Advanced Programming

Lab 09

1 CONTENTS

- Learn makefile

2 Knowledge Points

2.1 Makefile

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 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:

```cpp
// functions.h
#pragma once

#define N 5


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

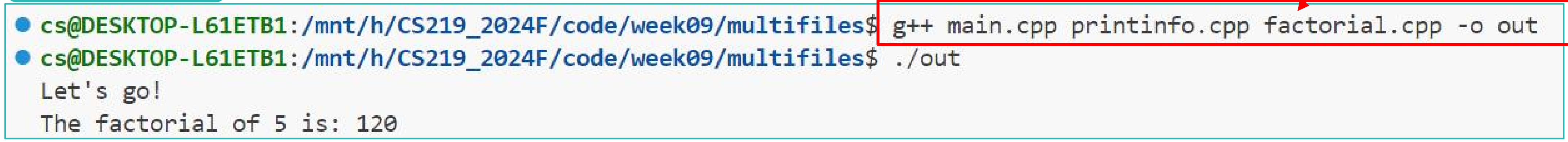
```cpp
// printinfo.cpp
#include <iostream>
#include "functions.h"

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

```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()
{
    printInfo();
    cout << "The factorial of " << N << " is: " << factorial(N) << endl;
    return 0;
}
```

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

Result:



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ g++ main.cpp printinfo.cpp factorial.cpp -o out
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ ./out
Let's go!
The factorial of 5 is: 120
```

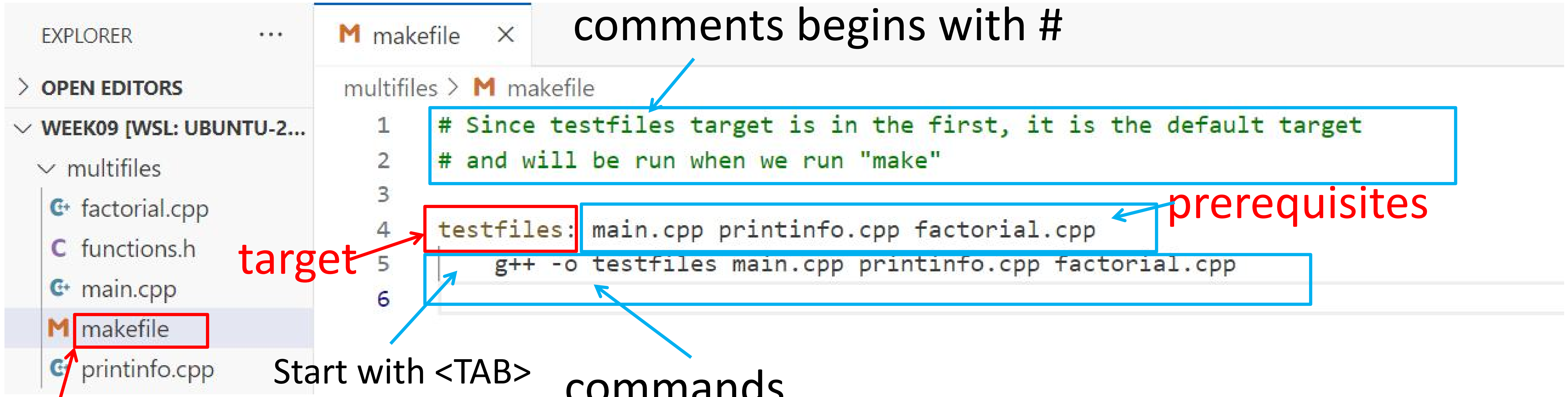
How about if there are hundreds of files need to compile? Do you think it is comfortable to write g++ or gcc compilation command by mentioning all these hundreds file names? Now you can choose **makefile**.

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.



```
# Since testfiles target is in the first, it is the default 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
```

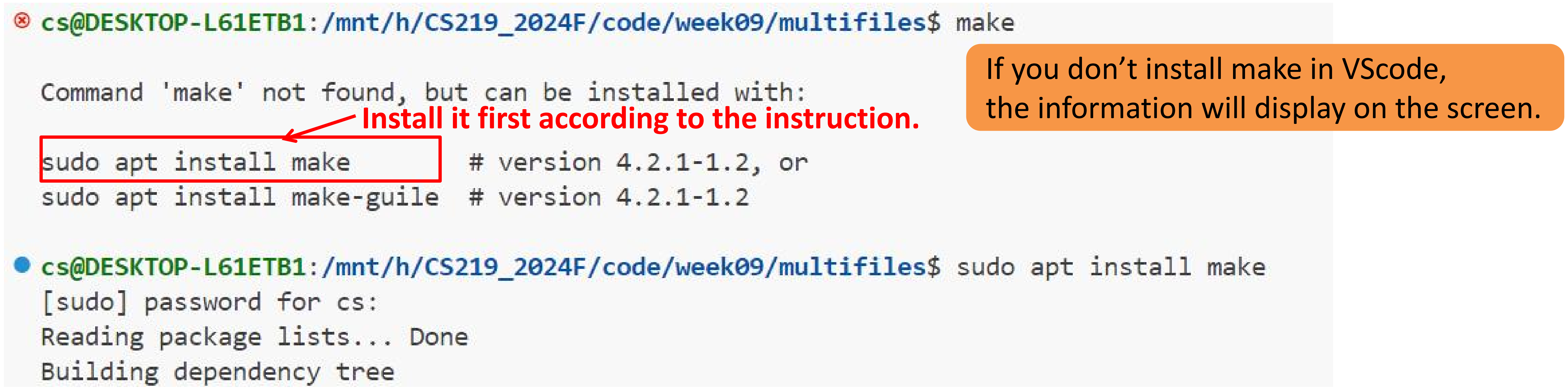
Place the **makefile** together with your programs.

commands

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

```
# Since testfiles target is in the first, it is the default 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
```

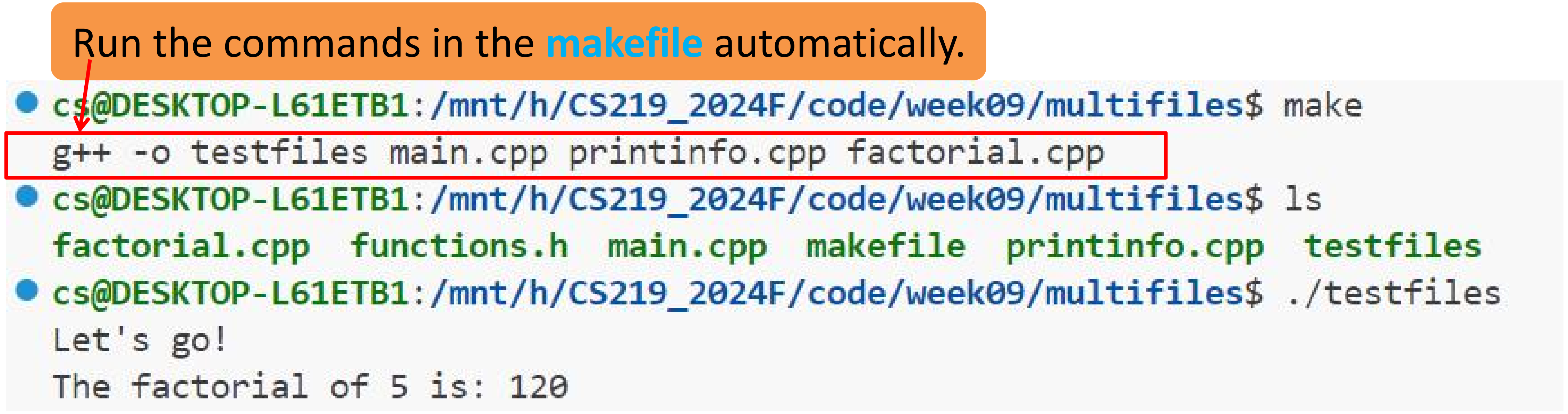
Type the command make in VScode



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ 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

cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ sudo apt install make
[sudo] password for cs:
Reading package lists... Done
Building dependency tree
```

Install it first according to the instruction.

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

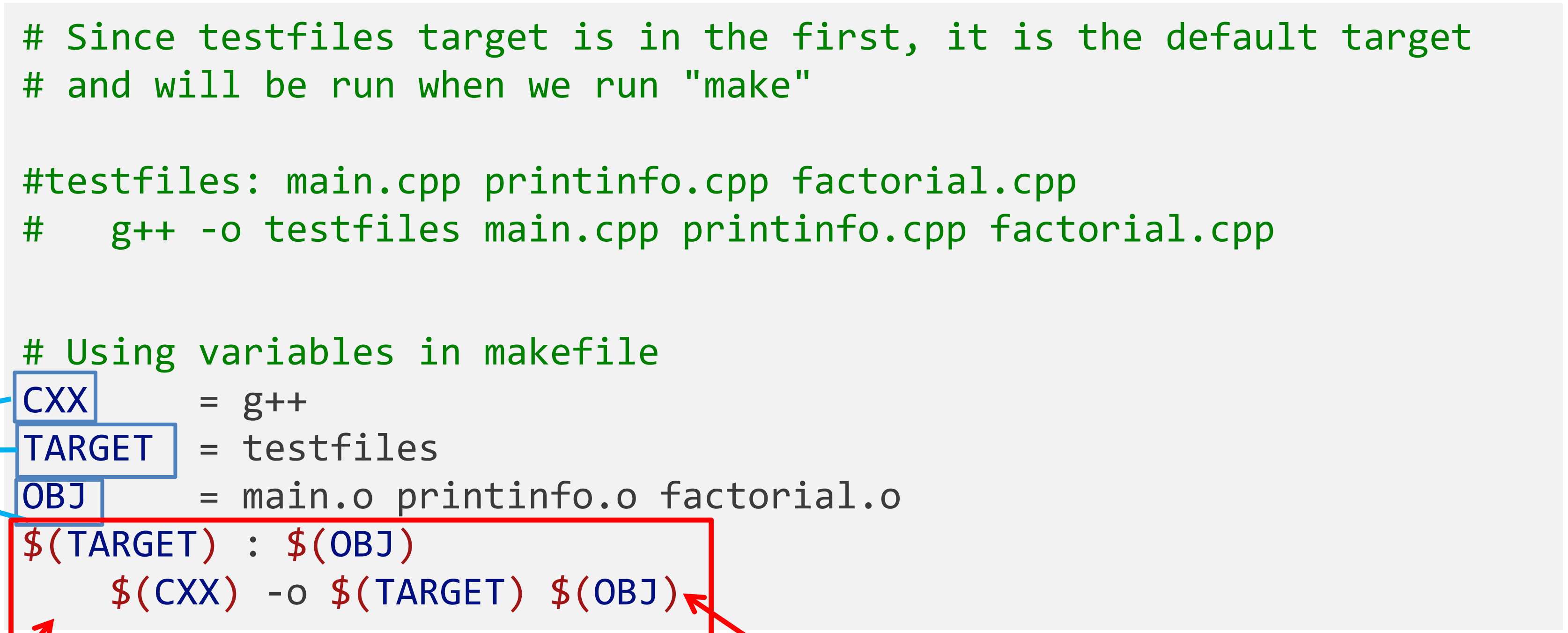


Run the commands in the makefile automatically.

```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++ -o testfiles main.cpp printinfo.cpp factorial.cpp
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ ls
factorial.cpp  functions.h  main.cpp  makefile  printinfo.cpp  testfiles
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ ./testfiles
Let's go!
The factorial of 5 is: 120
```

Define Macros/Variables in the makefile

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

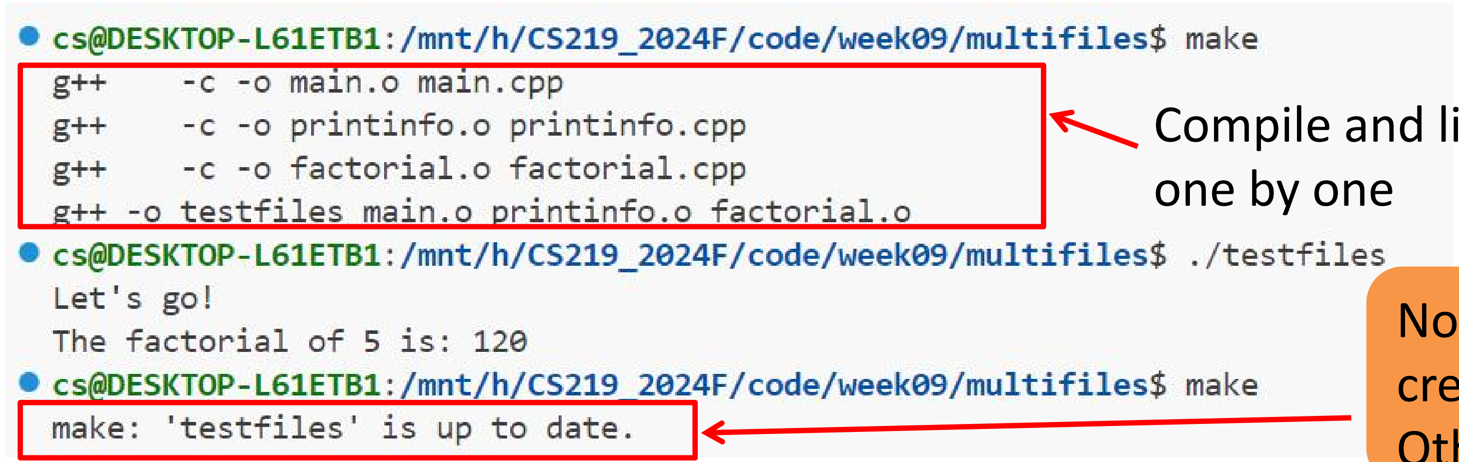


```
# Since testfiles target is in the first, it is the default target
# and will be run when we run "make"

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


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

variables

Start with <TAB>

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

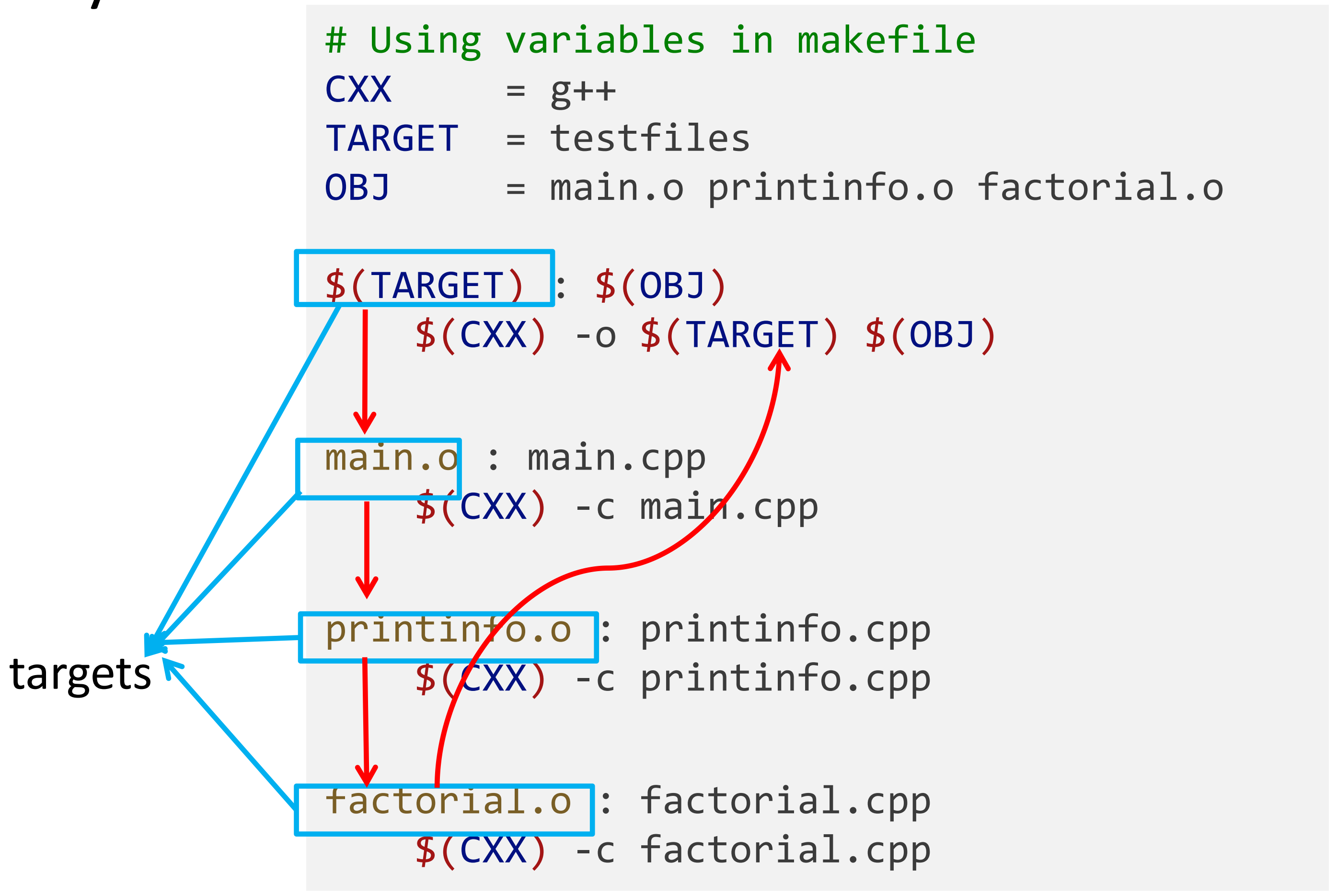


```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++     -c -o main.o main.cpp
g++     -c -o printinfo.o printinfo.cpp
g++     -c -o factorial.o factorial.cpp
g++ -o testfiles main.o printinfo.o factorial.o
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ ./testfiles
Let's go!
The factorial of 5 is: 120
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
make: 'testfiles' is up to date.
```

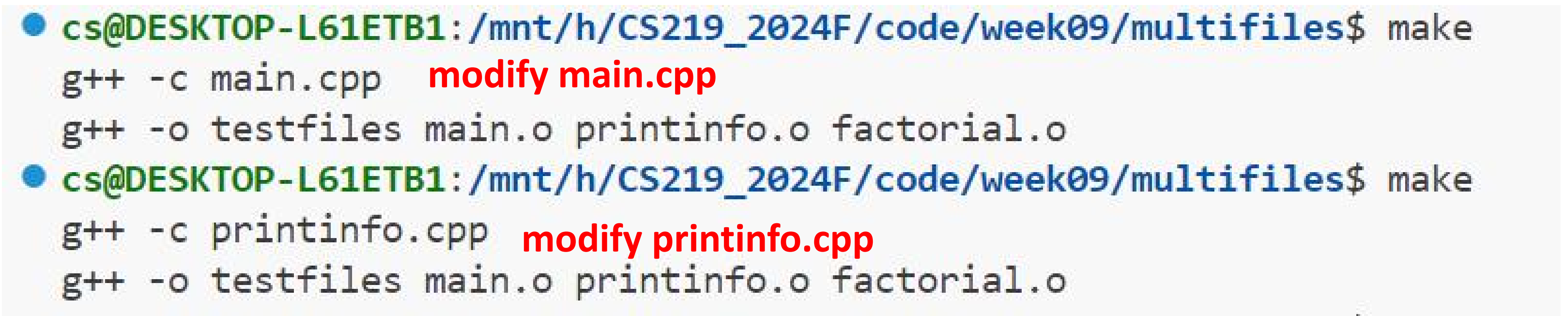
Compile and link the source file one by one

Note: Deletes all the .o files and executable file created previously before using make command. Otherwise, it'll display:

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



```
# Using variables in makefile
CXX      = g++
TARGET   = testfiles
OBJ      = main.o printinfo.o factorial.o

$(TARGET) : $(OBJ)
	$(CXX) -o $(TARGET) $(OBJ)

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

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

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



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++ -c main.cpp
g++ -o testfiles main.o printinfo.o factorial.o
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++ -c printinfo.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

modify main.cpp

modify printinfo.cpp

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

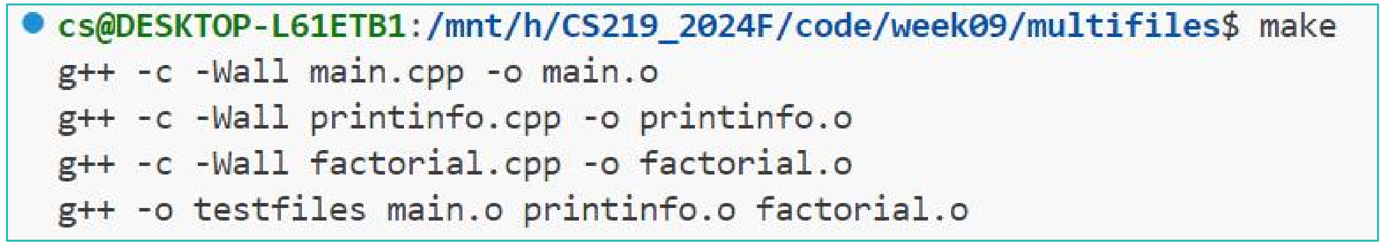
```
# Using several rules and targets
CXX = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o

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


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


%.o : %.cpp
	$(CXX) $(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

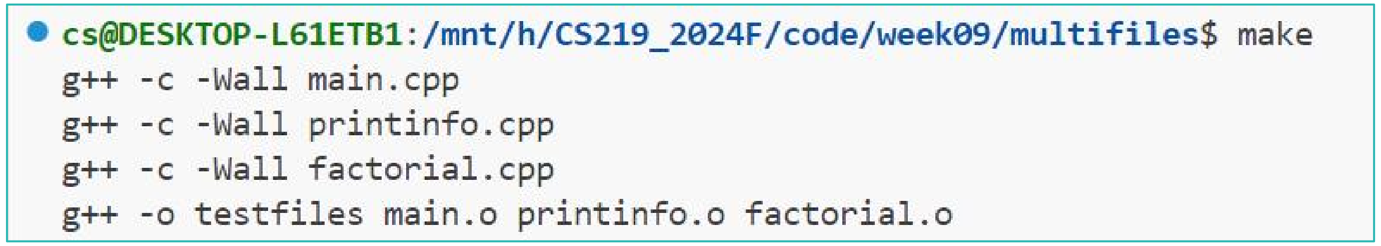


```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++ -c -Wall main.cpp -o main.o
g++ -c -Wall printinfo.cpp -o printinfo.o
g++ -c -Wall factorial.cpp -o factorial.o
g++ -o testfiles main.o printinfo.o factorial.o
```

```
%.o : %.cpp
	$(CXX) $(CFLAGS) $<
```

or

```
%.o : %.cpp
	$(CXX) $(CFLAGS) $^
```



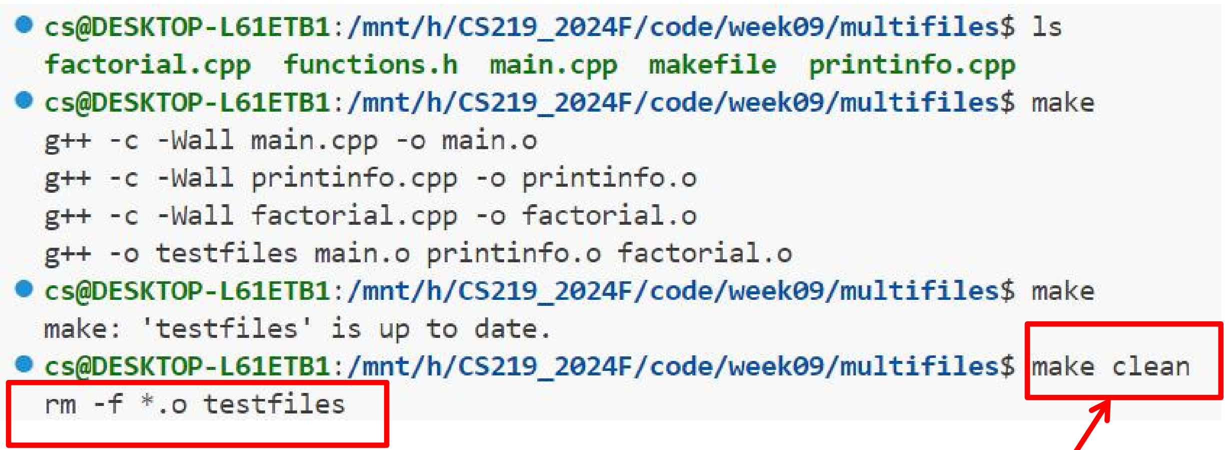
```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++ -c -Wall main.cpp
g++ -c -Wall printinfo.cpp
g++ -c -Wall factorial.cpp
g++ -o testfiles main.o printinfo.o factorial.o
```

Using phony target to clean up compiled results automatically

```
# Using several rules and targets
CXX = g++
TARGET = testfiles
OBJ = main.o printinfo.o factorial.o

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


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


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

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

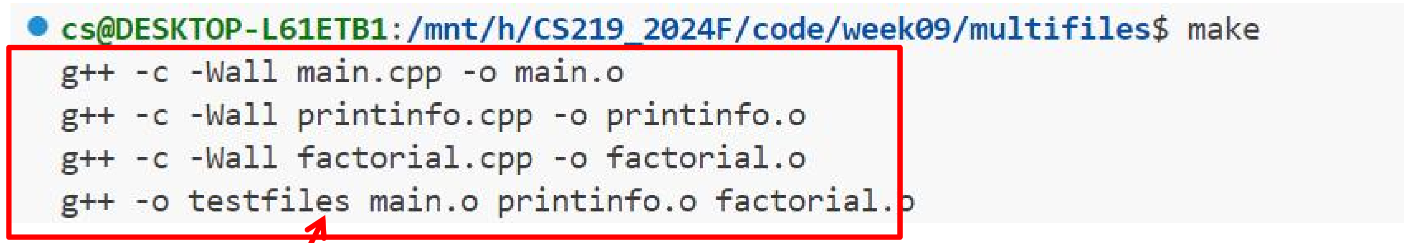
Start with <TAB>

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



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ ls
factorial.cpp  functions.h  main.cpp  makefile  printinfo.cpp
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/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
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
make: 'testfiles' is up to date.
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make clean
rm -f *.o testfiles
```

Because **clean** is a label not a target, the command **make clean** can execute the clean part. Only **make** command can not execute clean part.



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/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

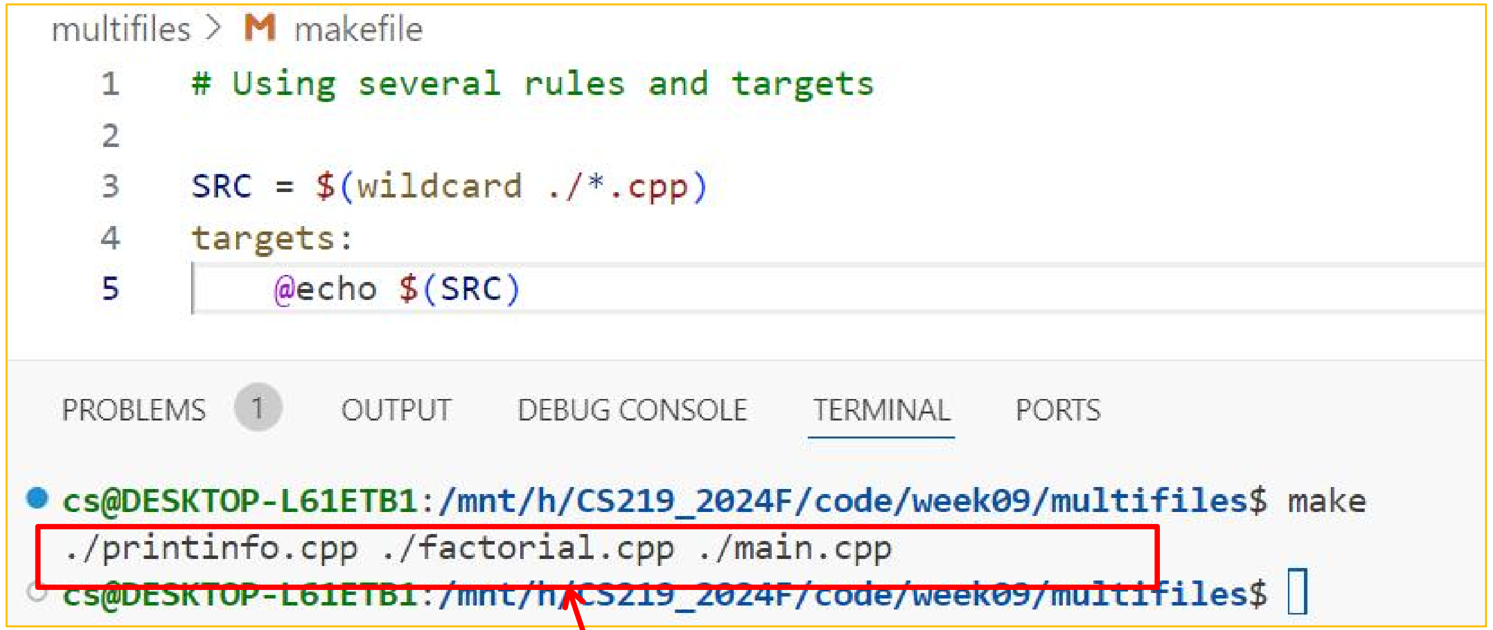
Functions in makefile

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

wildcard: search file

for example:

$$\boxed{\texttt{SRC = \$(wildcard ./*.cpp)}}$$


multifiles > makefile

```
# Using several rules and targets

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

```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$
```

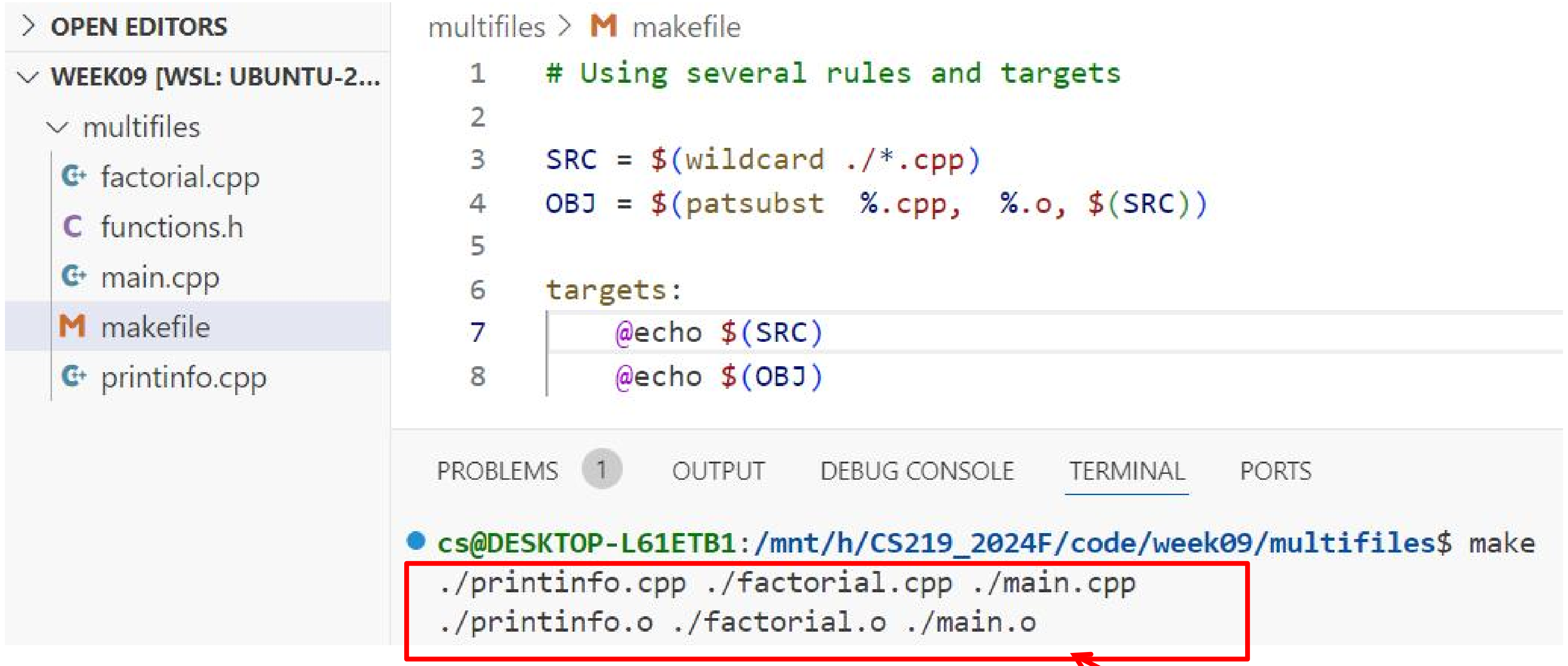
All .cpp files in the current directory

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))
```



```
# Using several rules and targets

SRC = $(wildcard ./*.cpp)
OBJ = $(patsubst  %.cpp,  %.o, $(SRC))

targets:
	@echo $(SRC)
	@echo $(OBJ)
```

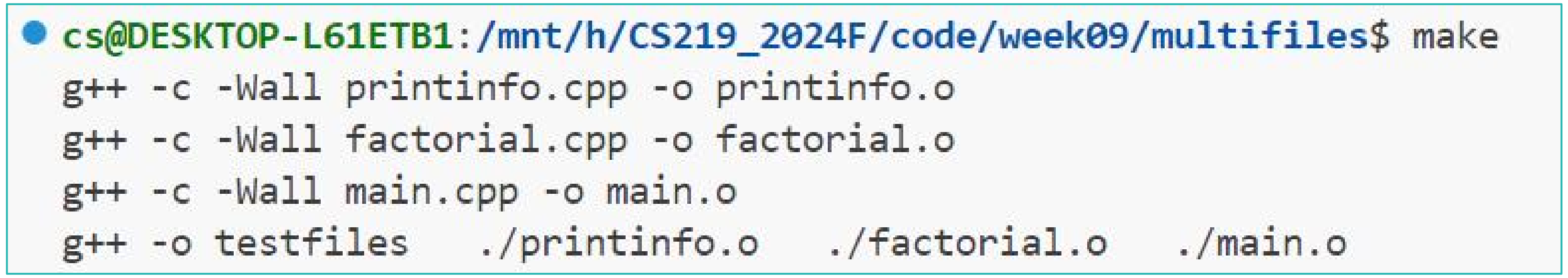
```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
./printinfo.o ./factorial.o ./main.o
```

Replace all .cpp files with .o files

```
# Using several rules and targets
CXX = g++
TARGET = testfiles
# OBJ = main.o printinfo.o factorial.o
SRC = $(wildcard ./*.cpp)
OBJ = $(patsubst  %.cpp,  %.o, $(SRC))

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

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

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

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

vs

```
OBJ = main.o printinfo.o factorial.o
```



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ make
g++ -c -Wall printinfo.cpp -o printinfo.o
g++ -c -Wall factorial.cpp -o factorial.o
g++ -c -Wall main.cpp -o main.o
g++ -o testfiles  ./printinfo.o  ./factorial.o  ./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

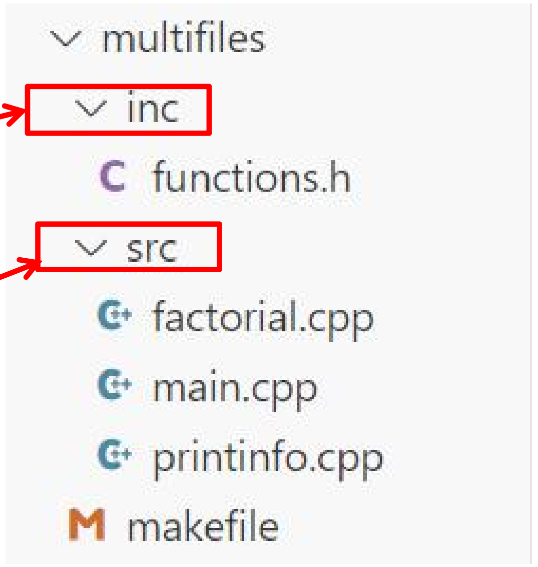
```
SRC_DIR = ./src
SOURCE  = $(wildcard $(SRC_DIR)/*.cpp)
OBJS    = $(patsubst %.cpp, %.o, $(SOURCE))

TARGET  = testfactorial
INCLUDE = -I./inc
```

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

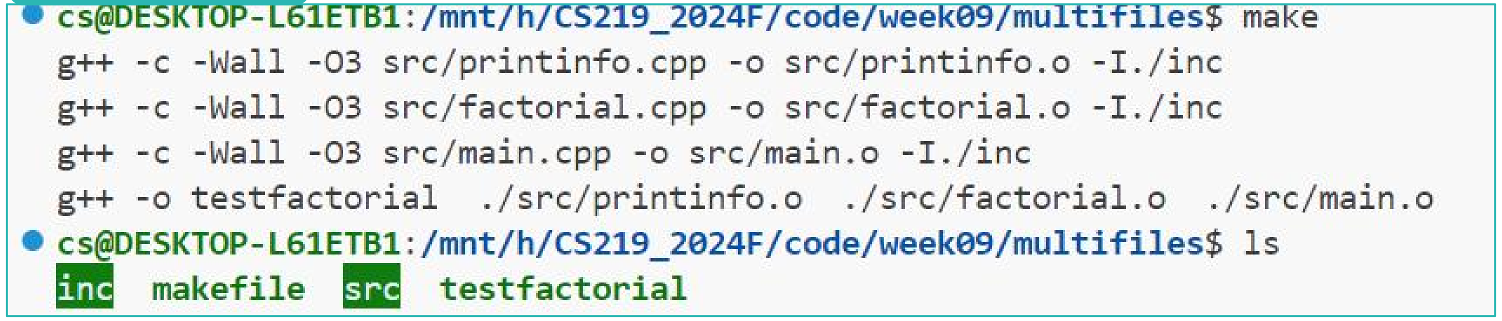
```
# options pass to the compiler
# -c: generates the object file
# -Wall: displays compiler warnings
# -O0: no optimizations
# -O1: default optimization
# -O2: represents the second-level optimization
# -O3: represents the highest level optimization


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


$(TARGET) : $(OBJS)
	$(CXX) -o $@ $(OBJS)


%.o : %.cpp
	$(CXX) $(CXXFLAGS) $< -o $@ $(INCLUDE)


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

All .h files are in inc

All .cpp files are in src



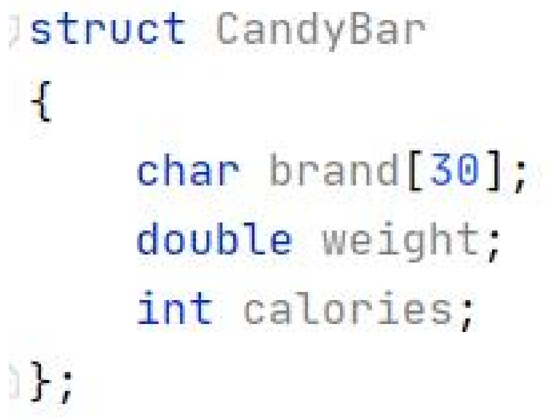
Result:



```
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ 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 testfactorial  ./src/printinfo.o  ./src/factorial.o  ./src/main.o
cs@DESKTOP-L61ETB1:/mnt/h/CS219_2024F/code/week09/multifiles$ ls
inc  makefile  src  testfactorial
```

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.

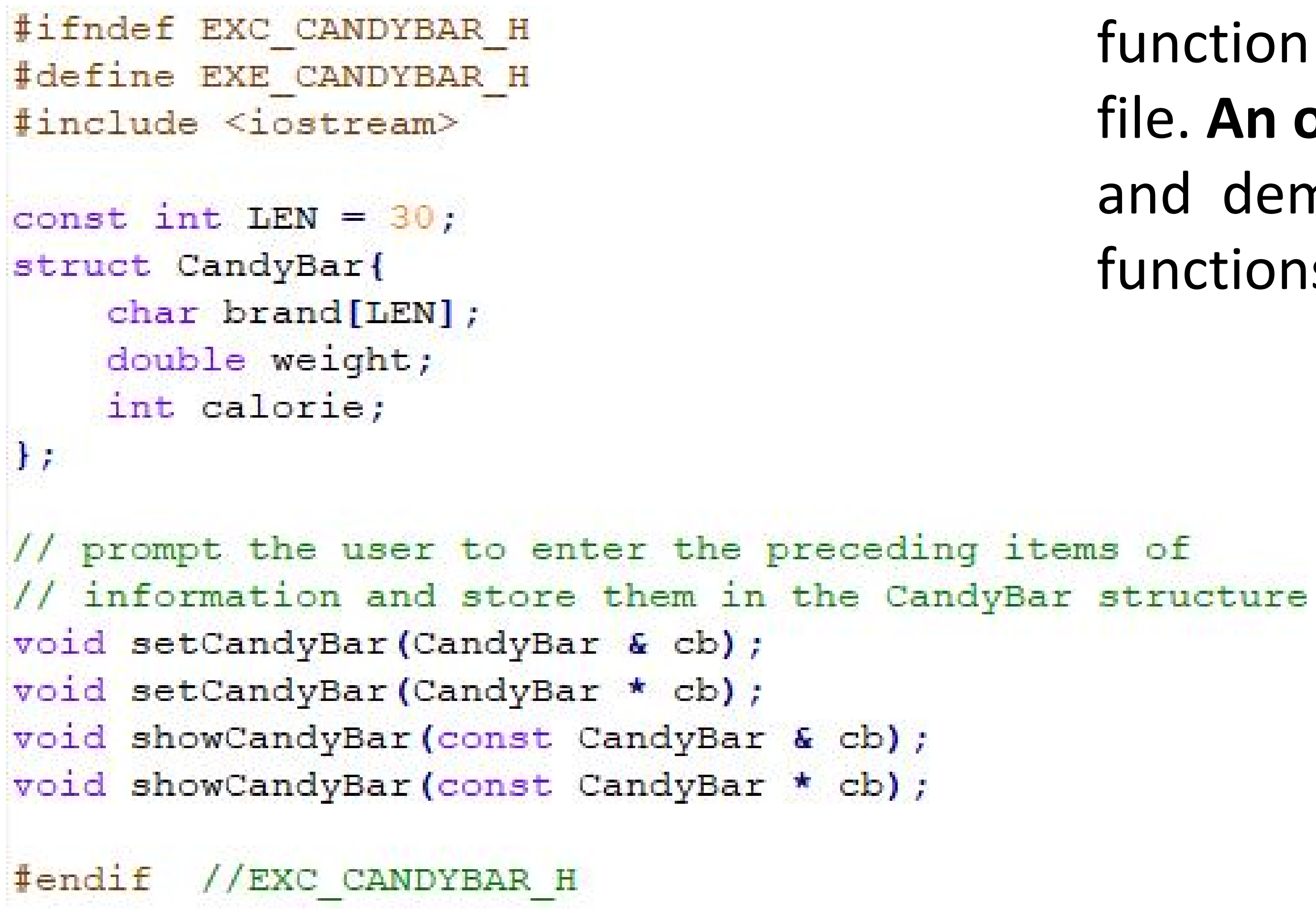


```
struct CandyBar
{
    char brand[30];
    double weight;
    int calories;
};
```

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
```

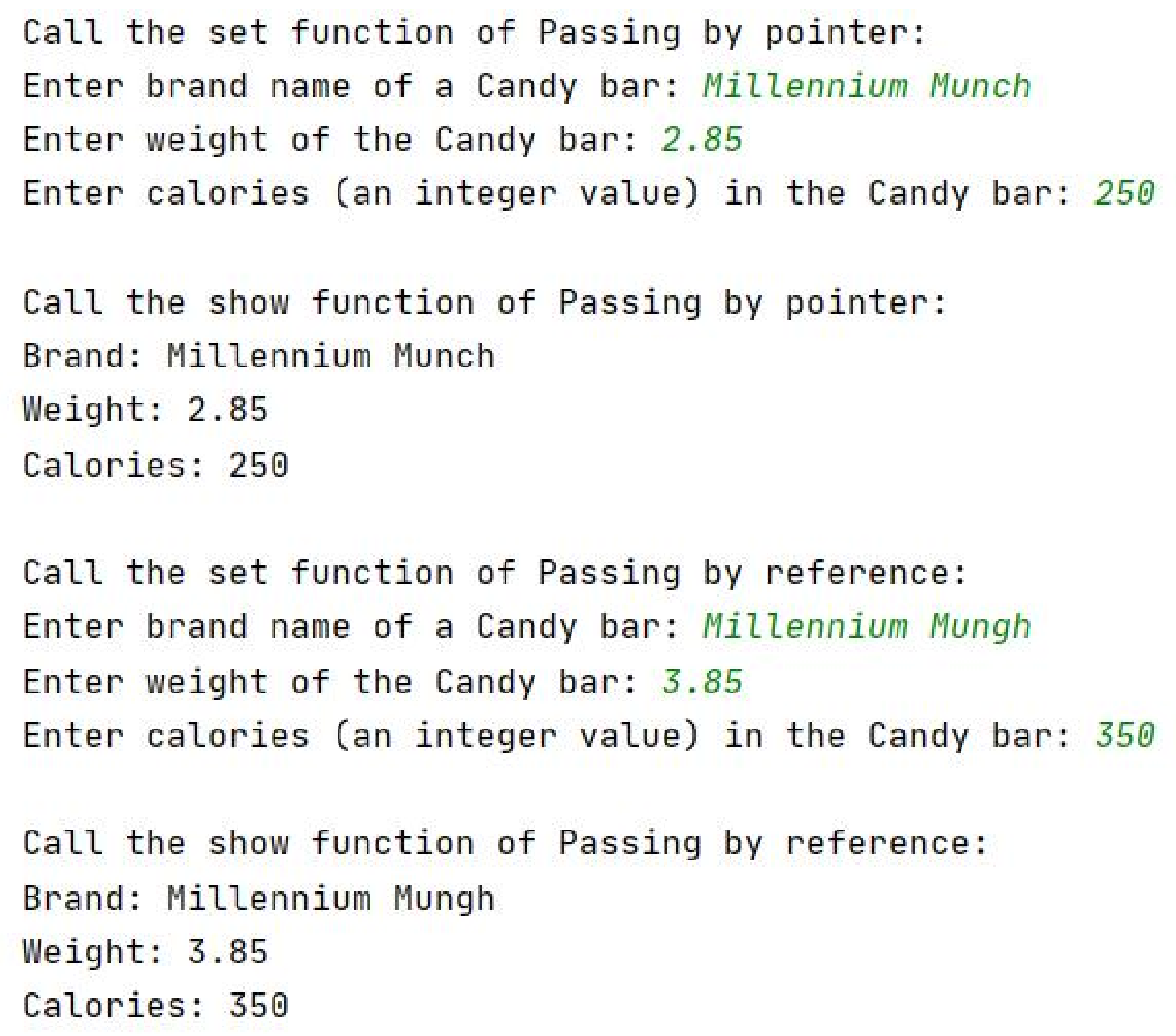
Put together a multi-file program based on this header. **One file**, **named candybar.cpp**, should provide suitable function definitions to match the prototypes in the header file. **An other file named main.cpp** should contain main() and demonstrate all the features of the prototyped functions.

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 CMakeLists.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
```