CMake for Complex Projects (Part 1): Building a C++ Game Engine from Scratch for Desktop and WebAssembly

Язык оригинала: en

# Оригинал

A practical guide to modern CMake through a real-world C++ game engine project - Compilation & Build Management  
If you've ever tried to build a C++ project with multiple dependencies and cross-platform support, you know the pain. Makefiles become unwieldy, Visual Studio solutions don't work on Linux, and don't even get me started on trying to distribute your library for others to use.  
This is where CMake shines. But most CMake tutorials show you toy examples with a single  
hello.cpp  
file. Today, we're going deep with a  
complex project  
- a complete game engine called ColumbaEngine (source code available  
here  
) with 80+ source files, multiple platforms (including web compilation), and a sophisticated build system.  
This is  
Part 1  
of a two-part series. In this article, we'll focus on the  
compilation and build management  
aspects: setting up the project, handling dependencies, cross-platform builds, and testing. In Part 2, we'll cover the  
deployment side  
: installation systems, package generation, and distribution.  
By the end of this article, you'll understand how to structure CMake for medium to large projects, handle complex dependencies, and create professional build systems that compile reliably across platforms.  
What Makes This Project Interesting?  
Before we dive into CMake, let's understand what we're building:  
80+ C++ source files  
organized in modules (ECS, Renderer, Audio, UI, etc.)  
Cross-platform  
: Windows, Linux, and Web (via Emscripten)  
Multiple dependencies  
: SDL2, OpenGL, FreeType, custom libraries  
Installable library  
: Other projects can use it with  
find\_package()  
Example applications  
: Several games and tools  
Package generation  
: Creates  
.deb  
,  
.rpm  
, and other installers  
This isn't a toy project - it's a real game engine that needs a robust build system.  
Project Structure: The Foundation  
Here's how our game engine is organized:  
ColumbaEngine/  
├── CMakeLists.txt # Main build file  
├── src/  
│ ├── Engine/ # Core engine code  
│ │ ├── ECS/ # Entity-Component-System  
│ │ ├── Renderer/ # Graphics rendering  
│ │ └── Audio/ # Sound system  
│ └── main.cpp # Editor application  
├── examples/ # Example games  
├── import/ # Vendored dependencies  
├── cmake/ # CMake modules  
└── test/ # Unit tests  
Enter fullscreen mode  
Exit fullscreen mode  
The  
CMakeLists.txt  
is our blueprint. Let's build it step by step.  
Step 1: Project Setup and Configuration  
cmake\_minimum\_required  
(  
VERSION 3.18  
)  
project  
(  
ColumbaEngine VERSION 1.0  
)  
# User-configurable options  
option  
(  
BUILD\_EXAMPLES  
"Build example applications"  
ON  
)  
option  
(  
BUILD\_STATIC\_LIB  
"Build static library only"  
OFF  
)  
option  
(  
ENABLE\_TIME\_TRACE  
"Add -ftime-trace to Clang builds"  
OFF  
)  
# Global settings  
set  
(  
CMAKE\_EXPORT\_COMPILE\_COMMANDS ON  
)  
# For IDE support  
set  
(  
CMAKE\_CXX\_STANDARD 17  
)  
set  
(  
CMAKE\_CXX\_STANDARD\_REQUIRED True  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Key concepts here:  
cmake\_minimum\_required  
: We use 3.18+ for modern features like precompiled headers  
option()  
: Creates user-configurable boolean flags. Users can run  
cmake -DBUILD\_EXAMPLES=OFF ..  
CMAKE\_EXPORT\_COMPILE\_COMMANDS  
: Generates  
compile\_commands.json  
for IDEs and tools like clangd  
Step 2: The Dependency Challenge  
Real projects have dependencies. Lots of them. Our game engine needs SDL2 for windowing, FreeType for text rendering, OpenGL for graphics, and more.  
We use a  
vendoring approach  
- including dependencies directly in our source tree:  
# Dependency paths - everything is self-contained  
set  
(  
SDL2\_DIR  
"import/SDL2-2.28.5"  
)  
set  
(  
SDL2MIXER\_DIR  
"import/SDL2\_mixer-2.6.3"  
)  
set  
(  
GLM\_DIR  
"import/glm"  
)  
set  
(  
TASKFLOW\_DIR  
"import/taskflow-3.6.0"  
)  
set  
(  
GTEST\_DIR  
"import/googletest-1.14.0"  
)  
# Configure dependencies before building them  
set  
(  
TF\_BUILD\_EXAMPLES OFF  
)  
# Don't build taskflow examples  
set  
(  
TF\_BUILD\_TESTS OFF  
)  
# Don't build taskflow tests  
set  
(  
BUILD\_SHARED\_LIBS OFF  
)  
# Force static linking  
set  
(  
SDL2\_DISABLE\_INSTALL ON  
)  
# We'll handle installation ourselves  
Enter fullscreen mode  
Exit fullscreen mode  
Why vendor dependencies?  
✅  
Pros  
: Exact version control, works offline, simplified build  
❌  
Cons  
: Larger repo size, manual updates  
For a game engine where stability is crucial, vendoring makes sense. For web services that need frequent security updates, you might prefer package managers.  
Deep Dive: Dependency Management Strategies  
Choosing how to handle dependencies is one of the most critical architectural decisions in C++. Let's compare the approaches:  
1. Vendoring (Our Current Approach)  
# Everything is self-contained in import/  
set  
(  
SDL2\_DIR  
"import/SDL2-2.28.5"  
)  
set  
(  
GLM\_DIR  
"import/glm"  
)  
add\_subdirectory  
(  
${  
SDL2\_DIR  
}  
)  
add\_subdirectory  
(  
${  
GLM\_DIR  
}  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Advantages:  
Reproducible builds  
: Exact same versions everywhere  
Offline builds  
: No internet required after initial clone  
Control  
: Can patch dependencies if needed  
Simplicity  
: No external tools or package managers  
Disadvantages:  
Repository size  
: Our import/ folder is 500MB+  
Update overhead  
: Manual process to update dependencies  
Security  
: Must manually track and update vulnerable dependencies  
License complexity  
: Must distribute all licenses  
2. Package Managers (Conan, vcpkg)  
# With Conan  
include  
(  
conan  
)  
conan\_cmake\_run  
(  
REQUIRES   
 SDL2/2.28.5  
 glm/0.9.9.8  
 BASIC\_SETUP CMAKE\_TARGETS  
 BUILD missing  
)  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE CONAN\_PKG::SDL2 CONAN\_PKG::glm  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Advantages:  
Smaller repos  
: Dependencies downloaded on-demand  
Easy updates  
:  
conan install --update  
Binary packages  
: Pre-built binaries for faster builds  
Ecosystem  
: Thousands of packages available  
Disadvantages:  
External dependency  
: Requires internet and package manager  
Version conflicts  
: Diamond dependency problems  
Platform support  
: Not all packages support all platforms  
Learning curve  
: Another tool to learn and maintain  
3. FetchContent (Modern CMake)  
include  
(  
FetchContent  
)  
FetchContent\_Declare  
(  
glm  
 GIT\_REPOSITORY https://github.com/g-truc/glm.git  
 GIT\_TAG 0.9.9.8  
 GIT\_SHALLOW TRUE  
)  
FetchContent\_Declare  
(  
SDL2  
 URL https://github.com/libsdl-org/SDL/releases/download/release-2.28.5/SDL2-2.28.5.tar.gz  
 URL\_HASH SHA256=332cb37d0be20cb9541739c61f79bae5a477427d79ae85e352089afdaf6666e4  
)  
FetchContent\_MakeAvailable  
(  
glm SDL2  
)  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE glm SDL2::SDL2  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Advantages:  
CMake native  
: No external tools required  
Flexible sources  
: Git repos, URLs, local paths  
Version pinning  
: Exact commits/tags  
Cross-platform  
: Works everywhere CMake works  
Disadvantages:  
Build time  
: Downloads and builds on first configure  
Internet required  
: At least for first build  
No binary caching  
: Always builds from source  
Hybrid Approach: The Best of Both Worlds  
For ColumbaEngine, we could evolve to a hybrid approach:  
# Critical, stable dependencies: Vendor them  
set  
(  
SDL2\_DIR  
"import/SDL2-2.28.5"  
)  
add\_subdirectory  
(  
${  
SDL2\_DIR  
}  
)  
# Development/testing dependencies: FetchContent  
if  
(  
BUILD\_TESTS  
)  
include  
(  
FetchContent  
)  
FetchContent\_Declare  
(  
googletest  
 GIT\_REPOSITORY https://github.com/google/googletest.git  
 GIT\_TAG v1.14.0  
 GIT\_SHALLOW TRUE  
)  
FetchContent\_MakeAvailable  
(  
googletest  
)  
endif  
()  
# Optional dependencies: find\_package with fallback  
find\_package  
(  
Doxygen  
)  
if  
(  
NOT Doxygen\_FOUND AND ENABLE\_DOCS  
)  
FetchContent\_Declare  
(  
doxygen  
 URL https://github.com/doxygen/doxygen/releases/download/Release\_1\_9\_8/doxygen-1.9.8.src.tar.gz  
)  
FetchContent\_MakeAvailable  
(  
doxygen  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Step 3: Cross-Platform Reality Check  
Our engine supports three platforms, with different flags and need:  
Native  
(Windows/Linux): Build everything from source  
Emscripten  
(Web): Use system-provided libraries  
if  
(  
${  
CMAKE\_SYSTEM\_NAME  
}  
MATCHES  
"Emscripten"  
)  
# Web build - use Emscripten's built-in libraries  
set  
(  
USE\_FLAGS  
"-O3 -sUSE\_SDL=2 -sUSE\_SDL\_MIXER=2 -sUSE\_FREETYPE=1 -fwasm-exceptions"  
)  
set  
(  
CMAKE\_EXE\_LINKER\_FLAGS  
"  
${  
CMAKE\_EXE\_LINKER\_FLAGS  
}  
${  
USE\_FLAGS  
}  
"  
)  
else  
()  
# Native build - compile dependencies from source  
add\_subdirectory  
(  
${  
SDL2\_DIR  
}  
)  
add\_subdirectory  
(  
${  
SDL2MIXER\_DIR  
}  
)  
add\_subdirectory  
(  
${  
GLEW\_DIR  
}  
)  
add\_subdirectory  
(  
${  
TTF\_DIR  
}  
)  
endif  
()  
# GLM is header-only, works everywhere  
add\_subdirectory  
(  
${  
GLM\_DIR  
}  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Platform detection patterns:  
CMAKE\_SYSTEM\_NAME  
: Target platform (Windows, Linux, Darwin, Emscripten)  
CMAKE\_HOST\_SYSTEM  
: Build machine platform  
Use generator expressions for conditional compilation:  
$<$<PLATFORM\_ID:Windows>:WIN32\_CODE>  
Step 4: Creating the Main Target  
Now for the heart of our build - the engine library itself:  
# List all source files (80+ files in our case!)  
set  
(  
ENGINESOURCE  
 src/Engine/window.cpp  
 src/Engine/configuration.cpp  
 src/Engine/logger.cpp  
# ... 70+ more files  
src/Engine/UI/uisystem.cpp  
)  
# Create the static library  
add\_library  
(  
ColumbaEngine STATIC  
${  
ENGINESOURCE  
}  
)  
# Modern CMake magic - precompiled headers for faster builds  
target\_precompile\_headers  
(  
ColumbaEngine PUBLIC src/Engine/stdafx.h  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Precompiled headers  
can cut build times by 50%+ in large projects. They compile commonly used headers once and reuse the result.  
Precompiled Headers - The Build Speed Game Changer  
Precompiled headers (PCH) are one of the most impactful optimizations for C++ build times, yet they're often overlooked. Let's understand how they work:  
The Problem  
: Header Parsing Overhead  
// Every .cpp file typically includes these  
#include  
<iostream>  
// ~15,000 lines when fully expanded  
#include  
<vector>  
// ~8,000 lines  
#include  
<string>  
// ~12,000 lines  
#include  
<memory>  
// ~6,000 lines  
#include  
<SDL2/SDL.h>  
// ~25,000 lines  
// Total: ~66,000 lines to parse per source file!  
Enter fullscreen mode  
Exit fullscreen mode  
With 80 source files, that's  
5.2 million lines  
of redundant header parsing!  
The Solution  
: Precompiled Headers  
target\_precompile\_headers  
(  
ColumbaEngine PUBLIC src/Engine/stdafx.h  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Our  
stdafx.h  
contains the most commonly used headers:  
// stdafx.h - precompiled header  
#pragma once  
// Standard library  
#include  
<iostream>  
#include  
<vector>  
#include  
<string>  
#include  
<memory>  
#include  
<unordered\_map>  
#include  
<algorithm>  
// Third-party libraries  
#include  
<SDL2/SDL.h>  
#include  
<glm/glm.hpp>  
#include  
<glm/gtc/matrix\_transform.hpp>  
// Engine fundamentals used everywhere  
#include  
"types.h"  
#include  
"logger.h"  
#include  
"configuration.h"  
Enter fullscreen mode  
Exit fullscreen mode  
How It Works  
:  
Compilation  
: CMake compiles  
stdafx.h  
once into  
stdafx.h.gch  
(GCC) or  
stdafx.pch  
(MSVC)  
Reuse  
: Every source file automatically uses the precompiled version  
Speed  
: 66,000 lines → 0 lines to parse per file  
Build Time Results  
(80 source files):  
Without PCH  
: ~8 minutes clean build  
With PCH  
: ~3 minutes clean build  
Incremental  
: Single file changes drop from 15s to 3s  
PUBLIC vs PRIVATE PCH  
:  
# PUBLIC: Users of your library get the PCH benefits too  
target\_precompile\_headers  
(  
ColumbaEngine PUBLIC stdafx.h  
)  
# PRIVATE: Only internal compilation uses PCH  
target\_precompile\_headers  
(  
ColumbaEngine PRIVATE stdafx.h  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Best Practices for PCH  
:  
// Good PCH content: Stable, frequently used headers  
#include  
<vector>  
// Used in 90% of files  
#include  
<SDL2/SDL.h>  
// Core dependency  
// Bad PCH content: Frequently changing headers  
#include  
"GameLogic.h"  
// Changes often, invalidates PCH  
#include  
"debug\_temp.h"  
// Temporary debugging code  
Enter fullscreen mode  
Exit fullscreen mode  
PCH Pitfalls  
:  
Overuse  
: Adding changing headers defeats the purpose  
Platform differences  
: MSVC vs GCC handle PCH differently  
Dependencies  
: PCH creates implicit dependencies between files  
Step 5: Usage Requirements - The Secret Sauce  
This is where modern CMake really shines. Instead of users manually figuring out include paths and linking, we specify  
usage requirements  
:  
target\_include\_directories  
(  
ColumbaEngine PUBLIC  
# When building this library  
$<BUILD\_INTERFACE:  
${  
CMAKE\_CURRENT\_SOURCE\_DIR  
}  
/src/Engine>  
 $<BUILD\_INTERFACE:  
${  
CMAKE\_CURRENT\_SOURCE\_DIR  
}  
/src/GameElements>  
# When using installed version  
$<INSTALL\_INTERFACE:include/ColumbaEngine>  
 $<INSTALL\_INTERFACE:include/ColumbaEngine/GameElements>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Generator expressions  
(  
$<...>  
) are evaluated during build generation:  
BUILD\_INTERFACE  
: Only when building this project  
INSTALL\_INTERFACE  
: Only when using the installed version  
$<CONFIG:Debug>  
: Only in Debug builds  
$<PLATFORM\_ID:Windows>  
: Only on Windows  
Why Generator Expressions Matter  
Let's understand why this is so powerful. Consider this traditional approach:  
# The old, problematic way  
if  
(  
CMAKE\_BUILD\_TYPE STREQUAL  
"Debug"  
)  
set  
(  
CMAKE\_CXX\_FLAGS  
"  
${  
CMAKE\_CXX\_FLAGS  
}  
-DDEBUG\_MODE"  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Problems with this approach:  
Build-time evaluation  
: The condition is checked when CMake configures, not when you build  
Global pollution  
: Affects ALL targets in the project  
Multi-config generators  
: Breaks with Visual Studio, Xcode which support multiple configs  
Generator expressions solve this:  
# Modern, correct approach  
target\_compile\_definitions  
(  
MyTarget PRIVATE   
 $<$<CONFIG:Debug>:DEBUG\_MODE>  
 $<$<CONFIG:Release>:NDEBUG>  
 $<$<PLATFORM\_ID:Windows>:WIN32\_LEAN\_AND\_MEAN>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
This is evaluated  
per-target  
,  
per-configuration  
,  
at build time  
. Much more flexible and robust.  
Common Generator Expression Patterns  
# Conditional compilation flags  
target\_compile\_options  
(  
MyTarget PRIVATE  
 $<$<CXX\_COMPILER\_ID:GNU,Clang>:-Wall -Wextra>  
 $<$<CXX\_COMPILER\_ID:MSVC>:/W4>  
)  
# Debug vs Release libraries  
target\_link\_libraries  
(  
MyTarget PRIVATE  
 $<$<CONFIG:Debug>:MyLib\_d>  
 $<$<CONFIG:Release>:MyLib>  
)  
# Platform-specific linking  
target\_link\_libraries  
(  
MyTarget PRIVATE  
 $<$<PLATFORM\_ID:Windows>:ws2\_32>  
 $<$<PLATFORM\_ID:Linux>:pthread>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Step 6: Dependency Linking - Getting Scope Right  
Here's a crucial concept:  
PUBLIC vs PRIVATE vs INTERFACE  
if  
(  
${  
CMAKE\_SYSTEM\_NAME  
}  
MATCHES  
"Emscripten"  
)  
# Web build  
target\_link\_libraries  
(  
ColumbaEngine PUBLIC glm  
)  
else  
()  
# Native build - note the scoping!  
target\_link\_libraries  
(  
ColumbaEngine PUBLIC   
 SDL2::SDL2-static  
# Users need SDL2 headers  
glm  
# Header-only math library  
OpenGL::GL  
# Graphics API  
)  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE   
 libglew\_static  
# Internal OpenGL extension loading  
freetype  
# Internal text rendering  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Scope meanings:  
PUBLIC  
: "I use this, and my users will need it too"  
PRIVATE  
: "I use this internally, but my users don't need to know"  
INTERFACE  
: "I don't use this, but my users will need it"  
Get this right, and users of your library automatically get the right dependencies. Get it wrong, and you'll have angry developers filing issues.  
Understanding Transitive Dependencies  
Let's look at a real example from our engine. Imagine this dependency chain:  
MyGame → ColumbaEngine → SDL2 → OpenGL  
Enter fullscreen mode  
Exit fullscreen mode  
If we declare SDL2 as  
PRIVATE  
to ColumbaEngine:  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE SDL2::SDL2-static  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Problem  
:  
MyGame  
won't be able to use SDL2 types in the public API. If ColumbaEngine's headers include  
<SDL2/SDL.h>  
, users get compile errors.  
If we declare SDL2 as  
PUBLIC  
:  
target\_link\_libraries  
(  
ColumbaEngine PUBLIC SDL2::SDL2-static  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Result  
:  
MyGame  
automatically gets SDL2 headers and libraries. Perfect for our engine where users need SDL2 types.  
The INTERFACE Scope Mystery  
INTERFACE  
is the trickiest to understand. It means "I don't use this, but my users will."  
Real-world example  
: Header-only libraries with dependencies  
# HeaderOnlyMath library depends on Eigen, but is itself header-only  
add\_library  
(  
HeaderOnlyMath INTERFACE  
)  
target\_link\_libraries  
(  
HeaderOnlyMath INTERFACE Eigen3::Eigen  
)  
# Users of HeaderOnlyMath automatically get Eigen  
add\_executable  
(  
MyApp main.cpp  
)  
target\_link\_libraries  
(  
MyApp PRIVATE HeaderOnlyMath  
)  
# Gets Eigen too!  
Enter fullscreen mode  
Exit fullscreen mode  
Mixing Scopes: The Real World  
Most real projects use mixed scopes:  
target\_link\_libraries  
(  
ColumbaEngine  
# Users need these APIs  
PUBLIC   
 SDL2::SDL2-static  
# Window/input handling in public API  
glm  
# Math types in public headers  
OpenGL::GL  
# Graphics API exposed to users  
# Internal implementation details  
PRIVATE  
 libglew\_static  
# OpenGL extension loading (wrapped)  
freetype  
# Text rendering (abstracted away)  
${  
CMAKE\_DL\_LIBS  
}  
# Dynamic library loading (platform detail)  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Step 7: Multiple Executables and Examples  
A game engine isn't just a library - it includes tools and examples:  
if  
(  
NOT BUILD\_STATIC\_LIB AND BUILD\_EXAMPLES  
)  
# Main editor application  
add\_executable  
(  
ColumbaEngineEditor src/main.cpp  
)  
target\_sources  
(  
ColumbaEngineEditor PRIVATE  
 src/application.cpp  
 src/Editor/Gui/inspector.cpp  
 src/Editor/Gui/projectmanager.cpp  
)  
target\_link\_libraries  
(  
ColumbaEngineEditor PRIVATE ColumbaEngine  
)  
# Game examples  
add\_executable  
(  
GameOff examples/GameOff/main.cpp  
)  
target\_sources  
(  
GameOff PRIVATE  
 examples/GameOff/application.cpp  
 examples/GameOff/character.cpp  
 examples/GameOff/inventory.cpp  
)  
target\_link\_libraries  
(  
GameOff PRIVATE ColumbaEngine  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Pattern  
: Create multiple executables that all link to your main library. This way, the library code is compiled once and reused.  
Step 8: Testing Infrastructure  
Professional projects need tests:  
if  
(  
NOT BUILD\_STATIC\_LIB  
)  
enable\_testing  
()  
add\_subdirectory  
(  
${  
GTEST\_DIR  
}  
)  
add\_executable  
(  
t1 test/maintest.cc  
)  
target\_sources  
(  
t1 PRIVATE  
 test/collision2d.cc  
 test/ecssystem.cc  
 test/interpreter.cc  
 test/renderer.cc  
)  
target\_link\_libraries  
(  
t1 PRIVATE gtest gtest\_main ColumbaEngine  
)  
# Auto-discover tests  
include  
(  
GoogleTest  
)  
gtest\_discover\_tests  
(  
t1  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
gtest\_discover\_tests()  
automatically finds all test cases and creates individual CTest entries. Run with  
ctest  
or  
make test  
.  
Common Build Pitfalls  
1. Global Variables vs Target Properties  
# BAD: Affects everything globally  
set  
(  
CMAKE\_CXX\_FLAGS  
"  
${  
CMAKE\_CXX\_FLAGS  
}  
-DDEBUG"  
)  
# GOOD: Only affects specific target  
target\_compile\_definitions  
(  
MyTarget PRIVATE DEBUG  
)  
Enter fullscreen mode  
Exit fullscreen mode  
2. Not Using Generator Expressions  
# BAD: Debug symbols in release builds  
target\_compile\_options  
(  
MyTarget PRIVATE -g  
)  
# GOOD: Only in debug builds  
target\_compile\_options  
(  
MyTarget PRIVATE $<$<CONFIG:Debug>:-g>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
3. Wrong Dependency Scopes  
# If users of your library need OpenGL headers:  
target\_link\_libraries  
(  
MyLib PUBLIC OpenGL::GL  
)  
# If only your implementation uses OpenGL:  
target\_link\_libraries  
(  
MyLib PRIVATE OpenGL::GL  
)  
Enter fullscreen mode  
Exit fullscreen mode  
What We've Achieved: Build System Results  
After implementing all these concepts, here's what we achieved:  
Developer Experience:  
# Simple build  
git clone https://github.com/user/ColumbaEngine  
cd  
ColumbaEngine  
mkdir  
build  
&&  
cd  
build  
cmake ..  
make  
-j8  
# Custom configuration  
cmake  
-DBUILD\_EXAMPLES  
=  
OFF  
-DCMAKE\_BUILD\_TYPE  
=  
Release ..  
# Run tests  
ctest  
Enter fullscreen mode  
Exit fullscreen mode  
Cross-platform Support:  
Same CMakeLists.txt works on Windows, Linux, and web  
Automatic dependency resolution per platform  
Platform-specific optimizations where needed  
Build Performance:  
Precompiled headers cut build times by 60%  
Parallel compilation across all targets  
Incremental builds under 10 seconds for single-file changes  
Coming Up in Part 2  
In the next article, we'll cover the  
deployment and distribution  
side of CMake:  
Installation Systems  
: How to make your library usable by others with  
find\_package()  
Export/Import Mechanisms  
: The complex but powerful system that makes modern CMake libraries work  
Package Configuration  
: Creating relocatable, dependency-aware packages  
CPack Integration  
: Generating professional installers for multiple platforms  
Real-world Distribution  
: Getting your library into the hands of users  
Conclusion  
Building a robust CMake build system for complex projects requires understanding several key concepts:  
Target-centric thinking  
- Everything revolves around targets and their usage requirements  
Proper dependency management  
- Choose the right strategy for your project's needs  
Cross-platform abstractions  
- Let CMake handle platform differences  
Generator expressions  
- Enable conditional behavior without complex if/else logic  
Build optimization  
- Use precompiled headers and other modern features  
The  
game engine  
we've built demonstrates these concepts in action with a real, production-ready build system that handles complex, multi-platform C++ projects.  
Remember: CMake is about expressing  
intent  
, not  
implementation details  
. Focus on what you want to achieve, and let CMake figure out how to do it on each platform.  
Don't miss [Part 2] where we'll cover installation, packaging, and distribution - making your library actually usable by the world!  
What's your biggest CMake build challenge?  
Share your experiences in the comments below!  
The complete source code for ColumbaEngine with all the CMake patterns from this series is available on  
GitHub  
. These patterns scale from small libraries to large, complex applications.

# Перевод на русский

A practical guide to modern CMake through a real-world C++ game engine project - Compilation & Build Management  
If you've ever tried to build a C++ project with multiple dependencies and cross-platform support, you know the pain. Makefiles become unwieldy, Visual Studio solutions don't work on Linux, and don't even get me started on trying to distribute your library for others to use.  
This is where CMake shines. But most CMake tutorials show you toy examples with a single  
hello.cpp  
file. Today, we're going deep with a  
complex project  
- a complete game engine called ColumbaEngine (source code available  
here  
) with 80+ source files, multiple platforms (including web compilation), and a sophisticated build system.  
This is  
Part 1  
of a two-part series. In this article, we'll focus on the  
compilation and build management  
aspects: setting up the project, handling dependencies, cross-platform builds, and testing. In Part 2, we'll cover the  
deployment side  
: installation systems, package generation, and distribution.  
By the end of this article, you'll understand how to structure CMake for medium to large projects, handle complex dependencies, and create professional build systems that compile reliably across platforms.  
What Makes This Project Interesting?  
Before we dive into CMake, let's understand what we're building:  
80+ C++ source files  
organized in modules (ECS, Renderer, Audio, UI, etc.)  
Cross-platform  
: Windows, Linux, and Web (via Emscripten)  
Multiple dependencies  
: SDL2, OpenGL, FreeType, custom libraries  
Installable library  
: Other projects can use it with  
find\_package()  
Example applications  
: Several games and tools  
Package generation  
: Creates  
.deb  
,  
.rpm  
, and other installers  
This isn't a toy project - it's a real game engine that needs a robust build system.  
Project Structure: The Foundation  
Here's how our game engine is organized:  
ColumbaEngine/  
├── CMakeLists.txt # Main build file  
├── src/  
│ ├── Engine/ # Core engine code  
│ │ ├── ECS/ # Entity-Component-System  
│ │ ├── Renderer/ # Graphics rendering  
│ │ └── Audio/ # Sound system  
│ └── main.cpp # Editor application  
├── examples/ # Example games  
├── import/ # Vendored dependencies  
├── cmake/ # CMake modules  
└── test/ # Unit tests  
Enter fullscreen mode  
Exit fullscreen mode  
The  
CMakeLists.txt  
is our blueprint. Let's build it step by step.  
Step 1: Project Setup and Configuration  
cmake\_minimum\_required  
(  
VERSION 3.18  
)  
project  
(  
ColumbaEngine VERSION 1.0  
)  
# User-configurable options  
option  
(  
BUILD\_EXAMPLES  
"Build example applications"  
ON  
)  
option  
(  
BUILD\_STATIC\_LIB  
"Build static library only"  
OFF  
)  
option  
(  
ENABLE\_TIME\_TRACE  
"Add -ftime-trace to Clang builds"  
OFF  
)  
# Global settings  
set  
(  
CMAKE\_EXPORT\_COMPILE\_COMMANDS ON  
)  
# For IDE support  
set  
(  
CMAKE\_CXX\_STANDARD 17  
)  
set  
(  
CMAKE\_CXX\_STANDARD\_REQUIRED True  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Key concepts here:  
cmake\_minimum\_required  
: We use 3.18+ for modern features like precompiled headers  
option()  
: Creates user-configurable boolean flags. Users can run  
cmake -DBUILD\_EXAMPLES=OFF ..  
CMAKE\_EXPORT\_COMPILE\_COMMANDS  
: Generates  
compile\_commands.json  
for IDEs and tools like clangd  
Step 2: The Dependency Challenge  
Real projects have dependencies. Lots of them. Our game engine needs SDL2 for windowing, FreeType for text rendering, OpenGL for graphics, and more.  
We use a  
vendoring approach  
- including dependencies directly in our source tree:  
# Dependency paths - everything is self-contained  
set  
(  
SDL2\_DIR  
"import/SDL2-2.28.5"  
)  
set  
(  
SDL2MIXER\_DIR  
"import/SDL2\_mixer-2.6.3"  
)  
set  
(  
GLM\_DIR  
"import/glm"  
)  
set  
(  
TASKFLOW\_DIR  
"import/taskflow-3.6.0"  
)  
set  
(  
GTEST\_DIR  
"import/googletest-1.14.0"  
)  
# Configure dependencies before building them  
set  
(  
TF\_BUILD\_EXAMPLES OFF  
)  
# Don't build taskflow examples  
set  
(  
TF\_BUILD\_TESTS OFF  
)  
# Don't build taskflow tests  
set  
(  
BUILD\_SHARED\_LIBS OFF  
)  
# Force static linking  
set  
(  
SDL2\_DISABLE\_INSTALL ON  
)  
# We'll handle installation ourselves  
Enter fullscreen mode  
Exit fullscreen mode  
Why vendor dependencies?  
✅  
Pros  
: Exact version control, works offline, simplified build  
❌  
Cons  
: Larger repo size, manual updates  
For a game engine where stability is crucial, vendoring makes sense. For web services that need frequent security updates, you might prefer package managers.  
Deep Dive: Dependency Management Strategies  
Choosing how to handle dependencies is one of the most critical architectural decisions in C++. Let's compare the approaches:  
1. Vendoring (Our Current Approach)  
# Everything is self-contained in import/  
set  
(  
SDL2\_DIR  
"import/SDL2-2.28.5"  
)  
set  
(  
GLM\_DIR  
"import/glm"  
)  
add\_subdirectory  
(  
${  
SDL2\_DIR  
}  
)  
add\_subdirectory  
(  
${  
GLM\_DIR  
}  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Advantages:  
Reproducible builds  
: Exact same versions everywhere  
Offline builds  
: No internet required after initial clone  
Control  
: Can patch dependencies if needed  
Simplicity  
: No external tools or package managers  
Disadvantages:  
Repository size  
: Our import/ folder is 500MB+  
Update overhead  
: Manual process to update dependencies  
Security  
: Must manually track and update vulnerable dependencies  
License complexity  
: Must distribute all licenses  
2. Package Managers (Conan, vcpkg)  
# With Conan  
include  
(  
conan  
)  
conan\_cmake\_run  
(  
REQUIRES   
 SDL2/2.28.5  
 glm/0.9.9.8  
 BASIC\_SETUP CMAKE\_TARGETS  
 BUILD missing  
)  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE CONAN\_PKG::SDL2 CONAN\_PKG::glm  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Advantages:  
Smaller repos  
: Dependencies downloaded on-demand  
Easy updates  
:  
conan install --update  
Binary packages  
: Pre-built binaries for faster builds  
Ecosystem  
: Thousands of packages available  
Disadvantages:  
External dependency  
: Requires internet and package manager  
Version conflicts  
: Diamond dependency problems  
Platform support  
: Not all packages support all platforms  
Learning curve  
: Another tool to learn and maintain  
3. FetchContent (Modern CMake)  
include  
(  
FetchContent  
)  
FetchContent\_Declare  
(  
glm  
 GIT\_REPOSITORY https://github.com/g-truc/glm.git  
 GIT\_TAG 0.9.9.8  
 GIT\_SHALLOW TRUE  
)  
FetchContent\_Declare  
(  
SDL2  
 URL https://github.com/libsdl-org/SDL/releases/download/release-2.28.5/SDL2-2.28.5.tar.gz  
 URL\_HASH SHA256=332cb37d0be20cb9541739c61f79bae5a477427d79ae85e352089afdaf6666e4  
)  
FetchContent\_MakeAvailable  
(  
glm SDL2  
)  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE glm SDL2::SDL2  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Advantages:  
CMake native  
: No external tools required  
Flexible sources  
: Git repos, URLs, local paths  
Version pinning  
: Exact commits/tags  
Cross-platform  
: Works everywhere CMake works  
Disadvantages:  
Build time  
: Downloads and builds on first configure  
Internet required  
: At least for first build  
No binary caching  
: Always builds from source  
Hybrid Approach: The Best of Both Worlds  
For ColumbaEngine, we could evolve to a hybrid approach:  
# Critical, stable dependencies: Vendor them  
set  
(  
SDL2\_DIR  
"import/SDL2-2.28.5"  
)  
add\_subdirectory  
(  
${  
SDL2\_DIR  
}  
)  
# Development/testing dependencies: FetchContent  
if  
(  
BUILD\_TESTS  
)  
include  
(  
FetchContent  
)  
FetchContent\_Declare  
(  
googletest  
 GIT\_REPOSITORY https://github.com/google/googletest.git  
 GIT\_TAG v1.14.0  
 GIT\_SHALLOW TRUE  
)  
FetchContent\_MakeAvailable  
(  
googletest  
)  
endif  
()  
# Optional dependencies: find\_package with fallback  
find\_package  
(  
Doxygen  
)  
if  
(  
NOT Doxygen\_FOUND AND ENABLE\_DOCS  
)  
FetchContent\_Declare  
(  
doxygen  
 URL https://github.com/doxygen/doxygen/releases/download/Release\_1\_9\_8/doxygen-1.9.8.src.tar.gz  
)  
FetchContent\_MakeAvailable  
(  
doxygen  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Step 3: Cross-Platform Reality Check  
Our engine supports three platforms, with different flags and need:  
Native  
(Windows/Linux): Build everything from source  
Emscripten  
(Web): Use system-provided libraries  
if  
(  
${  
CMAKE\_SYSTEM\_NAME  
}  
MATCHES  
"Emscripten"  
)  
# Web build - use Emscripten's built-in libraries  
set  
(  
USE\_FLAGS  
"-O3 -sUSE\_SDL=2 -sUSE\_SDL\_MIXER=2 -sUSE\_FREETYPE=1 -fwasm-exceptions"  
)  
set  
(  
CMAKE\_EXE\_LINKER\_FLAGS  
"  
${  
CMAKE\_EXE\_LINKER\_FLAGS  
}  
${  
USE\_FLAGS  
}  
"  
)  
else  
()  
# Native build - compile dependencies from source  
add\_subdirectory  
(  
${  
SDL2\_DIR  
}  
)  
add\_subdirectory  
(  
${  
SDL2MIXER\_DIR  
}  
)  
add\_subdirectory  
(  
${  
GLEW\_DIR  
}  
)  
add\_subdirectory  
(  
${  
TTF\_DIR  
}  
)  
endif  
()  
# GLM is header-only, works everywhere  
add\_subdirectory  
(  
${  
GLM\_DIR  
}  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Platform detection patterns:  
CMAKE\_SYSTEM\_NAME  
: Target platform (Windows, Linux, Darwin, Emscripten)  
CMAKE\_HOST\_SYSTEM  
: Build machine platform  
Use generator expressions for conditional compilation:  
$<$<PLATFORM\_ID:Windows>:WIN32\_CODE>  
Step 4: Creating the Main Target  
Now for the heart of our build - the engine library itself:  
# List all source files (80+ files in our case!)  
set  
(  
ENGINESOURCE  
 src/Engine/window.cpp  
 src/Engine/configuration.cpp  
 src/Engine/logger.cpp  
# ... 70+ more files  
src/Engine/UI/uisystem.cpp  
)  
# Create the static library  
add\_library  
(  
ColumbaEngine STATIC  
${  
ENGINESOURCE  
}  
)  
# Modern CMake magic - precompiled headers for faster builds  
target\_precompile\_headers  
(  
ColumbaEngine PUBLIC src/Engine/stdafx.h  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Precompiled headers  
can cut build times by 50%+ in large projects. They compile commonly used headers once and reuse the result.  
Precompiled Headers - The Build Speed Game Changer  
Precompiled headers (PCH) are one of the most impactful optimizations for C++ build times, yet they're often overlooked. Let's understand how they work:  
The Problem  
: Header Parsing Overhead  
// Every .cpp file typically includes these  
#include  
<iostream>  
// ~15,000 lines when fully expanded  
#include  
<vector>  
// ~8,000 lines  
#include  
<string>  
// ~12,000 lines  
#include  
<memory>  
// ~6,000 lines  
#include  
<SDL2/SDL.h>  
// ~25,000 lines  
// Total: ~66,000 lines to parse per source file!  
Enter fullscreen mode  
Exit fullscreen mode  
With 80 source files, that's  
5.2 million lines  
of redundant header parsing!  
The Solution  
: Precompiled Headers  
target\_precompile\_headers  
(  
ColumbaEngine PUBLIC src/Engine/stdafx.h  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Our  
stdafx.h  
contains the most commonly used headers:  
// stdafx.h - precompiled header  
#pragma once  
// Standard library  
#include  
<iostream>  
#include  
<vector>  
#include  
<string>  
#include  
<memory>  
#include  
<unordered\_map>  
#include  
<algorithm>  
// Third-party libraries  
#include  
<SDL2/SDL.h>  
#include  
<glm/glm.hpp>  
#include  
<glm/gtc/matrix\_transform.hpp>  
// Engine fundamentals used everywhere  
#include  
"types.h"  
#include  
"logger.h"  
#include  
"configuration.h"  
Enter fullscreen mode  
Exit fullscreen mode  
How It Works  
:  
Compilation  
: CMake compiles  
stdafx.h  
once into  
stdafx.h.gch  
(GCC) or  
stdafx.pch  
(MSVC)  
Reuse  
: Every source file automatically uses the precompiled version  
Speed  
: 66,000 lines → 0 lines to parse per file  
Build Time Results  
(80 source files):  
Without PCH  
: ~8 minutes clean build  
With PCH  
: ~3 minutes clean build  
Incremental  
: Single file changes drop from 15s to 3s  
PUBLIC vs PRIVATE PCH  
:  
# PUBLIC: Users of your library get the PCH benefits too  
target\_precompile\_headers  
(  
ColumbaEngine PUBLIC stdafx.h  
)  
# PRIVATE: Only internal compilation uses PCH  
target\_precompile\_headers  
(  
ColumbaEngine PRIVATE stdafx.h  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Best Practices for PCH  
:  
// Good PCH content: Stable, frequently used headers  
#include  
<vector>  
// Used in 90% of files  
#include  
<SDL2/SDL.h>  
// Core dependency  
// Bad PCH content: Frequently changing headers  
#include  
"GameLogic.h"  
// Changes often, invalidates PCH  
#include  
"debug\_temp.h"  
// Temporary debugging code  
Enter fullscreen mode  
Exit fullscreen mode  
PCH Pitfalls  
:  
Overuse  
: Adding changing headers defeats the purpose  
Platform differences  
: MSVC vs GCC handle PCH differently  
Dependencies  
: PCH creates implicit dependencies between files  
Step 5: Usage Requirements - The Secret Sauce  
This is where modern CMake really shines. Instead of users manually figuring out include paths and linking, we specify  
usage requirements  
:  
target\_include\_directories  
(  
ColumbaEngine PUBLIC  
# When building this library  
$<BUILD\_INTERFACE:  
${  
CMAKE\_CURRENT\_SOURCE\_DIR  
}  
/src/Engine>  
 $<BUILD\_INTERFACE:  
${  
CMAKE\_CURRENT\_SOURCE\_DIR  
}  
/src/GameElements>  
# When using installed version  
$<INSTALL\_INTERFACE:include/ColumbaEngine>  
 $<INSTALL\_INTERFACE:include/ColumbaEngine/GameElements>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Generator expressions  
(  
$<...>  
) are evaluated during build generation:  
BUILD\_INTERFACE  
: Only when building this project  
INSTALL\_INTERFACE  
: Only when using the installed version  
$<CONFIG:Debug>  
: Only in Debug builds  
$<PLATFORM\_ID:Windows>  
: Only on Windows  
Why Generator Expressions Matter  
Let's understand why this is so powerful. Consider this traditional approach:  
# The old, problematic way  
if  
(  
CMAKE\_BUILD\_TYPE STREQUAL  
"Debug"  
)  
set  
(  
CMAKE\_CXX\_FLAGS  
"  
${  
CMAKE\_CXX\_FLAGS  
}  
-DDEBUG\_MODE"  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Problems with this approach:  
Build-time evaluation  
: The condition is checked when CMake configures, not when you build  
Global pollution  
: Affects ALL targets in the project  
Multi-config generators  
: Breaks with Visual Studio, Xcode which support multiple configs  
Generator expressions solve this:  
# Modern, correct approach  
target\_compile\_definitions  
(  
MyTarget PRIVATE   
 $<$<CONFIG:Debug>:DEBUG\_MODE>  
 $<$<CONFIG:Release>:NDEBUG>  
 $<$<PLATFORM\_ID:Windows>:WIN32\_LEAN\_AND\_MEAN>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
This is evaluated  
per-target  
,  
per-configuration  
,  
at build time  
. Much more flexible and robust.  
Common Generator Expression Patterns  
# Conditional compilation flags  
target\_compile\_options  
(  
MyTarget PRIVATE  
 $<$<CXX\_COMPILER\_ID:GNU,Clang>:-Wall -Wextra>  
 $<$<CXX\_COMPILER\_ID:MSVC>:/W4>  
)  
# Debug vs Release libraries  
target\_link\_libraries  
(  
MyTarget PRIVATE  
 $<$<CONFIG:Debug>:MyLib\_d>  
 $<$<CONFIG:Release>:MyLib>  
)  
# Platform-specific linking  
target\_link\_libraries  
(  
MyTarget PRIVATE  
 $<$<PLATFORM\_ID:Windows>:ws2\_32>  
 $<$<PLATFORM\_ID:Linux>:pthread>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Step 6: Dependency Linking - Getting Scope Right  
Here's a crucial concept:  
PUBLIC vs PRIVATE vs INTERFACE  
if  
(  
${  
CMAKE\_SYSTEM\_NAME  
}  
MATCHES  
"Emscripten"  
)  
# Web build  
target\_link\_libraries  
(  
ColumbaEngine PUBLIC glm  
)  
else  
()  
# Native build - note the scoping!  
target\_link\_libraries  
(  
ColumbaEngine PUBLIC   
 SDL2::SDL2-static  
# Users need SDL2 headers  
glm  
# Header-only math library  
OpenGL::GL  
# Graphics API  
)  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE   
 libglew\_static  
# Internal OpenGL extension loading  
freetype  
# Internal text rendering  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Scope meanings:  
PUBLIC  
: "I use this, and my users will need it too"  
PRIVATE  
: "I use this internally, but my users don't need to know"  
INTERFACE  
: "I don't use this, but my users will need it"  
Get this right, and users of your library automatically get the right dependencies. Get it wrong, and you'll have angry developers filing issues.  
Understanding Transitive Dependencies  
Let's look at a real example from our engine. Imagine this dependency chain:  
MyGame → ColumbaEngine → SDL2 → OpenGL  
Enter fullscreen mode  
Exit fullscreen mode  
If we declare SDL2 as  
PRIVATE  
to ColumbaEngine:  
target\_link\_libraries  
(  
ColumbaEngine PRIVATE SDL2::SDL2-static  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Problem  
:  
MyGame  
won't be able to use SDL2 types in the public API. If ColumbaEngine's headers include  
<SDL2/SDL.h>  
, users get compile errors.  
If we declare SDL2 as  
PUBLIC  
:  
target\_link\_libraries  
(  
ColumbaEngine PUBLIC SDL2::SDL2-static  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Result  
:  
MyGame  
automatically gets SDL2 headers and libraries. Perfect for our engine where users need SDL2 types.  
The INTERFACE Scope Mystery  
INTERFACE  
is the trickiest to understand. It means "I don't use this, but my users will."  
Real-world example  
: Header-only libraries with dependencies  
# HeaderOnlyMath library depends on Eigen, but is itself header-only  
add\_library  
(  
HeaderOnlyMath INTERFACE  
)  
target\_link\_libraries  
(  
HeaderOnlyMath INTERFACE Eigen3::Eigen  
)  
# Users of HeaderOnlyMath automatically get Eigen  
add\_executable  
(  
MyApp main.cpp  
)  
target\_link\_libraries  
(  
MyApp PRIVATE HeaderOnlyMath  
)  
# Gets Eigen too!  
Enter fullscreen mode  
Exit fullscreen mode  
Mixing Scopes: The Real World  
Most real projects use mixed scopes:  
target\_link\_libraries  
(  
ColumbaEngine  
# Users need these APIs  
PUBLIC   
 SDL2::SDL2-static  
# Window/input handling in public API  
glm  
# Math types in public headers  
OpenGL::GL  
# Graphics API exposed to users  
# Internal implementation details  
PRIVATE  
 libglew\_static  
# OpenGL extension loading (wrapped)  
freetype  
# Text rendering (abstracted away)  
${  
CMAKE\_DL\_LIBS  
}  
# Dynamic library loading (platform detail)  
)  
Enter fullscreen mode  
Exit fullscreen mode  
Step 7: Multiple Executables and Examples  
A game engine isn't just a library - it includes tools and examples:  
if  
(  
NOT BUILD\_STATIC\_LIB AND BUILD\_EXAMPLES  
)  
# Main editor application  
add\_executable  
(  
ColumbaEngineEditor src/main.cpp  
)  
target\_sources  
(  
ColumbaEngineEditor PRIVATE  
 src/application.cpp  
 src/Editor/Gui/inspector.cpp  
 src/Editor/Gui/projectmanager.cpp  
)  
target\_link\_libraries  
(  
ColumbaEngineEditor PRIVATE ColumbaEngine  
)  
# Game examples  
add\_executable  
(  
GameOff examples/GameOff/main.cpp  
)  
target\_sources  
(  
GameOff PRIVATE  
 examples/GameOff/application.cpp  
 examples/GameOff/character.cpp  
 examples/GameOff/inventory.cpp  
)  
target\_link\_libraries  
(  
GameOff PRIVATE ColumbaEngine  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
Pattern  
: Create multiple executables that all link to your main library. This way, the library code is compiled once and reused.  
Step 8: Testing Infrastructure  
Professional projects need tests:  
if  
(  
NOT BUILD\_STATIC\_LIB  
)  
enable\_testing  
()  
add\_subdirectory  
(  
${  
GTEST\_DIR  
}  
)  
add\_executable  
(  
t1 test/maintest.cc  
)  
target\_sources  
(  
t1 PRIVATE  
 test/collision2d.cc  
 test/ecssystem.cc  
 test/interpreter.cc  
 test/renderer.cc  
)  
target\_link\_libraries  
(  
t1 PRIVATE gtest gtest\_main ColumbaEngine  
)  
# Auto-discover tests  
include  
(  
GoogleTest  
)  
gtest\_discover\_tests  
(  
t1  
)  
endif  
()  
Enter fullscreen mode  
Exit fullscreen mode  
gtest\_discover\_tests()  
automatically finds all test cases and creates individual CTest entries. Run with  
ctest  
or  
make test  
.  
Common Build Pitfalls  
1. Global Variables vs Target Properties  
# BAD: Affects everything globally  
set  
(  
CMAKE\_CXX\_FLAGS  
"  
${  
CMAKE\_CXX\_FLAGS  
}  
-DDEBUG"  
)  
# GOOD: Only affects specific target  
target\_compile\_definitions  
(  
MyTarget PRIVATE DEBUG  
)  
Enter fullscreen mode  
Exit fullscreen mode  
2. Not Using Generator Expressions  
# BAD: Debug symbols in release builds  
target\_compile\_options  
(  
MyTarget PRIVATE -g  
)  
# GOOD: Only in debug builds  
target\_compile\_options  
(  
MyTarget PRIVATE $<$<CONFIG:Debug>:-g>  
)  
Enter fullscreen mode  
Exit fullscreen mode  
3. Wrong Dependency Scopes  
# If users of your library need OpenGL headers:  
target\_link\_libraries  
(  
MyLib PUBLIC OpenGL::GL  
)  
# If only your implementation uses OpenGL:  
target\_link\_libraries  
(  
MyLib PRIVATE OpenGL::GL  
)  
Enter fullscreen mode  
Exit fullscreen mode  
What We've Achieved: Build System Results  
After implementing all these concepts, here's what we achieved:  
Developer Experience:  
# Simple build  
git clone https://github.com/user/ColumbaEngine  
cd  
ColumbaEngine  
mkdir  
build  
&&  
cd  
build  
cmake ..  
make  
-j8  
# Custom configuration  
cmake  
-DBUILD\_EXAMPLES  
=  
OFF  
-DCMAKE\_BUILD\_TYPE  
=  
Release ..  
# Run tests  
ctest  
Enter fullscreen mode  
Exit fullscreen mode  
Cross-platform Support:  
Same CMakeLists.txt works on Windows, Linux, and web  
Automatic dependency resolution per platform  
Platform-specific optimizations where needed  
Build Performance:  
Precompiled headers cut build times by 60%  
Parallel compilation across all targets  
Incremental builds under 10 seconds for single-file changes  
Coming Up in Part 2  
In the next article, we'll cover the  
deployment and distribution  
side of CMake:  
Installation Systems  
: How to make your library usable by others with  
find\_package()  
Export/Import Mechanisms  
: The complex but powerful system that makes modern CMake libraries work  
Package Configuration  
: Creating relocatable, dependency-aware packages  
CPack Integration  
: Generating professional installers for multiple platforms  
Real-world Distribution  
: Getting your library into the hands of users  
Conclusion  
Building a robust CMake build system for complex projects requires understanding several key concepts:  
Target-centric thinking  
- Everything revolves around targets and their usage requirements  
Proper dependency management  
- Choose the right strategy for your project's needs  
Cross-platform abstractions  
- Let CMake handle platform differences  
Generator expressions  
- Enable conditional behavior without complex if/else logic  
Build optimization  
- Use precompiled headers and other modern features  
The  
game engine  
we've built demonstrates these concepts in action with a real, production-ready build system that handles complex, multi-platform C++ projects.  
Remember: CMake is about expressing  
intent  
, not  
implementation details  
. Focus on what you want to achieve, and let CMake figure out how to do it on each platform.  
Don't miss [Part 2] where we'll cover installation, packaging, and distribution - making your library actually usable by the world!  
What's your biggest CMake build challenge?  
Share your experiences in the comments below!  
The complete source code for ColumbaEngine with all the CMake patterns from this series is available on  
GitHub  
. These patterns scale from small libraries to large, complex applications.