C/C++ Program Design

LAB 9

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:

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 #

EXPLORER

CCODEVS [WSL: UBUNTU]

- multifiles
  - factorial.cpp
  - functions.h
  - main.cpp
  - makefile
  - printinfo.cpp

makefile

lab09_examples > multifiles > makefile

```
# 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
```

target

prerequisites

start with <TAB>

commands

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

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

Type the command **make** in VScode

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_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
```

Install it first according to the instruction.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_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/lab09_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.

lab09_examples > multifiles > makefile

```
# 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

# 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 '$()'

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_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

Note: Deletes all the .o files and executable file created previously before using make command.

Otherwise, it'll display:

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/multifiles$ make
make: 'testfiles' is up to date.
```

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
```

targets

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

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_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 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
CFLAGES = -c -Wall

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

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

$@: Object Files

$^: all the prerequisites files

$<: the first prerequisite file

```
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
```

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

or

```
$(CXX) $(CFLAGES) $^
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/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
```

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

Using phony target to clean up compiled results

```
# 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
CFLAGES = -c -Wall

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

%.o : %.cpp
	$(CXX) $(CFLAGES) $< -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.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/multifiles$ make
make: 'testfiles' is up to date.
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/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.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_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

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/mycode/CcodeVS/lab09_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/lab09_examples/multifiles$ make
./printinfo.cpp ./factorial.cpp ./main.cpp
./printinfo.o ./factorial.o ./main.o
```

Replace all .cpp files with .o files

```makefile
# Using functions
SRC     = $(wildcard ./*.cpp)
OBJS    = $(patsubst %.cpp, %.o, $(SRC))
TARGET  = testfiles

CXX     = g++
CFLAGES = -c -Wall

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

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

vs

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

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/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

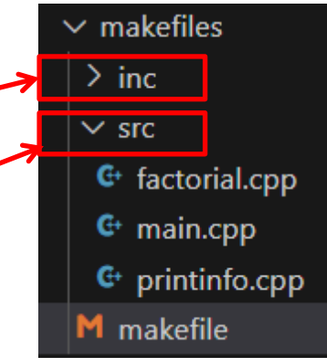
https://blog.csdn.net/xinianbuxiu/article/details/51844994

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

TARGET  = testfactorial
INCLUDE = -I./inc

# 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 teh highest level optimization

CXX       = g++
CFLAGES   = -c -Wall
CXXFLAGES = $(CFLAGES) -O3

$(TARGET) : $(OBJS)
    $(CXX) -o $@ $(OBJS)
%.o : %.cpp
    $(CXX) $(CXXFLAGES) $< -o $@ $(INCLUDE)

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

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

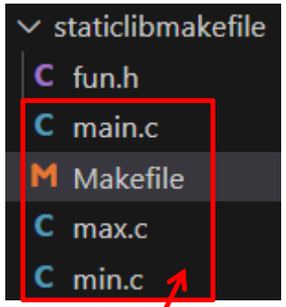
All .h files are in inc

All .cpp files are in src




```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/makefiles$ 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
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/makefiles$ ls
inc  makefile  src  testfactorial
```

Static library in makefile



All the files are in the same folder.

```
#makefile with static library

.PHONY:liba testliba clean

liba: libfun.a
libfun.a: max.o min.o
    ar cr $@ max.o min.o
max.o : max.c
    gcc -c max.c
min.o : min.c
    gcc -c min.c

testliba: main.out
main.out : main.c
    gcc main.c -L. -lfun -o main.out

clean:
    rm -f *.o *.a
```

three targets

the first target with its prerequisite

the second target with its prerequisite

the third target with no prerequisite

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefile$ make
gcc -c max.c
gcc -c min.c
ar cr libfun.a max.o min.o
```

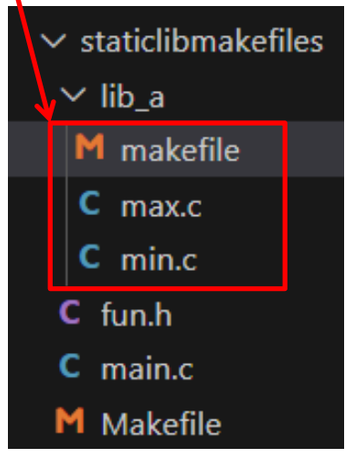
By default, the first target can run with only make command.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefile$ make testliba
gcc main.c -L. -lfun -o main.out
```

The target name followed make command can run the target.

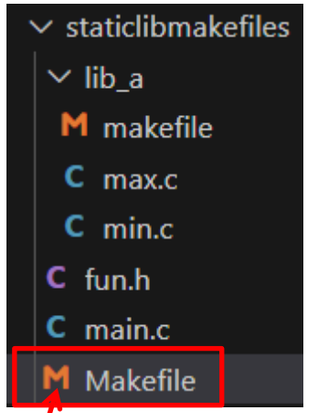
This time we put the functions in the “lib_a” folder, and create a makefile int this folder.

The makefile creates a static library file with the .o files respectively.




```makefile
# makefile with all the .c files created static library

OBJ = $(patsubst %.c, %.o, $(wildcard ./*.c))
TARGET = libmyfun.a
CC = gcc

$(TARGET): $(OBJ)
	ar -r $(TARGET) $^

%.o : %.c
	$(CC) -c $^ -o $@

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

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles$ cd lib_a
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles/lib_a$ make
gcc -c min.c -o min.o
gcc -c max.c -o max.o
ar -r libmyfun.a min.o max.o
ar: creating libmyfun.a
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles/lib_a$ ls
libmyfun.a  makefile  max.c  max.o  min.c  min.o
```



Creates another makefile in the upper-level folder.

```makefile
#link with static library in makefile

OBJS = $(patsubst %.c, %.o, $(wildcard ./*.c))
TARGET = main
CC = gcc

LDFLAGE = -L./lib_a
LIB = -lmyfun

$(TARGET): $(OBJS)
    $(CC)  $^ -o $@  $(LIB) $(LDFLAGE)

%.o : %.c
    $(CC) -c $^ -o $@

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

Link the executable file with the static library.

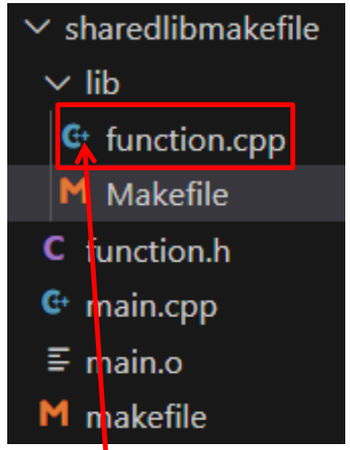
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles/lib_a$ cd ..
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles$ make
gcc -c main.c -o main.o
gcc  main.o -o main  -lmyfun -L./lib_a
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles$ ./main
Please input two integers:4 9
maxNum = 9, minNum = 4
```

```makefile
#link with static library in makefile

OBJS = $(patsubst %.c, %.o, $(wildcard ./*.c))
TARGET = main
CC = gcc

LDFLAGE = -L./lib_a
LIB = -lmyfun

$(TARGET): $(OBJS)
	$(CC)   $(LIB) $(LDFLAGE)  $^ -o $@

%.o : %.c
	$(CC) -c $^ -o $@

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

If you put the flag before $^, it will cause error.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/staticlibmakefiles$ make
gcc -c main.c -o main.o
gcc -lmyfun -L./lib_a main.o -o main
/usr/bin/ld: main.o: in function `main':
main.c:(.text+0x53): undefined reference to `max'
/usr/bin/ld: main.c:(.text+0x65): undefined reference to `min'
collect2: error: ld returned 1 exit status
make: *** [Makefile:11: main] Error 1
```

Shared library in makefile

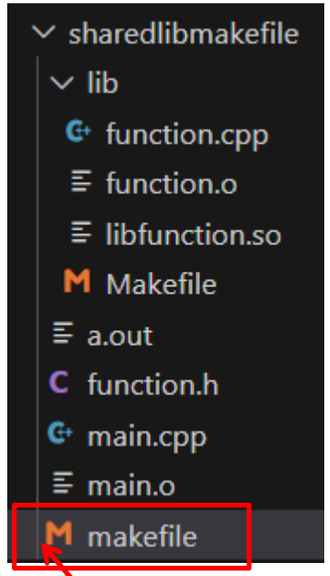


Put the function in the "lib" folder, and create a makefile int this folder.

```makefile
# create dynamic library in makefile

OBJ = $(patsubst %.cpp, %.o, $(wildcard ./*.cpp))
TARGET = libfunction.so
CXX = g++

$(TARGET) : $(OBJ)
    $(CXX)  -shared -fPIC $^ -o $@

    sudo cp $(TARGET) /usr/lib/

%.o : %.cpp
    $(CXX) -c $^ -o $@

clean:
    rm -f *.o $(TARGET) libfunction.so
    sudo rm -f /usr/lib/libfunction.so
```

Copy the .so file into the /usr/lib/folder, which can be accessed by default.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlibmakefile/lib$ make
g++ -c function.cpp -o function.o
g++  -shared -fPIC function.o -o libfunction.so
sudo cp libfunction.so /usr/lib/
[sudo] password for maydlee:
```



Another makefile in the upper-level folder.

```makefile
# makefile with dynamic library

OBJS = $(patsubst %.cpp, %.o, $(wildcard ./*.cpp))
TARGET = main
CXX = g++

LDFLAGE = -L./lib
LIB = -lfunction

$(TARGET) : $(OBJS)
	$(CXX) $^ $(LIB) $(LDFLAGE)  -o $@

%.o : %.cpp
	$(CXX) -c $^ -o $@

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

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlibmakefile$ make
g++ -c main.cpp -o main.o
g++ main.o -lfunction -L./lib  -o main
```

```makefile
# makefile with dynamic library

OBJS = $(patsubst %.cpp, %.o, $(wildcard ./*.cpp))
TARGET = main
CXX = g++

LDFLAGE = -L./lib
LIB = -lfunction

$(TARGET) : $(OBJS)
	$(CXX) $(LIB) $(LDFLAGE) $^ -o $@

%.o : %.cpp
	$(CXX) -c $^ -o $@

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

If you put the flag before $^, it will cause error.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/lab09_examples/sharedlibmakefile$ make
g++ -c main.cpp -o main.o
g++ -lfunction -L./lib main.o  -o main
/usr/bin/ld: main.o: in function `main':
main.cpp:(.text+0x17): undefined reference to `add(int, int)'
collect2: error: ld returned 1 exit status
make: *** [makefile:11: main] Error 1
```

3 Exercises

Define four functions that implement the operations of addition, subtraction, multiplication and division respectively.(one function one .cpp file) Write a test program to test these functions.

1.Write a Makefile file to organize all of the three files for compilation.

2. Write another Makefile file with static library. You can choose one (or two) of these functions for your .a file.

3. Write the third Makefile file with dynamic library. You can choose one function for your .so file.

Run make to test your Makefile. Run your program at last.