Managing dependencies with CMake

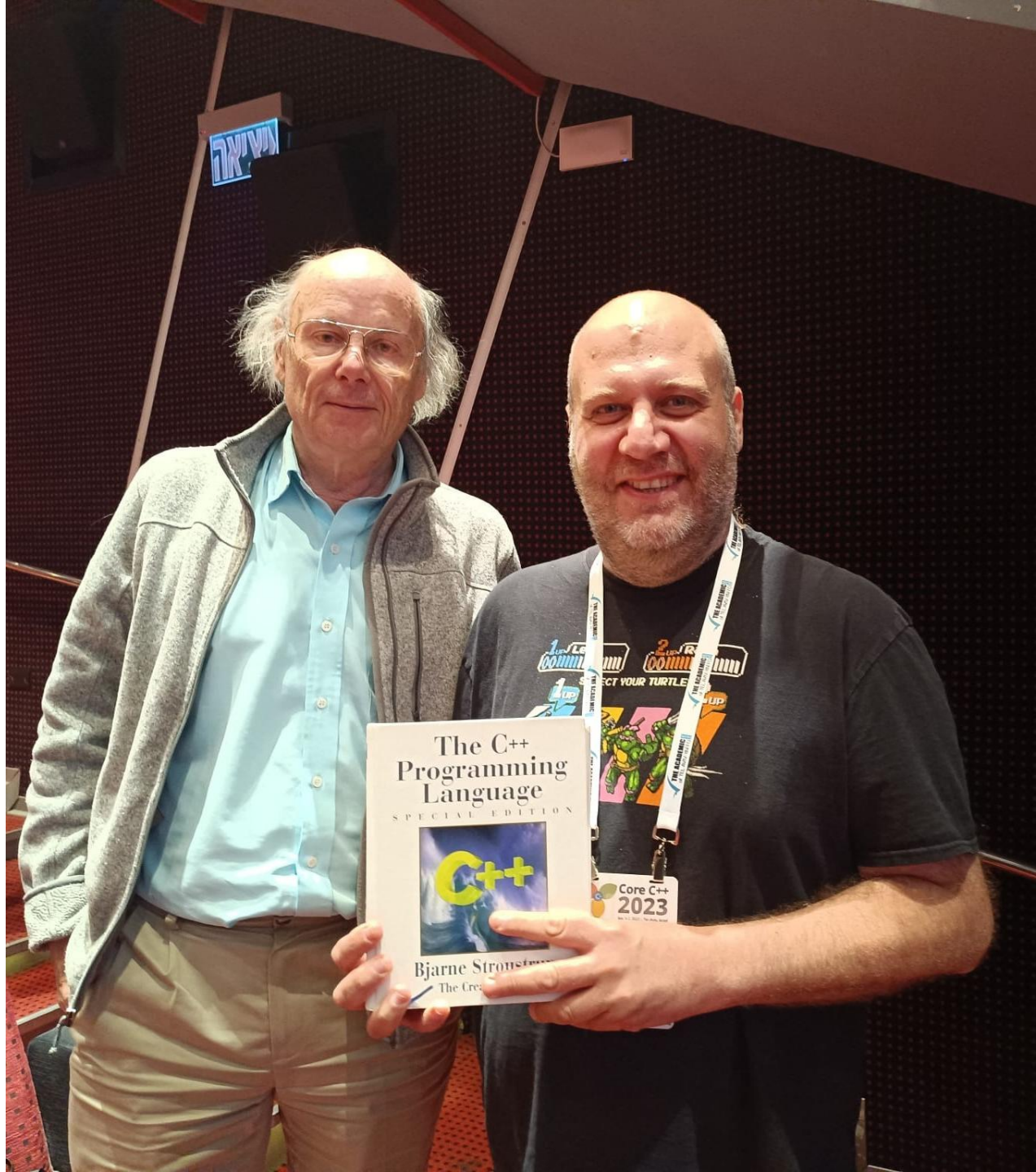
Alex Kushnir
Software engineer, Johnson&Johnson MedTech
19-Oct-2025
Core C++ 2025 conference, Tel-Aviv

Johnson&Johnson
MedTech

Electrophysiology

About Me

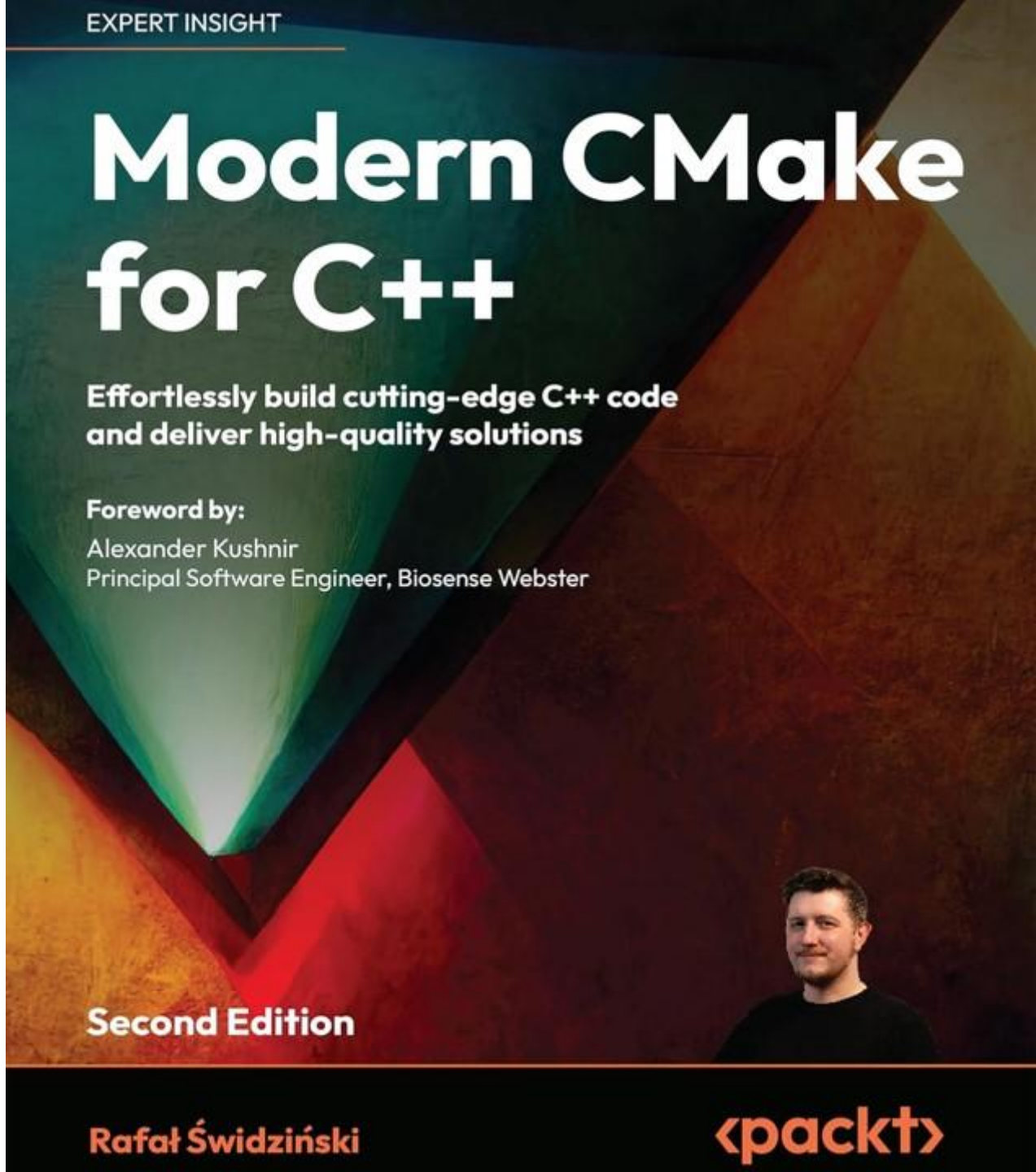
- Software engineer since 2007
- Mostly in embedded and low-level domains
- C++ = ♡
- Focusing on methodologies and tools



Modern CMake for C++

- Wrote a foreword for the book
- Reached out via LinkedIn
- Amazon bestseller

Best Sellers Rank: #131,969 in Books (See Top 100 in Books)
#8 in Software Programming Compilers
#18 in C++ Programming Language
#24 in Software Design Tools

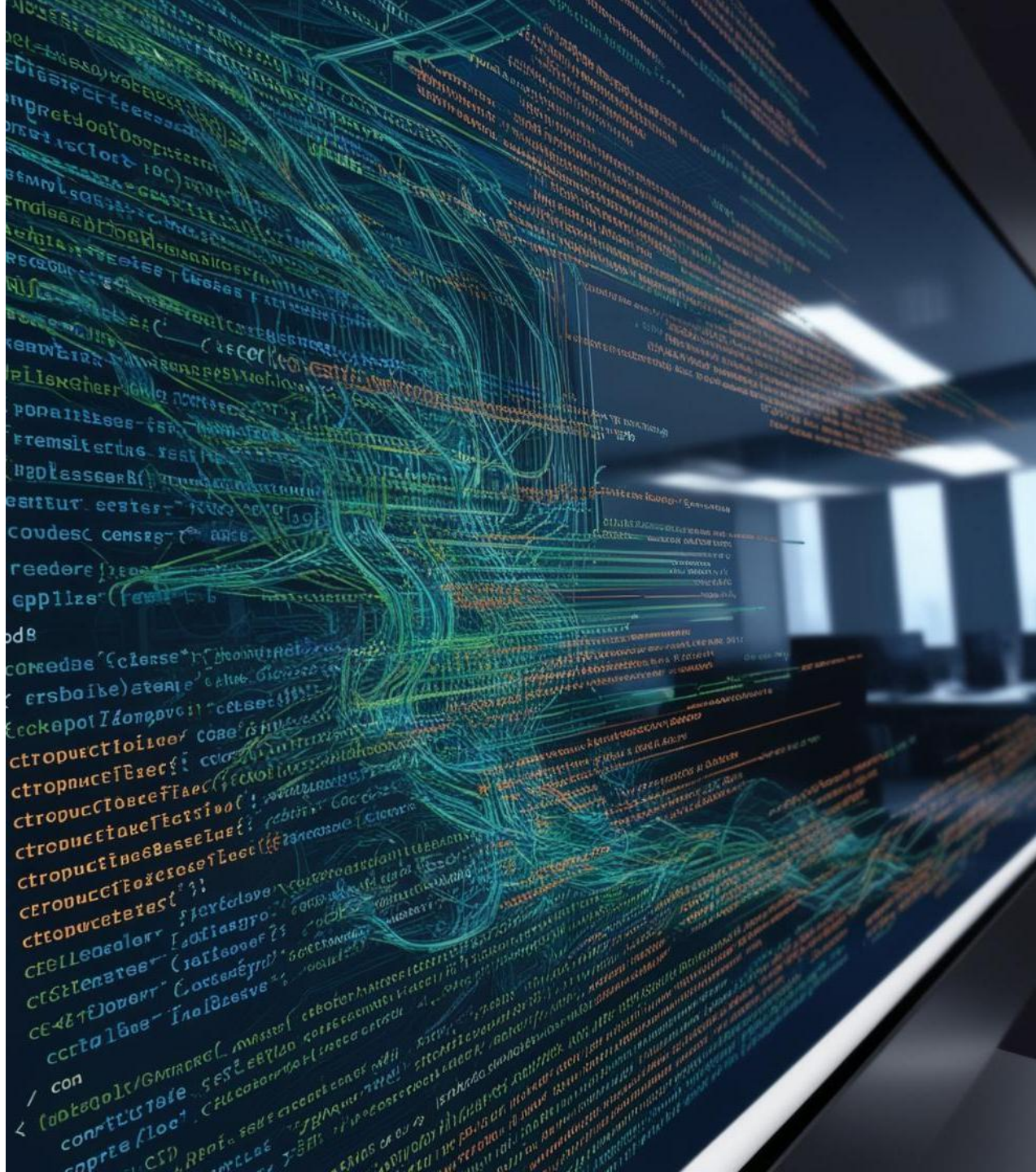


J&J MedTech - Electrophysiology

- A global leader in diagnosing and treating complex arrhythmias
- Our mission is to cure AFib
- We are a team of professionals within wide spectrum of domains
- Software and hardware engineers, physicians, clinical specialists and many more

Our Software Team

- ~70 engineers
- ~4.5M LOC
- Modern C++
- C# for UI
- Windows, embedded devices
- We are hiring!
  Visit our booth outside



Agenda

- Introduction
- The dependency problem in C++
- Handling dependencies with CMake
- Summary
- Q&A

The importance of build system

- Simplifies and manages complex dependencies and configurations
- Improves build performance and maintainability
- Smooth CI/CD workflows, which are essential for efficient development.

CMake Crash Course

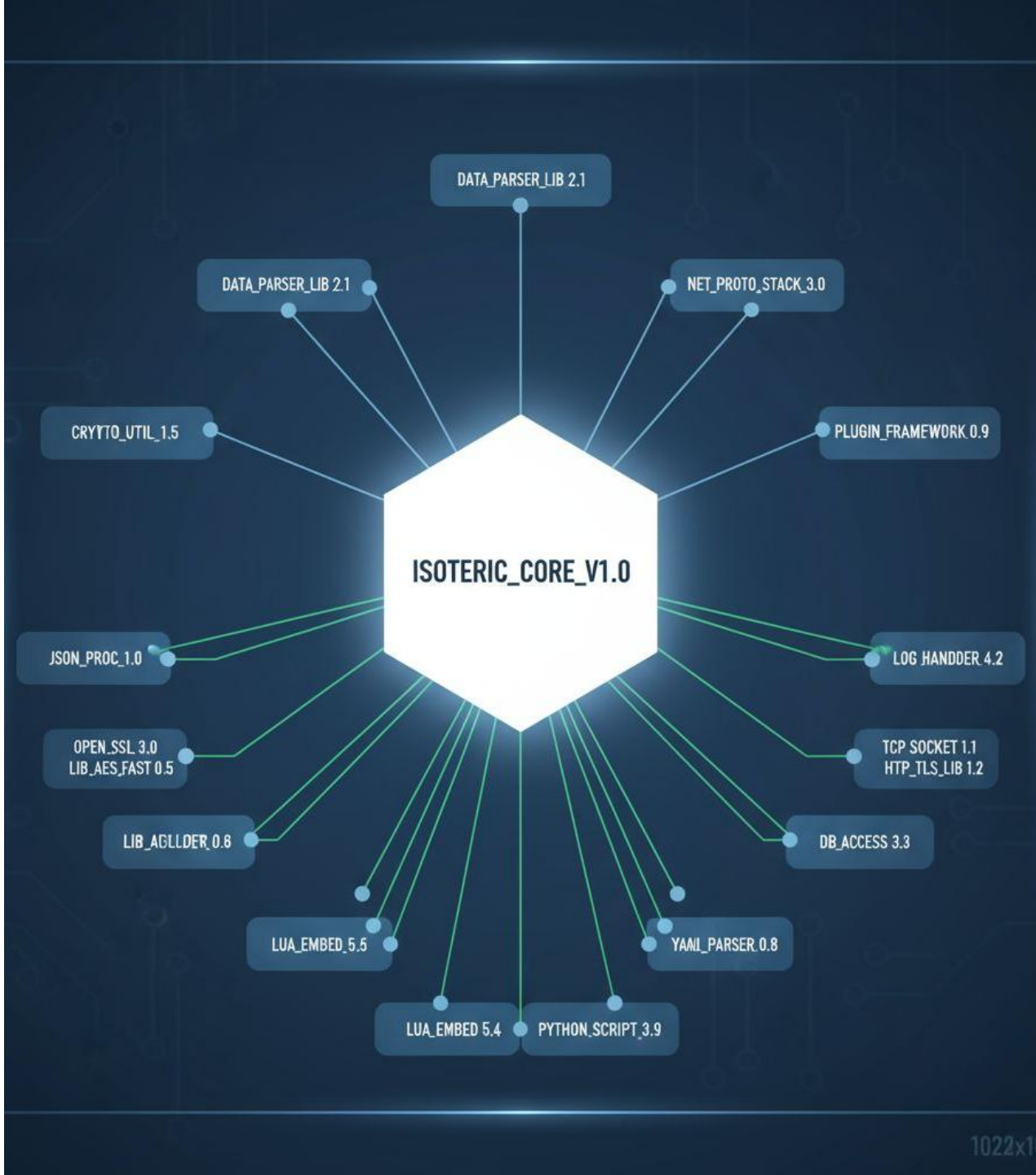
CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28)
project(minimal_project)

add_executable(minimal_project Main.cpp)
```

- **CMake targets** - named entities representing buildable components created by various commands that encapsulate their source files, compile options, include directories, and dependencies as properties
- **CMake properties** are key-value pairs attached to specific CMake objects that store configuration settings, build options, and metadata
- **CMake variables** are temporary named storage containers for strings, paths, flags, and configuration values with different scopes (normal, cache, environment) for controlling visibility and persistence

The Dependency Problem In C++

- No standard package manager (*)
- Manual header/library management
- Versions conflict
- Cross-platform build complexity
- CMake to the rescue!



Handling Dependencies in CMake

- Use pre-installed packages
- Fetch from network
- Use package managers
- Integrate package managers with CMake

Using Pre-installed Packages

- Find out where is the package in the file system
- Not all environments are the same
- Finding these paths should happen during the build process
- CMake built-in `find_package()` command does that for us
- Looks for a specific configuration files, and populates several variables that can be used in the listfile

find_package() - Deep Dive

- Built-in command
- A package should provide a config file
  to allow CMake to discover it using `find_package()`
- CMake comes with over 180 modules that can find popular libraries:
  - Boost, curl, OpenSSL, zlib, X11, Qt, and many more
- Several standard search locations (user-provided, system standard, etc.)
- Can be overridden by hinting CMake where the package is

find_package() - Example (1)

Main.cpp

```cpp
#include <boost/timer/timer.hpp>
#include <print>

int main()
{
    boost::timer::auto_cpu_timer t1;
    for (auto i = 0; i < 100000; ++i)
    {
        std::println("Index is {}", i);
    }
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28)
project(boost-timer-demo)
find_package(Boost REQUIRED ALL)
if (Boost_FOUND)
Main.cpp)
target_include_directories(boost_timer PRIVATE ${Boost_INCLUDE_DIRS})
target_link_libraries(boost_timer PRIVATE ${Boost_LIBRARIES})
    message(FATAL_ERROR "Boost libraries were not found!")
endif()
```

Core C++ 2025

Boost libraries found

/bin/bash

```
Boost libraries found: Boost::atomic;Boost::chrono;Boost::container;Boost::context;
Boost::coroutine;Boost::date_time;Boost::exception;Boost::fiber;Boost::filesystem;
Boost::graph;Boost::graph_parallel;Boost::iostreams;Boost::json;Boost::locale;
Boost::log;Boost::log_setup;Boost::math_c99;Boost::math_c99f;Boost::math_c99l;
Boost::math_tr1;Boost::math_tr1f;Boost::math_tr1l;Boost::mpi;Boost::mpi_python;
Boost::nowide;Boost::numpy;Boost::prg_exec_monitor;Boost::program_options;Boost::python;
Boost::random;Boost::regex;Boost::serialization;Boost::stacktrace_addr2line;
Boost::stacktrace_backtrace;Boost::stacktrace_basic;Boost::stacktrace_noop;Boost::system;
Boost::test_exec_monitor;Boost::thread;Boost::timer;Boost::type_erasure;Boost::unit_test_framework;
Boost::url;Boost::wave;Boost::wserialization
```

find_package() - Working With Targets

- Usage requirement propagation
  - Automatically include only what you need, and its dependencies
  - Automatically sets required compile flags
- Transitive dependencies
  - Imported targets carry both build requirements and usage requirements
    - PRIVATE: Used only when building the target
    - INTERFACE: Propagated to the consumers of the target
    - PUBLIC: Both private and interface (used for both building and consuming)
- Platform abstraction
  - Works identically on all platforms
    - Agnostic to library naming conventions
    - Debug/Release configurations are set automatically
- Namespace protection
  - Modern imported targets use namespaces (e.g., Boost::), preventing conflicts and making depxplicit

find_package() - Example (2)

Main.cpp

```cpp
#include <boost/timer/timer.hpp>
#include <print>

int main()
{
    boost::timer::auto_cpu_timer t1;
    for (auto i = 0; i < 100000; ++i)
    {
        std::println("Index is {}", i);
    }
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28)
project(boost-timer-demo)
set(CMAKE_CXX_STANDARD 23)

find_package(Boost REQUIRED ALL)

if (Boost_FOUND)
    add_executable(boost_timer Main.cpp)
    target_link_libraries(boost_timer PRIVATE Boost::timer)
    message(FATAL_ERROR "Boost libraries were not found!")
endif()
```

Core C++ 2025

FetchContent module

- Modern approach to package management
- Downloads and integrates projects during configure stage
- Treats the dependencies as if they were a part of the project
- Able to build the dependencies from source code
- All targets become available
- Interfaces are propagated to the project

FetchContent - Example

Main.cpp

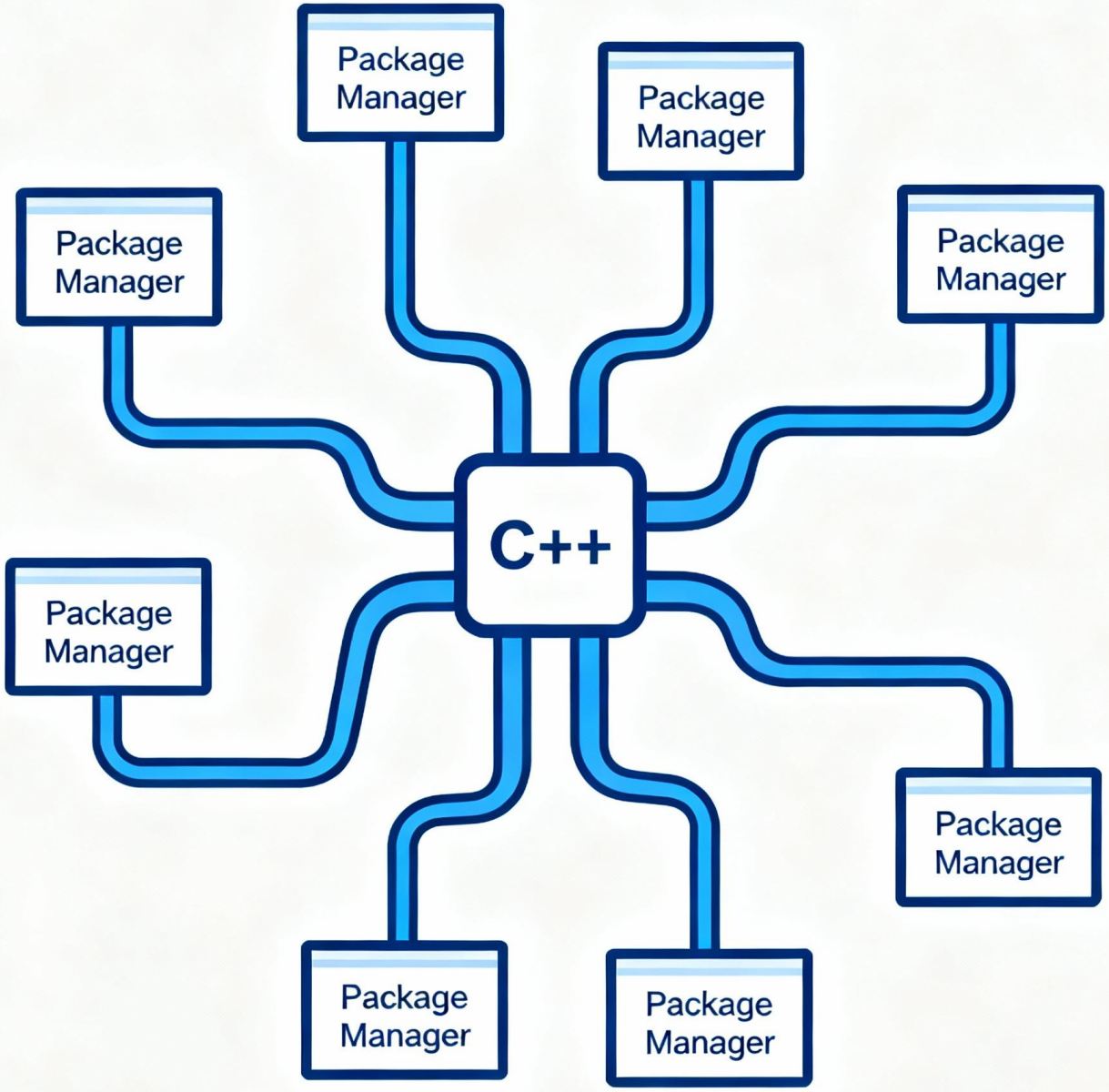
```cpp
#include "gtest/gtest.h"
#include <map>
#include <exception>

TEST(FetchContent, Test1)
{
    ASSERT_EQ(2 + 2, 4);
}

TEST(FetchContent, Test2)
{
    std::map<int, int> m;

    ASSERT_THROW(m.at(7), std::out_of_range);
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.28)
project(fetch-content-demo)

include(FetchContent)

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG main
)

FetchContent_MakeAvailable(googletest)

add_executable(fetch_content_tests Main.cpp)

target_link_libraries(fetch_content_tests gtest_main gmock)
```

But...we want C++ package manager!

- Several package managers exist
- No standard solution
- Conan has emerged as the most comprehensive solution



Integration with Conan – Example (1)

conanfile.txt

```
[requires]
nlohmann_json/3.12.0

[generators]
CMakeDeps
CMakeToolchain
```

CMakeLists.txt

```
if (NOT CMAKE_TOOLCHAIN_FILE)
    set(CMAKE_TOOLCHAIN_FILE conan_toolchain.cmake)
endif()
```

```
if (NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release)
endif()

project(json_parser)

find_package(nlohmann_json REQUIRED)

add_executable(json_parser Main.cpp)
target_link_libraries(json_parser
    PRIVATE
    nlohmann_json::nlohmann_json)

configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/example.json
    ${CMAKE_CURRENT_BINARY_DIR}
    COPYONLY)
```

Integration with Conan – Example

Main.cpp

```cpp
#include "nlohmann/json.hpp"
#include <fstream>
#include <iostream>

int main(int argc, char** argv)
{
    if (argc < 2)
    {
        std::cout << "Usage: json_parser <path_to_json>\n";
        return 1;
    }

    auto json_path = argv[1];

    nlohmann::json data = nlohmann::json::parse(
        std::ifstream{json_path});

    for (const auto& field : data.items())
    {
        std::cout << field.key() << " : " <<  field.value() << "\n";
    }

    return 0;
}
```

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.28)

if (NOT CMAKE_TOOLCHAIN_FILE)
    set(CMAKE_TOOLCHAIN_FILE conan_toolchain.cmake)
endif()

    set(CMAKE_BUILD_TYPE Release)
endif()

find_package(nlohmann_json REQUIRED)

target_link_libraries(json_parser
    PRIVATE
    nlohmann_json::nlohmann_json)

configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/example.json
    ${CMAKE_CURRENT_BINARY_DIR}
    COPYONLY)
```

Dependency providers

- Unifies dependency acquisition methods
- Transparent integration
- Centralized control
- Flexible fallbacks
- Team standardization
- Simply set the `DCMAKE_PROJECT_TOP_LEVEL_INCLUDES` with your custom provider script

Dependency providers – how it works?

- Intercepts find_package() and FetchContent_MakeAvailable()
- Allowing external package managers to handle dependency resolution transparently
- Represent the evolution toward language-standard package management
- Enabling seamless migration between package managers without project code changes.
- This is CMake's answer to the fragmented C++ ecosystem

Conan as dependency provider

Main.cpp

```cpp
#include "nlohmann/json.hpp"
#include <fstream>
#include <iostream>

int main(int argc, char** argv)
{
    if (argc < 2)
    {
        std::cout << "Usage: json_parser <path_to_json>\n";
        return 1;
    }

    auto json_path = argv[1];

    nlohmann::json data = nlohmann::json::parse(
        std::ifstream{json_path});

    for (const auto& field : data.items())
    {
        std::cout << field.key() << " : " <<  field.value() << "\n";
    }

    return 0;
}
```

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.28)
if (NOT CMAKE_PROJECT_TOP_LEVEL_INCLUDES)
    message("CMAKE_PROJECT_TOP_LEVEL_INCLUDES was not set!")
    set(CMAKE_PROJECT_TOP_LEVEL_INCLUDES
        ${CMAKE_SOURCE_DIR}/cmake/conan_provider.cmake)
endif()
project(json_parser)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

if (NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release)
endif()

find_package(nlohmann_json REQUIRED)

add_executable(json_parser main.cpp)
target_link_libraries(json_parser PRIVATE nlohmann_json::nlohmann_json)
configure_file(
    ${CMAKE_CURRENT_SOURCE_DIR}/example.json ${CMAKE_CURRENT_BINARY_DIR} COPYONLY)
```


```cmake
set(CMAKE_PROJECT_TOP_LEVEL_INCLUDES
    ${CMAKE_SOURCE_DIR}/cmake/conan_provider.cmake)
```

```cmake
find_package(nlohmann_json REQUIRED)
```

Methods comparison

Aspect	FetchContent	Conan + find_package
Build Speed	❌ **Slow** - Builds from source every time	✅ **Fast** - Uses precompiled binaries
Setup Complexity	✅ **Simple** - Built into CMake 3.11+	❌ **Complex** - Requires external tool and profiles
Dependency Resolution	❌ **Basic** - "First declare wins" conflicts	✅ **Advanced** - Handles complex dependency graphs
Package Ecosystem	⚠️ **Limited** - CMake-compatible projects only	✅ **Extensive** - 2000+ packages in ConanCenter
CI/CD Performance	❌ **Poor** - Rebuilds dependencies each run	✅ **Excellent** - Downloads prebuilt packages

- **Choose FetchContent** for small projects with few dependencies where simplicity matters
- **Choose Conan** for large projects where build speed and complex dependency management are critical

The modern hierarchy

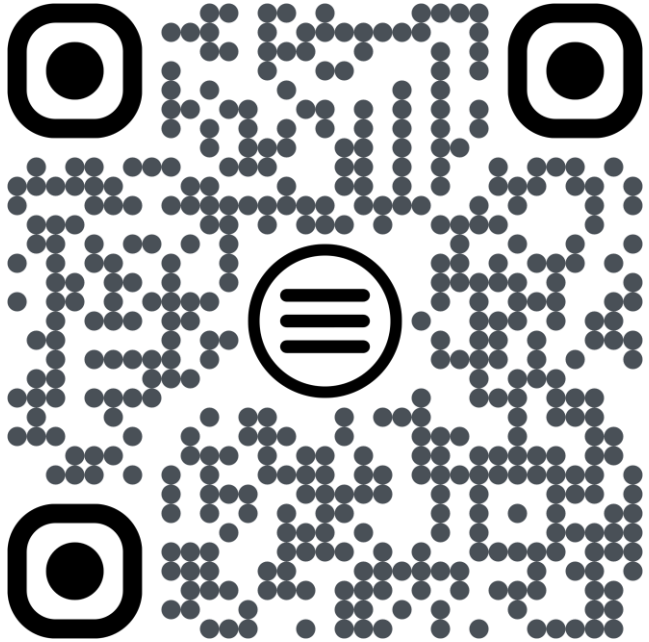
Priority	Method	When to use
1	**Dependency Providers**	Future-proof solution, unified interface
2	**FetchContent**	Source-based builds, modern CMake projects
3	**find_package + Conan**	Complex dependency graphs, enterprise projects
Legacy	**Git Submodules/Manual**	Avoid for new projects

Summary

- Choose the right tool for project
  - **Small project** – FetchContent for simplicity and full control
  - **Medium project** – Consider Conan for better build performance
  - **Large project** – Dependency providers

Questions?

Thank you



My e-card

https://github.com/alexkushnir/core_cpp_2025_talk