CROSS PLATFORM C++ DEPENDENCY MANAGEMENT WITH CMAKE

CppAfrica Discord Meetup December 2025

WELCOME TO CPPAFRICA

- started a few years ago on discord as a place to meet up with fellow developers in C++
- purpose is to be able to discuss and build up each other in learning and skill improvement
- In 2025 we started an online meetup, where we have an online session on some topics of interest
- We have had about 5 successful sessions so far.
- The format isn’t fixed, we are still experimenting and welcome speakers, comments and feedback

AGENDA

- Structuring, Compilation and Linking of Software in C++
- Build systems
- CMake
- Project Examples

INTRODUCTION

- Let us start with some basics

C++ BASIC COMPILATION

- A C++ program is a sequence of text files (typically header and source files) that contain declarations. They undergo translation to become an executable program, which is executed when the C++ implementation calls its main function.
- C++, being a native programming language, provides the ability to join together programs with libraries of binary code from various sources.
- C++ is a language standard. It comprises of a standard set of rules that implementors of the language can implement in various ways as long as the standard is adhered to.
- A program written in standard C++ should be able to be run, after compilation and linking, on various types of systems

PROGRAM DEPENDENCIES(LIBRARIES)

STATIC LIBRARIES AND STATIC LINKING

- A group of C++ source files can be grouped together to form a static library.
- It is generally just an archive of object files joined together to form a single file.
- Think of it like a zip file, storing the object files
- When an executable(the runnable program) is linked statically with a library, the code in the library is simply included in the final binary at compile time.
- This can make a statically linked binary very large, but it is a single file that is simple to deploy.

SHARED LIBRARIES AND DYNAMIC LINKING

- With Shared Libraries, the object files are joined together to a form similar to an executable.
- It is designed to be provided by the operating system mechanisms to any program that requires it dynamically.
- When an executable is compiled, it is marked with the names of the libraries it requires to function.
- When it is run, it requests the operating system for these libraries.
- With shared libraries, all processes in the operating system will share access to the same library loaded in memory.
- This makes the size of a program that uses shared libraries significantly smaller.
- Also sharing of memory, makes the use of shared libraries give us significant memory savings.

OPERATING SYSTEMS

- Different operating systems, expose their functionality using libraries of binary code that can be linked to by user programs.
- For example the C API is usually used in most programs in linux and unix will link with libc, and the same with windows.
- C++ programs will generally link with a library that implements the standard library like libc++(clang), libstdc++(GNU), MSVC(windows), etc.

BUILD SYSTEMS

- The process of producing runnable programs can get compilicated when dealing with complex and real world projects.
- This is due to a large number of source files as well as managing complex dependencies between libraries of compiled code.

CMAKE

- CMake is a tool to manage building of source code.
- It does so primarily by enabling the generation of Build files for various build systems
- Originally this was just Makefiles, but now generates build system files for MSBuild(Visual Studio), ninja, XCode and others.
- The nature of programming with languages like C++ involves creating a large number of files that have to be orchestrated to create a working program.

HOW TO USE CMAKE

CMAKELISTS.TXT

- This is a special file that is included in the source tree of a directory that we want to be managed by cmake
- It consists of commands in CMake format that are run when the **cmake** command is executed.
- The top level directory of a project contains one with the following structure

```
cmake_minimum_required(VERSION 3.29)
project(mytestproject CXX)
...
```

THE CMAKE PROGRAM

- We execute the **cmake** program to process the commands in the CMakeLists.txt file
- We generally give it a source directory and a build directory
- The Source directory is one with a CMakeLists.txt file
- The build directory is where the generated buildsystem is written to
- There are other features available through this command like installing and others

CMAKE TARGETS

- CMake(from version 3) uses the concept of targets to express “what we want to build”.
- Older versions of CMake used folders(directories) as the base “element”
- There are two main cartegories of Targets, Executables and Libraries.
- Executables have runnable programs as the final Result(Target).
- Libraries have a form of library which is a collection of functionality.
- There are other types of targets within those main types.
- Another division of targets, or way or cartegorizing is binary targets vs pseudo targets.

TARGET_LINK_LIBRARIES COMMAND

- The primary tool for creating relationships between targets is the **target_link_libraries()** command.
- This is used to create a connection between a target and one or more other targets
- It serves a very important purpose in managing the “Flag Soup” that would arise when flags are just put together aimlessly
- It ensures that duplicates are weeded out and that a logical structure is established.

STRUCTURE OF TARGETS IN CMAKE

- Targets in CMake behave a bit like Objects in C++
- They have private properties which are stored in the private section
- They have public properties which are stored in the interface section
- There is a “section” called PUBLIC which means storage of properties in both the private and interface properties
- These properties are referenced during the build system generation to specify the build system
- The **private** properties are used when compiling the target and are reffered to as BUILD REQUIREMENTS
- THE **interface** properties are specified for use in targets that reference this target and are referred to at USAGE REQUIREMENTS

BINARY TARGETS

- These are the most commonly known targets that describe what we want to build, i.e. an executable or a library
- There are the various types of library types which have a corresponding target type.
- There is only one type of executable target, but several library targets.

PSEUDO TARGETS

- These are targets that do not involve the compilation of source files
- They do provide artifacts to be used in the binary targets to enable compilation
- They generally only have **interface** properties to specify usage requirements

CMAKE DEPENDENCY MANAGEMENT

THE THREE TREES IN CMAKE PROJECTS

- There are three trees that are involved in CMake Projects.
- These are
- The Source Tree
- The Build Tree
- The Install Tree

THE SOURCE TREE

- This is usually what is checked into source control.
- It comprises of the source code and configuration files that are the basis for building the software project

THE BUILD TREE

- This is where the build articacts are stored.
- It is generally good practice to separate the Build and Source Trees.
- It is possible to generate multiple build trees with different configurations in separate directories

THE INSTALL TREE

- This is a tree made up of a selection of files from the Build Tree that can be packaged and installed to a system or user directory involved with runnable programs.
- It is the basis for distribution of software through packages in package management systems.
- It generally comprises of an include directory for headers, a library directory for libraries as well as a binary directory for runnable programs.
- It is also where Imported Targets that are the basis for using a package are stored using the cmake **install** command
- This particular tree is not very well understood by many
- When we talk about using the CMake **FindPackage** command, It generally assumes that other dependencies are produced in a specific format based on the install tree.

SOURCE BASED DEPENDENCIES

- One way of getting external dependencies is by obtaining them in source code form and including them in our project and then compiling them
- This can be done in various ways such as git submodules, manually downloading the code, etc.
- The obtained sources are then added to the main project with commands like the **add_subdirectory** command
- The **FetchContent** module has a more modern way of achieving the same result.
- However, A lot of care must be taken when dealing with large projects with multiple source dependencies as their corresponding CMake Configurations may have some intricate requiremnts in terms of CMake Properties and Options and the process of resolving these can be “Finicky”
- Some features like C++ modules can only be done in this way…for now.

BINARY PRE-COMPILED DEPENDENCIES

- These can be system installed packages or even downloads of packaged software.
- They correspond the the Install Tree of the “external projects”
- The **FindPackage** CMake Command generally assumes this format for tracking down dependencies.

HYBRID DEPENDENCY MANAGEMENT

- VCPKG is a kind of hybrid dependency management that handles various dependencies by obtaining them as source code but offering them to CMake Packages as Binary Pre-Compiled Dependencies
- The CMake FetchContent Module is also capable of doing something similar in a native way but also has compatibility with the FindPackage Command.
- The FetchContent Module can first check for the availability of a Pre-Compiled Solution using FindPackage. If no suitable package is found it can fall back to obtaining and compiling from source.

HOW THE FINDPACKAGE COMMAND WORKS

- It works by looking for CMake ConfigFiles that are stored in certain system level directories or in directories specified in the **CMAKE_PREFIX_PATH**
- There are special CMake Configuration Files that expose “Imported Targets” that expose Usage Requirements for the libraries and headers in their packages

EXAMPLES WITH MORE COMPLEX PROJECTS(GITHUB LINKS)

- RendererEngine
- Buyer-Backend
- Tristan Brindle AoC2025

Speaker notes