

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:

```
// factorial.cpp

#include "functions.h"
int factorial(int n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

```
// printhello.cpp

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

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

```
// 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;
}
```

```
// 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.

The screenshot shows the Visual Studio Code interface with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'TESTMAKEFILE [WSL: UBUNTU]' containing files: 'factorial.cpp', 'functions.h', 'hello', 'main.cpp', 'makefile', and 'printhello.cpp'. The 'makefile' file is selected. The code editor shows the contents of 'makefile' with the following text:

```
1 # Since the hello target is the first, it is the default target
2 # and will be run when we run "make"
3
4 hello: main.cpp printhello.cpp factorial.cpp
5     g++ -o hello main.cpp printhello.cpp factorial.cpp
6
```

Annotations are present:

- A yellow arrow points to the '#' character in line 1, with the text 'comments begins with #'.
- A red box highlights the 'makefile' file in the file explorer, with a red arrow pointing to it from the text 'Put the makefile together with your programs.'
- A red box highlights the 'hello:' target in line 4, with a red arrow pointing to it from the text 'target'.
- A blue box highlights the prerequisites 'main.cpp printhello.cpp factorial.cpp' in line 4, with a blue arrow pointing to it from the text 'prerequisites'.
- A blue box highlights the command 'g++ -o hello main.cpp printhello.cpp factorial.cpp' in line 5, with a blue arrow pointing to it from the text 'commands'.

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-U1M00N2F:/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-U1M00N2F:/mnt/d/cstudy/testmakefile$ make  
g++ -o hello main.cpp printhello.cpp factorial.cpp
```

Run the commands in the makefile automatically.

```
maydlee@LAPTOP-U1M00N2F:/mnt/d/cstudy/testmakefile$ ./hello  
Hello World!  
This is main:  
The factorial of 6 is: 720
```

Run your program

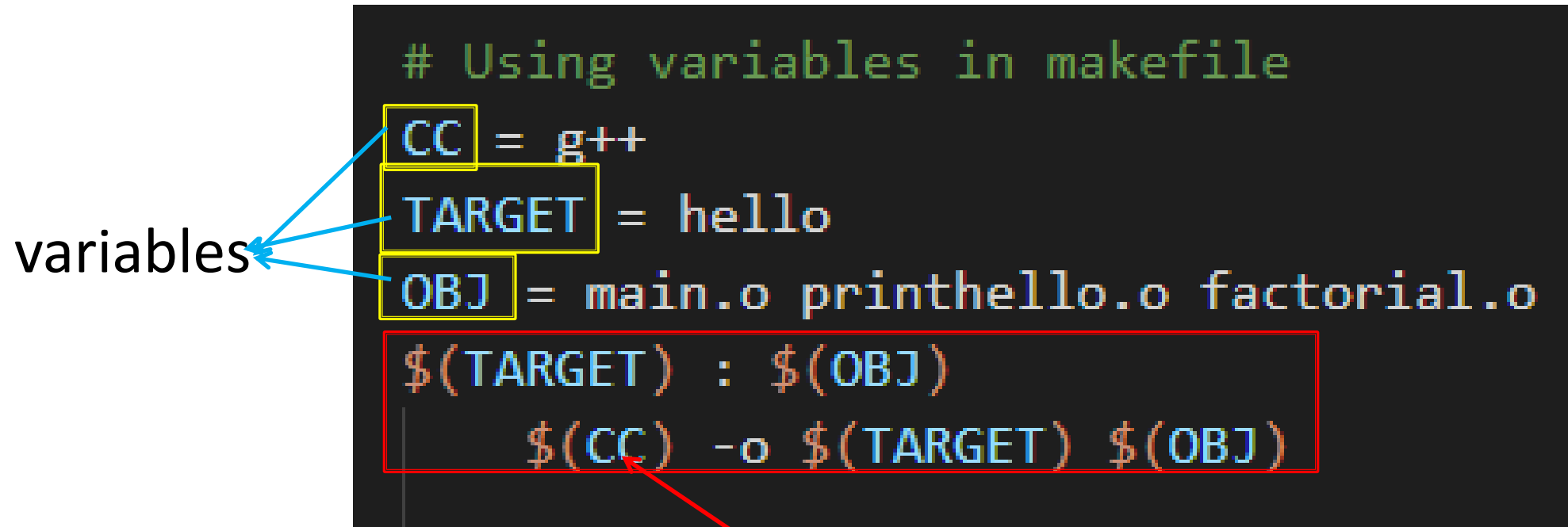
output

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-U1MO8N2F:/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-U1MO8N2F:/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 compiler warning
CFLAGS = -c -Wall

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

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

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++ -c -Wall main.cpp printhello.cpp factorial.cpp
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 compiler warning
CFLAGS = -c -Wall

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

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

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

```
maydlee@LAPTOP-U1M00N2F:/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-U1M00N2F:/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)

for example:  Replace all .cpp files with .o files

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

```
# 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, compiler 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)
```

此电脑 > 新加卷 (D:) > cstudy > testmakefile >

☐ 名称

inc

← All .h files are in inc

src

← All .cpp files are in src

makefile

```
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>

```

# 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 compiler 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-U1M00N2F:/mnt/d/cstudy/makefileby03$ 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.

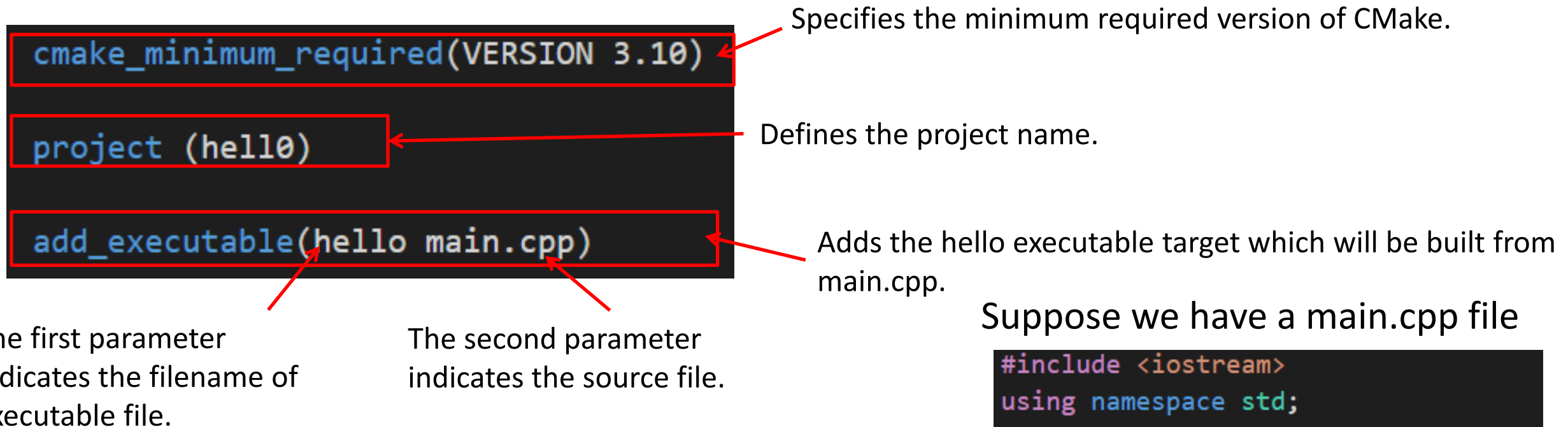
CMake needs **CMakeLists.txt** to run properly.

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.



The diagram shows a CMakeLists.txt file with three lines of code, each highlighted with a red box and annotated with a red arrow pointing to an explanatory text block.

```
cmake_minimum_required(VERSION 3.10)
project(hello)
add_executable(hello main.cpp)
```

- cmake_minimum_required(VERSION 3.10)**: Specifies the minimum required version of CMake.
- project(hello)**: Defines the project name.
- add_executable(hello main.cpp)**: Adds the hello executable target which will be built from main.cpp.

Additional annotations for the **add_executable** line:

- The first parameter indicates the filename of executable file.
- The second parameter indicates the source file.

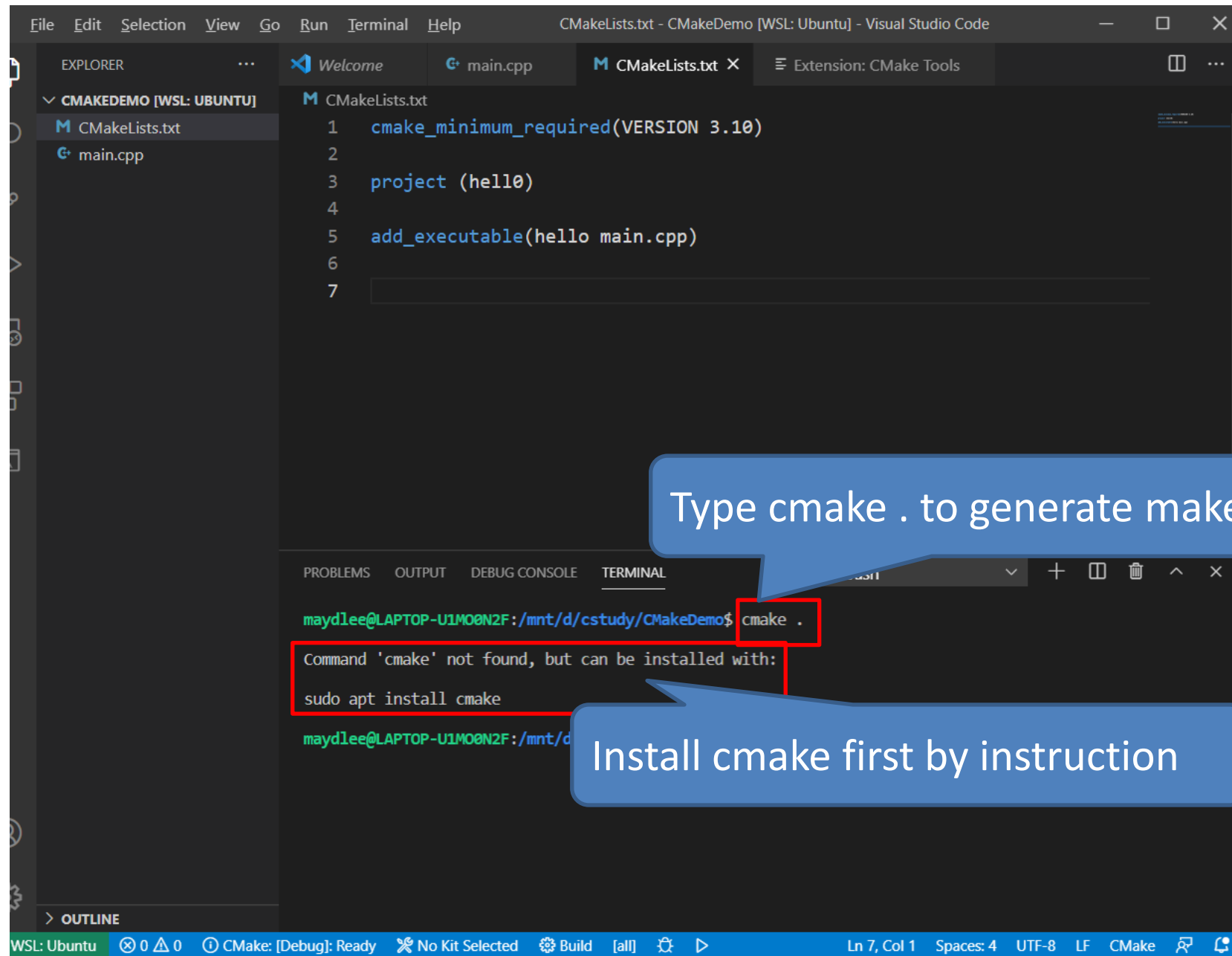
Store the CMakeLists.txt file in the same directory as the main.cpp.

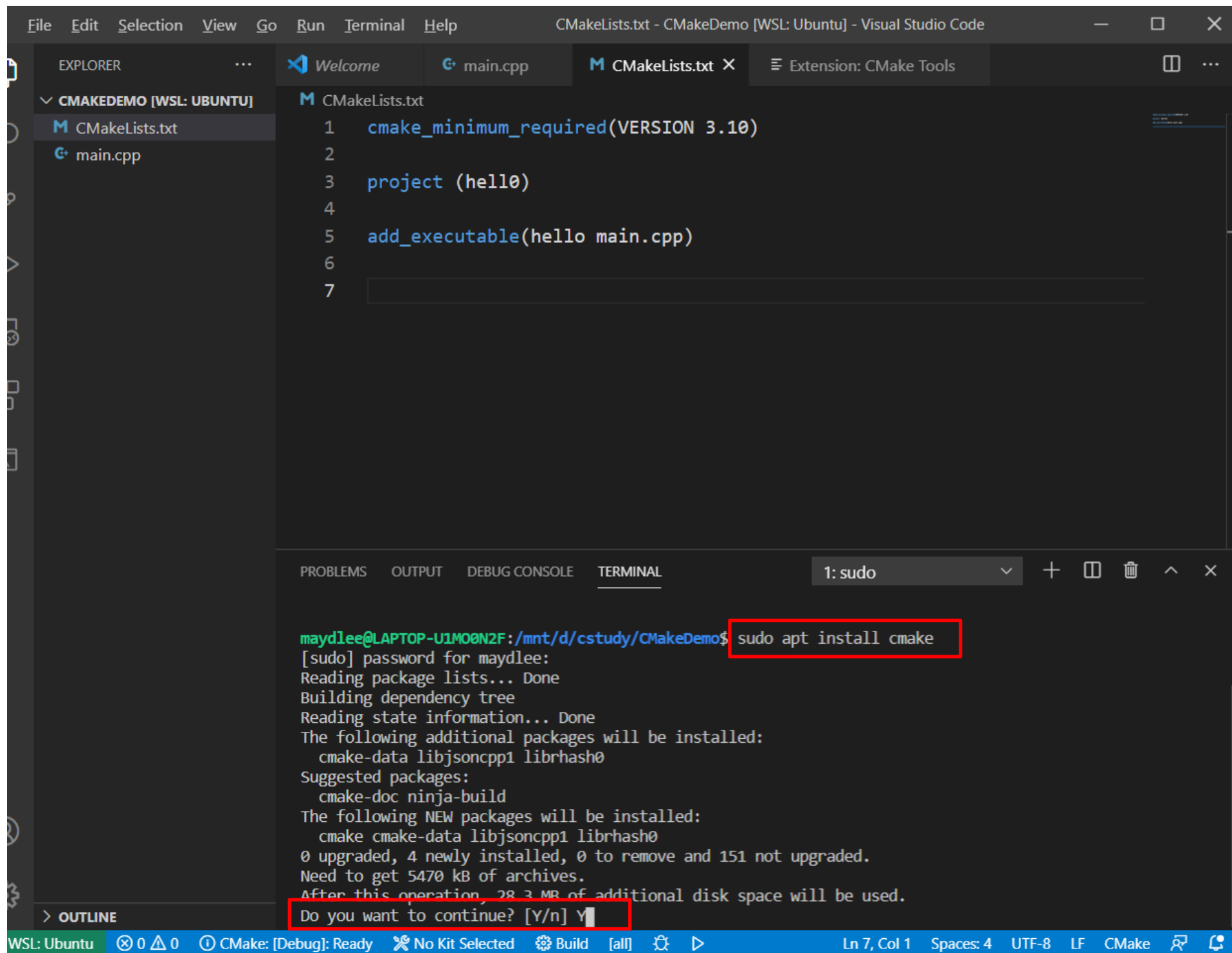
Suppose we have a main.cpp file

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

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

    return 0;
}
```





```
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/cstudy/CMakeDemo
```

Run cmake to generate makefile, . 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-U1M00N2F:/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-U1M00N2F:/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.

```
cmake_minimum_required(VERSION 3.10)

project(CmakeDemo2)

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

Put the function.cpp into the add_executable command.

./CmakeDemo2

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

```
maydlee@LAPTOP-U1M00N2F:/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-U1M00N2F:/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
```

All .cpp files are in the src directory

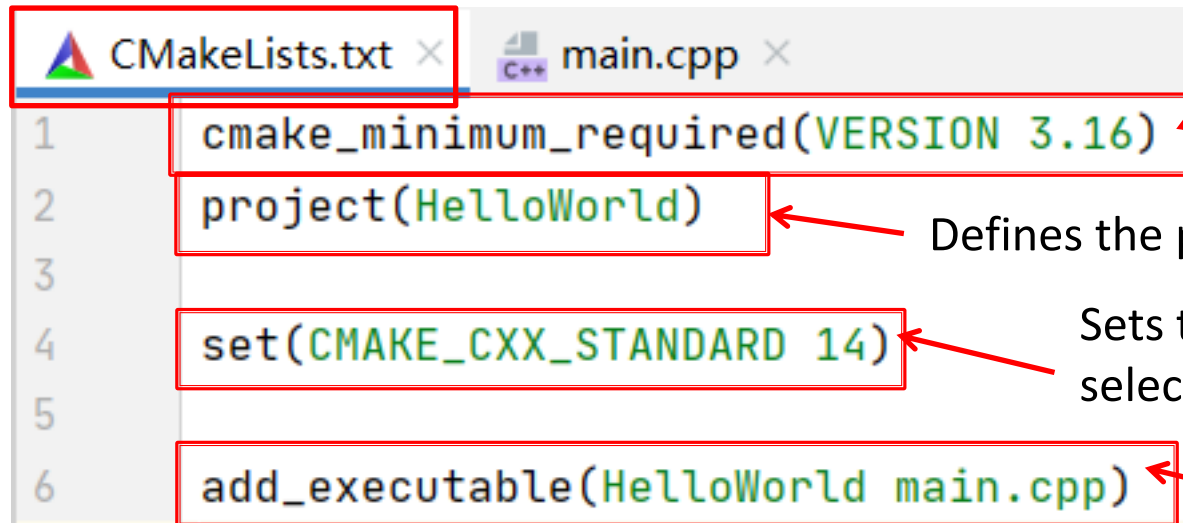
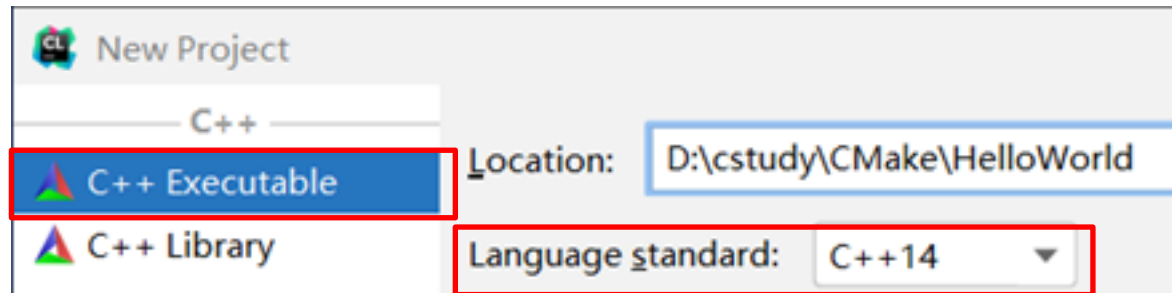
Include the header file which is stored in include directory.

We write CMakeLists.txt in CmakeDemo3 folder.

```
# 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})
```

```
maydlee@LAPTOP-U1M00N2F:/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-U1M00N2F:/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 CMakeLists.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>