg with CMake

To make vcpkg work with your CMake project, you need to set the appropriate environment variable and tell CMake where vcpkg is located.

1. Set the CMake Toolchain File: When invoking CMake, specify the vcpkg toolchain file to let CMake know that you want it to use vcpkg for package management:

```
cmake
↪  -DCMAKE_TOOLCHAIN_FILE=<path_to_vcpkg>/scripts/buildsystems/vcpkg.cmake
↪  ..
```

Replace <path_to_vcpkg> with the actual path to your vcpkg installation.

2. Linking Installed Libraries: After installing a package using vcpkg, you can link it to your project using find_package() or target_link_libraries() as you would with any other CMake-managed library.

   For example, after installing fmt with vcpkg, you can use:

```
find_package(fmt REQUIRED)
target_link_libraries(my_project PRIVATE fmt::fmt)
```

   vcpkg automatically configures find_package() for most libraries, so you don't have to manually specify paths or flags.

6.5.4 Integrating Conan with CMake

Conan is another excellent option for C++ package management, providing an alternative approach to managing dependencies with a focus on flexibility, versioning, and decentralized package repositories. Let's go over how to integrate Conan with your CMake project.

Installing Conan

1. Install Conan: You can install Conan via pip (Python's package manager):

```
pip install conan
```

2. Initialize a conanfile.txt or conanfile.py: In your project's root directory, create a conanfile.txt or conanfile.py file. This file specifies which dependencies your project requires. A basic conanfile.txt for a project that depends on fmt might look like:

```
[requires]
fmt/8.0.1

[generators]
cmake
```

   Here, fmt/8.0.1 specifies the version of the fmt library required.

Using Conan with CMake

To use Conan with CMake, you need to configure the build system to generate the appropriate CMake files.

1. Install Dependencies: Use conan install to install the dependencies listed in your conanfile.txt or conanfile.py:

```
conan install . --build=missing
```

   The --build=missing flag ensures that any missing packages are built from source.

2. Link with CMake: After installing dependencies, you need to tell CMake to use the Conan-generated configuration files. Conan generates CMake files that set up environment variables and paths for the dependencies.

In your CMakeLists.txt, add the following line to include the Conan-generated files:

```
include_directories(${CMAKE_BINARY_DIR}/conan_includes)
```

Alternatively, if you are using the cmake generator, include the Conan CMake module:

```
include(${CMAKE_BINARY_DIR}/conan.cmake)
```

3. Link the Libraries: With the dependencies installed and configured by Conan, you can link the libraries to your project in the usual way:

```
target_link_libraries(my_project PRIVATE fmt::fmt)
```

6.5.5 Comparing vcpkg and Conan

Both vcpkg and Conan are popular and powerful tools for package management in C++. However, they have different strengths and use cases:

Feature	vcpkg	Conan
Package Management	Centralized (single repository)	Decentralized (supports multiple repositories)
Cross-Platform	Yes (Windows, macOS, Linux)	Yes (Windows, macOS, Linux)
CMake Integration	Seamless integration via toolchain file	Seamless integration with conan.cmake

Feature	vcpkg	Conan
Precompiled Binaries	Many packages have precompiled binaries	Many packages, but some require building from source
Version Control	Manages versions via vcpkg tool	Full version control with multiple repositories
Package Hosting	Managed by Microsoft (official repository)	Community-driven and user-defined repositories

When to Use vcpkg:

- If you are working on a cross-platform C++ project and want a simple, streamlined way to manage dependencies.
- If you need precompiled binaries for faster builds.
- If you prefer to work with a single, centralized repository.

When to Use Conan:

- If you need decentralized package management and control over your package repository.
- If you need detailed version control and fine-grained control over the dependencies of your project.
- If your project has complex or custom dependencies that need to be versioned carefully.

6.5.6 Conclusion

Both vcpkg and Conan provide excellent solutions for managing external dependencies in CMake-based C++ projects. While vcpkg offers a simpler, centralized approach with easy integration into CMake, Conan offers more flexibility, version control, and decentralized package management. Depending on your project's needs, either tool can significantly simplify dependency management and ensure a more consistent, reproducible build process.

By leveraging the power of these package managers, you can focus more on developing your application and less on handling dependency configurations.

Chapter 7

Working with CMake GUI & CLI

7.1 Using the CMake GUI on Windows and Linux

CMake is a cross-platform tool that provides users with the ability to manage the build process of software projects in a compiler-independent manner. One of the most convenient ways to interact with CMake is through its graphical user interface (GUI), which offers a user-friendly way to configure projects, set up build paths, and generate platform-specific build files. This section focuses on how to use the CMake GUI on both Windows and Linux platforms, giving you insights into its interface and functionality.

7.1.1 Overview of the CMake GUI

The CMake GUI is an interactive application that provides a straightforward way to configure and generate CMake project files. It eliminates the need to remember

specific commands or command-line options by offering an intuitive interface. With CMake GUI, you can:

- Configure Projects: Set CMake variables, choose generators, and set specific paths for building your project.
- Generate Build Files: Create platform-specific build files (e.g., Makefiles, Visual Studio project files).
- Troubleshoot Configuration Errors: View error messages and warnings related to the project configuration.

7.1.2 Installing the CMake GUI

Before you can use the CMake GUI, you'll need to install it. Here are the installation steps for both Windows and Linux.

1. Installing CMake GUI on Windows
    1. Download the CMake Installer:
        - Go to the CMake official download page and download the latest version of the CMake installer for Windows (e.g., cmake-x.y.z-win64-x64.msi).
    2. Run the Installer:
        - Launch the downloaded .msi installer.
        - Follow the on-screen instructions. Ensure that the option to add CMake to the system PATH is checked. This will allow you to use CMake from both the GUI and the command line.
    3. Verify Installation:

- Once the installation completes, open the CMake GUI application from the Start menu or by searching for “CMake” in the Windows search bar.
- If you installed it correctly, the CMake GUI should open, and you'll be ready to begin using it.

2. Installing CMake GUI on Linux

    1. Install via Package Manager:
        - On most Linux distributions, CMake is available through the system's package manager. Use the appropriate command for your distribution.

    For Ubuntu or Debian:

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

    For Fedora:

    ```
    sudo dnf install cmake-qt-gui
    ```

    2. Verify Installation:
        - After installation, you can open the CMake GUI by running the following command in a terminal:

    ```
    cmake-gui
    ```

    This should launch the CMake GUI application, ready to configure and generate build files.

7.1.3 The CMake GUI Interface

Upon launching the CMake GUI, the interface will look slightly different on Windows and Linux but functions similarly across both platforms. Below is a breakdown of the interface elements:

1. Initial Configuration

   When you first open the CMake GUI, you will be prompted to provide paths for both your source directory (where the CMakeLists.txt file is located) and the binary directory (where the build files will be generated).

   - Source Directory: This is the location of the source code of your project, typically the folder containing your CMakeLists.txt.
   - Binary Directory: This is the folder where the generated build files will be stored. It's common to create a separate build folder within your project's root directory for this purpose.

2. Setting CMake Variables

   In the CMake GUI, the main window consists of several buttons and areas where you can input information:

   - CMake Cache Entries
     : On the left side of the GUI, there is a list of variables that CMake uses for the configuration. These variables can be set manually, or CMake can automatically populate them based on the system configuration. For instance:
     - CMAKE_BUILD_TYPE: Specifies the build type (e.g., Debug, Release).
     - CMAKE_INSTALL_PREFIX: Defines the installation directory after building the project.

  - CMAKE_CXX_COMPILER: Specifies the C++ compiler to use.
- Edit Variables: You can double-click on any variable in the list to modify its value. If you're unsure about a variable, you can click the Help button for a brief description.

3. Generating Build Files

   Once you have configured your project in the GUI, you can generate the appropriate build files for your platform. The “Configure” and “Generate” buttons in the CMake GUI allow you to:

   1. Configure: CMake will analyze the source and binary directories, attempt to detect your system's properties, and populate the GUI with relevant values (e.g., compilers, libraries). If there are issues or warnings, they will be displayed in the output window.
   2. Generate: After configuration, you can press the Generate button to create the build files. This step will generate files such as:
      - Makefiles (on Linux/macOS)
      - Visual Studio project files (on Windows)
      - Ninja build files (if Ninja is selected as the generator)

4. Advanced Options

   The CMake GUI also allows you to specify advanced settings:

   - Choose a Generator
     : CMake supports different generators for different platforms, such as:
     - On Windows, you may choose Visual Studio or MinGW.
     - On Linux, you may select Unix Makefiles or Ninja.
   - CMake Log: The log section provides output messages that help debug or verify configurations.

5. Building the Project

   While the CMake GUI does not directly build the project, it helps set up the build environment. After generating the build files, you will typically use your chosen build system (e.g., make, Visual Studio) to actually compile and link your project. For instance:

   - On Linux, you would navigate to the build directory and run make or ninja (depending on the generator you chose).
   - On Windows, you would open the generated Visual Studio solution and build from there.

7.1.4 Using the CMake GUI on Linux

Though the CMake GUI is very similar on both platforms, some Linux-specific behavior should be noted:

- On Linux, the CMake GUI relies on the Qt framework for its interface, so having the appropriate Qt libraries installed is essential.
- The Generate button will allow you to choose between different build systems, such as Unix Makefiles or Ninja.

7.1.5 Using the CMake GUI on Windows

On Windows, the process is mostly the same as on Linux, with a few key differences:

- The most notable difference is the choice of Visual Studio generators, which allow you to create Visual Studio project files directly from the GUI. Once

you generate these files, you can open and build them inside the Visual Studio IDE.

- If you're using MinGW or another alternative to Visual Studio, the CMake GUI will configure for those environments as well.

7.1.6 Troubleshooting

The CMake GUI offers helpful error messages if something goes wrong during configuration or generation:

- Missing Dependencies: If CMake can't find required libraries or tools, it will show warnings or errors. You can often resolve these by specifying the correct paths in the GUI or ensuring that necessary software is installed.
- Configuration Errors: Errors during configuration (e.g., wrong compiler or mismatched versions) will be displayed in the log output. You can fix these by adjusting the configuration or modifying the CMakeLists.txt file.

7.1.7 Conclusion

Using the CMake GUI is an effective way to simplify the process of configuring and building C++ projects. Whether you're working on Windows or Linux, the GUI provides an intuitive interface that guides you through configuring your project, setting CMake variables, and generating platform-specific build files. With its built-in error messages and advanced options, it is an excellent tool for both beginners and advanced developers alike.

7.2 Command-Line Interface (CLI) with CMake

While the CMake GUI is a powerful tool for configuring and generating build files with a graphical interface, many advanced users prefer the Command-Line Interface (CLI) because of its flexibility, automation capabilities, and faster workflows. The CMake CLI allows you to execute the same tasks that you would with the GUI, but through terminal commands, making it easier to integrate CMake into scripts, continuous integration (CI) pipelines, and large-scale automated builds.

In this section, we will walk through how to use CMake's Command-Line Interface to configure, generate, and build CMake projects. We'll also cover important commands and options that you can use to streamline your build process.

7.2.1 Overview of the CMake CLI

The CMake CLI is a text-based tool that uses commands to control how CMake configures and generates build files. The basic syntax of the cmake command is as follows:

```
cmake [options] <path-to-source>
```

Where:

- options are various flags and parameters to control the configuration.
- <path-to-source> is the directory containing the project's CMakeLists.txt file, which CMake uses to configure the project.

You can use the CMake CLI on both Windows and Linux (and other Unix-based systems), and its commands are consistent across platforms, though the underlying build system may differ.

7.2.2 Basic CMake Command-Line Workflow

Let's go through a typical CMake CLI workflow, from configuring the project to generating build files and building the project.

- Step 1: Creating a Build Directory

  In most CMake workflows, it's a best practice to create a separate build directory outside of the source directory. This keeps the source directory clean and avoids mixing source files with build artifacts. From the root of your project directory, create a new build directory:

  ```
  mkdir build
  cd build
  ```

  This step ensures that all generated files (such as Makefiles or Visual Studio project files) are placed in the build directory rather than in the source directory.

- Step 2: Running CMake to Configure the Project

  After navigating to the build directory, you can run the cmake command to configure your project and generate the appropriate build files. The general syntax is:

  ```
  cmake <path-to-source>
  ```

For example, if your project's source code is in the parent directory, you would run:

```
cmake ..
```

When you run this command, CMake will:

- Find the CMakeLists.txt file in the specified source directory.
- Detect the environment and configure variables (such as compilers, libraries, and system settings).
- Display a summary of the configuration process in the terminal, listing paths, flags, and other settings.

- Step 3: Specifying Build Types and Generators

  You can also pass additional options to the cmake command to control various aspects of the configuration. Two important options are:

  - CMAKE_BUILD_TYPE: Specifies the build type, which typically defines whether you want a Debug or Release build.
    For example:

    ```
    cmake -DCMAKE_BUILD_TYPE=Release ..
    ```

    This sets the build type to Release, optimizing the code for performance.
  - CMAKE_GENERATOR: Defines the build system generator, such as Makefiles, Ninja, or Visual Studio.
    Example for Unix Makefiles:

    ```
    cmake -G "Unix Makefiles" ..
    ```

    Example for Ninja:

```
cmake -G "Ninja" ..
```

On Windows, you could specify a Visual Studio version generator:

```
cmake -G "Visual Studio 16 2019" ..
```

- Step 4: Generating Build Files

  Once you run the cmake command with the necessary options, CMake will generate the build files in the build directory. The generated files are specific to the platform and generator you've selected (e.g., Makefiles, Visual Studio solution files, etc.).

7.2.3 Important CMake CLI Commands and Options

CMake provides a range of commands and options for advanced configurations. Here are some key ones that you will likely encounter when using the CLI:

1. cmake --build

   After configuring the project with CMake, you can invoke the build process directly from the CLI using the cmake --build command. This command allows you to build your project using the same configuration you generated earlier, without needing to switch to a separate build tool or IDE.

   For example, to build the project:

   ```
   cmake --build .
   ```

   This will use the default generator (e.g., Makefiles or Ninja) to build the project in the current directory.

You can also specify the number of parallel jobs to run during the build process (useful for speeding up builds, especially in large projects):

```
cmake --build . --parallel 4
```

2. cmake -D (Setting Cache Variables)

   You can set specific cache variables directly from the CLI using the -D flag. This allows you to override default values defined in CMakeLists.txt or other configuration files.

   For example, to set a custom installation path:

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

   Another example is setting the compiler:

   ```
   cmake -DCMAKE_CXX_COMPILER=g++-9 ..
   ```

3. cmake --install

   Once your project is built, you can use the cmake --install command to install the project into the specified location. This is especially useful for projects that require installation steps, such as libraries or applications that need to be copied to a system-wide location.

   For example:

   ```
   cmake --install .
   ```

   This will install the project using the installation paths defined during the configuration process.

4. cmake --version

   To check the installed version of CMake, you can use the following command:

   ```
   cmake --version
   ```

   This is useful when you want to confirm that you are using the correct version of CMake, particularly in environments with multiple versions.

5. cmake --help

   To view a list of available options and commands for CMake, you can use the help command:

   ```
   cmake --help
   ```

   This provides a comprehensive list of CMake commands, options, and usage instructions.

7.2.4 Advanced CMake CLI Usage

For more advanced workflows, the CMake CLI provides several additional features to support complex build processes.

1. Using CMake Presets

   CMake allows you to define build configurations using preset files, making it easier to standardize configurations across different systems or team members. You can use CMakePresets.json and CMakeUserPresets.json to define options like compilers, build types, and toolchains.

   For example, you can run CMake with a preset configuration using:

```
cmake --preset mypreset
```

This reduces the need to manually set multiple flags and ensures consistency in how your project is configured.

2. Running CMake from Scripts

   One of the biggest advantages of using the CMake CLI is the ability to automate the configuration and build process by including CMake commands in scripts. For example, you can write a shell script to configure and build your project automatically:

```
#!/bin/bash

# Create a build directory
mkdir -p build
cd build

# Configure the project
cmake -DCMAKE_BUILD_TYPE=Release -G "Unix Makefiles" ..

# Build the project
cmake --build .

# Optionally, install the project
cmake --install .
```

Such automation is especially useful in CI/CD pipelines and for teams working on large projects with many dependencies.

7.2.5 Troubleshooting Common Issues

While the CMake CLI is powerful and flexible, there are a few common issues you might encounter:

- Missing Dependencies: If CMake cannot find a required library or tool, it will display an error message. In such cases, ensure the necessary software is installed and available in your PATH, or specify custom paths using the -D option.
- Build Type Issues: If you're not specifying the CMAKE_BUILD_TYPE, CMake might default to an empty configuration, leading to unexpected results. Always explicitly set the build type if needed.
- Generator Errors: If you specify an invalid generator (e.g., Visual Studio version mismatch), CMake will return an error. Ensure that the chosen generator matches your system's available build tools.

7.2.6 Conclusion

The Command-Line Interface (CLI) provides CMake users with the flexibility to configure, generate, and build projects directly from the terminal. This is especially useful for automating builds, integrating with continuous integration systems, and streamlining workflows. Mastery of the CMake CLI is a valuable skill for C++ developers, as it gives them complete control over the build process while maintaining portability across platforms. Through the combination of simple commands and powerful flags, the CLI allows for highly customizable and efficient project management.

7.3 Configuring Different Generators (-G "Ninja", -G "Unix Makefiles", ...)

One of the most powerful features of CMake is its ability to generate build files for different platforms, compilers, and build systems. This flexibility is enabled by CMake's use of generators, which control how the build process is handled once CMake has configured the project. A generator specifies the type of build system (e.g., Makefiles, Visual Studio project files, Ninja build files) that CMake will use to create the final output.

In this section, we will explore how to configure different generators in CMake, such as -G "Ninja", -G "Unix Makefiles", and others. We'll discuss when and why you might choose one generator over another, and how to properly configure them using the command-line interface (CLI).

7.3.1 What is a Generator in CMake?

A generator in CMake is a template that defines how build files will be created based on the current platform and build tools. When you run the cmake command with a specific generator, CMake uses that generator to produce the build system files that your chosen platform requires.

For instance:

- On Linux, CMake may generate Makefile files that can be processed by the make tool.
- On Windows, CMake might generate Visual Studio project files.
- On macOS, it might generate Xcode project files or Makefiles.

The -G option in the cmake command is used to specify which generator to use. You can also choose from a wide variety of generators depending on your project's needs.

7.3.2 Common Generators in CMake

Here are some of the most common CMake generators, how they work, and when you might want to use them.

1. -G "Unix Makefiles"

   The Unix Makefiles generator creates traditional Makefiles that can be processed by the make build tool. This is one of the most widely used generators on Linux and other Unix-like systems, including macOS.

   Use Case:

   - When working on Linux, BSD, or macOS systems with make as the build tool.
   - For projects that do not require a specific IDE and prefer the command-line interface.

   Example:

   ```
   cmake -G "Unix Makefiles" ..
   ```

   This command will generate a Makefile that can later be used to build the project with make:

   ```
   make
   ```

   Advantages:

- Works well on Unix-like systems.
- Supported by nearly all distributions and environments.
- Well-suited for small-to-medium-sized projects.

Disadvantages:

- Slower compared to some other build tools like Ninja, especially for large projects.
- Lacks advanced parallelization features found in other build systems.

2. -G "Ninja"

   The Ninja generator is designed for fast builds and provides better parallelization than traditional make-based builds. Ninja is often preferred for large projects where speed is a priority. It also has a simpler, more efficient design than Make, which leads to faster incremental builds.

   Use Case:

   - When building large projects that require fast incremental builds.
   - Projects that need to leverage parallel builds for better performance.
   - When working with CI/CD pipelines that prioritize build speed.

   Example:

   ```
   cmake -G "Ninja" ..
   ```

   Once the project is configured with Ninja, you can build it using:

   ```
   ninja
   ```

   Advantages:

   - Faster builds than Unix Makefiles, especially on large projects.

- Efficient parallel builds by default.
- Simple and easy to use with no extra dependencies.

Disadvantages:

- Requires Ninja to be installed separately. While it is lightweight, it is another tool to install and maintain.
- May not be as familiar to developers who are used to make or other traditional build systems.

3. -G "Visual Studio <version>"

   On Windows, CMake supports generating project files for Microsoft's Visual Studio IDE. By specifying a version number (e.g., Visual Studio 2019), CMake will generate .sln (solution) and .vcxproj (project) files that can be opened directly in Visual Studio. This generator is particularly useful for developers who want to take advantage of Visual Studio's graphical interface, debugging tools, and other features.

   Use Case:

   - When working on Windows and targeting Visual Studio as the development environment.
   - Developers who prefer using Visual Studio for debugging, profiling, and GUI-based project management.

   Example: For Visual Studio 2019:

   ```
   cmake -G "Visual Studio 16 2019" ..
   ```

   Advantages:

   - Provides seamless integration with Visual Studio.

- Supports advanced IDE features like debugging, code navigation, and profiling.
- Can be used for both C++ and C# projects in Visual Studio.

Disadvantages:

- Only works on Windows.
- Requires Visual Studio to be installed.
- The build process is slower compared to other generators like Ninja or Makefiles.

4. -G "Xcode"

   The Xcode generator creates an Xcode project on macOS, enabling developers to build their projects using the Xcode IDE. This is an ideal choice for macOS developers who want to integrate with Xcode's graphical interface, testing tools, and simulator support for iOS/macOS applications.

   Use Case:

   - When working on macOS and targeting the Xcode IDE for development and testing.
   - Ideal for iOS and macOS applications, especially if you need to use the Xcode interface for design, profiling, or app store submissions.

   Example:

   ```
   cmake -G "Xcode" ..
   ```

   Advantages:

   - Direct integration with Xcode, which provides a rich development environment.

- Full support for iOS/macOS-specific tools such as simulators, testing, and app submissions.
- Useful for cross-platform C++ projects that are targeting Apple's ecosystem.

Disadvantages:

- Only available on macOS.
- Not suitable for non-Apple platforms.

5. -G "MinGW Makefiles"

   The MinGW Makefiles generator is used when you are building on Windows with the MinGW (Minimalist GNU for Windows) toolchain. MinGW provides a set of GNU tools (including GCC) to enable a Unix-like development environment on Windows.

   Use Case:

   - When building on Windows using the MinGW toolchain.
   - If you prefer to work with make rather than Visual Studio or other IDEs.

   Example:

   ```
   cmake -G "MinGW Makefiles" ..
   ```

   Advantages:

   - Allows using make on Windows with the MinGW toolchain.
   - Suitable for cross-platform C++ development targeting both Windows and Unix-like systems.

   Disadvantages:

   - Requires MinGW to be installed and properly configured.

- Not as widely used as Visual Studio on Windows, so you may encounter compatibility issues in certain environments.

7.3.3 Choosing the Right Generator

Selecting the right generator depends on several factors, including:

- Platform: Different generators are available for different platforms (Windows, Linux, macOS).
- Toolchain: If you are using a specific build tool (e.g., Ninja, Make, or Visual Studio), select the corresponding generator.
- Project Size: For larger projects, generators like Ninja are preferred due to their faster incremental builds and parallelization features.
- IDE Preferences: If you prefer working in a specific IDE, such as Visual Studio or Xcode, select the generator that integrates with that IDE.

Example Scenarios for Choosing a Generator

- Linux/Unix Project: If you are on a Linux or Unix system and working with a command-line environment, you would typically use -G ”Unix Makefiles” or -G ”Ninja”.
- Windows Developer: On Windows, if you are using Visual Studio, you would use -G ”Visual Studio 16 2019”. If you are using a different build system like MinGW, you would use -G ”MinGW Makefiles”.
- macOS Development: On macOS, if you're building a project for Xcode or iOS/macOS, you would use -G ”Xcode”. For non-Xcode workflows, -G ”Unix Makefiles” or -G ”Ninja” might also work.

7.3.4 Specifying Multiple Generators

In some cases, you may need to test or build your project with different generators. You can simply run CMake with different -G flags, specifying a different generator for each configuration. Keep in mind that some build systems (such as Visual Studio) may create multiple configuration files (e.g., solution files, project files) for different build types (Debug, Release).

7.3.5 Troubleshooting Generator Issues

Sometimes, you may run into issues while configuring CMake with a specific generator. Some common issues include:

- Incorrect Generator Syntax: Make sure the generator string is entered correctly. CMake is very specific about the exact names of the generators.
- Missing Tools: Some generators, like Ninja or MinGW, require the corresponding tools to be installed. If CMake cannot find the necessary toolchain, it will generate an error message.
- Generator Compatibility: Some generators might not work well on certain platforms or with specific versions of CMake. Always refer to the CMake documentation to verify compatibility.

7.3.6 Conclusion

CMake's ability to support multiple generators makes it a highly flexible tool for managing cross-platform and cross-toolchain projects. Whether you are targeting traditional Makefiles, Ninja for speed, Visual Studio for IDE integration, or Xcode

for macOS/iOS development, CMake makes it easy to configure and generate the right build files for your platform and toolchain. By understanding the various generator options and when to use them, you can streamline your build process and improve productivity in your C++ projects.

7.4 Managing Options and Settings with ccmake

While CMake offers powerful configuration capabilities through the command-line interface (CLI) and graphical user interface (GUI), there's a middle ground: ccmake. This tool is a terminal-based user interface that combines the simplicity of a GUI with the power and flexibility of the command line. It is particularly useful when working in environments where a GUI is not available or where you prefer a text-based interface.

In this section, we'll explore ccmake, how it works, how to manage project settings and options through it, and how it compares to both the full CMake GUI and the command-line interface.

7.4.1 What is ccmake?

ccmake stands for CMake curses interface, and it's a command-line tool that provides an interactive interface to configure CMake options for a project. The main purpose of ccmake is to provide a text-based configuration interface that can be run inside a terminal. This tool allows users to:

- View and set CMake cache variables.
- Modify project settings such as build types, installation paths, and optional features.
- Save and apply changes interactively without needing to edit configuration files manually.

Unlike the full graphical CMake GUI, ccmake works entirely in the terminal, making it useful for remote development or environments where a GUI is

impractical. It's also often preferred by advanced users who are comfortable with terminal-based workflows but still want to take advantage of the interactive nature of CMake's configuration process.

7.4.2 Installing and Using ccmake

ccmake is included as part of the CMake distribution, so it should be available as long as you have CMake installed on your system. To check if ccmake is installed, simply type the following in the terminal:

```
ccmake --version
```

If the tool is not installed, you may need to install CMake via your system's package manager. For example:

- On Ubuntu (or other Debian-based systems):

  ```
  sudo apt-get install cmake
  ```

- On macOS (using Homebrew):

  ```
  brew install cmake
  ```

- On Windows, ccmake is included with the standard CMake installer.

7.4.3 Basic Workflow with ccmake

Using ccmake is straightforward and involves running it in the build directory where CMake has been previously configured. Here's the basic workflow:

1. Navigate to the Build Directory: It's a best practice to use an out-of-source build directory (to keep source files clean). If you haven't already created a build directory, do so:

```
mkdir build
cd build
```

2. Run ccmake: Once you're inside the build directory, run the ccmake command, pointing to the project's source directory (typically the parent directory containing the CMakeLists.txt file).

```
ccmake ..
```

   This command will start the interactive configuration interface, displaying the current configuration options and settings for your project.

3. Navigate the ccmake Interface: The ccmake interface uses keyboard navigation. Once the interface appears, you'll see a list of cache variables with their current values. You can use the following keys to interact with ccmake:
   - Arrow keys: Navigate through the list of options.
   - Enter: Select an option to edit its value.
   - Type the value: Modify the value for the selected option (for variables, paths, etc.).
   - c: Configure the project (this updates the cache and displays any errors or warnings).
   - g: Generate the build files after configuration is complete.
   - q: Quit the ccmake interface.

   This interface is highly interactive, making it easy to change settings without needing to manually edit configuration files.

7.4.4 Managing Cache Variables in ccmake

A key feature of ccmake is the ability to view and modify the CMake cache variables. These variables control the configuration of the project, such as the paths to dependencies, compiler flags, or build types. The cache variables are saved in the CMakeCache.txt file in your build directory.

In ccmake, the cache variables are displayed in a table-like structure, with each row representing a variable, its current value, and its description. Some common types of cache variables include:

- Boolean variables: These represent on/off or true/false options. For example, enabling or disabling specific features or modules in the project.
- String variables: These represent paths, filenames, or any other string value, such as installation directories or compiler paths.
- Path variables: These are used for specifying the location of libraries or tools that your project depends on.

You can change the values of these variables using the arrow keys to navigate to the desired row, pressing Enter to edit the value, and typing the new value.

Example: Let's say your project has an option to enable or disable a specific feature, and it is defined as a Boolean cache variable, ENABLE_FEATURE_X. If the default value is OFF, you can change it to ON using ccmake:

1. Scroll down to the ENABLE_FEATURE_X variable.
2. Press Enter to select it.
3. Change the value from OFF to ON.
4. Press Enter again to save the change.

3. Important ccmake Features

    1. Configuration with ccmake

       Once you've modified the desired options, you can trigger a reconfiguration by pressing c. This will run CMake to re-evaluate the configuration and update the CMakeCache.txt file accordingly. During this process, CMake will check for any errors or missing dependencies and will notify you if any problems arise.

       For example:

       - If CMake detects missing dependencies or invalid paths, it will alert you with an error message.
       - If everything is valid, CMake will proceed and update the cache.

    2. Generating Build Files with ccmake

       After configuration is complete, you can use the g key to generate the appropriate build files for your project. These files will be created in the build directory, and they will be tailored to the generator and settings you've selected.

       For example, if you are using the Unix Makefiles generator, running g will create a Makefile that you can later use with the make tool to build the project.

    3. Viewing CMake Warnings and Errors

       ccmake displays useful warnings and errors during the configuration process. If there are any issues with your settings or missing dependencies, CMake will provide a message in the terminal window, helping you quickly identify and resolve configuration problems. This feature makes ccmake a great option for troubleshooting build problems interactively.

4. Advanced Options in ccmake

   For more advanced workflows, ccmake also allows you to:

   - Set cache entries directly: If you know the specific cache variable you want to change, you can type its name and value directly in the interface.
   - Clear cached variables: If you want to reset the configuration to its initial state, you can delete or clear the cache variables.
   - Use advanced mode: By pressing the t key, you can switch to advanced mode, which reveals additional configuration options that are typically hidden. This is useful for more complex projects or when you need to fine-tune the build configuration.

7.4.5 Differences Between ccmake, CMake GUI, and CLI

While both ccmake and the graphical CMake GUI serve similar purposes, they have distinct use cases and benefits:

- ccmake is a text-based tool that works within the terminal, ideal for environments without graphical interfaces. It's best suited for users who prefer terminal-based workflows but still want interactive configuration.
- CMake GUI provides a more visually intuitive interface, ideal for those who prefer using a mouse and require a more user-friendly experience.
- CMake CLI is fully command-line based, offering the highest level of automation and flexibility, particularly for integrating CMake into scripts or CI/CD systems.

7.4.6 Example ccmake Workflow

Let's go through an example of configuring a simple project using ccmake:

1. Create a build directory:

```
mkdir build
cd build
```

2. Run ccmake to configure the project:

```
ccmake ..
```

3. Modify some options:
   - Use the arrow keys to select CMAKE_BUILD_TYPE and change it from Debug to Release.
   - Enable or disable certain features by modifying Boolean cache variables (e.g., ENABLE_TESTING).
4. Configure the project:
   - Press c to configure and update the cache.
5. Generate the build files:
   - Press g to generate the build files (e.g., Makefiles or Ninja files).
6. Build the project: After generating the files, you can use the make or ninja command (depending on your generator) to build the project.

7.4.7 Conclusion

ccmake provides a convenient, interactive text-based interface for managing and modifying CMake configuration settings. It strikes a balance between the full GUI and the more automated CLI workflows, offering developers a simple yet powerful tool for configuring build options without needing to edit configuration files manually. Whether you're working on a remote server, prefer terminal-based tools, or need a lightweight configuration interface, ccmake is a valuable addition to your CMake toolkit. By mastering ccmake, you can easily fine-tune your project's settings and quickly troubleshoot build configuration issues.

Appendices

Appendix A: CMake Command Reference

This appendix is a comprehensive list of the most commonly used CMake commands, providing a quick reference for when you're working on CMake-based projects.

Key CMake Commands:

1. cmake
   The main command to configure, generate, and build a project.

   ```
   cmake <path-to-source>
   ```

   Example:

   ```
   cmake /path/to/project
   ```

2. add_executable
   Defines an executable target in a CMake project.

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

3. add_library
   Defines a library target (static or shared).

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

4. target_link_libraries
   Links a target with libraries or other targets.

```
target_link_libraries(MyApp MyLibrary)
```

5. find_package
   Searches for and configures external dependencies, such as libraries or tools.

```
find_package(OpenCV REQUIRED)
```

6. include_directories
   Adds directories to the compiler's search path for include files.

```
include_directories(${CMAKE_SOURCE_DIR}/include)
```

7. set
   Sets a variable or cache entry.

```
set(SOURCE_FILES main.cpp app.cpp)
```

8. message
   Displays messages during the configuration process.

```
message(STATUS "Building MyApp with OpenCV")
```

9. install
   Defines installation rules for files and targets.

```
install(TARGETS MyApp DESTINATION /usr/local/bin)
```

10. enable_testing / add_test
    Enables the testing framework and adds tests.

```
enable_testing()
add_test(NAME MyTest COMMAND MyTestExecutable)
```

This appendix allows you to quickly locate command syntax and descriptions, which can be especially helpful when debugging or experimenting with advanced CMake configurations.

Appendix B: CMake Best Practices

This appendix provides a collection of best practices for writing clean, efficient, and maintainable CMakeLists.txt files. These best practices are essential for scaling projects and ensuring that your build system is easy to manage in the long term.

1. Keep CMakeLists.txt Simple and Modular

   Avoid placing all logic in a single CMakeLists.txt file. Use subdirectories to break your project into logical modules. This makes the project easier to maintain and scale.

   ```
   add_subdirectory(src)
   add_subdirectory(tests)
   ```

2. Use Variables for Paths and Repeated Values

   Instead of hardcoding paths and values, store them in variables to avoid duplication and simplify updates.

   ```
   set(MY_LIBRARY_PATH ${CMAKE_SOURCE_DIR}/libs)
   ```

3. Use target_include_directories and target_link_libraries

   For better target management, always use target_include_directories and target_link_libraries for linking dependencies, rather than global commands like include_directories or link_directories.

   ```
   target_include_directories(MyApp PRIVATE ${CMAKE_SOURCE_DIR}/include)
   target_link_libraries(MyApp PRIVATE MyLibrary)
   ```

4. Prefer Modern CMake Syntax

   Use target_* commands instead of global commands whenever possible, as they are more specific and help avoid accidental global settings.

```
target_compile_options(MyApp PRIVATE -Wall)
```

5. Use find_package for External Dependencies

   Whenever possible, rely on CMake's find_package command for managing third-party libraries. This allows your project to work seamlessly with external dependencies across various platforms.

```
find_package(OpenCV REQUIRED)
```

6. Ensure Consistent Formatting

   Follow consistent naming conventions, indentation, and spacing. This makes your CMakeLists.txt files easier to read and maintain.

7. Add Comments for Clarity

   Even though CMake code is relatively readable, adding clear comments will help anyone reviewing or updating the configuration.

```
# Add the main application
add_executable(MyApp main.cpp)
```

8. Handle Build Types Properly

   Ensure you configure different build types (e.g., Debug, Release, RelWithDebInfo) to provide the best development and performance experience.

```
set(CMAKE_BUILD_TYPE ”Release”)
```

Appendix C: CMake Troubleshooting Guide

The troubleshooting guide in this appendix offers solutions to common issues and tips for diagnosing problems during the CMake configuration or build process.

1. Common CMake Errors and Solutions:
    1. Error: "Could not find CMakeLists.txt"
       This error occurs when you run CMake from a directory that does not contain a CMakeLists.txt file. To fix this, ensure you are in the correct directory containing the root CMakeLists.txt.
    2. Error: "No CMAKE_CXX_COMPILER could be found"
       This usually means that CMake cannot find a suitable C++ compiler. Ensure that your system has a valid C++ compiler installed and available in the system path.
    3. Error: "Target already exists"
       This error occurs if a target is defined more than once in your CMakeLists.txt. Check for duplicate calls to add_executable or add_library for the same target name.
    4. Error: "Unknown CMake command"
       If you receive an error about an unknown command, double-check the spelling of the CMake command and verify that the required version of CMake supports the command.
    5. Error: "FindXXX.cmake module not found"
       This error appears when CMake cannot locate a FindXXX.cmake module for a required package. Ensure that the package is installed and that CMake's CMAKE_MODULE_PATH variable is correctly set.
2. General Debugging Tips:

- Use cmake --trace to log all actions taken by CMake during configuration.
- Run CMake with -DCMAKE_VERBOSE_MAKEFILE=ON for verbose output during the build process.
- Look at CMake's cache file (CMakeCache.txt) for stored variables that might be affecting the build.

Appendix D: CMake Project Examples

Here, we provide practical, real-world project examples with detailed CMakeLists.txt files. These examples will help you understand how to structure your CMake-based projects and handle common CMake configurations.

- Example 1: Simple CMake Project

  This example shows how to set up a basic CMake project with a single executable target.

```
cmake_minimum_required(VERSION 3.10)
project(SimpleProject)

add_executable(MyApp main.cpp)
```

- Example 2: Multi-Target Project

  This example demonstrates how to organize a project with multiple targets (executables and libraries).

```
cmake_minimum_required(VERSION 3.10)
project(MultiTargetProject)

add_library(MyLibrary STATIC lib.cpp)
add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE MyLibrary)
```

- Example 3: External Dependency (OpenCV)

  This example shows how to configure CMake to use an external library (OpenCV) via find_package.

```
cmake_minimum_required(VERSION 3.10)
project(OpenCVExample)

find_package(OpenCV REQUIRED)
add_executable(MyApp main.cpp)
target_link_libraries(MyApp PRIVATE ${OpenCV_LIBS})
```

These examples provide a foundation for building larger, more complex CMake projects. You can adapt these templates as needed for your own projects.

Appendix E: CMake Tools and Integrations

This appendix lists various tools and IDE integrations that work seamlessly with CMake to streamline your development process.

1. CMake GUI

   A graphical interface that simplifies the configuration process for projects. It allows you to set variables and configure your project without using the command line.

2. Visual Studio Code (VSCode)

   VSCode has excellent CMake support via the CMake Tools extension, providing integration with CMake commands, build systems, and debugging tools.

3. CLion

   A powerful IDE for C++ development, CLion has native CMake support, which makes it a great choice for managing large C++ projects with CMake.

4. Ninja

   A small build system with a focus on speed, Ninja can be used with CMake to speed up builds and optimize the build process.

5. Continuous Integration Tools (GitHub Actions, GitLab CI, Jenkins)

   These CI tools integrate seamlessly with CMake to automate builds, tests, and deployment pipelines. Configuration is typically handled via .yml files and CMake's CMakeLists.txt setup.

Final Thoughts

The appendices in this book offer essential reference material and troubleshooting advice to support your ongoing CMake journey. By utilizing these resources, you can navigate common pitfalls, write more efficient CMakeLists.txt files, and manage complex builds with ease.

References

Books

1. ”Professional CMake: A Practical Guide” by Craig Scott

   This book is widely regarded as one of the best resources for learning CMake. It provides a practical, hands-on approach, covering basic to advanced topics in great detail. Craig Scott's writing style makes complex topics easy to understand, and the book is full of real-world examples, making it an invaluable resource for developers who want to master CMake.

   - Key Topics: CMake fundamentals, writing portable CMake code, handling external dependencies, and continuous integration with CMake.
   - Why It’s Useful: It complements this book by diving deeper into advanced techniques, offering more extensive coverage of CMake in production-level projects.

2. ”Mastering CMake” by Ken Martin and Bill Hoffman

   As the creators of CMake, Ken Martin and Bill Hoffman offer a definitive guide to the tool in this book. It goes into the inner workings of CMake, covering not just how to use the tool, but also why it behaves in certain ways and how to customize it for specific needs.

- Key Topics: CMake internals, writing custom CMake modules, advanced topics in toolchain file configurations, and understanding the build process at a low level.
- Why It's Useful: This is an excellent resource for developers who need to write highly customized CMake files or troubleshoot complex build systems.

3. "Modern CMake for C++" by Rafal Swidzinski

   This book focuses on best practices for using CMake with modern C++ projects, covering everything from simple applications to large-scale projects. It emphasizes the use of modern CMake commands and strategies to create highly maintainable and scalable C++ codebases.

   - Key Topics: Modern CMake syntax, integration with external tools like CI/CD pipelines, writing clean and reusable CMake code, and utilizing new CMake features.
   - Why It's Useful: It offers a practical, example-driven approach to modern CMake, which is great for developers looking to move away from legacy CMake practices.

Online Documentation and Official Resources

1. CMake Official Documentation

   The official documentation for CMake is the authoritative source for everything related to CMake syntax, commands, modules, and configuration options. It is constantly updated and covers all versions of CMake. The documentation is rich with examples, command references, and explanations of CMake's internal workings.

- Key Features: Comprehensive command reference, tutorials for beginners, detailed module documentation, and platform-specific guides.
- Why It's Useful: For any question or issue related to CMake, the official documentation is the go-to resource. It will help you understand the tool's core functionality and how to use it in various scenarios.

2. CMake GitLab Repository

   This repository contains the source code of CMake itself. It's a valuable resource if you need to see how CMake is implemented or want to report a bug, contribute to the development of CMake, or explore the change history.

   - Key Features: CMake source code, bug tracking, feature requests, and contributions.
   - Why It's Useful: Developers who want to go beyond using CMake and contribute to its development or learn about its internal mechanics will find this a great resource.

3. CMake Wiki

   The CMake Wiki is a community-driven resource that supplements the official documentation with additional guides, how-tos, and solutions to common problems.

   - Key Features: Community-contributed tutorials, tips and tricks, case studies, and frequently asked questions.
   - Why It's Useful: The Wiki often contains solutions to real-world CMake problems, contributed by experienced users and experts from the community. It's a great place to find practical solutions.

4. CMake Discourse

   CMake's official forum is an excellent place to discuss issues, ask for help, and find discussions around CMake features. It's a valuable resource for

troubleshooting and learning from others' experiences.

- Key Features: Discussion threads on CMake topics, feature requests, bug reports, and user-provided examples.
- Why It's Useful: The CMake Discourse forum allows you to connect with the community, ask specific questions, and read about other users' experiences and solutions to problems.

Websites and Blogs

1. Modern CMake

   This website is dedicated to modern CMake best practices and provides an excellent summary of the most current CMake features, syntax, and patterns.

   - Key Features: Quick reference, guides on how to use CMake in modern C++ development, and examples of best practices.
   - Why It's Useful: This website is a great starting point for developers who want to learn modern CMake practices in a structured and concise way.

2. Kitware Blog

   Kitware, the company behind CMake, regularly publishes blog posts on new features, best practices, case studies, and tutorials. The blog provides insights into how CMake is used in various industries, including scientific computing, game development, and more.

   - Key Features: Updates on CMake features, case studies, tutorials, and advice from CMake experts.
   - Why It's Useful: For developers looking to stay up-to-date with new features in CMake or explore case studies of CMake used in complex environments, the Kitware blog is a valuable resource.

3. CMake Best Practices

   This open-source GitHub repository compiles a collection of CMake best practices, tips, and tricks from the community. It's an excellent place to find clean, maintainable examples of how to structure and organize CMake files.

   - Key Features: Best practices for writing reusable CMake code, guidelines for organizing large projects, and solutions to common problems.
   - Why It's Useful: This repository is a useful guide for writing clean and efficient CMake code. It contains advice on how to structure your build system to make your projects more scalable and maintainable.

4. CMake Examples

   This GitHub repository provides a collection of CMake examples covering various use cases, from simple projects to more complex configurations.

   - Key Features: Practical examples, explanations of CMake techniques, and diverse use cases.
   - Why It's Useful: If you are learning CMake through hands-on examples, this repository is a goldmine for realistic configurations. You can find examples on advanced topics like cross-compiling, multi-platform support, and complex dependency management.

Open-Source Projects and Repositories

1. LLVM/Clang

   LLVM is a highly modular project that uses CMake as its build system. It demonstrates complex, cross-platform CMake configurations in a large-scale codebase.

- Why It's Useful: By studying LLVM's CMake setup, you can learn how to handle large codebases, manage external dependencies, and optimize build processes using CMake.

2. Boost Libraries

   Boost is one of the most widely-used C++ libraries, and its CMake setup provides examples of managing complex dependencies, creating libraries, and structuring large C++ projects.

   - Why It's Useful: Boost's CMake files provide a clear example of handling external libraries, setting up multi-platform builds, and supporting various build configurations.

3. OpenCV

   OpenCV is a popular open-source computer vision library that uses CMake. The OpenCV project is a great example of how to manage cross-platform builds, link to external libraries, and organize large projects with CMake.

   - Why It's Useful: Studying OpenCV's CMake configuration will give you practical examples of using CMake with external dependencies and handling complex C++ codebases.

Final Thoughts on References

These references will provide you with the necessary tools and information to continue expanding your knowledge of CMake. Whether you're troubleshooting specific issues, looking for advanced techniques, or seeking hands-on examples, the resources listed here will help you develop a deeper understanding of CMake and its role in managing and building C++ projects.