BUILD SYSTEM USING CMAKE

Commands :
cmake commands -
cmake -G "build system" <cmakelists path>
cmake --build . --target help
cmake --system-information info.txt
cmake -GNinja -D CMAKE_CXX_COMPILER=cl <path_to_cmakelists_file>
cmake -Bbuild -> same as doing mkdir build, cd build , cmake ..

What is Build System :
A build system is a collection of tools, scripts, and processes used to automate the creation of software from source code. It manages the compilation, linking, and packaging of code into executables, libraries, or other output artifacts, ensuring that code is built efficiently, consistently, and with proper dependency tracking.

Types of Build Systems:

- **Manual Scripts**: Simple but not scalable; good for small projects.
- **Make-Based Systems**: Traditional and widespread but platform-dependent.
- **CMake**: A cross-platform meta-build system that generates project files for native build systems.
- **Ninja**: Ultra-fast, often used as a backend for other build tools.
- **SCons**: Python-based and highly flexible, but slower and less commonly used.
- **Visual Studio/MSBuild**: Windows-centric with deep IDE integration.
- **Xcode Build System**: Native to macOS/iOS, but not cross-platform.
- **Bazel**: Scalable for large projects, especially for multi-language builds.

CMAKE
Overview :

Why is CMake Needed?
CMake (Cross-Platform Make) is a meta-build system that generates build files for different native build systems like Make, Ninja, or Visual Studio. It was created to solve several problems with traditional build tools:

Cross-Platform Support: CMake is designed to work on multiple platforms (Linux, macOS, Windows) and with different compilers. It abstracts away platform-specific details and allows developers to write a single configuration file (CMakeLists.txt) for all platforms.

Ease of Use: CMake provides a higher-level syntax that is easier to understand and manage than Makefiles, especially for large projects. It supports modern C++ features and can handle complex dependency graphs more gracefully.

Dependency Management: CMake can automatically find and configure external libraries and dependencies (like Boost, OpenSSL, etc.) and ensure they are properly linked with the project.

Integration with Other Tools: CMake can generate build files for a variety of build systems (Make, Ninja, Visual Studio, Xcode, etc.), providing flexibility to developers working in different environments.

Out-of-Source Builds: CMake supports out-of-source builds, which means that all build artifacts (object files, executables, etc.) can be placed in a separate directory, keeping the source directory clean.

Maintainability and Readability: CMake's configuration files are easier to maintain, especially for complex projects with multiple modules, libraries, and dependencies.

CMake is a powerful tool for managing the build process of C++ projects, especially when dealing with complex projects or cross-platform development.

Step 1: Understanding CMake
- CMake is a cross-platform build system generator. It generates build files for native build systems like Make, Ninja, Visual Studio,Scons, Ant , nmake etc.

- CMakeLists.txt is the configuration file where you define the structure of your project.

Step 2: Install CMake
Make sure you have CMake installed on your system.
- Windows: Download from [cmake.org](https://cmake.org/download/).
- Linux: Install via package manager (e.g., `sudo apt-get install cmake`).
- macOS: Install via Homebrew (`brew install cmake`).

Step 3: Create a Simple C++ Project
Let's start with a basic "Hello, World!" project.
1. Project Structure:

```
MyProject/
├── src/
│   └── main.
└── CMakeLists.txt
```

2. src/main.:

```
#include <iostream>

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

```
}
```

Step 4: Writing the CMakeLists.txt

Now, let's write a simple `CMakeLists.txt` file for this project.

1. CMakeLists.txt:

```
cmake
cmake_minimum_required(VERSION 3.10)

# Set the project name
project(MyProject)

# Add an executable
add_executable(MyProject src/main.)
```

2. Explanation:
   - `cmake_minimum_required`: Specifies the minimum CMake version.
   - `project`: Names the project.
   - `add_executable`: Defines an executable target from the given source file.

Step 5: Build the Project

1. Navigate to your project directory:

```
cd MyProject
```

2. Create a build directory:

```
mkdir build
cd build
```

3. Generate build files using CMake:

```
cmake ..
```

   This will generate build files in the `build` directory.

4. Compile the project:

```
cmake --build .
```

   This will compile the `main.` and generate the executable.

5. Run the executable:

```
./MyProject
```

   You should see `Hello, World!` printed on the console.

Step 6: Adding More Files

Let's expand the project to include multiple source files.

1. Update the Project Structure:

```
MyProject/
├── src/
│   ├── main.
│   └── greeter.
├── include/
│   └── greeter.h
└── CMakeLists.txt
```

2. src/greeter.:

```
#include "greeter.h"

void greet() {
    std::cout << "Hello from Greeter!" << std::endl;
}
```

3. include/greeter.h:

```
#pragma once

void greet();
```

4. Update src/main.:

```
#include <iostream>
#include "greeter.h"

int main() {
    greet();
    return 0;
}
```

5. Update CMakeLists.txt:

```
cmake
cmake_minimum_required(VERSION 3.10)

# Set the project name
project(MyProject)

# Add include directories
include_directories(include)
```

```
# Add sources
set(SOURCES
    src/main.cpp
    src/greeter.cpp
)

# Add an executable
add_executable(MyProject ${SOURCES})
```

Step 7: Build and Run Again

Repeat the build steps to compile and run the updated project. You should see the output from the `greet()` function.

Step 8: Advanced Topics

1. Target Properties:
   - Learn how to set C++ standards, compiler options, etc.
   - Example:
     cmake
     set_target_properties(MyProject PROPERTIES CXX_STANDARD 17)

2. Linking Libraries:
   - Link external libraries using `target_link_libraries`.
   - Example:
     cmake
     target_link_libraries(MyProject PUBLIC MyLibrary)

3. Using Find Packages:
   - Use `find_package()` to find and link libraries like Boost, OpenCV, etc.
   - Example:
     cmake
     find_package(OpenCV REQUIRED)
     include_directories(${OpenCV_INCLUDE_DIRS})
     target_link_libraries(MyProject ${OpenCV_LIBS})

CMAKE Usage

example :

```
cmake_minimum_required (2.6)
project (HELLO)

set (HELLO_SRCS Hello.c File2.c File3.c)

if (WIN32)
  set(HELLO_SRCS ${HELLO_SRCS} WinSupport.c)
else ()
  set(HELLO_SRCS ${HELLO_SRCS} UnixSupport.c)
endif ()

add_executable (Hello ${HELLO_SRCS})

# look for the Tcl library
find_library (TCL_LIBRARY
  NAMES tcl tcl84 tcl83 tcl82 tcl80
  PATHS /usr/lib /usr/local/lib
  )

if (TCL_LIBRARY)
  target_link_library (Hello ${TCL_LIBRARY})
endif ()
```

CMAKE Structure :
Directories: CMake uses two main directories:
Source Directory: Contains the source code and CMakeLists files.
Binary Directory: Where CMake places the resulting object files, libraries, and executables.
Build Types:
Out-of-Source Build: Source and binary directories are different.
In-Source Build: Source and binary directories are the same.

```
Usage: cmake --build <dir> [options] [-- [native-options]]
Options:
  <dir>          = Project binary directory to be built.
  --target <tgt> = Build <tgt> instead of default targets.
  --config <cfg> = For multi-configuration tools, choose <cfg>.
  --clean-first  = Build target 'clean' first, then build.
                   (To clean only, use --target 'clean'.)
  --             = Pass remaining options to the native tool.
```

Add Custom Command :

```
add_custom_command (
  TARGET target
  PRE_BUILD | PRE_LINK | POST_BUILD
  COMMAND command [ARGS arg1 arg2 arg3 …]
  [COMMAND command [ARGS arg1 arg2 arg3 …] …]
  [COMMENT comment]
  )
```

Build Directory :

Variables: EXECUTABLE_OUTPUT_PATH and LIBRARY_OUTPUT_PATH are used in CMake to control where binary executables and libraries are placed after the build process.
Purpose: These variables help in organizing the output files, especially useful for projects with many subdirectories.
Benefit: Using these variables can save time by centralizing the placement of libraries and executables.

```
# Setup output directories.
set (LIBRARY_OUTPUT_PATH
  ${PROJECT_BINARY_DIR}/bin
  CACHE PATH
  "Single directory for all libraries."
  )

set ( EXECUTABLE_OUTPUT_PATH
  ${PROJECT_BINARY_DIR}/bin
  CACHE PATH
  "Single directory for all executables."
  )

mark_as_advanced (
  LIBRARY_OUTPUT_PATH
  EXECUTABLE_OUTPUT_PATH
  )
```

CMAKE TUTORIAL :

windows : https://cmake.org/download
linux :
steps :
# Update the package list
sudo apt update

# Install build-essential (includes GCC and other development tools)
sudo apt install build-essential -y

# Install CMake
sudo apt install cmake -y

sudo apt-get install tree - to see the tree structure of the folder in linux

After installation is completed , Let's write a simple calculator code and compile and run it on windows and linux .

Code for simple calculator :

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

int main() {
    double num1, num2;
    char op;

    std::cout << "Enter first number: ";
    std::cin >> num1;
    std::cout << "Enter an operator (+, -, *, /): ";
    std::cin >> op;
    std::cout << "Enter second number: ";
    std::cin >> num2;

    switch (op) {
        case '+':
            std::cout << "Result: " << num1 + num2 << std::endl;
            break;
        case '-':
            std::cout << "Result: " << num1 - num2 << std::endl;
            break;
        case '*':
            std::cout << "Result: " << num1 * num2 << std::endl;
```

```
            break;
        case '/':
            if (num2 != 0)
                std::cout << "Result: " << num1 / num2 << std::endl;
            else
                std::cout << "Error: Division by zero!" << std::endl;
            break;
        default:
            std::cout << "Error: Invalid operator!" << std::endl;
            break;
    }

    return 0;
}
```

compile it using g++ :
g++ -o main main.cpp

and run it -> ./main

Now make a CMakeLists.txt file and use CMAKE to compile and build the executable

CMakeLists.txt file :

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.20)

# Set the project name and version
project(SimpleCalculator VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Add the executable
add_executable(SimpleCalculator src/main.cpp)
```

Now run following command to build the project on linux(out of source build) :

Create a build directory
mkdir build
cd build

Run CMake to generate the Makefile

cmake ..

Build the project
make (cmake --build .)

Do the same on Windows machines .
To build using the windows "cl" compiler , open msvc command prompt and run this command :
cl /EHsc src\main.cpp /Fe:SimpleCalculator.exe

Next is Have calculator code in different directory -

folder structure :
SimpleCalculator/
├── CMakeLists.txt
├── src/
│ ├── main.cpp
├── calculator/
│ ├── Calculator.h
│ ├── Calculator.cpp
└── build/ (generated during the build process)

```
// calculator/Calculator.h
#ifndef CALCULATOR_H
#define CALCULATOR_H

class Calculator {
public:
    double add(double a, double b);
    double subtract(double a, double b);
    double multiply(double a, double b);
    double divide(double a, double b);
};

#endif // CALCULATOR_H
```

// calculator/Calculator.cpp

```
#include "calculator.h"
#include <stdexcept>

double Calculator::add(double a, double b) {
    return a + b;
}
```

```
double Calculator::subtract(double a, double b) {
    return a - b;
}

double Calculator::multiply(double a, double b) {
    return a * b;
}

double Calculator::divide(double a, double b) {
    if (b == 0) {
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}
```

// src/main.cpp

```
#include <iostream>
#include "calculator.h"

int main() {
    Calculator calc;
    double num1, num2;
    char op;

    std::cout << "Enter first number: ";
    std::cin >> num1;
    std::cout << "Enter an operator (+, -, *, /): ";
    std::cin >> op;
    std::cout << "Enter second number: ";
    std::cin >> num2;

    try {
        double result;
        switch (op) {
            case '+':
                result = calc.add(num1, num2);
                break;
            case '-':
                result = calc.subtract(num1, num2);
                break;
            case '*':
                result = calc.multiply(num1, num2);
                break;
            case '/':
```

```
                result = calc.divide(num1, num2);
                break;
            default:
                std::cout << "Error: Invalid operator!" << std::endl;
                return 1;
        }
        std::cout << "Result: " << result << std::endl;
    } catch (const std::invalid_argument& e) {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

# Set the project name and version
project(SimpleCalculator VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Include directories
include_directories(${PROJECT_SOURCE_DIR}/calculator)

# Add the executable
add_executable(SimpleCalculator src/main.cpp calculator/calculator.cpp)
```

Now after this - Make the calculator as static library which can be used by the main project to link against this library and use its functionalities.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)

# Set the project name and version
project(SimpleCalculatorWithSLib VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

```
set(SOURCES
src/main.cpp
calculator/src/calculator.cpp
)

set(INCLUDES
    calculator/include/calculator.h
)

# Include directories
include_directories(${PROJECT_SOURCE_DIR})

# Add the library target for Calculator (static library)
add_library(CalculatorLib STATIC calculator/src/calculator.cpp)

# Add the executable
add_executable(SimpleCalculatorWithSLib ${SOURCES} ${INCLUDES})

# Link the Calculator library with the main executable
target_link_libraries(SimpleCalculatorWithSLib CalculatorLib)
```

How to make release build :
Linux/macos -> *cmake -DCMAKE_BUILD_TYPE=Release ..*
windows -> *cmake .. and then cmake --build . --config Release*

Note - for release build with debug info just replace **Release** with **RelWithDebInfo** in above commands

Check if the library is built with debug info or release build :
file <library_name>
nm -C <library_name> | grep debug

clean build -> *cmake --build . --target clean (can also manually delete the "build" folder)*

How to add options from user while building :
You can define custom options or variables in your CMakeLists.txt file and use them to control the build process. For example, you might want to:

Enable or disable certain features.
Choose between different libraries.
Set specific build parameters.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

# Set the project name and version
project(SimpleCalculator VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Add an option to enable or disable advanced operations
option(ENABLE_ADVANCED_OPERATIONS "Enable advanced operations like
multiply and divide" ON)

# Include directories for the calculator library
include_directories(${PROJECT_SOURCE_DIR}/calculator)

# Add the library target for Calculator (static library)
add_library(CalculatorLib STATIC calculator/Calculator.cpp)

# Conditionally add definitions based on the option
if(ENABLE_ADVANCED_OPERATIONS)
    target_compile_definitions(CalculatorLib PRIVATE
ENABLE_ADVANCED_OPERATIONS)
endif()

# Add the executable target for the main application
add_executable(SimpleCalculator src/main.cpp)

# Link the Calculator library with the main executable
target_link_libraries(SimpleCalculator CalculatorLib)
```

in code we should change :

```
#ifdef ENABLE_ADVANCED_OPERATIONS
    double multiply(double a, double b);
    double divide(double a, double b);
#endif
```

```
#ifdef ENABLE_ADVANCED_OPERATIONS
double Calculator::multiply(double a, double b) {
    return a * b;
}

double Calculator::divide(double a, double b) {
    if (b == 0) {
```

```
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}
#endif
```

To enable advanced operations:
Run CMake with custom option
cmake -DENABLE_ADVANCED_OPERATIONS=ON ..

cmake -DENABLE_ADVANCED_OPERATIONS=OFF ..

multiplication and division methods won't be compiled, and any code that tries to use these methods will result in a compilation error (if not handled).

meaning of this line -
target_compile_definitions(<target> [<INTERFACE|PUBLIC|PRIVATE>] <definition> ...)
<target> is CalculatorLib, which is the library target created earlier in the CMakeLists.txt file.
PRIVATE is the visibility specifier. It means that the definition ENABLE_ADVANCED_OPERATIONS will only be used when compiling the CalculatorLib target and will not propagate to other targets that link against it.
<definition> is ENABLE_ADVANCED_OPERATIONS. When this is defined, it is equivalent to having #define ENABLE_ADVANCED_OPERATIONS in the source code.

- Now create a maths folder just like calculator but it should have its own CMakeLists.txt file which should be called from the main CMakeLists.txt file(specify different output directories for this library) :

maths/CMakeLists.txt

```
# Minimum required CMake version
cmake_minimum_required(VERSION 3.10)

# Define the project
project(MathsFunctions)

# Include directory for header files
include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include)

# Specify source files for the static library
set(MATHS_SOURCES src/MathsFunctions.cpp)

# Create the static library target
add_library(MathsFunctionsLib STATIC ${MATHS_SOURCES})
```

```
# Specify the output directories for the static library and headers
set_target_properties(MathsFunctionsLib PROPERTIES
    ARCHIVE_OUTPUT_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}/bin"
    LIBRARY_OUTPUT_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}/bin"
    RUNTIME_OUTPUT_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}/bin"
)

# Install headers in the include directory (not needed for building)
install(FILES include/MathsFunctions.h DESTINATION
${CMAKE_CURRENT_SOURCE_DIR}/include)

# Make include directory available for dependents
target_include_directories(MathsFunctionsLib PUBLIC
${CMAKE_CURRENT_SOURCE_DIR}/include)
```

top-level CMakeLists.txt file :

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

# Set the project name and version
project(SimpleCalculator VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Add subdirectories for calculator and maths libraries
add_subdirectory(calculator)
add_subdirectory(maths)

# Include directories for the calculator library
include_directories(${PROJECT_SOURCE_DIR}/calculator)

# Add the library target for Calculator (static library)
add_library(CalculatorLib STATIC calculator/Calculator.cpp)

# Add the executable target for the main application
add_executable(SimpleCalculator src/main.cpp)

# Link the Calculator library with the main executable
target_link_libraries(SimpleCalculator CalculatorLib)
```

```
# Link the Maths library with the main executable using target name
target_link_libraries(SimpleCalculator MathsFunctionsLib)
```

---

1. **install**()
Purpose: The install() command specifies how files, targets, or directories should be installed to a system or a distribution location when the project is installed.

Think of it as: Instructions for where to put files once your project is built and ready to be deployed or shared.
example :
install(FILES myfile.txt DESTINATION /usr/local/share/myproject)

2. **include_directories**()
Purpose: The include_directories() command specifies directories where the compiler should look for header files when compiling your source code.

Think of it as: A way to tell the compiler where to find additional code files (headers) that your source files need to use.

Example:
include_directories(/path/to/myheaders)

3. **target_include_directories**()
Purpose: The target_include_directories() command specifies which directories should be searched for header files for a specific target (like a library or executable) and how those directories are exposed.

Think of it as: Customizing where each individual target (e.g., a library or executable) looks for header files, and how those include paths are shared with other targets.

Example:
target_include_directories(MyLibrary PUBLIC /path/to/myheaders)

Now create the shared library of physics project :

```
// PhysicsFunctions.h
#ifndef PHYSICS_FUNCTIONS_H
#define PHYSICS_FUNCTIONS_H

namespace Physics {

// Handle export/import for cross-platform shared libraries
```

```
#if defined(_WIN32) || defined(_WIN64)
    #ifdef PHYSICS_FUNCTIONS_EXPORTS
        #define PHYSICS_API __declspec(dllexport)
    #else
        #define PHYSICS_API __declspec(dllimport)
    #endif
#elif defined(__linux__) || defined(__APPLE__)
    #ifdef PHYSICS_FUNCTIONS_EXPORTS
        #define PHYSICS_API __attribute__((visibility("default")))
    #else
        #define PHYSICS_API
    #endif
#else
    #define PHYSICS_API
#endif

// Function to calculate force given mass and acceleration (F = m * a)
PHYSICS_API double calculateForce(double mass, double acceleration);

// Function to calculate kinetic energy given mass and velocity (KE =
0.5 * m * v^2)
PHYSICS_API double calculateKineticEnergy(double mass, double velocity);

// Function to calculate potential energy given mass, gravitational
acceleration, and height (PE = m * g * h)
PHYSICS_API double calculatePotentialEnergy(double mass, double gravity,
double height);

} // namespace Physics

#endif // PHYSICS_FUNCTIONS_H
```

// PhysicsFunctions.cpp

```
#include "PhysicsFunctions.h"

namespace Physics {

double calculateForce(double mass, double acceleration) {
    return mass * acceleration;  // F = m * a
}

double calculateKineticEnergy(double mass, double velocity) {
    return 0.5 * mass * velocity * velocity;  // KE = 0.5 * m * v^2
}
```

```
double calculatePotentialEnergy(double mass, double gravity, double
height) {
    return mass * gravity * height;  // PE = m * g * h
}

} // namespace Physics
```

PhysicsProject/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(PhysicsProject)

# Set the output directory for all binaries (executables and libraries)
to be the same as the build directory
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}")
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}")

# Add shared library for PhysicsFunctions
add_library(PhysicsFunctionsLib SHARED src/PhysicsFunctions.cpp)

# Define the export symbol for platforms (needed for proper symbol
export)
target_compile_definitions(PhysicsFunctionsLib PRIVATE
PHYSICS_FUNCTIONS_EXPORTS)

# Include directories for headers
target_include_directories(PhysicsFunctionsLib PUBLIC
${CMAKE_CURRENT_SOURCE_DIR}/include)

# Install the library and header files to the specified directory
install(TARGETS PhysicsFunctionsLib
    LIBRARY DESTINATION "${CMAKE_BINARY_DIR}"
    RUNTIME DESTINATION "${CMAKE_BINARY_DIR}"
    ARCHIVE DESTINATION "${CMAKE_BINARY_DIR}"
)
install(FILES include/PhysicsFunctions.h DESTINATION
"${CMAKE_CURRENT_SOURCE_DIR}/include")
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)

# Set the project name and version
project(SimpleCalculatorWithSLib VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

set(SOURCES
src/main.cpp
calculator/src/calculator.cpp
)

set(INCLUDES
    calculator/include/calculator.h
)

# Include directories
include_directories(${PROJECT_SOURCE_DIR})


add_subdirectory(maths)
add_subdirectory(physics)

# Add the library target for Calculator (static library)
add_library(CalculatorLib STATIC calculator/src/calculator.cpp)

# Add the executable
add_executable(SimpleCalculatorWithSLib ${SOURCES} ${INCLUDES})

message("Executable output path: ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}")
message("Shared library output path: ${CMAKE_LIBRARY_OUTPUT_DIRECTORY}")

# Link the Calculator library with the main executable
#target_link_libraries(SimpleCalculatorWithSLib CalculatorLib)

# Link the Maths library with the main executable using target name
target_link_libraries(SimpleCalculatorWithSLib CalculatorLib
MathsFunctionsLib PhysicsFunctionsLib)
```

- By above steps , it will create the shared library and executable in the same folder and it will be able to read the exported symbols and use it .

Why Use target_compile_definitions in above project?
By defining PHYSICS_FUNCTIONS_EXPORTS using target_compile_definitions, we ensure that this macro is only defined when building the PhysicsFunctionsLib library itself. This is necessary to set PHYSICS_API to __declspec(dllexport) (on Windows) or __attribute__((visibility("default"))) (on Linux/macOS) to properly export symbols.
When another project links to PhysicsFunctionsLib, PHYSICS_FUNCTIONS_EXPORTS is not defined, and thus, PHYSICS_API is set to __declspec(dllimport) (on Windows) or left empty (on Linux/macOS), properly importing the symbols.

Additional Examples of target_compile_definitions :

1. **Platform-Specific Definitions**
You may want to define a macro for different platforms to include platform-specific code:

For a library called MyLibrary
```
add_library(MyLibrary src/MyLibrary.cpp)

# Define a platform-specific macro
if (WIN32)
    target_compile_definitions(MyLibrary PRIVATE PLATFORM_WINDOWS)
elseif (APPLE)
    target_compile_definitions(MyLibrary PRIVATE PLATFORM_MACOS)
elseif (UNIX)
    target_compile_definitions(MyLibrary PRIVATE PLATFORM_LINUX)
endif()
```

Usage in Code:
```
#ifdef PLATFORM_WINDOWS
// Windows-specific code
#elif defined(PLATFORM_MACOS)
// macOS-specific code
#elif defined(PLATFORM_LINUX)
// Linux-specific code
#endif
```

2. **Enabling Debug or Release-Specific Code**
You can define different macros depending on the build type (Debug or Release):
For a target called MyApp
```
add_executable(MyApp src/main.cpp)

# Define a macro for the Debug build type
if (CMAKE_BUILD_TYPE STREQUAL "Debug")
```

```
    target_compile_definitions(MyApp PRIVATE DEBUG_MODE)
endif()
```

Usage in Code:

```
#ifdef DEBUG_MODE
// Debug-only code
#endif
```

3. **Conditional Feature Compilation**
You may have optional features that are controlled via compile-time definitions:

For a library called FeatureLib

```
add_library(FeatureLib src/Feature.cpp)

# Conditionally define a macro for an optional feature
option(ENABLE_FEATURE_X "Enable Feature X" ON)
if (ENABLE_FEATURE_X)
    target_compile_definitions(FeatureLib PUBLIC FEATURE_X_ENABLED)
endif()
```

Usage in Code:

```
#ifdef FEATURE_X_ENABLED
void featureX() {
    // Feature X code
}
#endif
```

Summary :
target_compile_definitions() is a versatile CMake command that allows you to control compile-time definitions for your targets. It is crucial for handling cross-platform differences, managing build types, and optionally including or excluding features.

How to use a third party library ?

There are two ways to find and link third party libraries using cmake :
find_package() and find_library()

find_package()
find_package() is a higher-level command used in CMake to find and load the configuration of an external package or library. It can locate multiple components of a package (libraries, headers, etc.) and set up all the necessary variables and targets for you.

Key Points about find_package():
Purpose: Finds a complete package that may consist of libraries, headers, and sometimes additional components like configuration files or executables.

Finds Multiple Components: find_package() is capable of finding multiple components related to a package. For example, find_package(Qt5 COMPONENTS Core Widgets REQUIRED) can find both the Qt5Core and Qt5Widgets components.

Finds Config Files: It often finds *-config.cmake files provided by the package itself, which contain all necessary information about the package's components.

Sets Multiple Variables: Automatically sets include directories, libraries, and other necessary variables needed for compiling and linking against the found package.

Target-based Modern CMake: When using modern CMake (3.0+), find_package() often provides imported targets (e.g., OpenSSL::SSL) that you can use directly with target_link_libraries() without worrying about manual configuration.

Flexibility and Configurability: With find_package(), you can specify components, versions, and optional or required packages.

Example Usage:

```
# Find OpenSSL package
find_package(OpenSSL REQUIRED)

# Include OpenSSL headers
include_directories(${OPENSSL_INCLUDE_DIR})

# Add executable target
add_executable(MyApp main.cpp)

# Link OpenSSL libraries
target_link_libraries(MyApp ${OPENSSL_LIBRARIES})
```

find_library() is a lower-level command used to find only a single library file. It does not handle finding associated headers or other components, nor does it provide imported targets or handle dependencies.

Key Points about find_library():
Purpose: Finds a single library file (e.g., .so, .a, .lib, .dll), usually used when you know the exact name of the library you want to find.

Only Finds Libraries: find_library() searches for a specific library file and sets a variable to its path.

Does Not Find Headers: Unlike find_package(), it does not find associated headers or other components.

Sets Only One Variable: find_library() only sets the variable you specify to the path of the found library.

No Imported Targets: It does not provide imported targets; you have to manually manage paths and linking.

Used for Custom or Uncommon Libraries: Often used for libraries that do not provide a *-config.cmake file or for custom-built libraries where find_package() is not applicable.

Example Usage:

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

# Specify the location where to search for libraries
set(LIBRARY_SEARCH_PATH "${CMAKE_CURRENT_SOURCE_DIR}/lib/mylib")

# Use find_library to locate the custom library
find_library(MYLIB_LIBRARY NAMES mylib PATHS ${LIBRARY_SEARCH_PATH}
REQUIRED)

# Display the found path (optional, useful for debugging)
message(STATUS "Found mylib library at: ${MYLIB_LIBRARY}")

# Add the executable target
add_executable(MyApp src/main.cpp)

# Link the found library to the executable
target_link_libraries(MyApp PRIVATE ${MYLIB_LIBRARY})
```

Example 2: Finding a System Library (zlib)

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

# Find the zlib library
find_library(ZLIB_LIBRARY NAMES z PATHS /usr/lib /usr/local/lib
REQUIRED)

# Display the found path (optional)
message(STATUS "Found zlib library at: ${ZLIB_LIBRARY}")

# Add the executable target
add_executable(MyApp src/main.cpp)

# Link the found library to the executable
target_link_libraries(MyApp PRIVATE ${ZLIB_LIBRARY})
```

Example 3: Using Environment Variables for Custom Library Paths

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

# Use an environment variable or default paths for library search
find_library(MY_CUSTOM_LIBRARY NAMES mycustomlib
    PATHS $ENV{MY_CUSTOM_LIB_PATH} /usr/local/lib /usr/lib
    REQUIRED
)

# Display the found path
message(STATUS "Found custom library at: ${MY_CUSTOM_LIBRARY}")

# Add the executable target
add_executable(MyApp src/main.cpp)

# Link the library to the executable
target_link_libraries(MyApp PRIVATE ${MY_CUSTOM_LIBRARY})
```

More use case :
Suppose you have to use openssl (a third party library) in your main project :
Steps required :

1. Install openssl -
 on linux - sudo apt-get update
sudo apt-get install libssl-dev

on macos -

brew install openssl

on windows : download prebuilt binaries from internet -
https://slproweb.com/products/Win32OpenSSL.html

change CMakeLists.txt file of the main project :
MainProject/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(MainProject)

# Set the output directory for all binaries (executables and libraries)
to be the same as the build directory
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}")
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}")

# Find OpenSSL package
find_package(OpenSSL REQUIRED)

# Include the OpenSSL headers
include_directories(${OPENSSL_INCLUDE_DIR})

# Add the executable target for the main project
add_executable(MainProject src/MainFunctions.cpp)

# Link OpenSSL libraries
target_link_libraries(MainProject ${OPENSSL_LIBRARIES})

# Also link the PhysicsFunctionsLib from PhysicsProject
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/PhysicsProject)
target_link_libraries(MainProject PhysicsFunctionsLib)

# Include directories for headers
target_include_directories(MainProject PRIVATE
${CMAKE_CURRENT_SOURCE_DIR}/include)
```

code to use openssl functions :
// MainProject/src/MainFunctions.cpp

```
#include "MainFunctions.h"
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <iostream>
```

```
void mainFunction() {
    // Initialize OpenSSL library
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();

    std::cout << "OpenSSL initialized successfully!" << std::endl;

    // Example: Create an SSL context
    const SSL_METHOD *method = SSLv23_client_method();
    SSL_CTX *ctx = SSL_CTX_new(method);
    if (!ctx) {
        std::cerr << "Failed to create SSL context." << std::endl;
        ERR_print_errors_fp(stderr);
        return;
    }

    // Cleanup OpenSSL
    SSL_CTX_free(ctx);
    EVP_cleanup();
    std::cout << "OpenSSL cleanup done!" << std::endl;
}
```

Similarly you can use find_package() to integrate popular libraries like **Boost, Qt, OpenCV, SQLite, Threads, OpenGL, and Eigen** in real-world CMake projects.

add_custom_command() :
The add_custom_command() function in CMake is a powerful tool that allows you to execute custom commands at specific points during the build process. This can be useful for various tasks such as generating files, copying files, running scripts, or performing any other custom operations needed by your build system.

```
add_custom_command(
  TARGET target_name
  [PRE_BUILD | POST_BUILD | PRE_LINK]
  COMMAND command1 [ARGS ...]
  [COMMAND command2 [ARGS ...]]
  [WORKING_DIRECTORY dir]
  [DEPENDS depend1 [depend2 ...]]
  [BYPRODUCTS byproduct1 [byproduct2 ...]]
  [COMMENT comment]
  [VERBATIM]
```

)

Parameters
TARGET target_name: Specifies the target for which this custom command is associated. This can be an executable or a library.

PRE_BUILD | POST_BUILD | PRE_LINK: Defines when the custom command should be executed relative to the build target:

PRE_BUILD: Before the target is built.
POST_BUILD: After the target is built.
PRE_LINK: Before the target is linked.
COMMAND command1 [ARGS ...]: Specifies the command to be executed. You can provide multiple commands.

WORKING_DIRECTORY dir: Sets the working directory for the command.

DEPENDS depend1 [depend2 ...]: Lists files or targets that this command depends on. The command will be executed only after these dependencies are built or available.

BYPRODUCTS byproduct1 [byproduct2 ...]: Lists files that are generated by this command. CMake uses this information to track the command's outputs.

COMMENT comment: Provides a message that is displayed when the command is executed.

VERBATIM: Ensures that arguments are passed exactly as specified, without additional escaping or modification.

Example 1: Copying a File After Build
copy a configuration file to the build directory after building your application:

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

# Define the path to the configuration file
set(CONFIG_FILE_PATH "${CMAKE_SOURCE_DIR}/config/my_config.cfg")

# Copy the configuration file to the build directory after building the
executable
add_custom_command(TARGET MyApp POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different
    "${CONFIG_FILE_PATH}"
    $<TARGET_FILE_DIR:MyApp>
    COMMENT "Copying configuration file to the build directory"
)
```

Example 2: Generating a File Before Build
Suppose you have a script that generates a header file needed by your project:

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

# Define the path to the generated header file
set(GENERATED_HEADER "${CMAKE_BINARY_DIR}/generated_header.h")

# Run a script to generate the header file before building the
executable
add_custom_command(OUTPUT ${GENERATED_HEADER}
    COMMAND ${CMAKE_COMMAND} -P
${CMAKE_SOURCE_DIR}/scripts/generate_header.cmake
    DEPENDS ${CMAKE_SOURCE_DIR}/scripts/generate_header.cmake
    COMMENT "Generating header file"
)

# Specify that the generated header file is a dependency of the target
add_custom_target(generate_header ALL DEPENDS ${GENERATED_HEADER})

# Ensure MyApp depends on the generated header
add_dependencies(MyApp generate_header)
```

Example 3: Running a Post-Build Script

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

# Define the script to run after building the executable
set(POST_BUILD_SCRIPT "${CMAKE_SOURCE_DIR}/scripts/post_build.sh")

# Run the script after the executable is built
add_custom_command(TARGET MyApp POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E env bash ${POST_BUILD_SCRIPT}
    COMMENT "Running post-build script"
)
```

Example 4: Cleaning Up Temporary Files post build

```
# Add the main executable
add_executable(MyApp src/main.cpp)
```

```
# Define the path to the temporary directory to be cleaned
set(TEMP_DIR "${CMAKE_BINARY_DIR}/temp")

# Create a custom command to clean up the temporary directory
add_custom_command(TARGET MyApp POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E remove_directory ${TEMP_DIR}
    COMMENT "Cleaning up temporary files"
)
```

Dependency management in CMAKE ：

The commonly used commands for managing dependencies in CMake are:

add_dependencies()
find_package()
FetchContent
ExternalProject

1. **add_dependencies()**
The add_dependencies() command is used to specify that one target depends on another target. This ensures that the dependent target is built before the target that depends on it. It's useful when you have custom commands or custom build steps that need to be executed in a specific order.

add_dependencies(target_name depend_target1 [depend_target2 ...])
example ：

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

# Define a library target
add_library(my_library STATIC my_library.cpp)

# Define an executable target
add_executable(my_executable main.cpp)

# Make sure the executable depends on the library being built first
add_dependencies(my_executable my_library)

# Link the library to the executable
target_link_libraries(my_executable PRIVATE my_library)
```

2. **find_package()**
This command is used to find and configure external libraries that are installed on your system. It sets up the necessary variables and includes directories for the libraries.Already discussed above.

3. **FetchContent**
The FetchContent module allows you to download and add dependencies from remote repositories during the configuration phase of the build.
example :

```
include(FetchContent)

FetchContent_Declare(
    spdlog
    GIT_REPOSITORY https://github.com/gabime/spdlog.git
    GIT_TAG v1.9.2
)

FetchContent_MakeAvailable(spdlog)

add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE spdlog::spdlog)
```

4. **ExternalProject**
The ExternalProject module handles downloading, building, and installing projects from source, which is useful for more complex dependencies.
example:

```
include(ExternalProject)

ExternalProject_Add(
    my_dependency
    GIT_REPOSITORY https://github.com/some/repo.git
    GIT_TAG some_tag
    CMAKE_ARGS -DCMAKE_INSTALL_PREFIX=<INSTALL_PREFIX>
)

# Use `ExternalProject` to build and link the dependency.
```

packaging :

other commonly used commands :
To use g++ as the compiler
cmake -DCMAKE_CXX_COMPILER=g++ .

build differnt build systems :

cmake -G Ninja ..

cmake -G "Visual Studio 16 2019" ..
cmake -G "Xcode" ..

build a particular target :
cmake --build . --target <target_name>

Specify a build configuration:
cmake --build . --target app1 --config Debug
cmake --build . --target <target_name> --config RelWithDebInfo

Specify the number of parallel jobs (for faster builds):
cmake --build . --target app1 -- -j4

PROJECT

LOGGING :

(Spdlog library)

Features of the Logging Library

The logging library should have the following features:

1. Log Levels:
   - Support various log levels such as `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`, and `FATAL`.
   - Allow users to set the log level threshold (e.g., display messages of level `INFO` and above).

2. Log Outputs:
   - Log messages to multiple outputs like console, files, and remote servers.
   - Support rotating log files by size and by date.
   - Provide options for log file naming patterns (e.g., `log_YYYYMMDD.txt`).

3. Formatting and Customization:
   - Customizable log formats (e.g., `"[LEVEL] [TIME] [FILE:LINE] MESSAGE"`).
   - Support for custom log message prefixes and suffixes.
   - Allow users to define their own log format using format strings.

4. Thread Safety:
   - Ensure that the logging library is thread-safe and can be used in multi-threaded environments.

5. Performance Optimization:
   - Use asynchronous logging to prevent blocking operations in performance-critical applications.
   - Offer options to control the buffer size for asynchronous logging.

6. Filtering:
   - Filter logs based on different criteria such as log level, source file, function, or module.

7. Configuration:
   - Provide runtime configuration using a configuration file (e.g., JSON, XML).
   - Allow configuration through environment variables or programmatic API.

8. Extendibility:
   - Allow users to add custom log sinks (e.g., send logs to a database or network service).
   - Provide a plugin interface for extending log processing (e.g., custom log filters, formatters).

9. Log Rotation and Compression:
   - Support log rotation based on size or date.
   - Automatically compress rotated log files to save space.

10. Platform Support:
   - The library should work seamlessly across Windows, Linux, and macOS.
   - Abstract platform-specific details like file handling and system calls.

11. Backwards Compatibility and Modern Features:
   - Compatibility with C++11 and later standards.
   - Make use of modern C++ features where applicable.

Sample test code - How Logger should be used ：

```
#include "logger.h"
#include "console_sink.h"
#include "file_sink.h"
#include <memory>
```

```
int main() {
    // Get the singleton instance of the Logger
    Logger& logger = Logger::instance();

    // Create and add console sink to the logger
    std::shared_ptr<LogSink> console_sink =
std::make_shared<ConsoleSink>();
    logger.add_sink(console_sink);

    // Create and add file sink to the logger
    std::shared_ptr<LogSink> file_sink =
std::make_shared<FileSink>("logs/app_log.txt");
    logger.add_sink(file_sink);

    // Set the logging level to INFO
    logger.set_log_level(LogLevel::INFO);

    // Log messages of different levels
    logger.log(LogLevel::DEBUG, "This is a DEBUG message.");
    logger.log(LogLevel::INFO, "This is an INFO message.");
    logger.log(LogLevel::WARNING, "This is a WARNING message.");
    logger.log(LogLevel::ERROR, "This is an ERROR message.");
    logger.log(LogLevel::CRITICAL, "This is a CRITICAL message.");
    logger.log(LogLevel::FATAL, "This is a FATAL message.");

    // Test if the log level threshold is working
    logger.set_log_level(LogLevel::ERROR);
    logger.log(LogLevel::INFO, "This INFO message should not be
logged.");   // Won't be logged
    logger.log(LogLevel::ERROR, "This ERROR message should be logged.");
// Will be logged

    // Test dynamic change of logging levels
    logger.set_log_level(LogLevel::DEBUG);
    logger.log(LogLevel::DEBUG, "Now, DEBUG messages will be logged
again.");

    return 0;
}
```

Logging config can be mentioned in log_config file and then logging framework can read this file and set the configuration .

Example code using config file ：

```cpp
#include "logger.h"

int main() {
    // Load logger configuration from file
    Logger::instance().load_config("logger_config.txt");

    // Log some messages
    Logger::instance().log(LogLevel::DEBUG, "This is a DEBUG message.");
    Logger::instance().log(LogLevel::INFO, "This is an INFO message.");
    Logger::instance().log(LogLevel::WARNING, "This is a WARNING
message.");
    Logger::instance().log(LogLevel::ERROR, "This is an ERROR
message.");
    Logger::instance().log(LogLevel::CRITICAL, "This is a CRITICAL
message.");
    Logger::instance().log(LogLevel::FATAL, "This is a FATAL message.");

    return 0;
}
```

logger_config.txt

```
log_level=INFO
sinks=console,file
file_path=logs/app_log.txt
```