CMake
-build system-

Content

- ❑ Introduction
- ❑ CMakeLists For two simple C++ Programs.
- ❑ Targets
- ❑ Targets dependencies
- ❑ Commands (include, sub_directory, message)
- ❑ CMake basics
  - Variables
  - Decisions
  - Loops
  - Functions
  - Macros
- ❑ GUI tools
- ❑ Custom targets
- ❑ Dependency management: FetchContent

Content

- ❑ Testing (GoogleTest, Catch2)
- ❑ Generating documentation(Doxygen)
- ❑ Dependency Management: CPM, VCPKG, Conan
- ❑ Presets
- ❑ CMake project template example
- ❑ Project generator: cmake_init

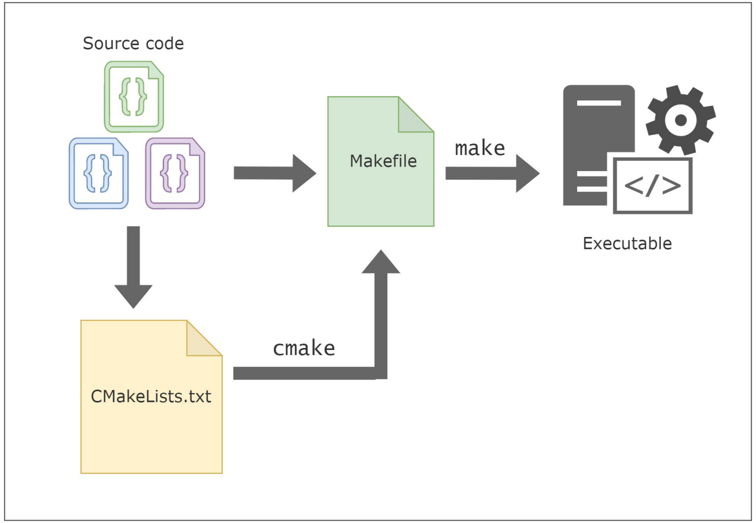
1) Introduction

What is CMake?

If you have just started learning C++, you are probably using a popular C++ IDE to edit and build/compile your programs. May be MS Visual studio, Code blocks, which are not only IDEs available to build your C++ programs. The only thing that the most popular IDEs have in common is CMake support.

In windows, running cpp files may be done only by pressing the run button in the IDE which is installed on your system. MS Visual studio for example creates a solution file (.sln) which tells the OS how to compile the program, but what if we want to run this cpp file on Linux?

Linux and other operating systems can not interpret this solution file. This is where CMake comes in letting us define the representation of our project, and it creates its own files that can be run by any environment with CMake support.



Build process

After creating the CMake project, it will go through a building process which is mainly broken down into two steps, configure and build.

1. Configure (triggered by the command: `cmake -S . -B build`):

   In this step, cmake searches for any usable toolchain available and decides which configuration it outputs. The toolchain typically includes the compiler and related tools that are used to build the project. In visual studio compiler (MSVC) `*.sln` file will be created.

   During this process, CMakeLists.txt file is parsed and executed to configure the build files relative to toolchains, architecture, and dependencies.

   CMake writes the build files based on the generators which can be changed using the command: `cmake . -G generator`.

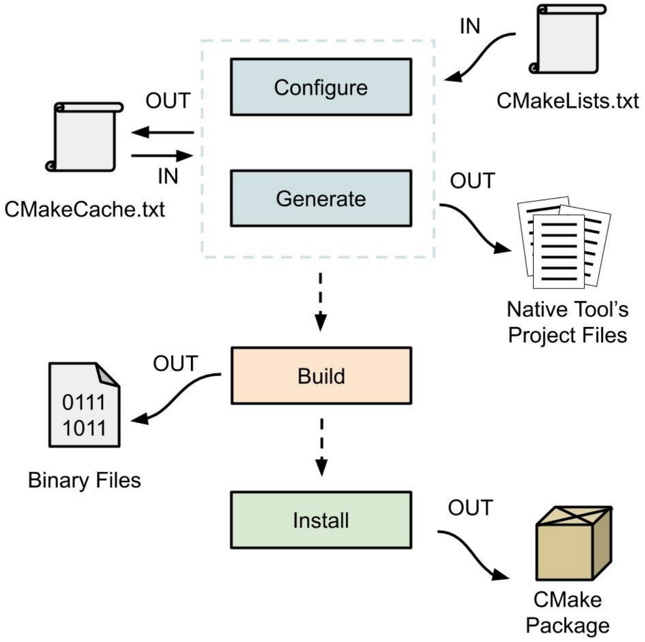
2. Build (triggered by the command: `cmake –build build`):

   CMake will execute the build files to compile and link libraries, and run tests.

   - The build directory will contain the generated binaries, build instructions, and cache.
   - Inside the build directory, CMakeCache.txt file will contain all the detected configurations.

Flow:

1. The input file is `CMakeLists.txt`, it contains commands and instructions that CMake uses to configure and generate the project build system.
2. Then, CMake reads the input file, processes it, and then creates or updates `CMakeCache.txt`, which stores persistent configuration options and variables used during the configuration.
3. After configuration, CMake generates the build system files required by the native build tool (`*.sln` for visual studio code). The output will be native tool's project file, which used by the native build tool to compile and link the project.
4. Then, the build tool uses the generated project files to compile the source code into binary files (executable files, libraries, etc.).
5. Finally, the compiled binaries and other files like headers, and documentation are installed to the designated install locations.



Most common variables and directories to know

`CMAKE_SOURCE_DIR`: Represents the top-level directory of your source tree, where `CMakeFiles.txt` file is located.

`CMAKE_BINARY_DIR`: Represents the top-level directory where the build is being performed. This directory is specified using the `-B` option in the cmake command.

`CMAKE_CURRENT_SOURCE_DIR`: Represents the directory currently being processed by CMake. This can be different from `CMAKE_SOURCE_DIR` while processing.

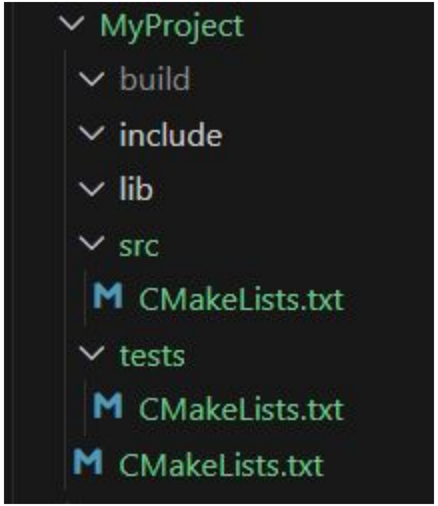
`CMAKE_CURRENT_BINARY_DIR`: Represents the binary directory currently being processed by CMake. This also can be different from `CMAKE_BINARY_DIR` while processing.

`PROJECT_SOURCE_DIR`: Equivalent to `CMAKE_SOURCE_DIR`, but scoped to the current project if you have multiple projects in the same directory.

`PROJECT_BINARY_DIR`: Equivalent to `CMAKE_BINARY_DIR`, but scoped to the current project if you have multiple projects in the same directory.

Common Directory structure

```
MyProject/
|-- CMakeLists.txt        # Top-level CMake script
|-- src/                  # Source files
    |-- CMakeLists.txt    # CMake script for source files
|-- include/              # Header files
|-- build/                # Build output (generated by CMake)
|-- lib/                  # Libraries (if any)
|-- tests/                # Test files
    |-- CMakeLists.txt    # CMake script for tests
```



CMake as a script language:

CMake is dynamically-typed language like Python. CMake script is composed of commands. Each command ends with parenthese with some arguments and keywords.

- Commands are case insensitive.
- Variables are case sensitive.
- Keywords are always written in uppercase.

Ex:

```
comman1(KEYWORD1 arg1)

COMMAND1(KEYWORD1 arg1)
```

These two lines are the same

Generators

Generator is a component that determines the type of the build system that CMake will produce. Each generator creates its own project files, such as makefiles, visual studio project files, or Ninja build files. The generator can be specified using `-G` option of cmake command.

Ex. `cmake -G "Unix Makefiles" -S . -B build`

Common CMake generators:

1. Unix MakeFiles ⇒ Used for make build tool on Unix-like systems.
2. Ninja ⇒ Used for Ninja build system, which is known for its fast build speeds.
3. Visual Studio (MSBuild) ⇒ Used for generating project files for Microsoft Visual Studio. The version must be specified (ex. `Cmake -G "Visual Studio 16 2019"`).
4. Xcode ⇒ Used for generating project files for Apple's Xcode IDE.

2) CMakeLists For two simple C++ Programs

CMakeLists for the simplest C++ program

main.cpp U ×

Slides > Simplest_program > main.cpp

```cpp
int main() {

}
```

CMakeLists.txt U ×

Slides > Simplest_program > CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.10)
project(simpleProgram)

add_executable(simple_program main.cpp)
```

Here, we have created a simple_program which holds the source file main.cpp, then simple_program is the root of th project as it contains all files. Given this directory setup, we will create a CMakeLists.txt file inside simple_program with the left part content.

example

```cmake
project("HelloWorld"
        VERSION 1.0
        DESCRIPTION "Hello world in CMAKE!"
        LANGUAGES CXX
)
```

CMakeLists.txt commands:

- `cmake_minimum_required(VERSION 3.10)`: It tells CMake the minimum version the user of CMake needs to have to build the project.
- `project(simpleProgram)`: It tells the CMake the name of the project which can be referenced with later. We can use this command to define versions, languages, and other data for the project as well.
- `add_executable(simple_program main.cpp)`: defines an executable target which holds the source file main.cpp, it can include multiple source files as well. Executable targets will be discussed later, but till now, we can say that it instructs CMake to compile the listed source files and link them into the executable binary(generated from the build process). It also can be referred in another CMake commands. (for example, linking libraries). In conclusion, we use it to setup a binary file whose name here is simple_program.

CMakeLists for another C++ program

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hello world\n";
    return 0;
}
```

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.10)
project("HelloWorld"
        VERSION 1.0
        DESCRIPTION "Hello world in CMAKE!"
        LANGUAGES CXX
)

add_executable(${PROJECT_NAME} main.cpp)
```

Here is another program. It is almost like the first one. From there, we can notice that `CMakeLists.txt` script isn't affected much by the source file implementation; as the main role for CMake is organizing and managing the build system. While it does reference source files to specify what needs to be built. It is responsible for defining the structure and configuration of the build process.

Multi file C++ project with CMake structure:

Basic structure:

dog.h
operations.h
dog.cpp
operations.cpp
main.cpp

More practical structure:

/include
dog.h
operations.h

/src
dog.cpp
operations.cpp

main.cpp
void main()

For the practical structure, we need to tell CMake about this structure, so that it can really understand our program.

- Firstly, we need to tell CMake where it can find header files/ the directory of header files (`*.h`):

  We will use command: `target_include_directories`, by adding a CMake variable `${CMAKE_CURRENT_SOURCE_DIR}/include`; where include is the file contains all header files.

  Ex: `target_include_directories(HelloBinary PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)`

  Note: PRIVATE keyword will be discussed later, but for now, we can consider that it specifies that the include directories are only needed for this target `HelloBinary` and are not propagated to other targets that depend on it for this target not for the whole source code.

- Secondly, We need to tell CMake the directory of the source file, we can add it as a parameter to `add_executable` command:

  Ex: `add_executable(HelloBinary main.cpp src/dog.cpp src/operations.cpp)`

In this way, we add the source files to the command file by file. For this reason, there is another way called BlueBend. It is a technique to tell CMake to go and scan a directory and find all the files that meet some criteria. We will store the found files in a variable `SRC_FILES`, to be added to the define the executable target against it (To set up the binary file whose name is HelloBinary).

Ex: `file(GLOB_RECURSE SRC_FILES src/*.cpp)` → CMake will globally scan the src directory and scan `.cpp` files

`add_executable(HelloBinary main.cpp ${SRC_FILES})`--> Grab the source files, `${SRC_FILES}` is used to get its content which is `files.cpp` here that are stored at the variable `SRC_FILES`.

Note: According to the CMake official documentation, it is not recommended to use globing feature; as it can lead to issues with detecting changes to the source files. CMake may not automatically regenerate build scripts when files are added or removed. It is often better to explicitly list the source files or use a script to manage the file list.

(To manage the list of source files in a CMake project using a script, we can create a separate script (e.g., Python) that generates the list of source files and writes them to a file. Then, we can include this file in your CMakeLists.txt)

CMakeLists.txt file for this structure:

CMakelists.txt M ×

2-multiFileProject > Source > CMakelists.txt

```
cmake_minimum_required(VERSION 3.5)

project(HelloApp
        VERSION 0.0.1
        DESCRIPTION "The leading Hello world App"
        LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 23)
file(GLOB_RECURSE SOURCE_FILES src/*.cpp)
add_executable(HelloAppBinary main.cpp ${SOURCE_FILES})
target_include_directories(HelloAppBinary PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

This specifies that the entire project will use the C++ 23 standard.

3) Targets

Target is something you want to get out of your project. It is generated by the build process. Targets can be executable targets, libraries, or custom targets.

1) **Executable targets:** The executable binary file used to run the code file. It is defined using `add_executable` command.

```
cmake_minimum_required(VERSION 3.10)
project("HelloWorld"
        VERSION 1.0
        DESCRIPTION "Hello world in CMAKE!"
        LANGUAGES CXX
)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

file(GLOB_RECURSE SOURCE_FILES src/*.cpp)
add_executable(${PROJECT_NAME} main.cpp ${SOURCE_FILES})

target_include_directories(HelloAppBinary PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

2) Library targets:

Libraries can be one of two types, static and shared-dynamic library, but what is the difference between them?

- **Static library** (integrated in the compile time)=> During the build process, the code of it is copied into the executable file. Which in turn will increase the size of the executable file. Additionally, it is necessary to re-compile the executable in case of updating the library, which can be time-consuming for large applications. HowEver, it performs portability which means that the executable can be run on any compatible system without needing additional library files.
- **Dynamic library** (integrated on run-time)=> The code of it is not copied into the executable. Instead, the executable has references to the shared/ dynamic library as a separated file. This in turn reduces the size of the executable files in the libraries. Additionally shared library allows easier updating and maintenance as it can be updated independently of the executable files. On the other hand, there is a small cost for loading the library at runtime.

Each library type has its own pros and cons. So, it is necessary to choose the type carefully based on our application. For example, the static library is suitable when the application portability and simplicity are more important than the disk space. But, shared/dynamic library is preferable in case of large applications where the ability to update parts of the application independently are crucial. It will performs well in environments where multiple applications use the same libraries; as it reduces memory usage and disk spaces.

Shared libraries are defined using `add_library` command, mentioning the directory of the library file. If we have multiple libraries, it is necessary to add the header files of each library separately to the target include directories; to ensure that each target knows where to find necessary files during compilation.

Ex:

```
# Add static library target
add_library(operations STATIC src/operations.cpp)
target_include_directories(operations PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include())
#target_compile_features(operations PUBLIC cxx_std_20)
```

Full code example

4) Target dependencies

CMake Target dependencies:

Cmake uses somewhat similar inheritance concepts to C++, especially in public and private access specifiers and inheritance types. We can think that CMake keywords public, private and interface used in target_link_directories and target_link_libraties are mixtures of access specifiers and inheritance type in C++.

Access specifiers: In C++ OOP, there are three types of access specifiers for classes:

Access specifiers	Description
PUBLIC	Members are accessible from anywhere outside the class.
PROTECTED	Members cannot be accessed from outside the class. However, they are accessible in derived classes.
PRIVATE	Members cannot be accessed or viewed outside the class.

Inheritance types : In OOP, there are also three types for class inheritance: public, protected, and private

Inheritance type	Description
PUBLIC	Public members of the base class becomes public members of derived class, and the protected members of the base class becomes protected members of the derived class. The private member of base class aren’t accessible directly from the derived class, but can be accessed through calls to the protected and public members from the base class..
PROTECTED	Both public and protected members of the base class becomes protected members for the derived class.
PRIVATE	Public and protected members of the base class become private members for the derived class.

CMake scope:

Before we move on to the cmake dependencies. Firstly, let’s know the scope in CMake:

Scope refers to the context in which a command is executed. For example, if we have a function operates on multiple targets, those targets are considered to be in the same scope.

The key points are:

1. **Function scope** : when we define a function in CMake using the function() command, the targets created within that function are considered to be in the same scope.
2. **Directory scope** : If we have multiple targets defined at the same directory (i.e. at the same `CMakeLists.txt`), those targets are also considered to be at the same scope.
3. **Nested scope** : When we have a function that calls another function or we have subdirectories with their own CMakeLists.txt files (i.e. using `CMakeLists.txt` file inside another `CMakeLists.txt` file).

Secondly, let's know the difference between two keywords which are `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES`:

- `INCLUDE_DIRECTORIES:` It specifies the include directories for the current targets or the targets within the same scope. The specified include directories are added to the compile command line for the current target and any targets within the same scope. This means that the include_directories are used when compiling the source files for the current target. => Used in `set_target_properties` command.
- `INTERFACE_INCLUDE_DIRECTORIES`: Specifies the include directories that are needed for the target's consumers (i.e. the targets that link to the current targets). When we use this command, the specified include directories are not added to the compile command line for the current target. Instead, they are propagated to the compile command line of the linked targets to the current target => Used with the `target_include_directories` command.

But, what is the difference between keyword and command?
Keyword: special terms to modify the behaviour of the commands. They are not standalone commands, but they are used within the commands to specify certain properties. They are always written in uppercase. Ex. `VERSION` keyword, used in `project` command.

Include Inheritance (A bit similar to access specifier in OOP)

In CMake, for any target, in the preprocessing stage, it comes with a `INCLUDE_DIRECTORIES` and a `INTERFACE_INCLUDE_DIRECTORIES` for searching for the header files building. target_include_directories will populate all the directories to `INCLUDE_DIRECTORIES` and/or `INTERFACE_INCLUDE_DIRECTORIES` depending on the keyword `PUBLIC | PRIVATE | INTERFACE` we specified. [Note that `INCLUDE_DIRECTORIES` is used to store directories of the target itself and INTERFACE_INCLUDE_DIRECTORIES is used to store the directories that are propagated to the target's consumers]. The `INCLUDE_DIRECTORIES` will be used for the current target only and the `INTERFACE_INCLUDE_DIRECTORIES` will be appended to the `INCLUDE_DIRECTORIES` of any other target which has dependencies on the current target, With such settings, the configurations of `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES` for all building targets are easy to compute and scale up even for multiple hierarchical layers for building dependencies and many building targets.

Include inheritance	Description
PUBLIC	All directories following `PUBLIC` will be used for the current target and the other targets that have dependencies on the current target (will be appended the directories to INCLUDE_DIRECTORIES and `INTERFACE_INCLUDE_DIRECTORIES`).
PRIVATE	All the include directories following `PRIVATE` will be used for the current target only (will be appended to `INCLUDE_DIRECTORIES`).
INTERFACE	All the include directories following `INTERFACE` will not be used for the current target, but will be accessible for the other targets that have dependencies on the current target (will be appended into `INTERFACE_INCLUDE_DIRECTORIES`).

When we do `target_link_libraries (<target> <PRIVATE | PUBLIC | INTERFACE> <item>)`, if the dependent <item> has been built in the same CMake project, `INTERFACE_INCLUDE_DIRECTORIES` of <item> would be appended to the `INCLUDE_DIRECTORIES` of <target>. By controlling the `INTERFACE_INCLUDE_DIRECTORIES`, we could eliminate some unwanted or conflicting declarations from <item> to the target.

For example, the fruit library has `INCLUDE_DIRECTORIES` of `fruit_h`, `tree_h`, and `INTERFACE_INCLUDE_DIRECTORIES` of `fruit_h`. If there is a `apple` library that is linked with the `fruit` library, the apple library would also have the `fruit_h` in its `INCLUDE_DIRECTORIES` as well. We could equivalently say, the `apple` library's include directory inherited the `fruit_h` of the `fruit` library.

LINK Inheritance

Similarly, for any target, in the linking stage. Given the item to be linked, we would need to decide whether we have to put the item in the link dependencies, or the link interface or both in the compiled target. The link dependencies means that the item has some implementations that target would use, and it is linked to the item. So, these implementations would always be mapped correctly to the implementations in item via the link, whereas the link interface means the target becomes an interface for linking the item for other targets which have dependencies on the target, and the target does not have to use item at all.

Link inheritance	Description
PUBLIC	All the directories following `PUBLIC` will be used for linking to the current target and the other targets that have dependencies on the current target (i.e. The directories will be appended into `INCLUDE_DIRECTORIES` and `INTERFACE_INCLUDE_DIRECTORIES`).
PRIVATE	All the objects following `PRIVATE` will be used for linking to the current target only.
INTERFACE	All the objects following `INTERFACE` will be used for linking to the targets that have dependencies on the current target (Only provides an interface to them)

EX.

```cmake
# Define the fruit library
add_library(fruit STATIC fruit.cpp)

# Speecify include directories for fruit library
target_include_directories(fruit
                           PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/fruit_h
                           PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/tree_h
                           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/fruit_h
                          )

# Define the apple library
add_library(apple STATIC apple.cpp)

# Line the apple library with the fruit library
target_link_libraries(apple PRIVATE fruit)
```

This command is used to link two targets

Include Inheritance:

1. `Target_include_directories` for fruit library:
   - `PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/fruit_h`:
     - it adds `fruit_h` to `fruit`'s include directories(`INCLUDE_DIRECORIES`).
     - Also makes `fruit_h` available to any target that links with `fruit` (`INTERFACE_INCLUDE_DIRECTORIES`); as it is set to be PUBLIC.

- `PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/tree_h`:
    - Adds `tree_h` only to `fruit`'s include directories (`INCLUDE_DIRECTORIES`).
    - `tree_h` is not available to targets linking with fruit.
- `INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/fruit_h`:
    - Adds `fruit_h` to fruit's `INTERFACE_INCLUDE_DIRECTORIES` without adding it to fruit's `INCLUDE_DIRECTORIES`.

In conclusion, `INCLUDE_DIRECTORIES` of `fruit` will include `fruit_h` as the relation is set to `PUBLIC`, and `tree_h` as the relation is set to `PRIVATE`.

But, `INTERFACE_INCLUDE_DIIRECTORIES` of `fruit` will include `fruit_h`, because the relation is set to both `INTERFACE` and `PUBLIC`.

Link Inheritance:

Links `apple` to `fruit` makes the `fruit`'s public and interface include directories available to `apple`. This means that `apple` inherits `fruit`'s `INTERFACE_INCLUDE_DIRECTORIES` which includes `fruit_h`. Thus, after linking, the `apple` library will also have `fruit_h` in its `INCLUDE_DIRECTORIES`.

Despite using `PRIVATE` in `target_link_directories(apple PRIVATE fruit)`, apple will still get the `INTERFACE` properties from fruit:

- The `INTERFACE_INCLUDE_DIRECTORIES` property of `fruit` is always intended for propagation to consumers (Linked targets).
- When `apple` linkes to `fruit` privately, `apple` will get the include directories specified by `fruit`'s `INTERFACE_INCLUDE_DIRECTORIES`.
- `apple` links against `fruit` means that `apple` depends on `fruit` which means that it's required that the compiled output of `fruit` be available when building and linking apple.
- The keyword `PRIVATE` here indicates that the `fruit` library is only used internally by `apple`. It ensures that the properties of `fruit` are not propagated to other targets that might link against apple. (i.e. any targets linking to `apple` will not automatically link against `fruit`).
- `apple` depends on `fruit`: This doesn't imply that `fruit` depends on `apple`. Instead, it means that `apple` needs `fruit` to be built and available during the linking phase of `apple`.

Propagation visual representation:

```
fruit
|-- INCLUDE_DIRECTORIES: fruit_h, tree_h
|-- INTERFACE_INCLUDE_DIRECTORIES: fruit_h

apple (linked PRIVATE to fruit)
|-- INCLUDE_DIRECTORIES: fruit_h (from fruit's INTERFACE_INCLUDE_DIRECTORIES)
```

If we want to summarize the target dependencies in three sentences, we can say that:

`PRIVATE` only cares about itself and doesn't allow sharing with others (doesn't allow inheritance). `INTERFACE` only cares about the others (allow inheritance to them). `PUBLIC` cares about itself and every other one (allows inheritance).

We can use CMAKE to restrict access to any number of header files; This can keep the code clean from unwanted dependencies. We want to disappear header files (don’t be found).

Ex. we don’t want to compile the following code:

```
#include "privateHeader.h"  // shouldn’t compile; "no such file or directory"
#include "privateHeader.h"  // shouldn’t compile; "no such file or directory"
```

To do this, we should find the paths belong to this particular target. Then for each path in the given target we should decide whether it should be accessible to others or not. We do this using `target_include_directory`

Ex: For the following project structure: If we want to give any target linked against libA access to headers inside include directory and the other files be private to libA and not accessible to other linked targets.

To do this, we will set include directory to PUBLIC and the other files inside the current source directory to PRIVATE.

This is a generator expression that evaluates to the content inside the angle brackets only when building the project.

```cmake
cmake_minimum_required(VERSION 3.5)

project(libA VERSION 1.0.0 LANGUAGES CXX)

# Add the library target
add_library(libA STATIC
    sourceA.cpp
    privateHeaderA1.h
    privateHeaderA2.h
)

# Set the include directories for the target
target_include_directories(libA
    PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}
)
```

$<INSTALL_INTERFACE: ......> ⇒ It is a complementary expression that evaluates to the content inside the angle brackets only when installing the project.

project structure

```
libA/
|--include/
    |--sourceA.h
|--privateHeaderA1.h
|--privateHeaderA2.h
|--sourceA.cpp

libB/
|--include/
    |--sourceB/
|--submodule/
    |--submodule.h
    |--submodule.cpp
|--privateHeaderB1.h
|--privateHeaderB2.h
|--sourceB.cpp
|--sourceB_impl.h
|--sourceB_impl.cpp
```

If we link a target called `libB` against `libA` and we want to limit the accessibility of `libB` files to itself and don't be propagated into any additional linked libraries against it:

We will set the linkage dependency to be `PRIVATE` through command `target_link_libraries`:

`target_link_libraries(libB PRIVATE libA)`, but firstly we will create a library(CMakeLists.txt) for libB:

```
cmake_minimum_required(VERSION 3.5)
project(libB VERSION 1.0.0 LANGUAGES CXX)

# Add the library target
add_library(libB STATIC
    sourceB.cpp
    sourceB_impl.cpp
    privateHeaderB1.h
    privateHeaderB2.h
    submodule/submodule.cpp
)

# Set the include directories for the target
target_include_directories(libB
    PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}
)

# Link libB to libA
target_link_libraries(libB PRIVATE libA)
```

By setting CMakeLists.txt file in this way, we will ensure that:

- `libB` can access `libA`'s public interface.
- `libB`'s internal headers and implementations are not accessible to other targets.
- `libA`'s linkage is not propagated to targets linking against `libB`.

We can break how this is happened in three parts:

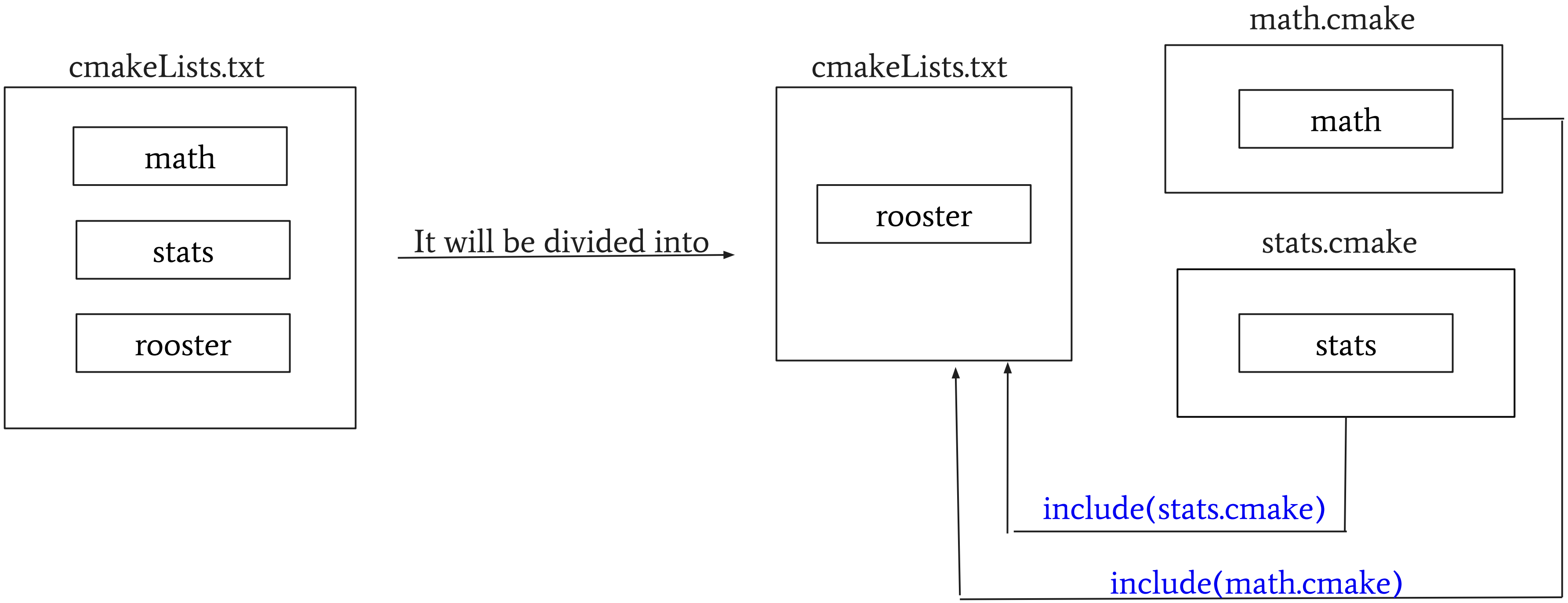
1. Include directories of `libA`:
    - The include directory is added to `libA`'s public interface, making it visible to any target linking against `libA`.
    - The `PRIVATE` part makes `libA`'s source directory be accessible only to `libA`.
2. Include directories for `libB`:
    - The include directory is added to `libB`'s public interface, making it accessible to any target linked against `libB`.
    - The `PRIVATE`'s part makes `libB`'s source directory accessible only to `libB` itself.
3. Linking `libB` to libA:
    - `target_link_libraries(libB PRIVATE libA)` ensures that `libB` is linked against `libA` but isn't propagated this linkage to the targets that are linked against `libB`.

5) Commands

5.1 include

If we want to divide our CMakeLists.txt file into other smaller parts/ blocks, each block can be used in different other files, we use:

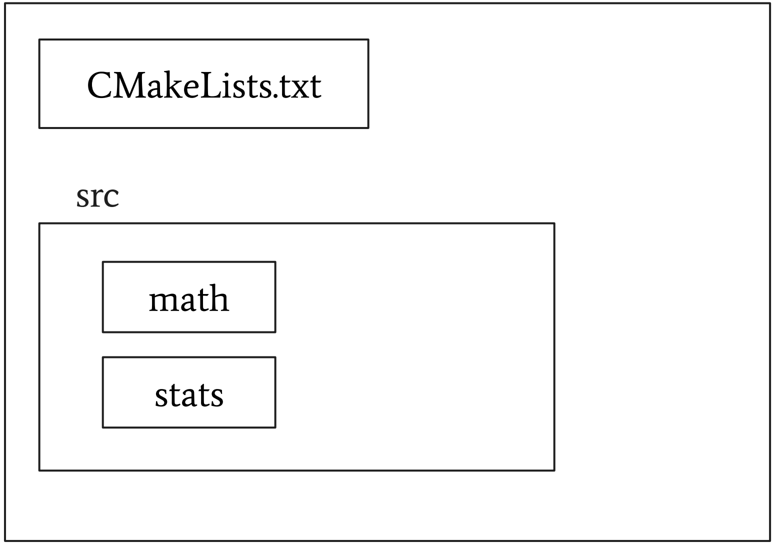
- Include command: to include and execute the contents of another CMake file within the current CMake file.
- Sub_directory command: to add subdirectory to the build, and processing its CMakeLists.txt file as part of the build process.



We will have 3 partitions for our cmake files, one for our binary file(rooster), and the other for math target binary logic, and another one for stats target logic.

Then, the project structure will become:

```
program /
|-- CMakeLists.txt /
|-- src /
   |-- main.cpp /
   |-- math /
      |-- math.h /
      |-- math.cpp
      |-- math.cmake /
   |-- stats
      |-- stats.h /
      |-- stats.cpp /
      |-- stats.cmake /
```



When using include command, cmake will copy the cmake script inside the include command only. This means that for our example, the content of the cmake script of both math and stats targets will be copied into the CMakeLists.txt file inside the root. This is the downside of include command, it's going to populate the global scope. It acts as the C/C++ preprocessor, copying code and making it available in the main CMakeLists.txt file.