C/C++ Program Design

LAB 8

CONTENTS

- Learn makefile
- Learn cmake

2 Knowledge Points

2.1 Makefile

2.2 CMake

2.1 Makefile

What is a makefile?

Makefile is a tool to simplify or to organize for compilation. **Makefile is a set of commands with variable names and targets .** You can compile your project(program) or only compile the update files in the project by using Makefile.

Suppose we have four source files as follows:

```cpp
// factorial.cpp

#include "functions.h"
int factorial(int n)
{

    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);

}
```

```cpp
// main.cpp

#include <iostream>
#include "functions.h"
using namespace std;

int main()
{
    print_hello();

    cout << "This is main:" << endl;
    cout << "The factorial of 5 is: " << factorial(5) << endl;

    return 0;

}
```

```cpp
// printhello.cpp

#include <iostream>
#include "functions.h"
using namespace std;

void print_hello()
{
    cout << "Hello World!" << endl;
}
```

```cpp
// functions.h
void print_hello();
int factorial(int n);
```

Normally, you can compile these files by the following command:

```
$ g++ -o hello main.cpp printhello.cpp factorial.cpp
```

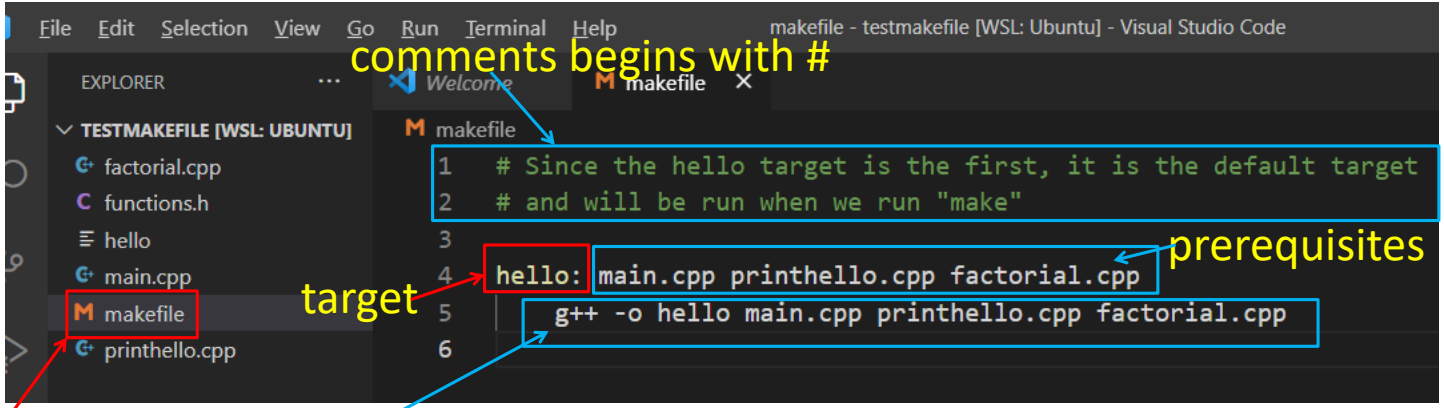
How about if there are hundreds of files need to compile? Do you think it is comfortable to write g++ or gcc compilation command by mentioning all these hundreds file names? Now you can choose makefile.

The name of makefile must be either **makefile** or **Makefile** without extension. You can write makefile in any text editor. A rule of makefile including three elements: **targets**, **prerequisites** and **commands**. There are many rules in the makefile.

A makefile consists of a set of rules. A rule including three elements: **target**, **prerequisites** and **commands**.

```
targets : prerequisites
<TAB> command
```

- The **target** is an object file, which means the program that need to compile. Typically, there is only one per rule.
- The **prerequisites** are file names, separated by spaces.
- The **commands** are a series of steps typically used to make the target(s). These need to start with a **tab character**, not spaces.



Put the makefile together with your programs.

commands

g++ is compiler name, -o is linker flag and hello is binary file name.

Type the command make in VScode

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make

Command 'make' not found, but can be installed with:

sudo apt install make          # version 4.2.1-1.2, or
sudo apt install make-guile    # version 4.2.1-1.2
```

If you don't install make in VScode, install it first according to the instruction.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
g++ -o hello main.cpp printhello.cpp factorial.cpp
```

Run the commands in the makefile automatically.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ ./hello
Hello World!
This is main:
The factorial of 6 is: 720
```

Run your program

output

Defining Macros/Variables in the makefile

To improve the efficiency of the makefile, we use variables.

variables

```
# Using variables in makefile
CC = g++
TARGET = hello
OBJ = main.o printhello.o factorial.o
$(TARGET) : $(OBJ)
    $(CC) -o $(TARGET) $(OBJ)
```

Write target, prerequisite and commands by variables using '$()'

If only one source file is modified, we need not compile all the files. So, let's modify the makefile.

```
# Using several rules and several targets

CC = g++
TARGET = hello
OBJ = main.o printhello.o factorial.o
$(TARGET) : $(OBJ)
    $(CC) -o $(TARGET) $(OBJ)

main.o: main.cpp
    $(CC) -c main.cpp

printhello.o: printhello.cpp
    $(CC) -c printhello.cpp

factorial.o: factorial.cpp
    $(CC) -c factorial.cpp
```

targets

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
g++ -c main.cpp
g++ -c printhello.cpp
g++ -c factorial.cpp
g++ -o hello main.o printhello.o factorial.o
```

If main.cpp is modified, it is compiled by make.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
g++ -c main.cpp
g++ -o hello main.o printhello.o factorial.o
```

All the **.cpp** files are compiled to the **.o** files, so we can modify the makefile like this:

```
# Using several rules and several targets

CC = g++
TARGET = hello
OBJ = main.o printhello.o factorial.o

# options pass to the compiler
# -c generates the object file
# -Wall displays complier warning
CFLAGES = -c -Wall

$(TARGET) : $(OBJ)
    $(CC) -o $@ $(OBJ)

%.o: %.cpp
    $(CC) $(CFLAGES) $< -O $@
```

warning 提示

This is a model rule, which indicates that all the .o objects depend on the .cpp files

$@: Object Files

$^: all the prerequisites files

$<: the first prerequisite file

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
g++ -o hello main.o printhello.o factorial.o
```

Using phony target to clean up compiled results automatically

```
# Using several rules and several targets

CC = g++
TARGET = hello
OBJ = main.o printhello.o factorial.o

# options pass to the compiler
# -c generates the object file
# -Wall displays complier warning
CFLAGES = -c -Wall

$(TARGET) : $(OBJ)
    $(CC) -o $@ $(OBJ)

%.o: %.cpp
    $(CC) $(CFLAGES) $< -O $@

.PHONY:clean
clean:
    rm -f *.o $(TARGET)
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make clean
rm -f *.o hello
```

Adding **.PHONY** to a target will prevent making from confusing the phony target with a file name.

Functions in makefile

wildcard: search file

for example:

Search all the .cpp files in the current directory, and return to SRC

```
SRC = $(wildcard ./*.cpp)
```

```
SRC = $(wildcard ./*.cpp)
target:
    @echo $(SRC)
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
./printhello.cpp ./factorial.cpp ./main.cpp
```

All .cpp files in the current directory

patsubst(pattern substitution): replace file

$(**patsubst** original pattern, target pattern, file list)

Replace all .cpp files with .o files

for example:

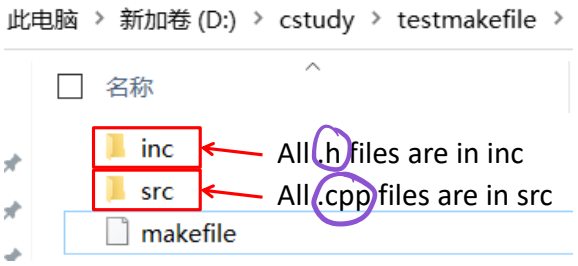
```
OBJ = $(patsubst  %.cpp,  %.o, $(SRC))
```

```
SRC = $(wildcard ./*.cpp)
OBJ = $(patsubst %.cpp, %.o, $(SRC))
target:
    @echo $(SRC)
    @echo $(OBJ)
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
./factorial.cpp ./printhello.cpp ./main.cpp
./factorial.o ./printhello.o ./main.o
```

Replace all .cpp files with .o files

```makefile
# Using functions

SRC_DIR = ./src
SOURCE  = $(wildcard $(SRC_DIR)/*.cpp)
OBJS    = $(patsubst %.cpp, %.o, $(SOURCE))
TARGET  = hello
INCLUDE = -I./inc


# options pass to the compiler
# -c  says to generate the object file
# -wall turns on most, but not all, complier warning
CC     = g++
CFLAGS = -c -Wall

$(TARGET):$(OBJS)
    $(CC) -o $@ $(OBJS)
%.o: %.cpp
    $(CC) $(CFLAGS) $< -o $@ $(INCLUDE)


.PHONY:clean
clean:
    rm -f $(SRC_DIR)/*.o $(TARGET)
```




```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/testmakefile$ make
g++ -c -Wall src/printhello.cpp -o src/printhello.o -I./inc
g++ -c -Wall src/factorial.cpp -o src/factorial.o -I./inc
g++ -c -Wall src/main.cpp -o src/main.o -I./inc
g++ -o hello  ./src/printhello.o  ./src/factorial.o  ./src/main.o
```

GNU Make Manual

http://www.gnu.org/software/make/manual/make.html

Use Options That Control Optimization

-O1, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-O2,Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O1, this option increases both compilation time and the performance of the generated code.

-O3 Optimize yet more. O3 turns on all optimizations specified by -O2.

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

https://blog.csdn.net/xinianbuxiu/article/details/51844994

```makefile
# Using function and optimization
SRC_DIR = ./src
SOURCE = $(wildcard $(SRC_DIR)/*.cpp)
OBJS   = $(patsubst %.cpp, %.o, $(SOURCE))
TARGET  = hello
INCLUDE = -I./inc

# options pass to the compiler
# -c  generates the object file
# -Wall displays complier warning
# -o0: no optimizations
# -o1: default optimization
# -o2: represents the second-level optimization
# -o3: represents the highest level optimization
# os: equivalent to -o2.5 optimization, but with no visible code size

CC     = g++
CFLAGS = -c -Wall
CXXFLAGS = $(CFLAGS) -O3

$(TARGET):$(OBJS)
    $(CC) -o $@ $(OBJS)
%.o: %.cpp
    $(CC) $(CXXFLAGS) $< -o $@ $(INCLUDE)


.PHONY:clean
clean:
    rm -f $(SRC_DIR)/*.o $(TARGET)
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/makefilebyO3$ make
g++ -c -Wall -O3 src/factorial.cpp -o src/factorial.o -I./inc
g++ -c -Wall -O3 src/printhello.cpp -o src/printhello.o -I./inc
g++ -c -Wall -O3 src/main.cpp -o src/main.o -I./inc
g++ -o hello  ./src/factorial.o  ./src/printhello.o  ./src/main.o
```

2.2 CMake

What is CMake?

Cmake is an open-source, cross-platform family of tools designed to build, test and package software. **Cmake** is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.

CMake needs **CMakeLists.txt** to run properly.

A CMakeLists.txt consists of **commands** , **comments** and **spaces**.

- The **commands** include command name, brackets and parameters ,

the parameters are separated by spaces. Commands are not case sensitive.

- **Comments** begins with ‘#’.

1. A single source file in a project

The most basic project is an executable built from source code files. For simple projects, a three-line **CMakeLists.txt** file is all that is required.

```
cmake_minimum_required(VERSION 3.10)

project (hell0)

add_executable(hello main.cpp)
```

Specifies the minimum required version of CMake.

Defines the project name.

Adds the hello executable target which will be built from main.cpp.

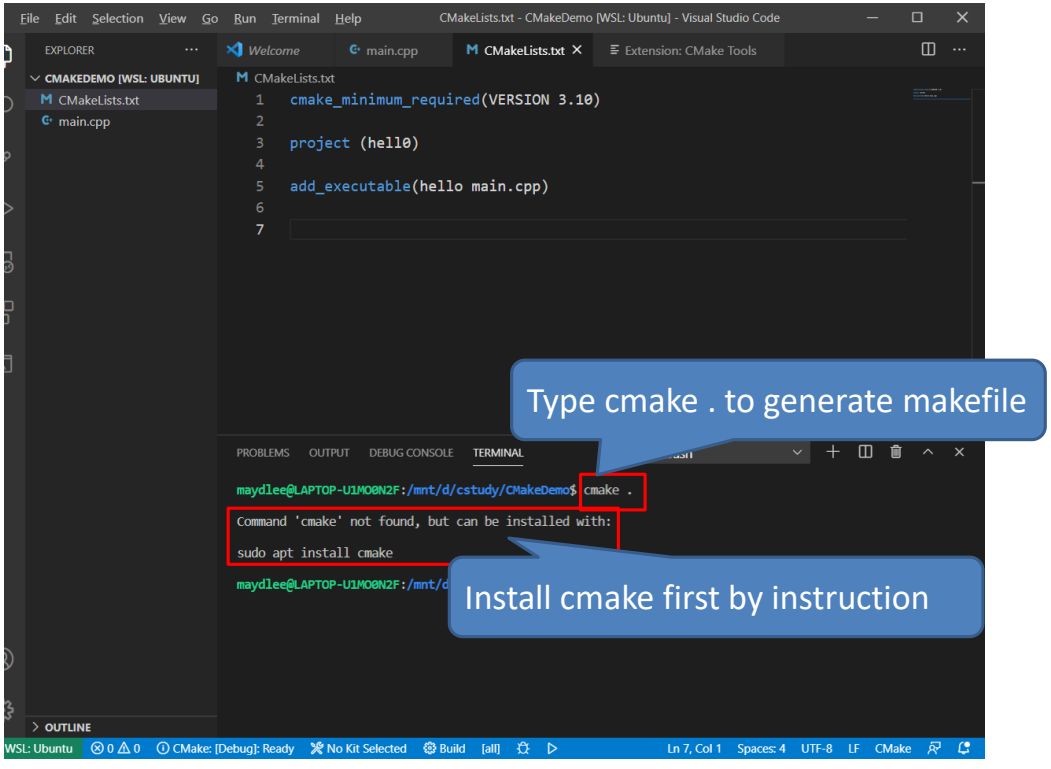
The first parameter indicates the filename of executable file.

The second parameter indicates the source file.

Suppose we have a main.cpp file

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!" << endl;

    return 0;
}
```

Store the CMakeLists.txt file in the same directory as the main.cpp.


File Edit Selection View Go Run Terminal Help
CMakeLists.txt - CMakeDemo [WSL: Ubuntu] - Visual Studio Code
EXPLORER
CMAKEDEMO [WSL: UBUNTU]
CMakeLists.txt
main.cpp
Welcome
main.cpp
CMakeLists.txt
Extension: CMake Tools
CMakeLists.txt
cmake_minimum_required(VERSION 3.10)
project (hello)
add_executable(hello main.cpp)
Type cmake . to generate makefile
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo$ cmake .
Command 'cmake' not found, but can be installed with:
sudo apt install cmake
Install cmake first by instruction
OUTLINE
WSL: Ubuntu
CMake: [Debug]: Ready
No Kit Selected
Build
[all]
Ln 7, Col 1
Spaces: 4
UTF-8
LF
CMake

File Edit Selection View Go Run Terminal Help CMakeLists.txt - CMakeDemo [WSL: Ubuntu] - Visual Studio Code

EXPLORER

CMAKEDEMO [WSL: UBUNTU]

- CMakeLists.txt
- main.cpp

Welcome | main.cpp | CMakeLists.txt | Extension: CMake Tools

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project (hell0)

add_executable(hello main.cpp)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: sudo

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo$ sudo apt install cmake
[sudo] password for maydlee:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  cmake-data libjsoncpp1 librhash0
Suggested packages:
  cmake-doc ninja-build
The following NEW packages will be installed:
  cmake cmake-data libjsoncpp1 librhash0
0 upgraded, 4 newly installed, 0 to remove and 151 not upgraded.
Need to get 5470 kB of archives.
After this operation, 28.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

OUTLINE

WSL: Ubuntu | 0 0 | CMake: [Debug]: Ready | No Kit Selected | Build [all] | Ln 7, Col 1 | Spaces: 4 | UTF-8 | LF | CMake

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/cs
```

Run cmake to generate makefle, . indicates the makefile is stored in the current directory.

Makefile file is created automatically after running cmake in the current directory.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo$ ls
CMakeCache.txt  CMakeFiles  CMakeLists.txt  Makefile  cmake_install.cmake  main.cpp
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo$ make
Scanning dependencies of target hello
[ 50%] Building CXX object CMakeFiles/hello.dir/main.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
```

Execute make to compile the program.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo$ ./hello
Hello World!
```

Run the program

2. Multi-source files in a project

There are three files in the same directory.

```
./CmakeDemo2
    |
    +--- main.cpp
    |
    +--- function.cpp
    |
    +--- function.h
```

```
cmake_minimum_required(VERSION 3.10)

project(CmakeDemo2)

add_executable(CmakeDemo2 main.cpp function.cpp)
```

Put the function.cpp into the add_executable command.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo2$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/cstudy/CMakeDemo2
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo2$ make
Scanning dependencies of target CmakeDemo2
[ 33%] Building CXX object CMakeFiles/CmakeDemo2.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/CmakeDemo2.dir/function.cpp.o
[100%] Linking CXX executable CmakeDemo2
[100%] Built target CmakeDemo2
```

2. Multi-source files in a project

If there are several files in directory, put each file into the add_executable command is not recommended. The better way is using **aux_source_directory** command.

aux_source_directory (<dir> <variable>)

The command finds all the source files in the specified directory indicated by <dir> and stores the results in the specified variable indicated by <variable>.

2. Multi-source files in a project

```
cmake_minimum_required(VERSION 3.10)

project(CmakeDemo2)

aux_source_directory(. DIR_SRCS)

add_executable(CmakeDemo2 ${DIR_SRCS})
```

Store all files in the current directory into DIR_SRCS.

Compile the source files in the variable by ${ } into an executable file named CmakeDemo2

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo2$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/cstudy/CMakeDemo2
```

3. Multi-source files in a project in different directories

```
./CMakeDemo3
    |
    +--- src/
    |     |
    |     +-- main.cpp
    |     +-- function.cpp
    |
    +--- include/
          |
          +--- function.h
```

We write CMakeLists.txt in CmakeDemo3 folder.

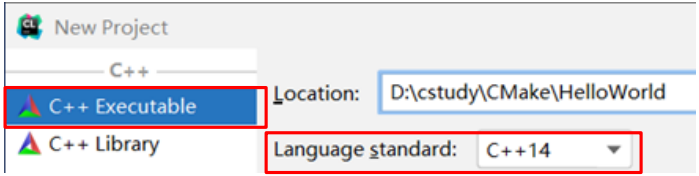
```cmake
# CMake minimum version
cmake_minimum_required(VERSION 3.10)

# project information
project(CMakeDemo3)

# Search the source files in the src directory
# and store them into the variable DIR_SRCS
aux_source_directory(./src DIR_SRCS)

# add the directory of include
include_directories(include)

# Specify the build target
add_executable(CMakeDemo3 ${DIR_SRCS})
```

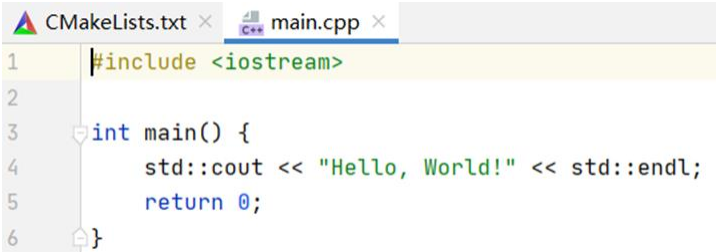
All .cpp files are in the src directory

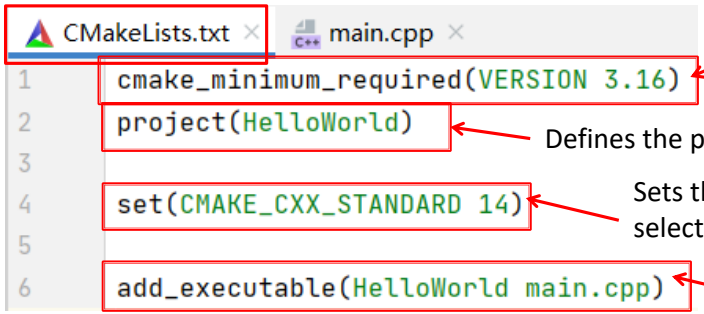
Include the header file which is stored in include directory.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo3$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /mnt/d/cstudy/CMakeDemo3
maydlee@LAPTOP-U1MO0N2F:/mnt/d/cstudy/CMakeDemo3$ make
Scanning dependencies of target CMakeDemo3
[ 33%] Building CXX object CMakeFiles/CMakeDemo3.dir/src/function.cpp.o
[ 66%] Building CXX object CMakeFiles/CMakeDemo3.dir/src/main.cpp.o
[100%] Linking CXX executable CMakeDemo3
[100%] Built target CMakeDemo3
```

Create a C++ project by CLion, the CMakeList.txt is created automatically.







Specifies the minimum required version of Cmake. It is set to the version of Cmake bundled in Clion (always one of the newest versions available).

Defines the project name according to what we provided during project creation.

Sets the CMAKE_CXX_STANDARD variable to the value of 14, as we selected when creating the project.

Adds the HelloWorld executable target which will be built from main.cpp.

For more about Cmake(cmake tutorial):
https://cmake.org/cmake/help/latest/guide/tutorial/index.html

3 Exercises

The **CandyBar** structure contains **three** members.The first member holds the brand **name** of a candy bar.The second member holds the **weight** (which may have a fractional part) of the candy bar, and the third member holds **the number of calories** (an integer value) in the candy bar.

```
struct CandyBar
{
    char brand[30];
    double weight;
    int calories;
};
```

Write the following functions:

- **void set(CandyBar & cb),** that should ask the user to enter each of the preceding items of information to set the corresponding members of the structure.
- **void set(CandyBar* const cb)** ,that is a overloading function .
- **void show(const CandyBar & cb),**that displays the contents of the structure.
- **void show(const CandyBar* cb),**that is a overloading function .

Here is a **header file named candybar.h**

Put together a multi-file program based on this header. **One file**, **named candybar.cpp**, should provide suitable function definitions to match the prototypes in the header file. **An other file named main.cpp** should contain main() and demonstrate all the features of the prototyped functions.

```
#ifndef EXC_CANDYBAR_H
#define EXE_CANDYBAR_H
#include <iostream>

const int LEN = 30;
struct CandyBar{
    char brand[LEN];
    double weight;
    int calorie;
};

// prompt the user to enter the preceding items of
// information and store them in the CandyBar structure
void setCandyBar(CandyBar & cb);
void setCandyBar(CandyBar * cb);
void showCandyBar(const CandyBar & cb);
void showCandyBar(const CandyBar * cb);

#endif  //EXC_CANDYBAR_H
```

Complete the following two tasks:

1.Write a Makefile file to organize all of the three files for compilation. Run make to test your Makefile. Run your program at last.

2.Create new folder and copy your code to the new folder. Write a MakeLists.txt file for cmake to create Makefile automatically. Run cmake and make, and then run your program at last.

A sample runs might look like this:

```
Call the set function of Passing by pointer:
Enter brand name of a Candy bar: Millennium Munch
Enter weight of the Candy bar: 2.85
Enter calories (an integer value) in the Candy bar: 250

Call the show function of Passing by pointer:
Brand: Millennium Munch
Weight: 2.85
Calories: 250

Call the set function of Passing by reference:
Enter brand name of a Candy bar: Millennium Mungh
Enter weight of the Candy bar: 3.85
Enter calories (an integer value) in the Candy bar: 350

Call the show function of Passing by reference:
Brand: Millennium Mungh
Weight: 3.85
Calories: 350
```