C/C++ Program Design

LAB 8

CONTENTS

2 Knowledge Points

2.1 Makefile

2.2 CMake

2.1 Multiple-File Structure

Both C and C++ allow and even encourage you to locate the component functions of a program in separate files. You can compile the files separately and then link them into the final executable program. Using **make**, if you modify just one file, you can recompile just that one file and then link it to the previously compiled versions of the other files. This facility makes it easier to manage large programs.

You can divide the original program into **three parts**:

- **A header file** that contains the structure declarations and prototypes for functions that use those structures
- **A source code file** that contains the code for the structure-related functions
- **A source code file** that contains the code that calls the structure-related functions

Commonly, header file includes:

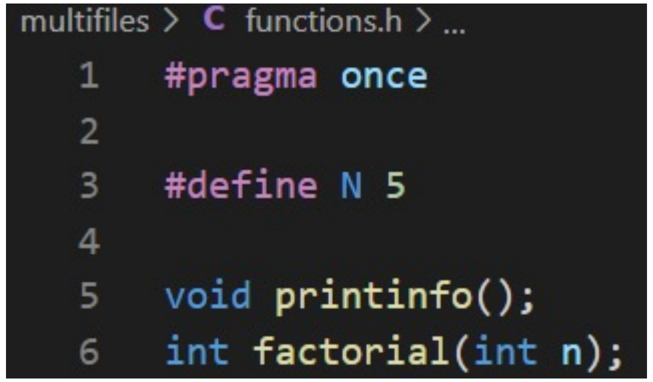
- Function prototype
- Symbolic constants define using **#define** or **const**
- Structure declarations
- Class declarations
- Template declarations
- Inline functions

2.2 Makefile

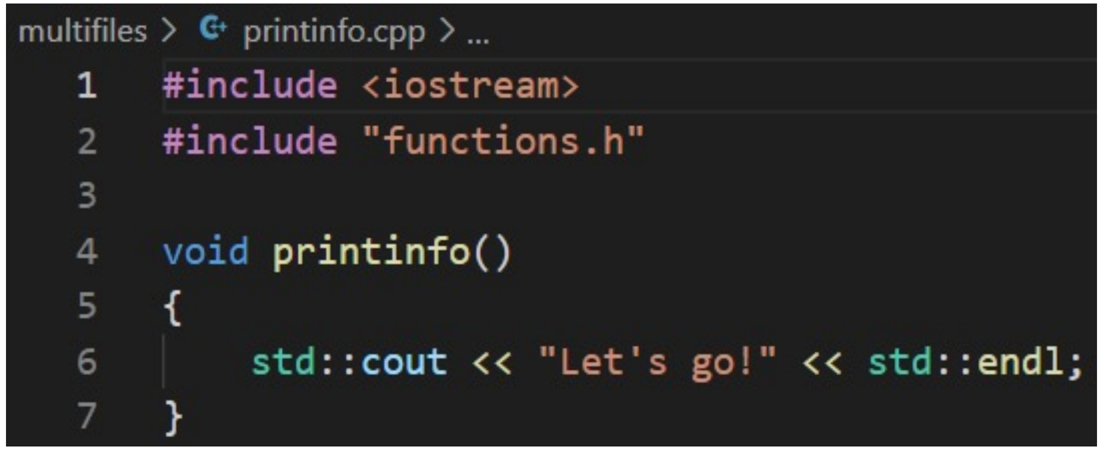
What is a makefile?

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.

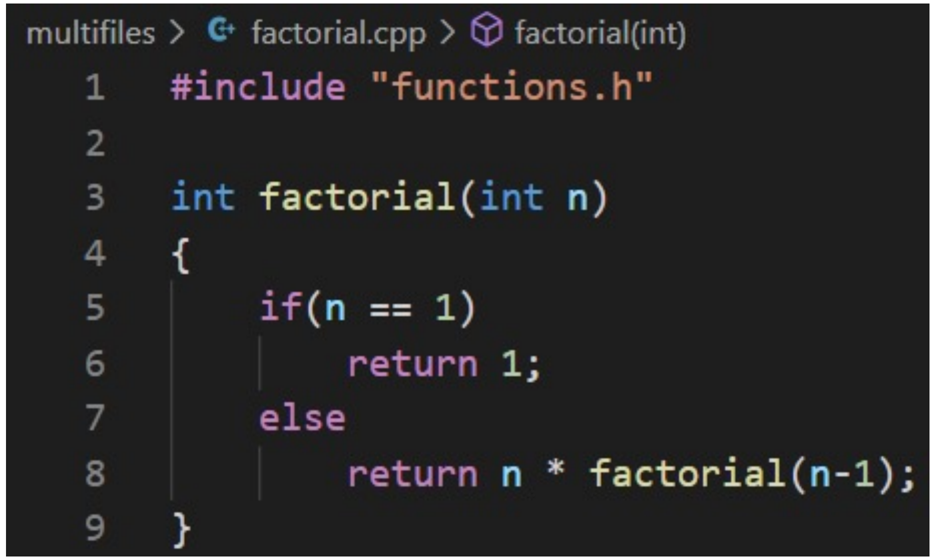
Suppose we have four source files as follows:


multifiles > functions.h > ...

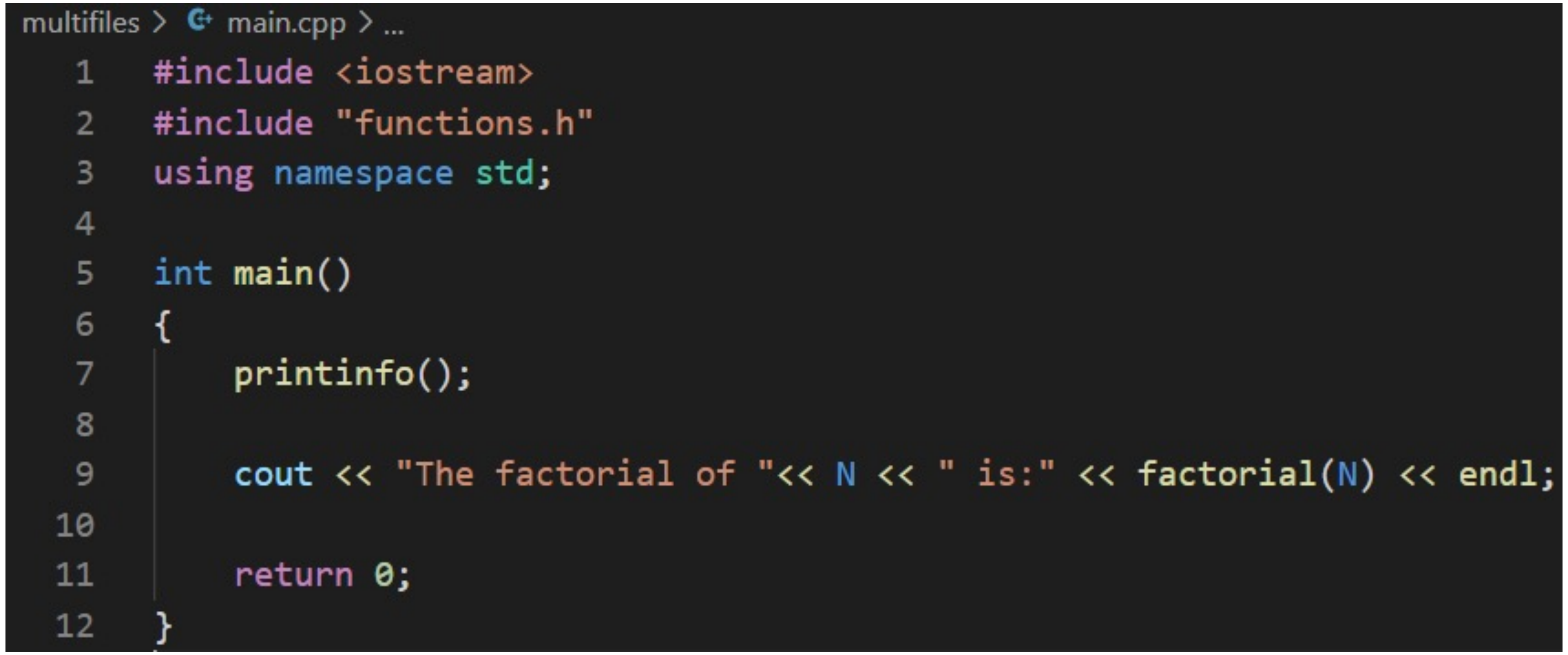
```cpp
#pragma once

#define N 5

void printinfo();
int factorial(int n);
```


multifiles > printinfo.cpp > ...

```cpp
#include <iostream>
#include "functions.h"

void printinfo()
{
    std::cout << "Let's go!" << std::endl;
}
```

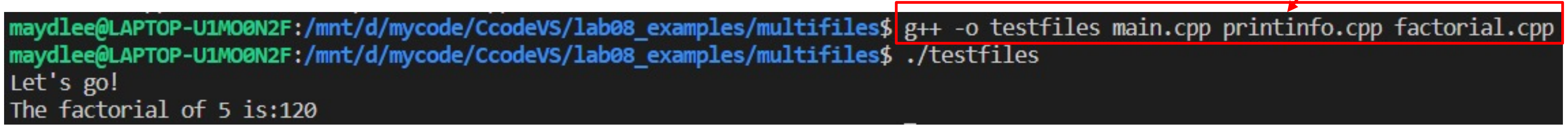

multifiles > factorial.cpp > factorial(int)

```cpp
#include "functions.h"

int factorial(int n)
{
    if(n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```


multifiles > main.cpp > ...

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

int main()
{
    printinfo();

    cout << "The factorial of "<< N << " is:" << factorial(N) << endl;

    return 0;
}
```

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



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ g++ -o testfiles main.cpp printinfo.cpp factorial.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ ./testfiles
Let's go!
The factorial of 5 is:120
```

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**.

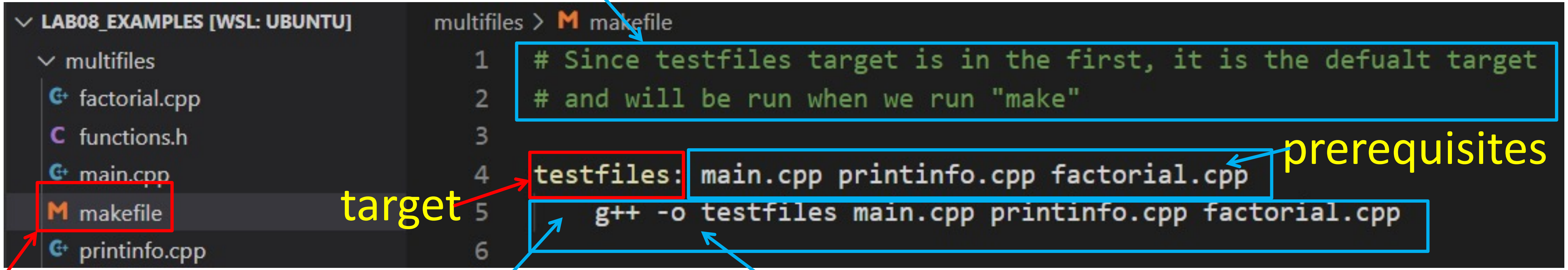
The name of makefile must be either **makefile** or **Makefile** without extension. You can write makefile in any text editor. A rule of makefile including three elements: **targets**, **prerequisites** and **commands**. There are many rules in the makefile.

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.

comments begins with #



```
# Since testfiles target is in the first, it is the defualt target
# and will be run when we run "make"

testfiles: main.cpp printinfo.cpp factorial.cpp
	g++ -o testfiles main.cpp printinfo.cpp factorial.cpp
```

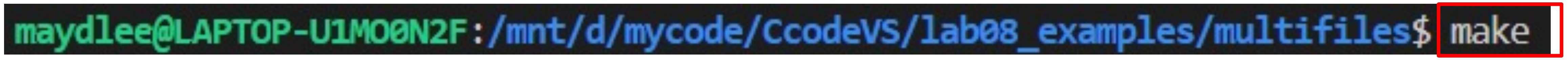
Start with <TAB>

Place the makefile together with your programs.

commands

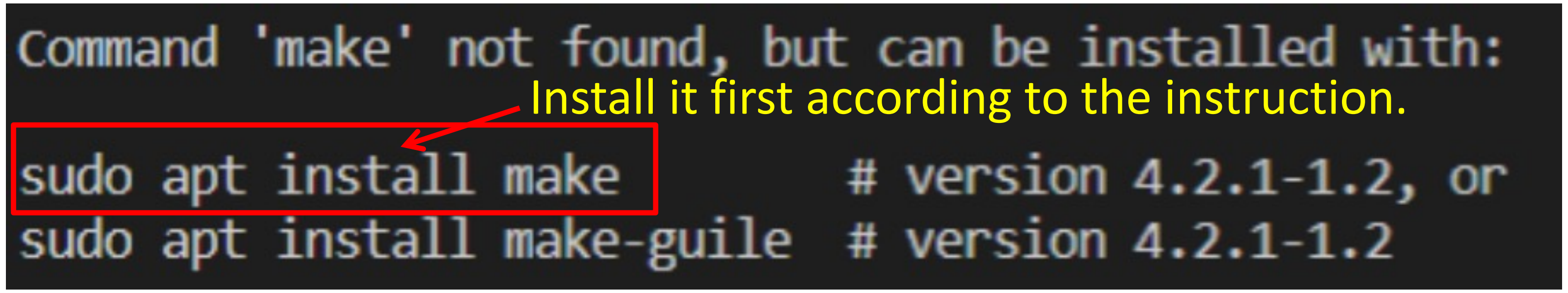
g++ is compiler name, -o is linker flag and testfiles is binary file name.

Type the command make in VScode

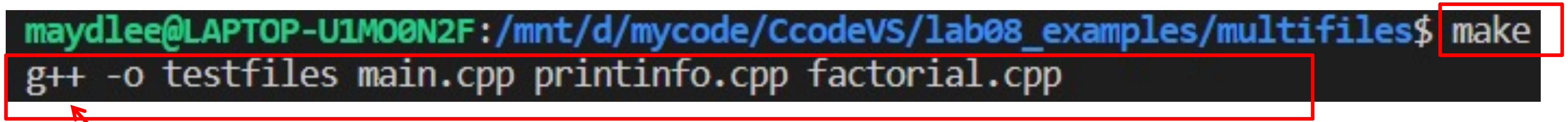


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
```

If you don’t install make in VScode, the information will display on the screen.



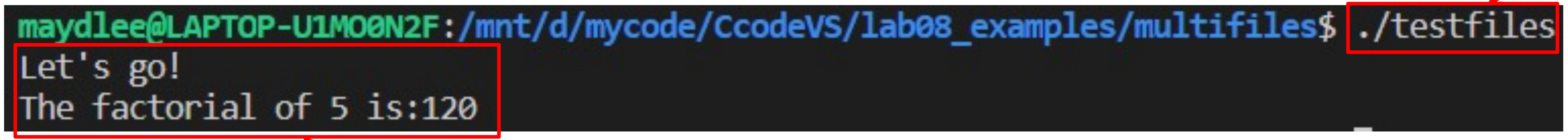
```
Command 'make' not found, but can be installed with:

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



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
g++ -o testfiles main.cpp printinfo.cpp factorial.cpp
```

Run the commands in the makefile automatically.

Run your program

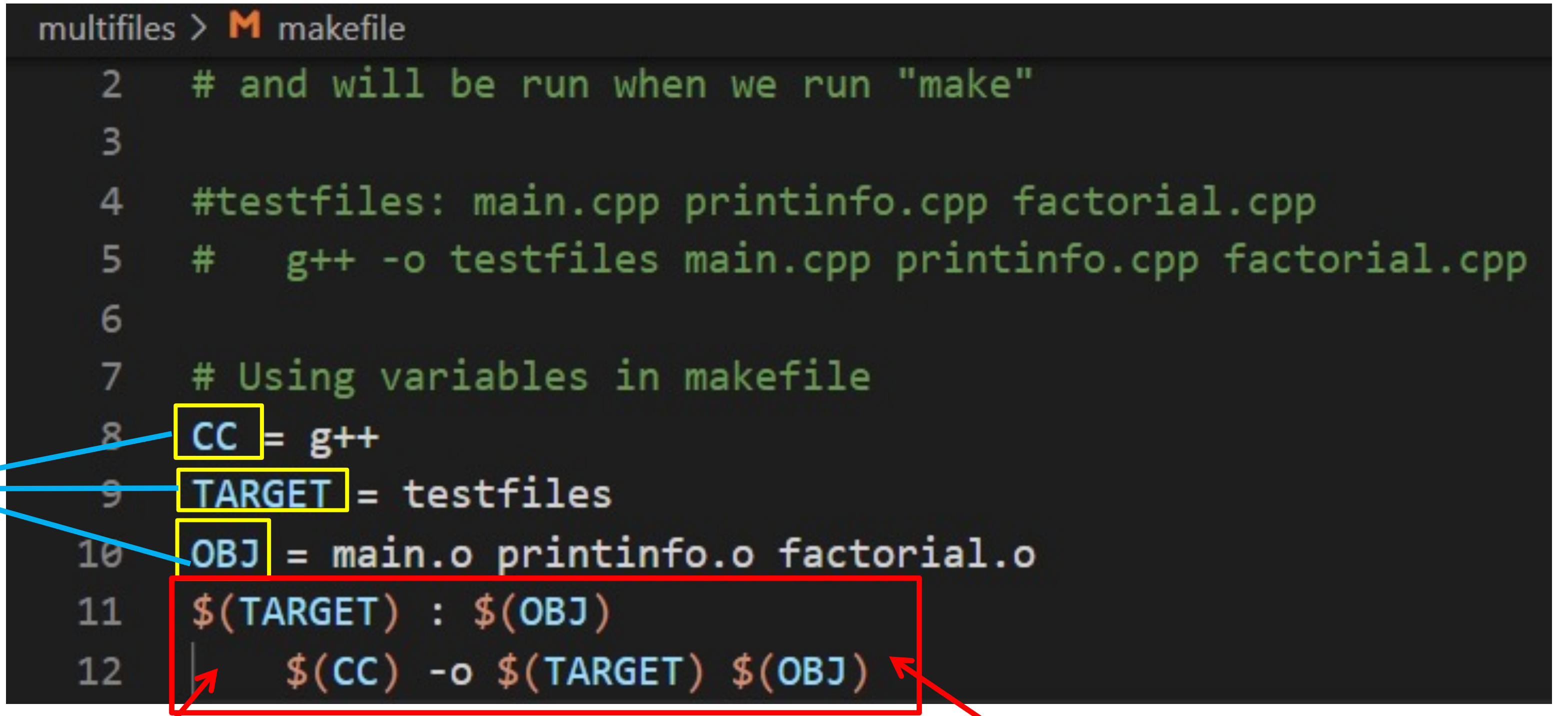


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ ./testfiles
Let's go!
The factorial of 5 is:120
```

output

Define Macros/Variables in the makefile

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

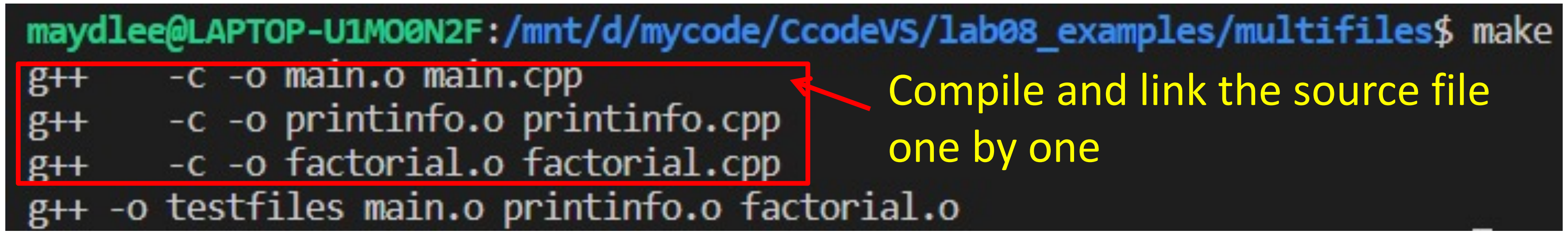


```
# and will be run when we run "make"

#testfiles: main.cpp printinfo.cpp factorial.cpp
#   g++ -o testfiles main.cpp printinfo.cpp factorial.cpp

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

variables

Start with <TAB>

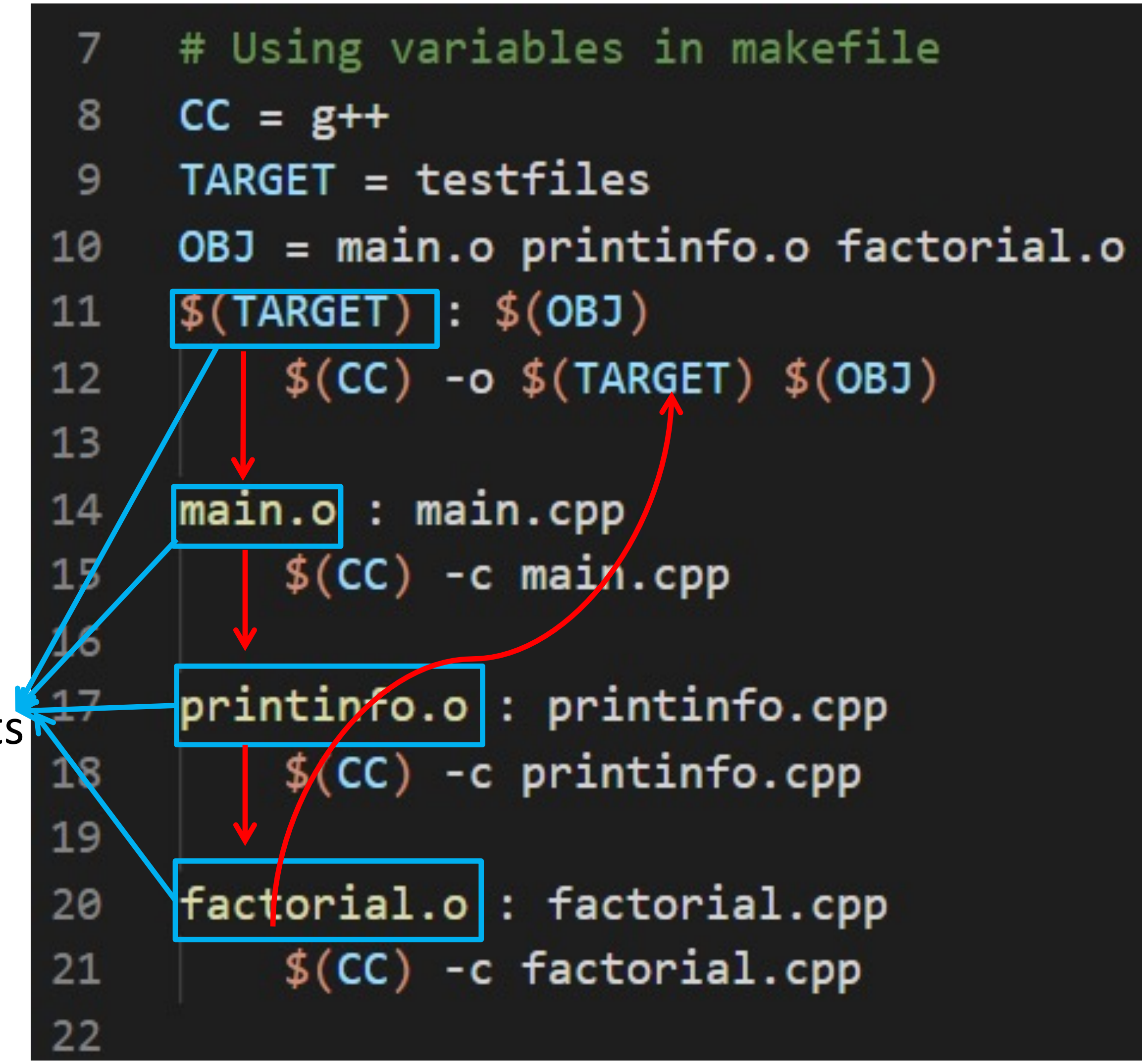
Write target, prerequisite and commands by variables using '$()'



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
g++     -c -o main.o main.cpp
g++     -c -o printinfo.o printinfo.cpp
g++     -c -o factorial.o factorial.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

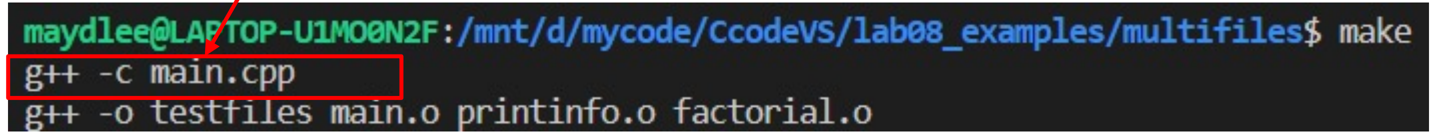
Compile and link the source file one by one

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



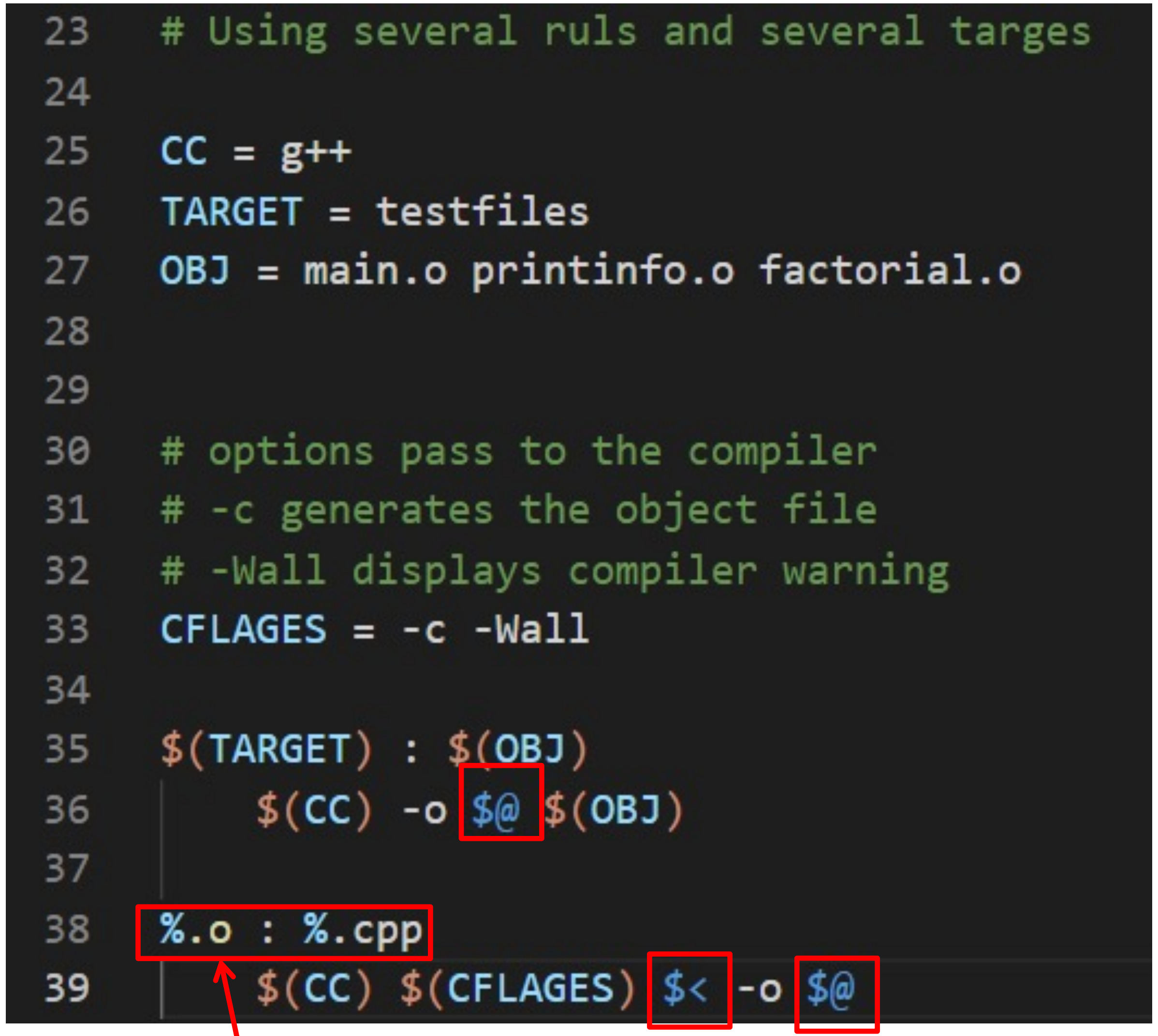
```
# Using variables in makefile
CC = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o
$(TARGET) : $(OBJ)
	$(CC) -o $(TARGET) $(OBJ)

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

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

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

```

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



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
g++ -c main.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

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



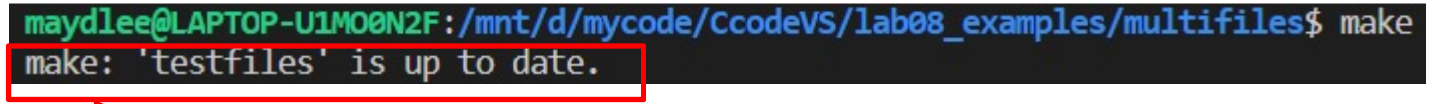
```
# Using several ruls and several targes

CC = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o


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

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

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

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

$@: Object Files

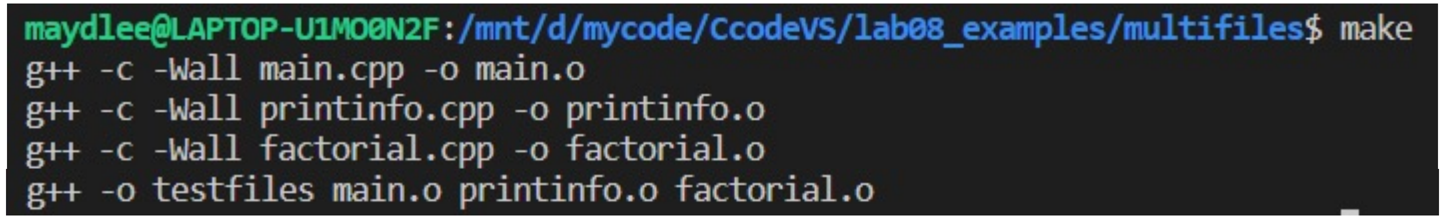
$^: all the prerequisites files

$<: the first prerequisite file



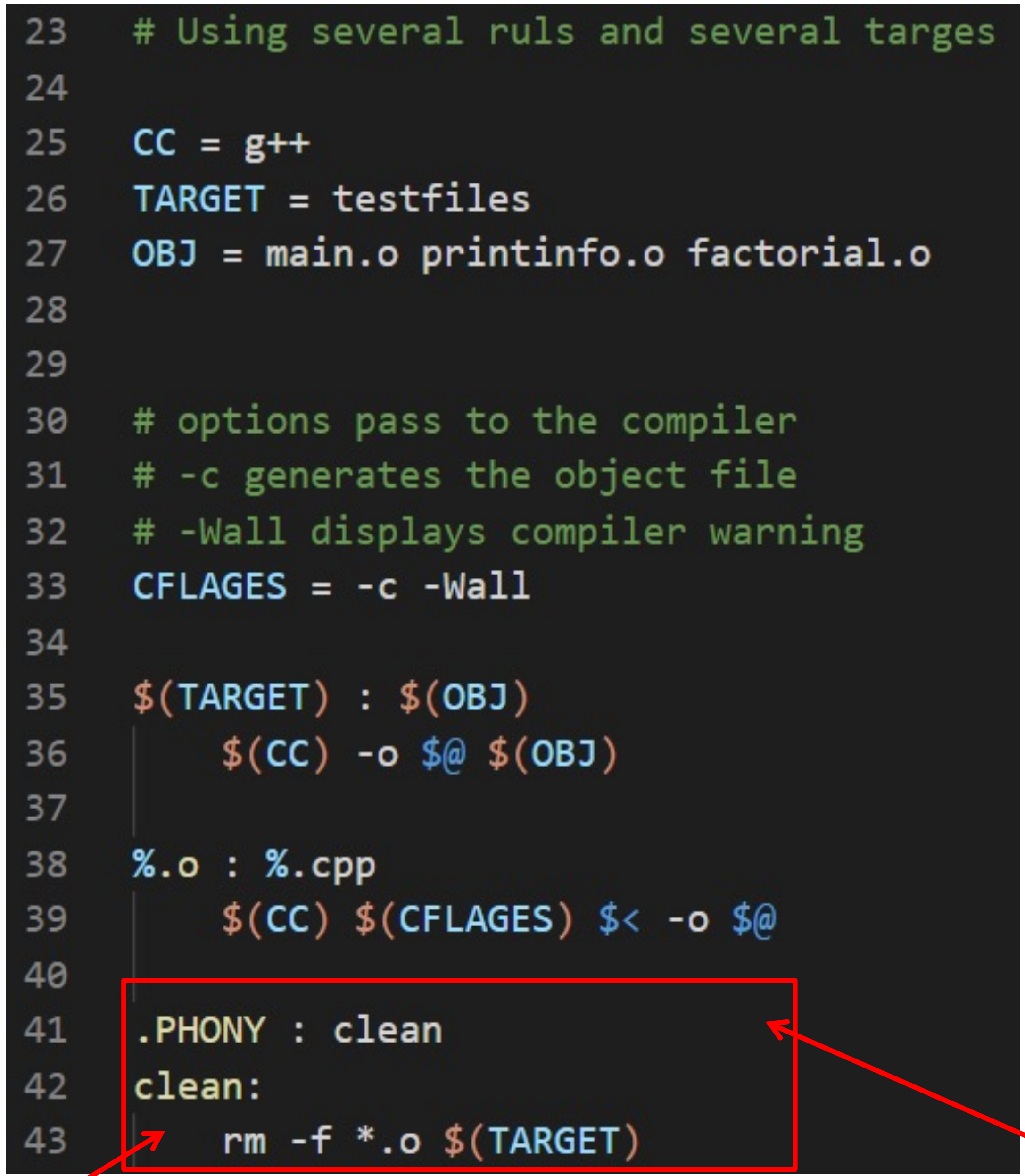
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
make: 'testfiles' is up to date.
```

This means your source files do not be update.
You should update the files and run make again.



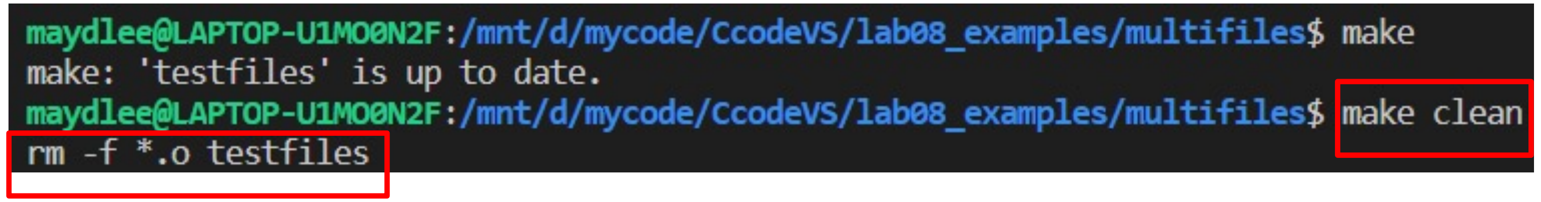
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
g++ -c -Wall main.cpp -o main.o
g++ -c -Wall printinfo.cpp -o printinfo.o
g++ -c -Wall factorial.cpp -o factorial.o
g++ -o testfiles main.o printinfo.o factorial.o
```

Using phony target to clean up compiled results automatically

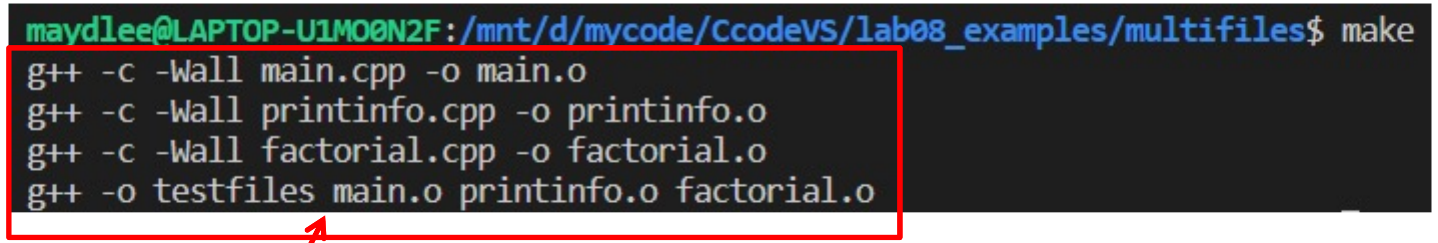


```
# Using several ruls and several targes

CC = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o


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

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

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

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

Start with <TAB>



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
make: 'testfiles' is up to date.
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make clean
rm -f *.o testfiles
```



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
g++ -c -Wall main.cpp -o main.o
g++ -c -Wall printinfo.cpp -o printinfo.o
g++ -c -Wall factorial.cpp -o factorial.o
g++ -o testfiles main.o printinfo.o factorial.o
```

After clean, you can run make again

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

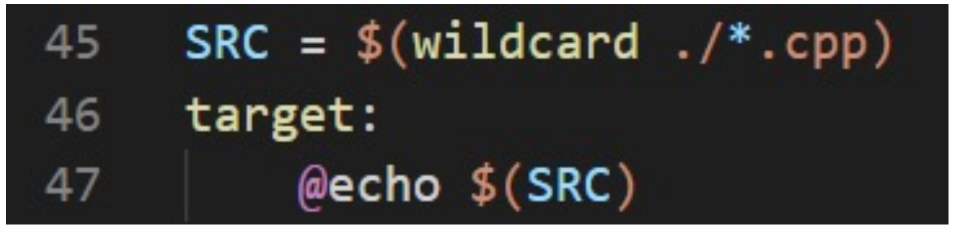
Functions in makefile

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

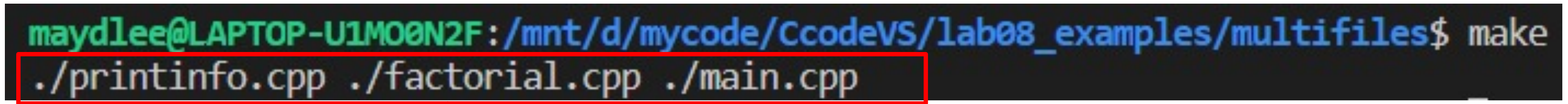
wildcard: search file

for example:

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



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



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
```

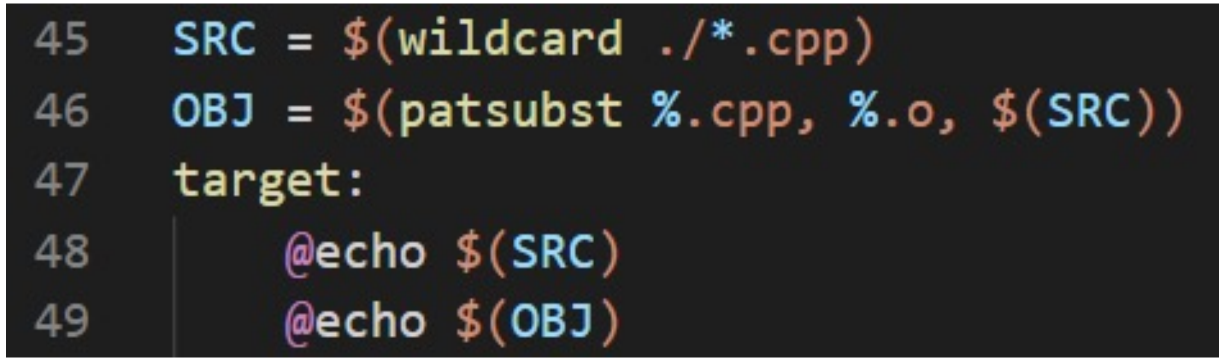
All .cpp files in the current directory

patsubst(pattern substitution): replace file
$(patsubst original pattern, target pattern, file list)

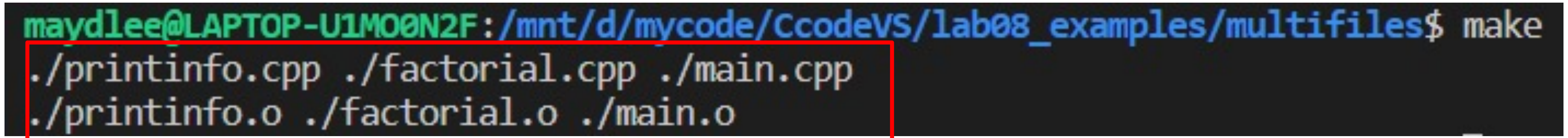
for example:

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)
```

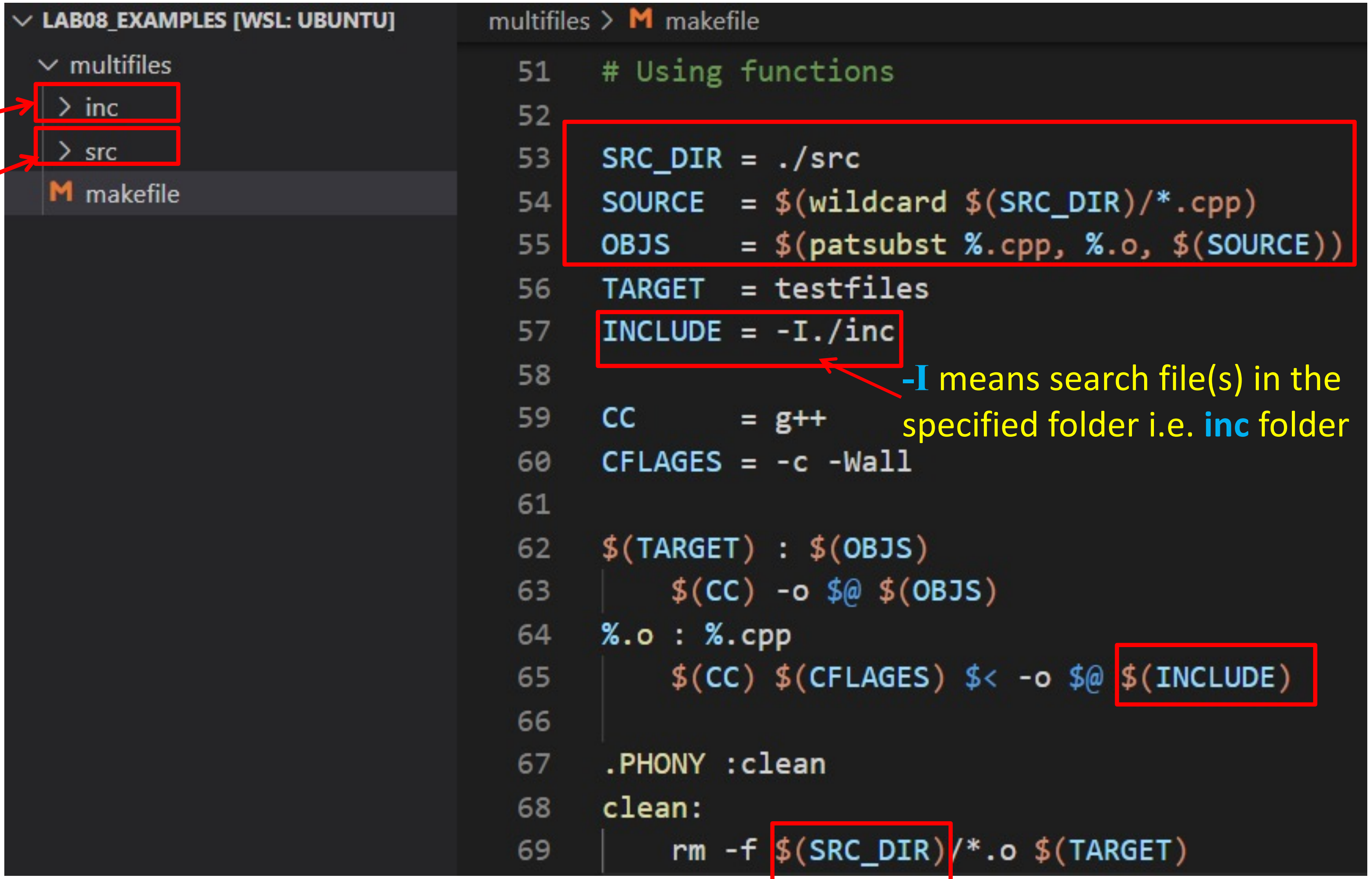


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
./printinfo.o ./factorial.o ./main.o
```

Replace all .cpp files with .o files

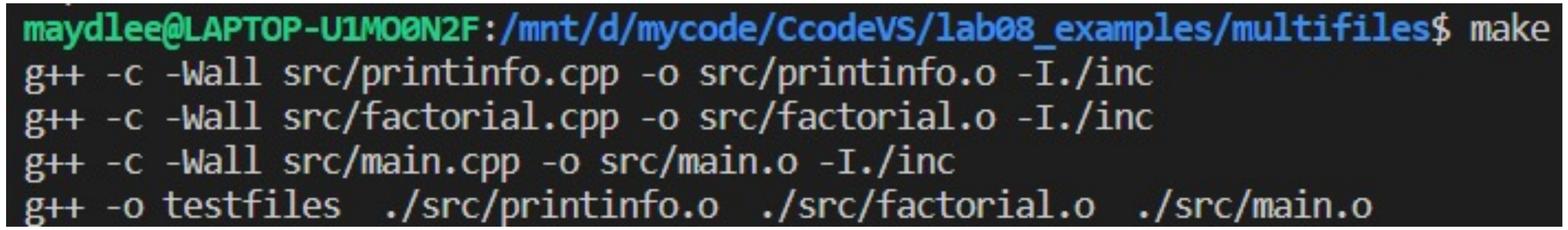
All .h files are in inc

All .cpp files are in src



```makefile
# Using functions

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

CC      = g++
CFLAGES = -c -Wall

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

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

-I means search file(s) in the specified folder i.e. **inc** folder



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/multifiles$ make
g++ -c -Wall src/printinfo.cpp -o src/printinfo.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 testfiles  ./src/printinfo.o  ./src/factorial.o  ./src/main.o
```

GNU Make Manual
http://www.gnu.org/software/make/manual/make.html

Use Options to Control Optimization

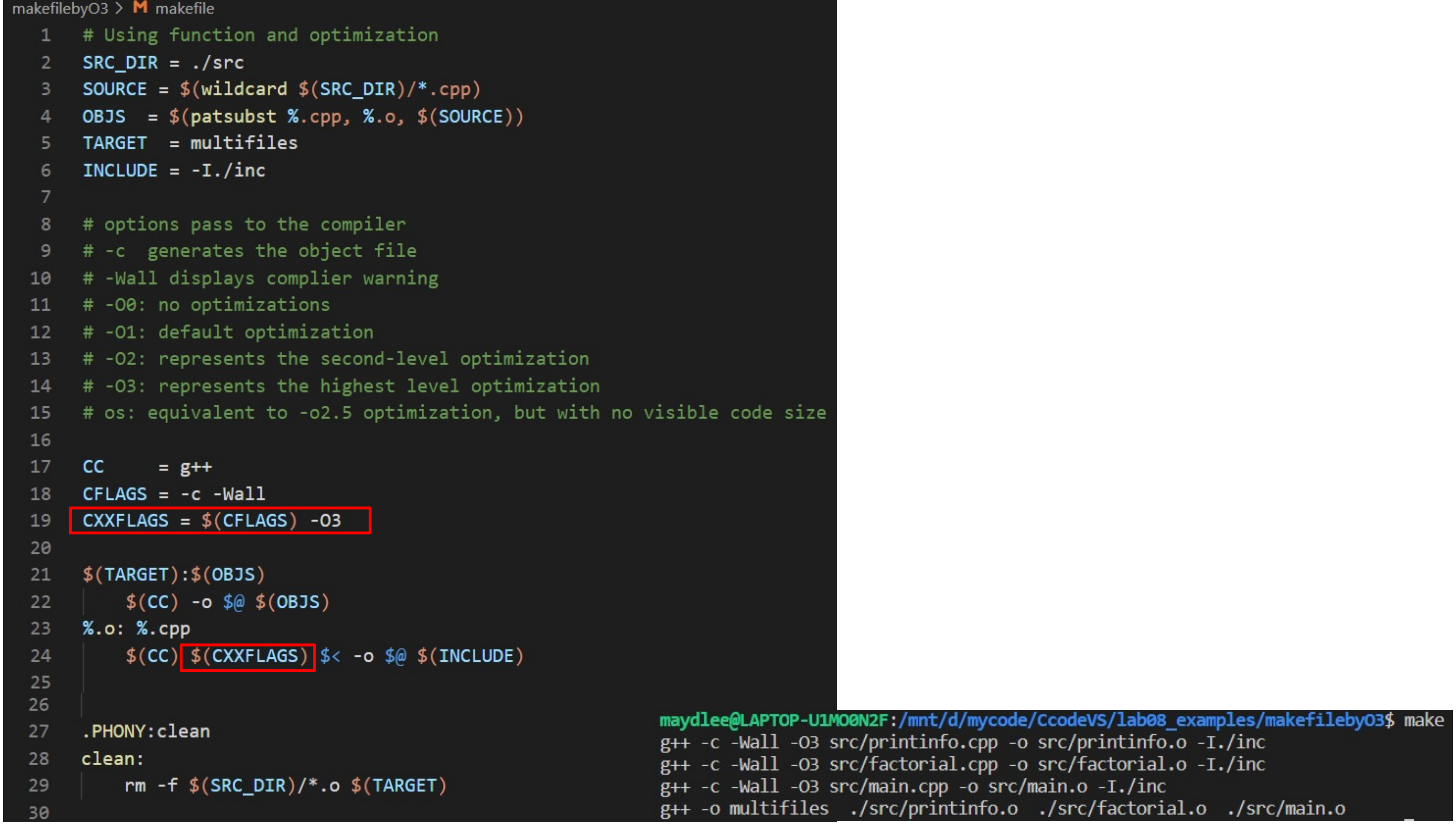
-O1, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

-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



makefilebyO3 > makefile

```makefile
# Using function and optimization
SRC_DIR = ./src
SOURCE = $(wildcard $(SRC_DIR)/*.cpp)
OBJS  = $(patsubst %.cpp, %.o, $(SOURCE))
TARGET  = multifiles
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/mycode/CcodeVS/lab08_examples/makefilebyO3$ make
g++ -c -Wall -O3 src/printinfo.cpp -o src/printinfo.o -I./inc
g++ -c -Wall -O3 src/factorial.cpp -o src/factorial.o -I./inc
g++ -c -Wall -O3 src/main.cpp -o src/main.o -I./inc
g++ -o multifiles  ./src/printinfo.o  ./src/factorial.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**.

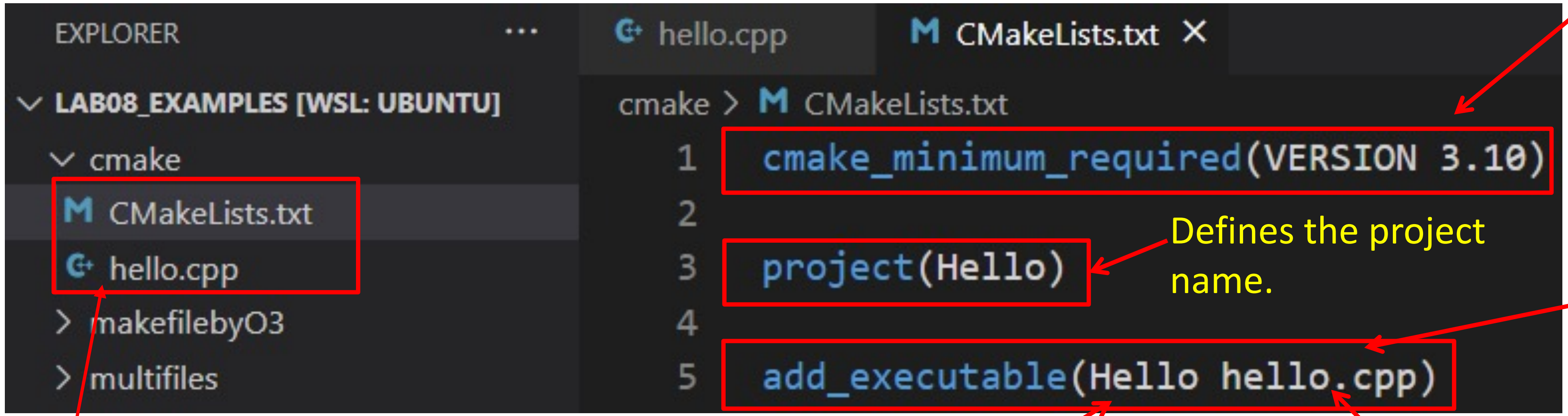
- The **commands** include command name, brackets and parameters ,

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(Hello)

add_executable(Hello hello.cpp)
```

Specifies the minimum required version of CMake. Use **cmake --version** in Vscode terminal window to check the cmake version in your computer.

Defines the project name.

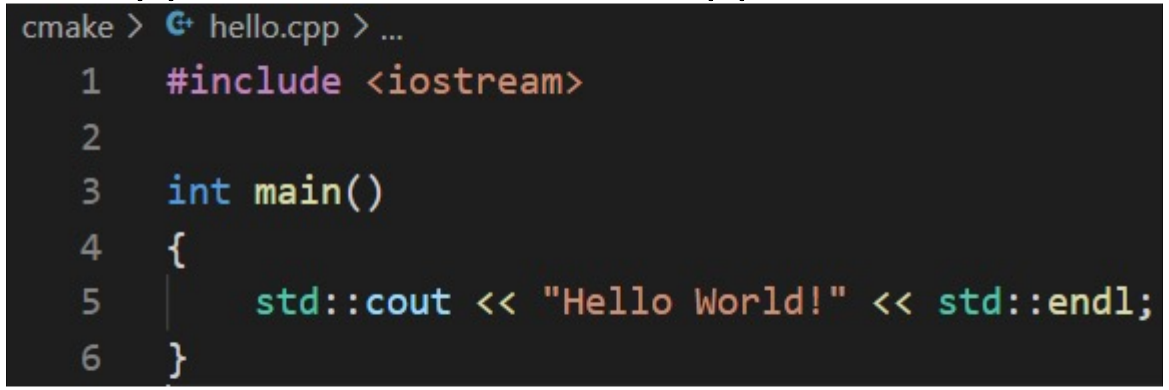
Adds the Hello executable target which will be built from hello.cpp.

The first parameter indicates the filename of executable file.

The second parameter indicates the source file.

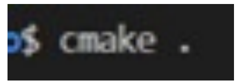
Store the CMakeLists.txt file in the same directory as the hello.cpp.

Suppose we have a hello.cpp file

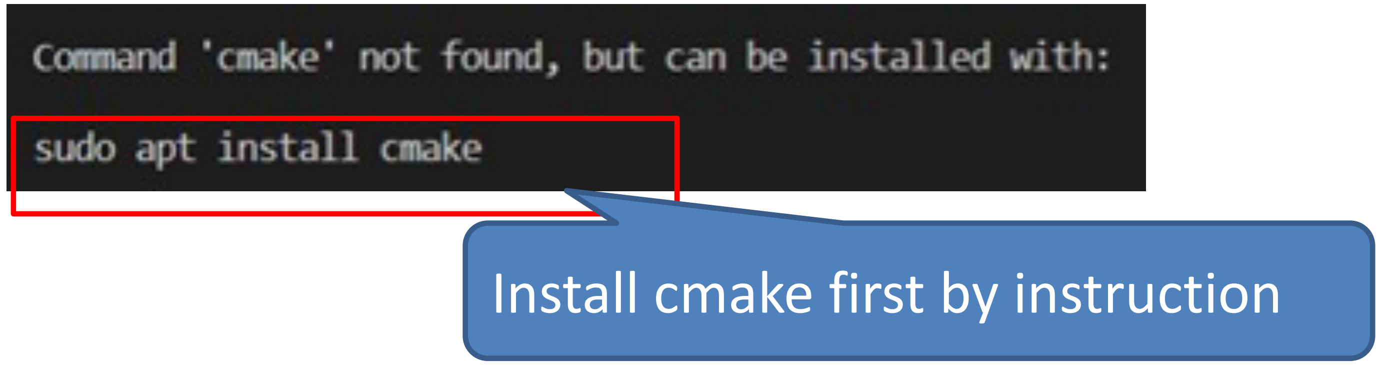


```cpp
#include <iostream>

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

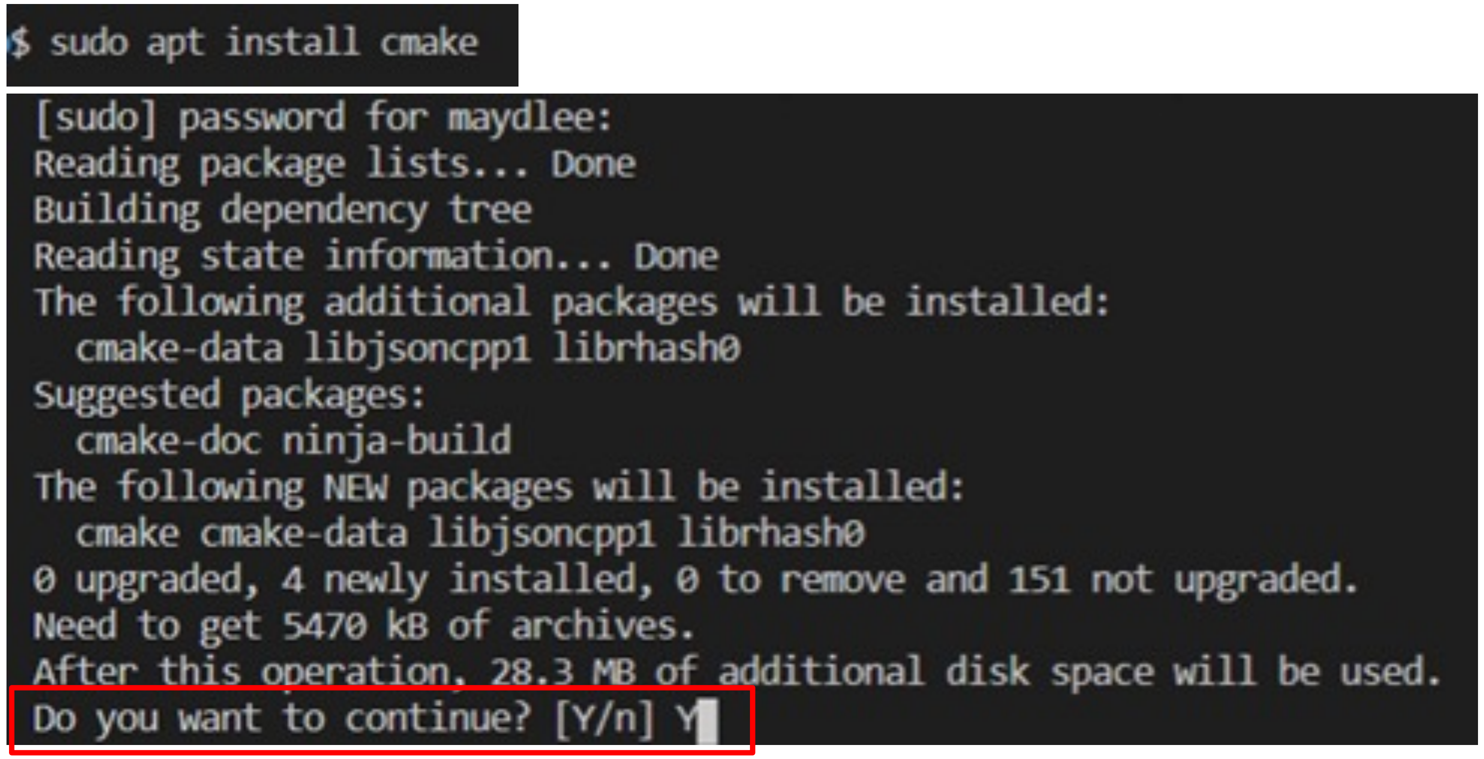
In current directory, type **cmake .** to generate makefile. If cmake does not be installed, follow the instruction to intall cmake.



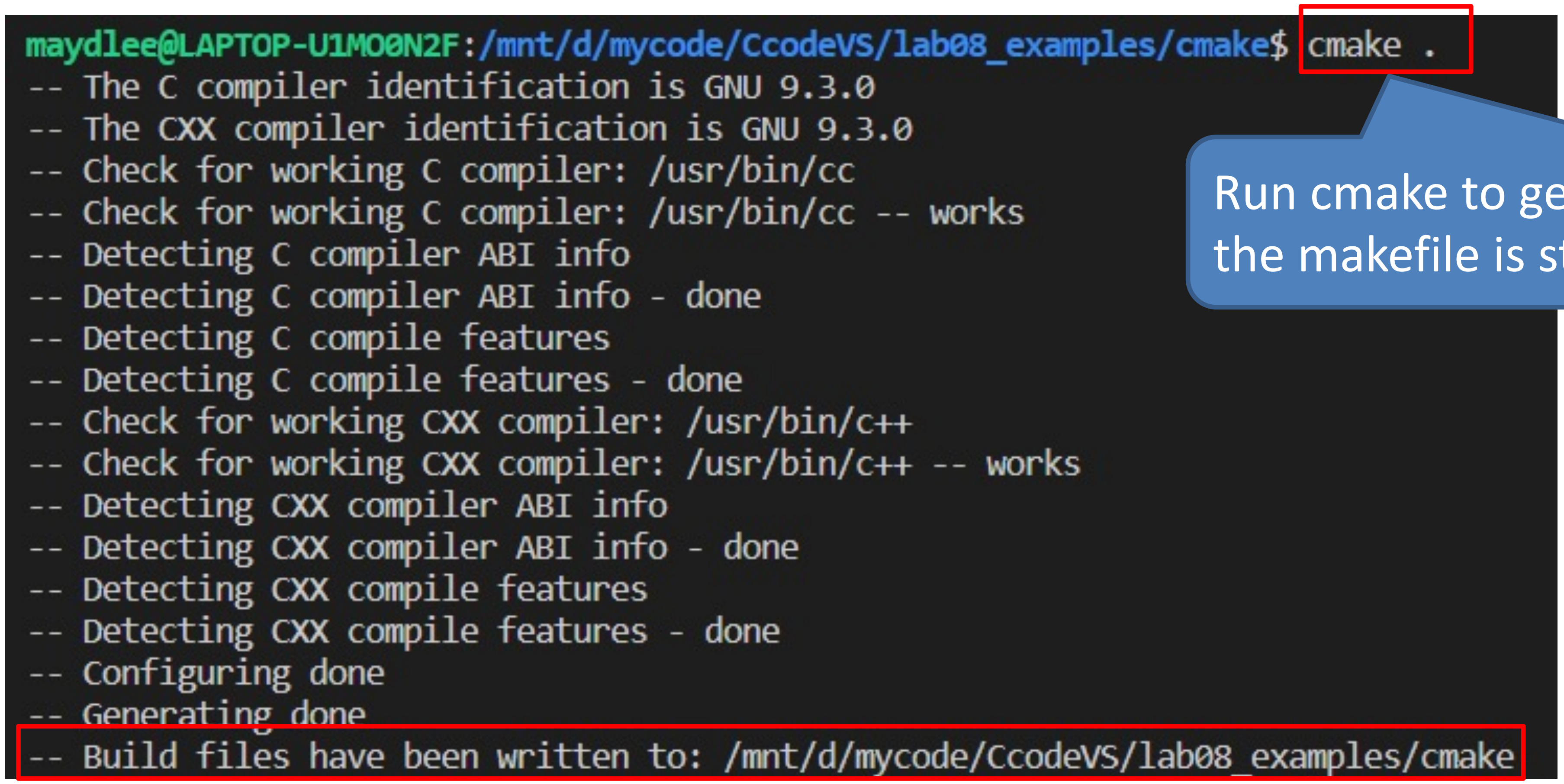
```
$ cmake .
```



```
Command 'cmake' not found, but can be installed with:

sudo apt install cmake
```



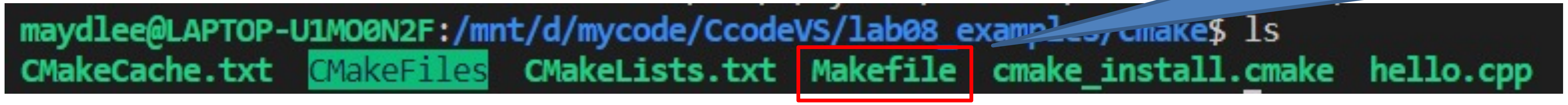
```
$ sudo apt install cmake
[sudo] password for maydlee:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  cmake-data libjsoncpp1 librhash0
Suggested packages:
  cmake-doc ninja-build
The following NEW packages 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
```



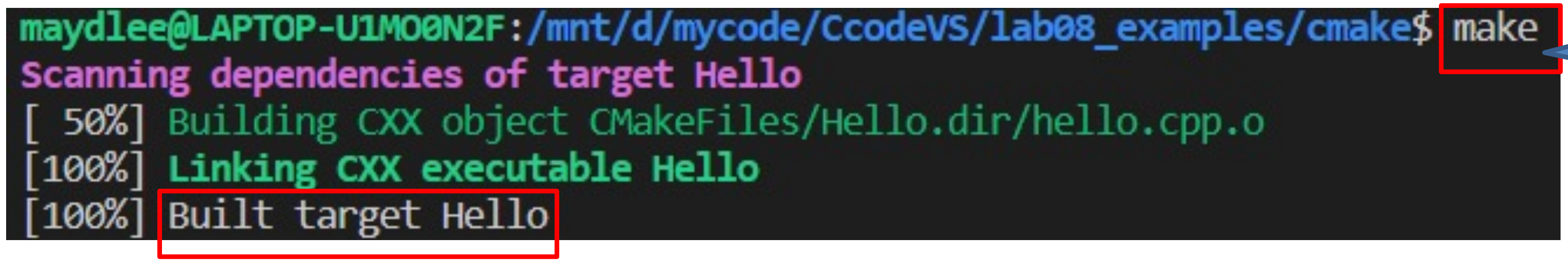
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/cmake$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI 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/mycode/CcodeVS/lab08_examples/cmake
```

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.

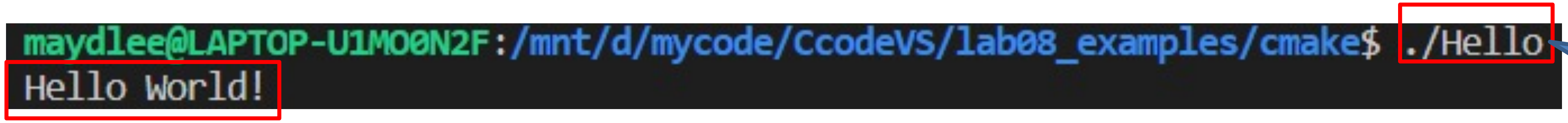


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/cmake$ ls
CMakeCache.txt  CMakeFiles  CMakeLists.txt  Makefile  cmake_install.cmake  hello.cpp
```



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/cmake$ make
Scanning dependencies of target Hello
[ 50%] Building CXX object CMakeFiles/Hello.dir/hello.cpp.o
[100%] Linking CXX executable Hello
[100%] Built target Hello
```

Execute make to compile the program.



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/cmake$ ./Hello
Hello World!
```

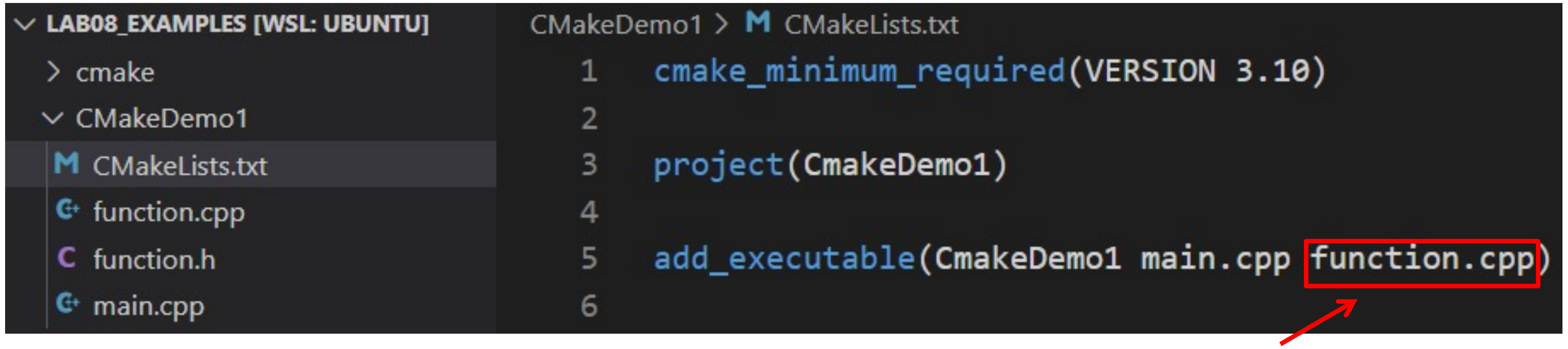
Run the program

2. Multi-source files in a project

There are three files in the same directory.

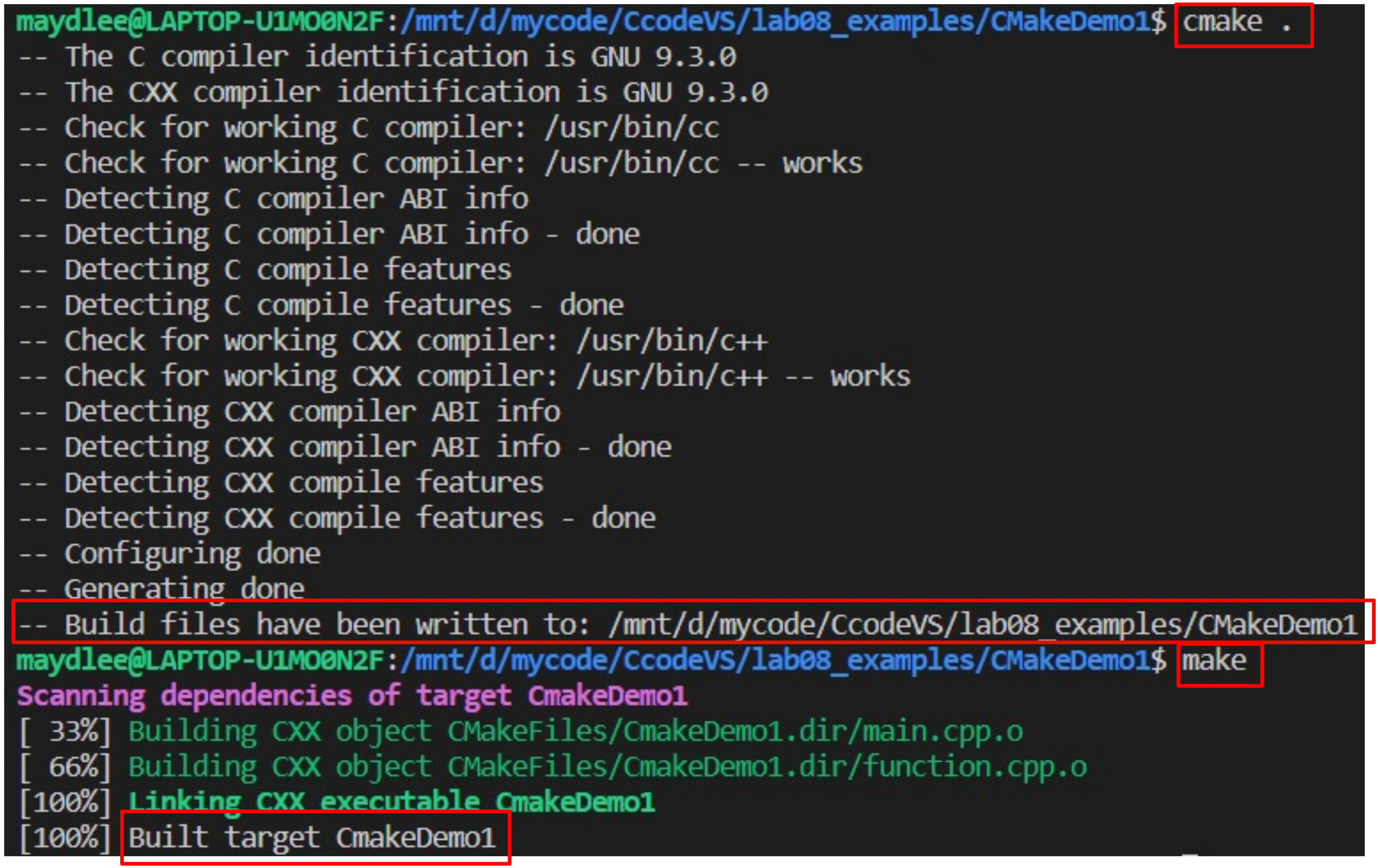
./CmakeDemo2

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



```
cmake_minimum_required(VERSION 3.10)

project(CmakeDemo1)

add_executable(CmakeDemo1 main.cpp function.cpp)
```

Put the function.cpp into the add_executable command.



```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/CMakeDemo1$ 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/mycode/CcodeVS/lab08_examples/CMakeDemo1
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/CMakeDemo1$ make
Scanning dependencies of target CmakeDemo1
[ 33%] Building CXX object CMakeFiles/CmakeDemo1.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/CmakeDemo1.dir/function.cpp.o
[100%] Linking CXX executable CmakeDemo1
[100%] Built target CmakeDemo1
```

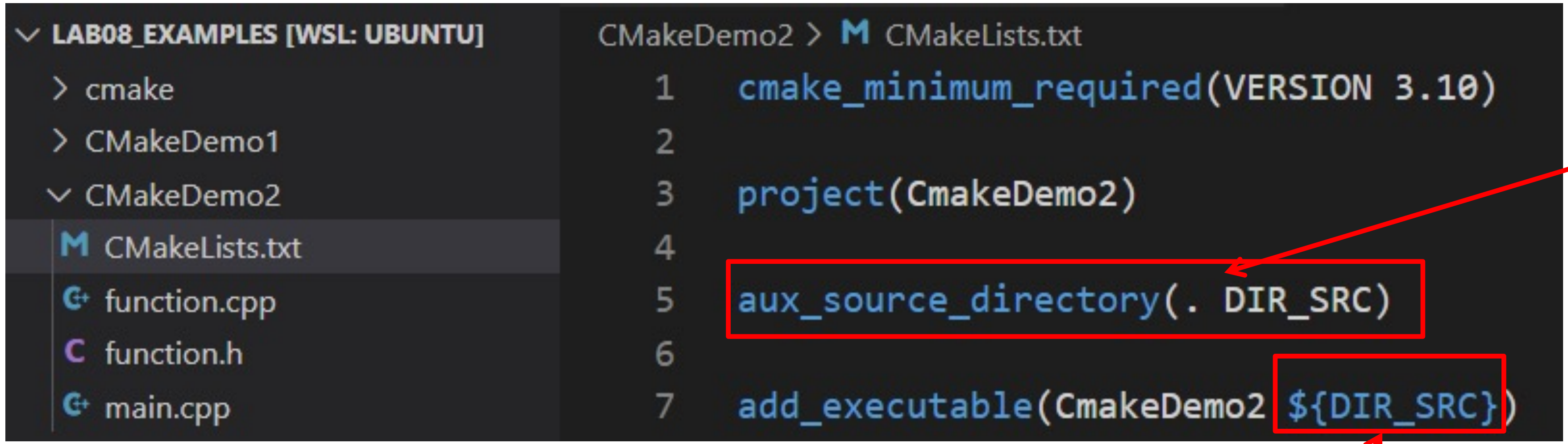
2. Multi-source files in a project

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>.

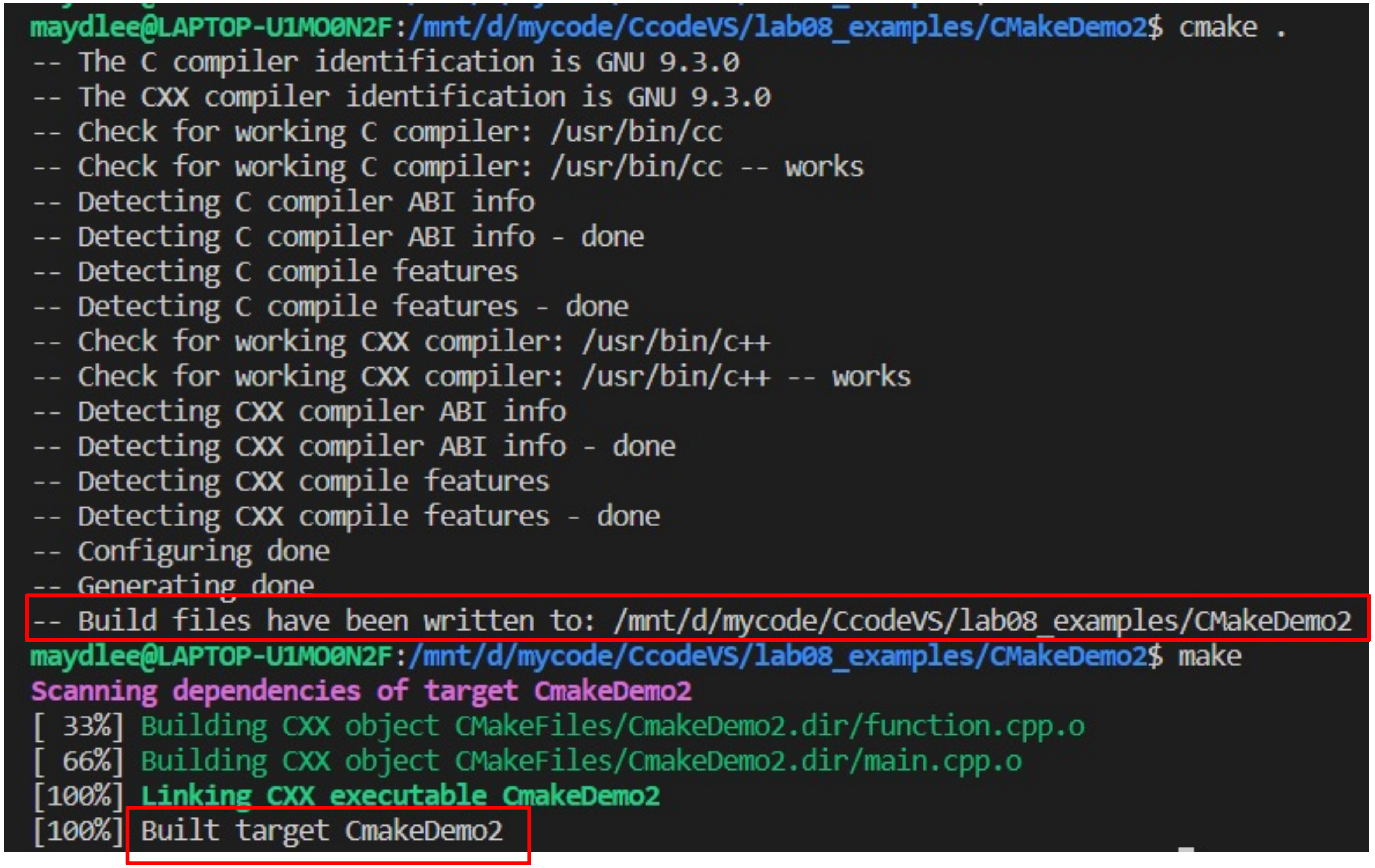
2. Multi-source files in a project



```cmake
cmake_minimum_required(VERSION 3.10)

project(CmakeDemo2)

aux_source_directory(. DIR_SRC)

add_executable(CmakeDemo2 ${DIR_SRC})
```

Store all files in the current directory into DIR_SRCS.

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



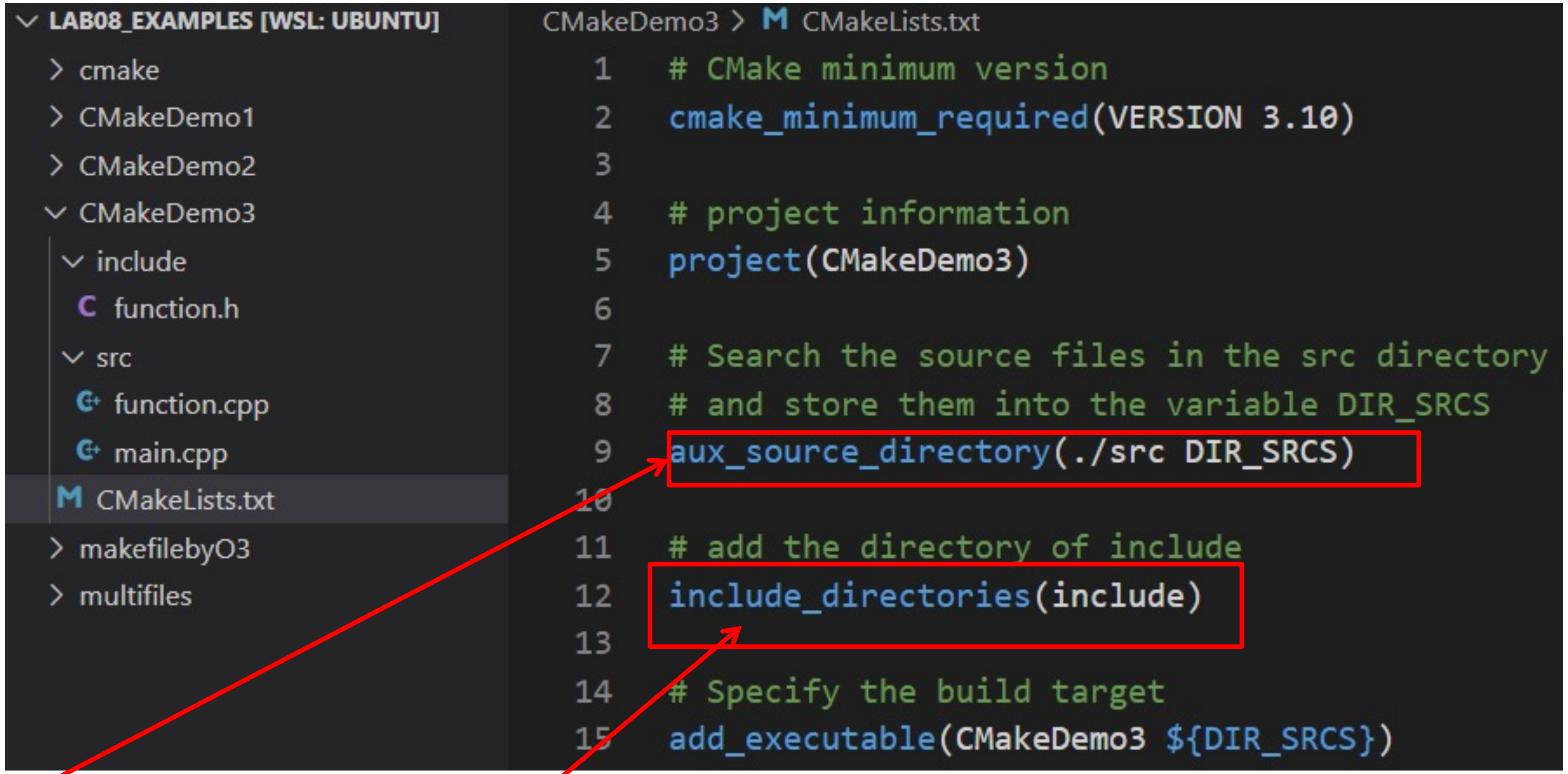
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/CMakeDemo2$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler 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/mycode/CcodeVS/lab08_examples/CMakeDemo2
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/CMakeDemo2$ make
Scanning dependencies of target CmakeDemo2
[ 33%] Building CXX object CMakeFiles/CmakeDemo2.dir/function.cpp.o
[ 66%] Building CXX object CMakeFiles/CmakeDemo2.dir/main.cpp.o
[100%] Linking CXX executable CmakeDemo2
[100%] Built target CmakeDemo2
```

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

./CMakeDemo3

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

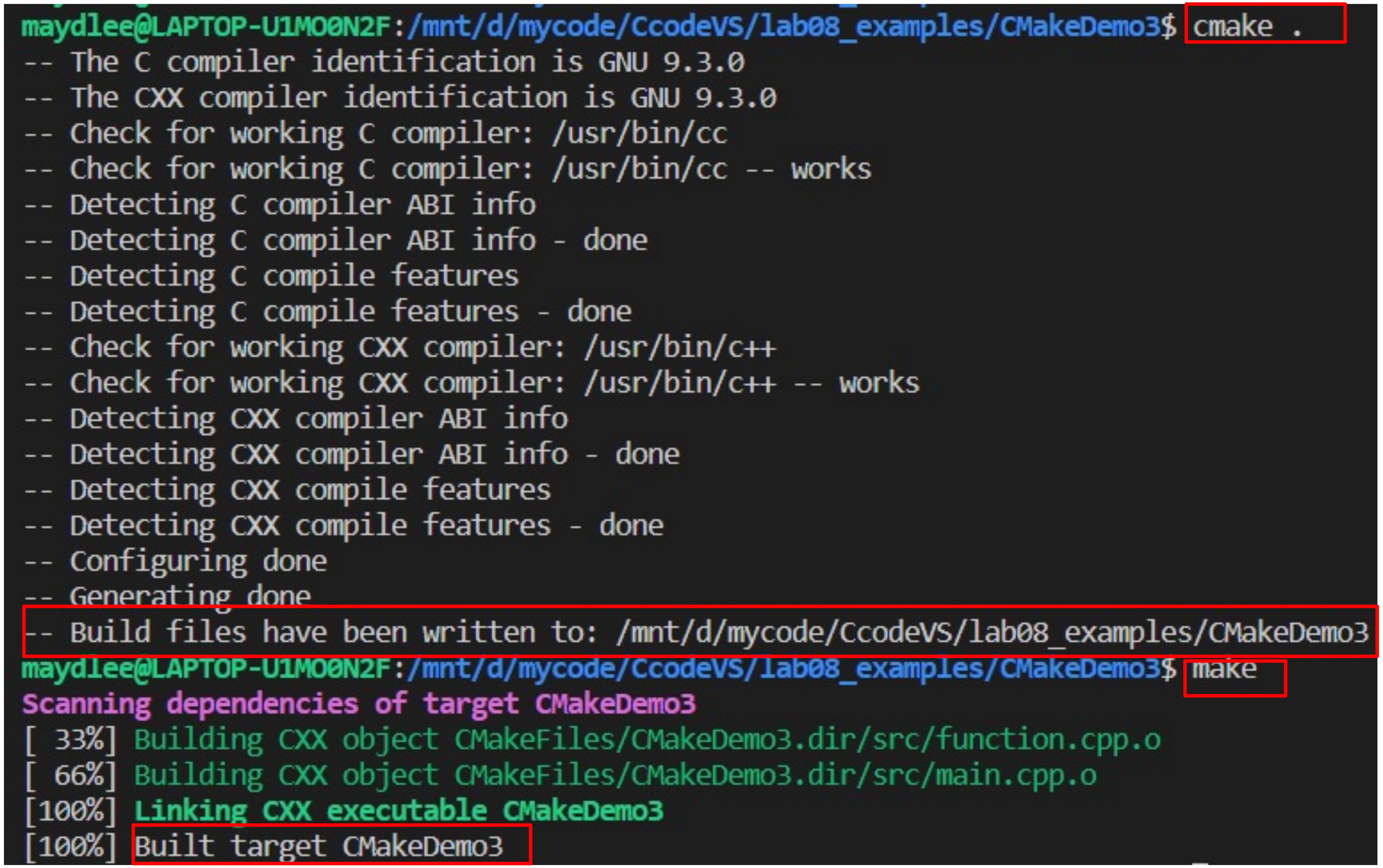
We write CMakeLists.txt in CmakeDemo3 folder.



```
# CMake 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})
```

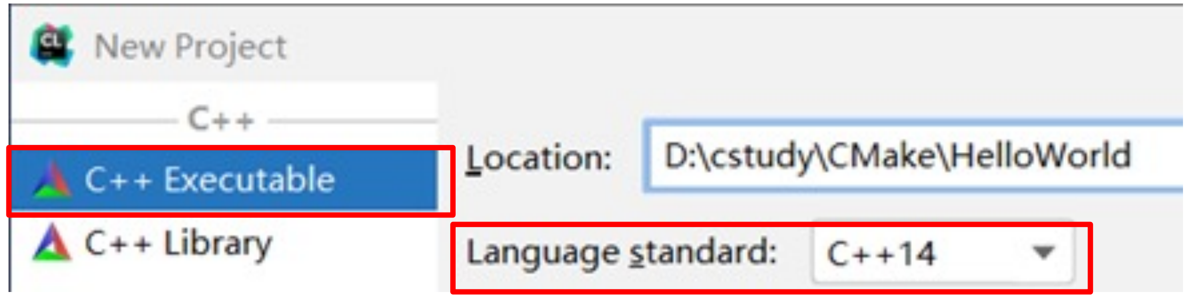
All .cpp files are in the src directory

Include the header file which is stored in include directory.



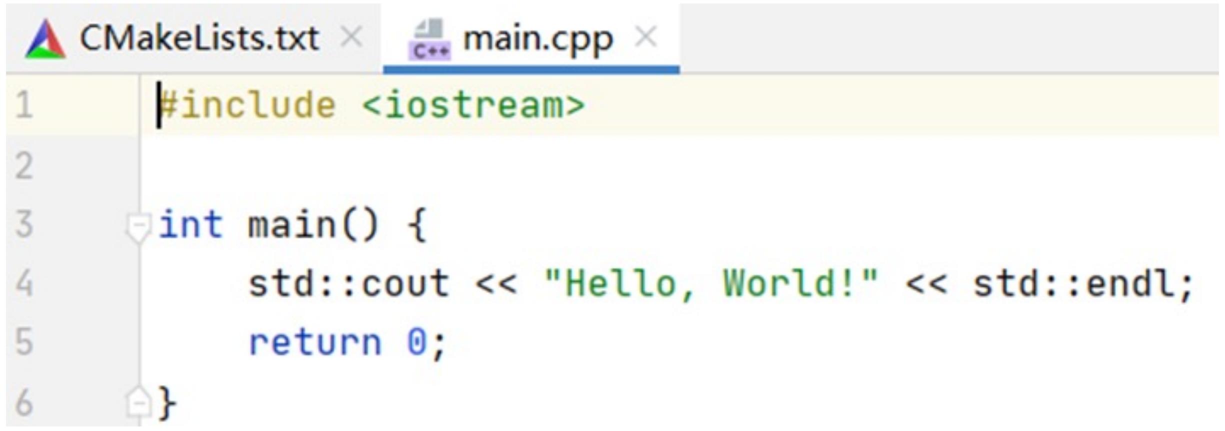
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/CMakeDemo3$ cmake .
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler 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/mycode/CcodeVS/lab08_examples/CMakeDemo3
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab08_examples/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.

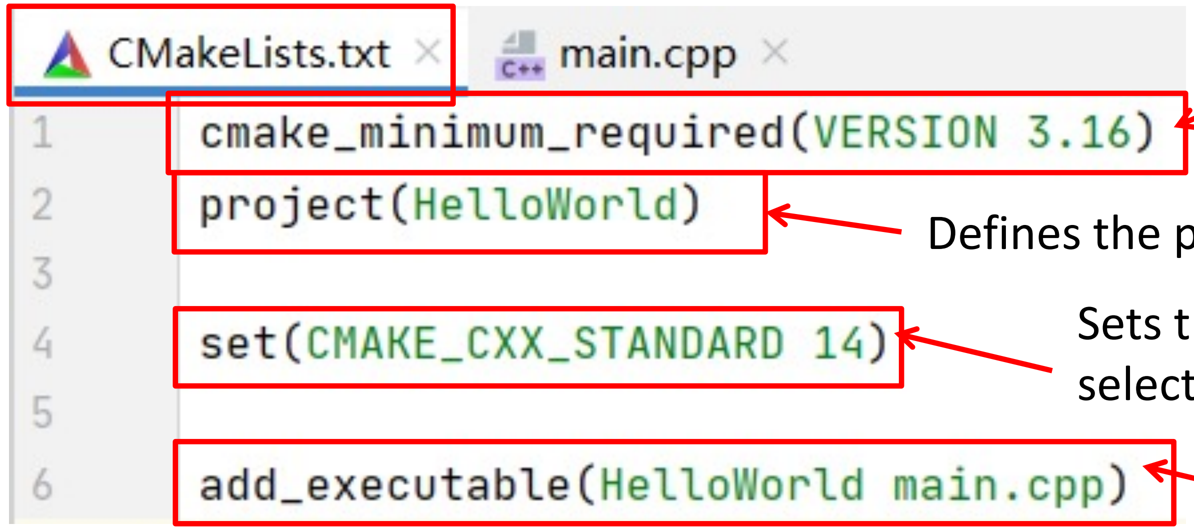


Location: D:\cstudy\CMake\HelloWorld

Language standard: C++14



```cpp
#include <iostream>

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



```cmake
cmake_minimum_required(VERSION 3.16)
project(HelloWorld)

set(CMAKE_CXX_STANDARD 14)

add_executable(HelloWorld main.cpp)
```

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.

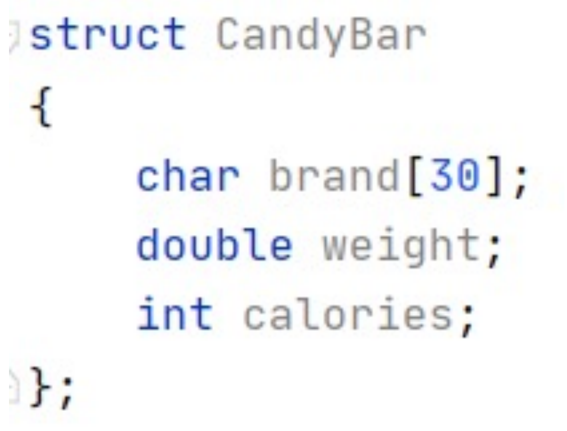
Sets the CMAKE_CXX_STANDARD variable to the value of 14, as we selected when creating the project.

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
https://www.jetbrains.com/help/clion/2016.3/quick-cmake-tutorial.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.

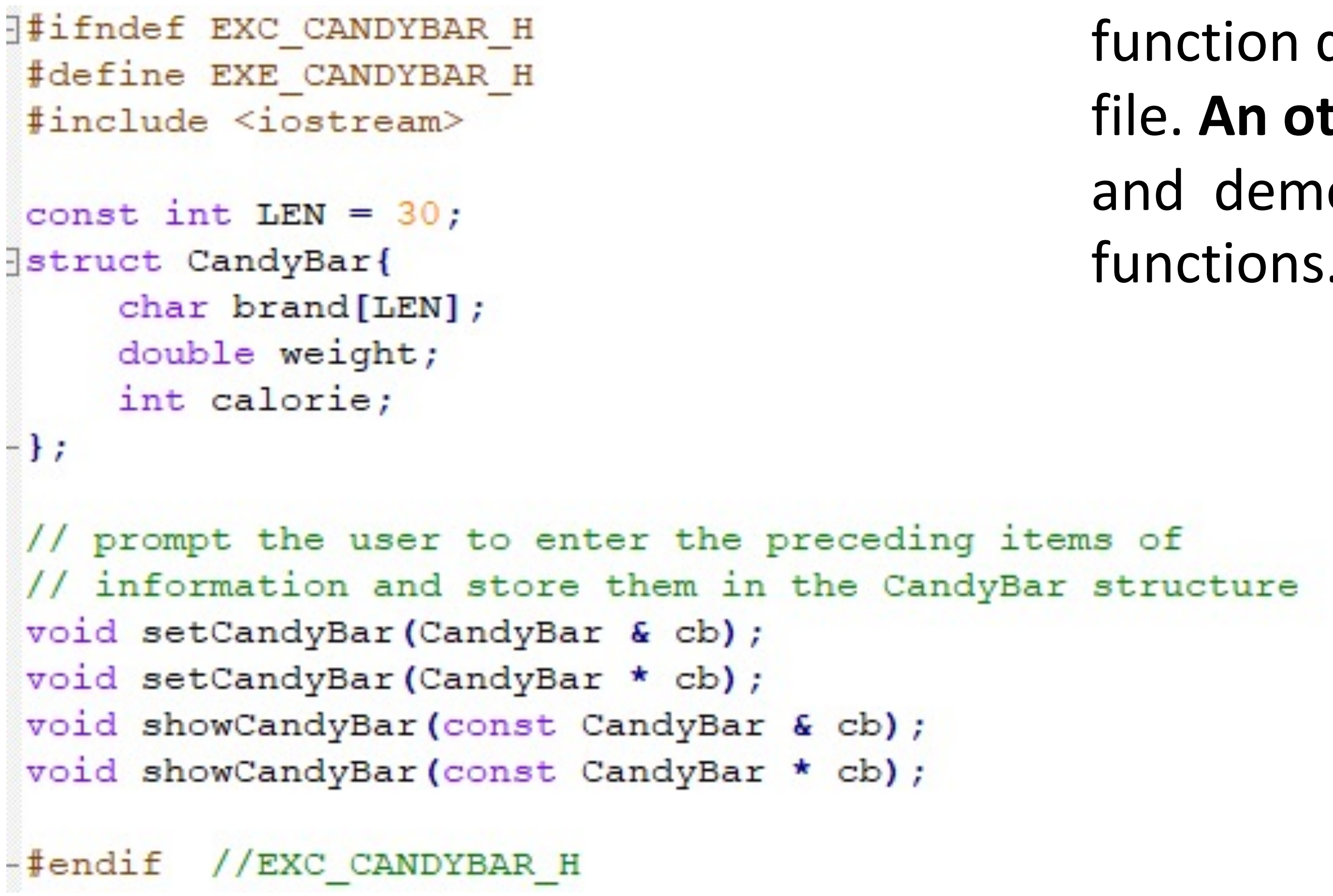


```
struct CandyBar
{
    char brand[30];
    double weight;
    int calories;
};
```

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**



```
#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
```

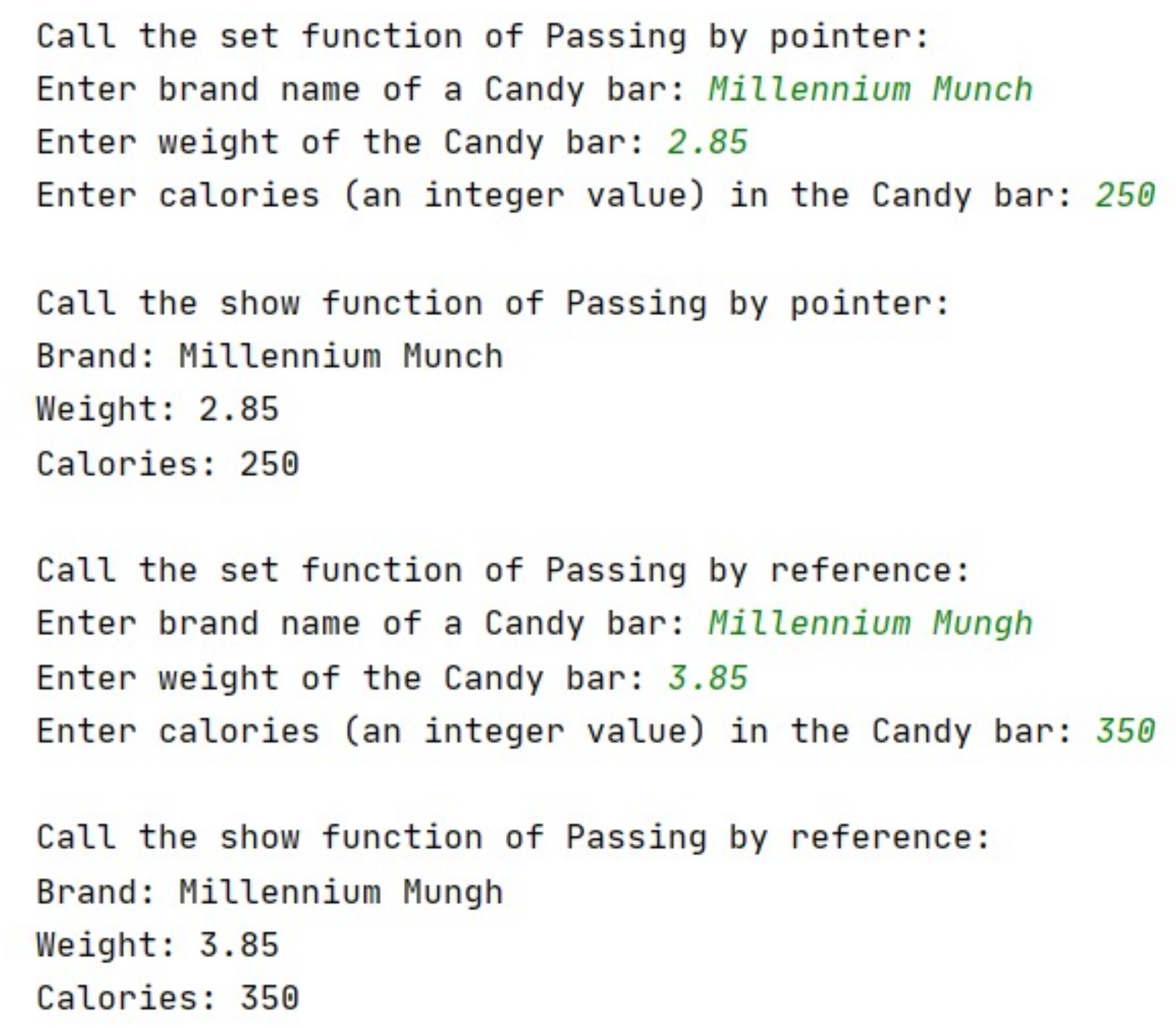
Put together a multi-file program based on this header. **One file**, **named candybar.cpp**, should provide suitable function definitions to match the prototypes in the header file. **An other file named main.cpp** should contain main() and demonstrate all the features of the prototyped functions.

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
```