1. Create separate folder for source code files
2. Create separate folder for build files typically with name "build"
3. Create CMakeLists.txt file in the source folder
4. CMakeLists.txt file example

cmake_minimum_required(VERSION 1.2.3)

This must be the first line in every cmake file to tell that what is the minimum version is required to work with this cmake file

project(PROJECT_NAME
VERSION 1.2.3
DESCRIPTION "A Sample Project"
LANGUAGES CXX)

This command sets the name of the project, its version number, description of the project and sets the compiler for the compiling the project

add_executable(PROJECT_BINARY_FILE source1.cpp source2.cpp)

This command creates an executable file for the project with the source file(s) mentioned next.

target_compile_features(PROJECT_BINARY_FILE PRIVATE cxx_std_20)

This command assigns the C++ standard 20 to generate PROJECT_BINARY_FILE.

5. Go to build directory and use command
cmake ../source/

This will build the project using the rules specified in CMakeLists.txt

6. Go to build directory and use command
cmake --build .

This will build the project using the default build system present in the system.

CMAKE GENERATORS

Cmake allows you to use different build projects such as

1. Visual studio
2. Mingw-make
3. Unix Makefiles
4. Ninja etc.

You can select a different build system using -G flag. For example lets say I want to build project using mingw-make.

Use below commands

Cmake –help

This will give the information about types of generators present in the cmake. Note that MS Visual studio does not know the *help* command. This command is valid only if we are using makefile or ninja build system

cmake -G "Mingw Makelfiles" ../source/

This will consider the mingw compiler to build the project

Then in build directory use *mingw32-make* command to build and create executable

Note: Ensure the corresponding binaries for the build system are present in the C drive and environmental variable path is set.

Similar approach can be followed for the other build systems.

We can use a unified way to build and run the binary using below command. We don't have to remember the native build system.

cmake --build . --target *PROJECT_BINARY_FILE*

Multi file C++ Project With Cmake

Generally, while developing CPP project, we separate out header files and source files.

```
Project
|--source
        |---include
        |---src
|--CMakelists.txt
|--main.cpp
```

We keep main.cpp and CMakeLists.txt file int the main source folder. The project related header files and source codes are stored in include and src folders respectively.

Then updated cmake includes

add_executables(PROJECT_BINARY main.cpp /src/dog.cpp /src/cat.cpp)

This will consider all source files required to build the project.

target_include_directories(*PROJECT_BINARY ${ CMAKE_CURRENT_SOURCE_DIR }/include*)

This will provide path for all header files. *CMAKE_CURRENT_SOURCE_DIR* is a cmake variable that holds the address of the folder where *CMakeLists.txt* file is saved.

We can also use the GLOBING to avoid the specifying each source file separately on add_executables() call!

file(GLOB_RECURSE SOURCE_FILES src/*.cpp)

This will pass all source .cpp files to the cmake variable SOURCE_FILES from *src* folder. The we use the add_executable() function as below

add_executable(*PROJECT_BINARY main.cpp ${SOURCE_FILES}*)

However GLOBING way of passing the source files is DISCOURAGED by cmake

SELECTING COMPILERS

If you are using windows OS the default compiler is MSVC i.e. Microsoft Visual Studio Compiler
If you are using Ninja Generator it also uses the MSVC
If you use MinGW Makefiles generator it uses g++ compiler

On Linux we don't have access to the MSVC.
If you use Ninja/MinGW Makefiles generator it uses g++
However, it is possible to change the compiler used by your generator as mentioned below.

Visual Studio Generator	MSVC	Clang	
Ninja Generator	MSVC	Clang	g++
MinGW Makefiles Generator	g++	Clang	

cmake --system-information info.txt

This command gives the all details of default configuration of cmake. You can find a variable in the info.txt file named CMAKE_CXX_COMPILER which stores the default compiler information cmake is using.
We want to change this default behaviour of cmake i.e. use a different compiler instead of the default one.
Example

cmake -G "Ninja" CMAKE_CXX_COMPILER=g++ ../source/

Here we are specifying a compiler g++ instead of using the default one. Note that if the specified compiler is not found then it will switch back to the default one.

Lets say I want to generate build system using Visual Studio generator but I would want to use clang compiler.

cmake -G "Visual Studio 16 2019" -T"ClangCL" ../source/

Make sur Clang must be installed at your system as part of visual studio IDE. If it is not installed then it will throw error.

cmake -GNinja -D CMAKE_CXX_COMPILER=cl ../source

Here build system is Ninja but compiler is clang

cmake -GNinja -D CMAKE_CXX_COMPILER=g++ ..\source

Here build system is Ninja but compiler is g++

cmake -G"MinGW Makefiles" -D CMAKE_CXX_COMPILER=g++ ..\source

Here build system is Ninja but compiler is g++

cmake -G"MinGW Makefiles" -D CMAKE_CXX_COMPILER=clang++ ..\source

Here build system is Ninja but compiler is clang++

CREATING MULTIPLE TARGETS USING STATIC LIBRARY

We can create a static library that is a complete functionality without having the main function. We can build the target for the static library just like we build target for entire project. Then the target build for static library can be used as input to build the entire project target.

Lets add

```
|--source
        |---include
            |--dog.h
            |--operations.h
            |--log.h
        |---src
            |--dog.c
            |--operations.c
            |--log.c
|--CMakelists.txt
|--main.cpp
```

```
cmake_minimum_required(VERSION 3.10)
project(HelloApp
        VERSION 1.0
        DESCRIPTION "Hello App Project"
        LANGUAGES CXX)

add_library(oprtaions src/operations.cpp)
target_compile_features(oprtaions PUBLIC CXX_std_20)
target_inlcude_directories(operations PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

add_executable(HelloAppBinary main.cpp
                              src/log.cpp
                              src/dog.cpp)
target_include_directories(HelloAppBinary PUBLIC ${CMAKE_CURENT_SOURCE_DIRECTORY}/include)
target_compile_features(HelloAppBinary PUBLIC cxx_std_20)
target_inlcude_directories(operations PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

target_link_libraries(HelloAppBinary PUBLIC operations)
```

SETTING UP C++ STANDARDS

set(CXX_STANDARD_REQUIRED on)

This sets hard requirement on C++ specific standard. If that standard not followed then it will throw error.

set(CMAKE_CXX_STANDARD 20)

This will set C++ standard as 20 for entire project. We do not have to specify C++ standard for every target using target_compile_features().

Build only specific target

cmake --build --target logger ../source
This will only build the logger target not the entire project. This will create STATIC logger library with name liblogger.a

If we specify the final target in the below command, it will build the entire project and understand the dependency automatically as mentioned in the CMakeLists.txt file
cmake --build --target HelloAppBinary ../source

TARGET DEPENDENCIES – PRIVATE, PUBLIC AND INTERFACE

```
|--source
        |---math
            |--include
                |--supermath.h
            |--supermath.cpp
        |---stats
            |--include
                |--stats.h
            |--stats.cpp
|--CMakelists.txt
|--main.cpp
```

Here we can create two static libraries libmath and libstats using add_library().

```
add_library(libmath STATIC ${CMAKE_CURRENT_SOURCE_DIR}/math/supermath.cpp)
target_include_directories(libmath PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/math/include)

add_library(libstats STATIC ${CMAKE_CURRENT_SOURCE_DIR}/stats/stats.cpp)
target_include_directories(libstats PUBLIC ${CMAKE_CURRENT_DIR}/stats/include)
```

For both libraries we will add respective include folders.
PRIVATE: Use when the requirement is only needed internally by the target (e.g., internal headers).
INTERFACE: Use when the requirement is only needed by others using the target (e.g., header-only library).
PUBLIC: Use when the requirement is needed by both the target and dependents.

libstat requires functionality from the libmath hence we need to first link these two libraris.

```
target_link_libraries(libstats PUBLIC libmath)
```
Here PUBLIC indicates all the dependencies will be taken by libstat from libmath and it will be forwarded to any other library inherits libstats.

These two libraries will be used to build the final target *rooster* that can be linked using target_link_libraries()

```
add_executable(rooster ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)
target_link_libraries(rooster PUBLIC libstats)
```

ORGNIZING CMAKE PROJECT

{CMAKE_CURRENT_SOURCE_DIR} does not give the source directory where the file lies. It is relative to the file CMakeLists.txt file no matter in which file your refer this variable.

```
|--source
        |---math
            |--include
                |--supermath.h
            |--supermath.cpp
            |--math.cmake
        |---stats
            |--include
                |--stats.h
            |--stats.cpp
            |--stats.cmake
|--CMakelists.txt
|--main.cpp
```

Here we are crating math.cmake and stats.cmake in their respective folder and include into the main CMakeLists.txt file. This will make cmake code modular and written into respective folder of static library.

math.cmake created in the math folder

```
add_library(libmath STATIC ${CMAKE_CURRENT_DIR}/math/supermath.cpp)
target_include_directories(libmath PUBLIC ${CMAKE_CURRENT_DIR}/math/include)
```

stats.cmake created in the stats folder

```
add_library(libstats STATIC STATIC ${CMAKE_CURRENT_DIR}/stats/stats.cpp)
target_include_directories(libstat PUBLIC ${CMAKE_CURRENT_DIR}/stats/include )
```

The main CMakeLists.txt simply includes these .cmake files from the respective folders.

CMakeLists.txt

```
include(math/math.cmake)
include(stats/stats.cmake)

target_link_libraries(libstats PUBLIC libmath)

add_executable(rooster ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)
target_link_libraries(rooster PUBLIC libstats)
```

Note that include() works like preprocessor like in C/C++ it simply just replace it with the code from included .cmake files. This may cause conflicting on some global variables so generally not recommended.

USING ADD_SUBIRECTORY COMMAND

```
|--source
        |---math
            |--include
                |--supermath.h
            |--supermath.cpp
            |--CMakeLists.txt
        |---stats
            |--include
                |--stats.h
            |--stats.cpp
            |-- CMakeLists.txt
|--CMakelists.txt
|--main.cpp
```

As you can observe now we have created a separate CMakeLists.txt file for each library. This way we do not need to specify the relative path w.r.t. main CMakeLists.txt. For individual CMakeLists.txt the source directory will be the same in which it belongs to.

```
|---math
        |--CMakeLists.txt

add_library(libmath STATIC ${CMAKE_CURRENT_SOURCE_DIR}/supermath.cpp)
target_include_directories(libmath PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

|---stats
        |--CMakeLists.txt
add_library(libstats STATIC ${CMAKE_CURRENT_SOURCE_DIR}/stats.cpp)
target_include_directories(libstats PUBLIC ${CMAKE_CURRENT_DIR}/include)
target_link_libraries(libstats PUBLIC libmath)
```

|--source
|--CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(rooster
    VERSION 1.0
    DESCRIPTION "Rooster to demonstrate PRIVATE PUBLIC and INTERFACE linking"
    LANGUAGES CXX)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_STANDARD 20)

add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/math)
add_subdirectory(${CMAKE_CURRENT_SOURCE_DIR}/stats)


add_executable(rooster ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)
target_link_libraries(rooster PUBLIC libstats)
```