南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Program Design

Lab 7, shared library

廖琪梅，王大兴

Building a shared library

- Suppose we have written the following code:

```
// mymath.h
#ifndef __MY_MATH_H__
#define __MY_MATH_H__
float arraySum(const float *array, size_t size);
#endif
```

```
// mymath.cpp
#include <iostream>
#include "mymath.h"

float arraySum(const float *array, size_t size)
{
   if(array == NULL)
   {
      std::cerr << "NULL pointer!" << std::endl;
      return 0.0f;
   }
   float sum = 0.0f;
   for(size_t i = 0; i < size; i++)
      sum += array[i];
   return sum;
}
```

```
// main.cpp
#include <iostream>
#include "mymath.h"
int main()
{
   float arr1[8]{1.f, 2.f, 3.f, 4.f, 5.f, 6.f, 7.f, 8.f};
   float * arr2 = NULL;

   float sum1 = arraySum(arr1, 8);
   float sum2 = arraySum(arr2, 8);

   std::cout << "The result1 is " << sum1 <<
std::endl;
   std::cout << "The result2 is " << sum2 <<
std::endl;

   return 0;
}
```

Building shared libraries

- A **shared library** packs compiled code of functionality that the developer wants to **share** with other developers.
- Shared libraries in linux are **.so** files.
- Remember to use arguments "**-shared**" and "**-fPIC**" when building it.
- Now we should see "libmymath.so" in the directory

The name of .so must be started with "**lib**" followed by the .cpp name in which a function is defined.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ g++ -shared -fPIC -o libmymath.so mymath.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ls
libmymath.so  main.cpp  mymath.cpp  mymath.h
```

Create a shared library.

Generate Position-Independent-Code.

Using shared library

- Now we can use the ".so" shared library.
- Let's compile "main":

"lmymath" indicates to use "libmymath.so"

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ g++ -o main main.cpp -L. -lmymath
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ls
libmymath.so  main  main.cpp  mymath.cpp  mymath.h
```

"-L." indicates to find a library file in the current directory.

- **-L**: indicates the directory of libraries
- **-l**: indicates the library name, the compiler can give the "**lib**" prefix to the library name and follows with **.so** as extension name.

Using shared library

- After the “main” has been compiled, try to run it:

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ls
libmymath.so  main  main.cpp  mymath.cpp  mymath.h
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ./main
./main: error while loading shared libraries: libmymath.so: cannot open shared object file: No such file or directory
```

- It failed because “main” now relys on “libmymath.so”. By default, libraries are located in **/usr/local/lib** or **/usr/lib** , but our “libmymath.so” is not in that directory. You must tell the terminal where to find “libmymath.so”.

Using a shared library

- Using **export** command to set environment variable "**LD_LIBRARY_PATH**"
- And then run "main" again

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ echo $LD_LIBRARY_PATH
.:
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ./main
NULL pointer!
The result1 is 36
The result2 is 0
```

export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH

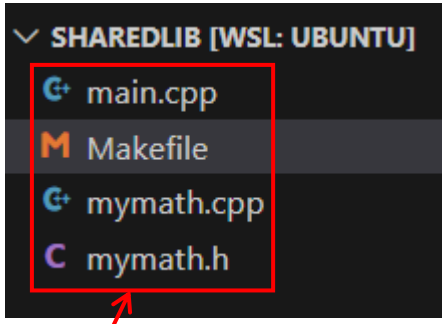
There is no space on either side of the equal sign

. indicates the current directory

Another choice is to move your .so file to **/usr/lib** folder by **mv** command

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ sudo mv libmymath.so /usr/lib
[sudo] password for maydlee:
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ./main
NULL pointer!
The result1 is 36
The result2 is 0
```

Shared library in makefile



All the files are in the same folder.

```
1   # makefile with dynamic library
2
3   .PHONY: libd testlibd clean
4
5   libd: libfunction.so
6   libfunction.so: mymath.cpp
7       g++ -shared -fPIC -o libfunction.so mymath.cpp
8
9   testlibd : main
10  main : main.cpp libfunction.so
11      g++ main.cpp -L. -lfunction -Wl,-rpath=. -o main
12
13  clean:
14      rm -rf *.o *.so main
```

–Wl option allows you to pass subsequent arguments to the linker.

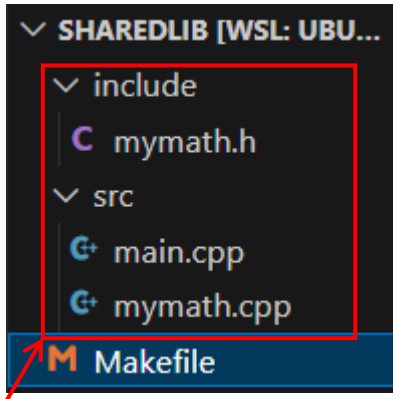
–rpath option is used to specify the directories that the runtime dynamic linker(ld.so) should search when looking for dynamic libraries during execution.

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ make
g++ -shared -fPIC -o libfunction.so mymath.cpp
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ make testlibd
g++ main.cpp -L. -lfunction -Wl,-rpath=. -o main
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ ./main
NULL pointer!
The result1 is 36
The result2 is 0
```

Run the first target to create the dynamic library.

Run the second target to link the dynamic library to the executable file.

Run the executable file.



This time we put all the source files in the “src” folder, the function header file in the “include” folder, and create a makefile in the current folder.

```
cpp_srcs := $(wildcard src/*.cpp)
cpp_objs := $(patsubst src/%.cpp, objs/%.o, $(cpp_srcs))
so_objs  := $(filter-out objs/main.o, $(cpp_objs))

include_path := ./include

I_options := $(include_path:%=-I%)

compile_flags := -g -O3 -w -fPIC $(I_options)

objs/%.o : src/%.cpp
    mkdir -p $(dir $@)
    g++ -c $^ -o $@ $(compile_flags)

compile : $(cpp_objs)

# ============== Generating dynamic library ==============
lib/libfunction.so : $(so_objs)
    mkdir -p $(dir $@)
    g++ -shared $^ -o $@

dynamic : lib/libfunction.so

clean:
    rm -rf ./objs ./lib
.PHONY : compile dynamic clean
```

The first part of the makefile just creates a dynamic library named **libfunction.so**

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ make
mkdir -p objs/
g++ -c src/mymath.cpp -o objs/mymath.o -g -O3 -w -fPIC -I./include
mkdir -p objs/
g++ -c src/main.cpp -o objs/main.o -g -O3 -w -fPIC -I./include
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ make dynamic
mkdir -p lib/
g++ -shared objs/mymath.o -o lib/libfunction.so
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ cd lib
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib/lib$ ls
libfunction.so
```

```
# ============== linking dynamic library ==============
library_path := ./lib

linking_libs := function

l_options := $(linking_libs:%=-l%)
L_options := $(library_path:%=-L%)
r_options := $(library_path:%=-Wl,-rpath=%)

objs/testdynamic : objs/main.o compile dynamic
    mkdir -p $(dir $@)
    g++ $< -o $@ $(linking_flags)

run : objs/testdynamic
    ./$<

clean :
    rm -rf lib objs

.PHONY : compile dynamic run clean
```

The second part of the makefile links the dynamic library **libfunction.so** to the executable file **testdynamic** in the "objs" folder.

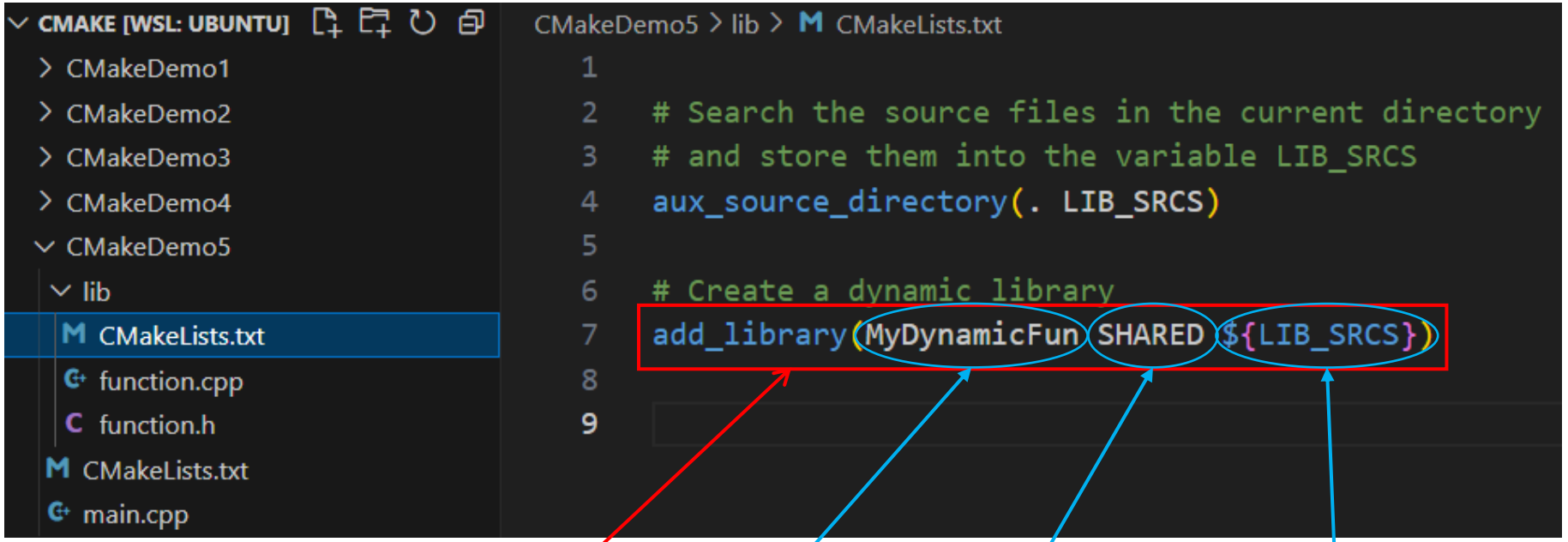
```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/mycode/CcodeVS/sharedlib$ make run
mkdir -p objs/
g++ objs/main.o -o objs/testdynamic -lfunction -L./lib -Wl,-rpath=./lib
./objs/testdynamic
NULL pointer!
The result1 is 36
The result2 is 0
```

Creating and linking a dynamic library by CMake

We want to create a dynamic library by function.cpp and call the dynamic library in main.cpp. This time we write two CMakeLists.txt files, one in **CmakeDemo5** folder and another in **lib** folder.

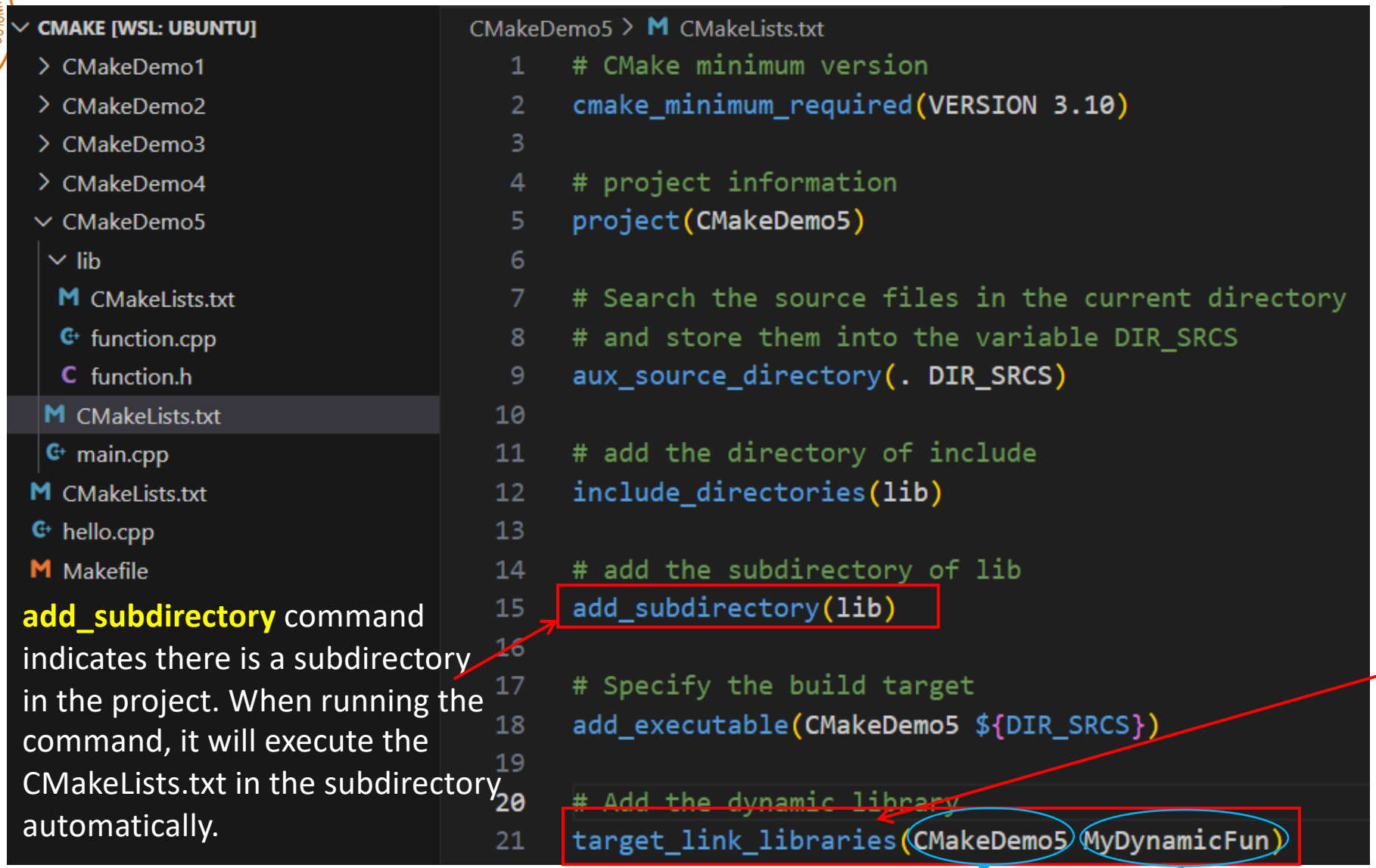
The CMakeLists.txt in **lib** folder creates a dynamic library.

```
./CMakeDemo5
        |
        +--- main.cpp
        |
        +--- lib/
                |
                +--- function.h
                |
                +--- function.cpp
```



Create a static library named libMyDynamicFun.so by the files in the current directory.

The CMakeLists.txt in **CMakeDemo5** folder creates the project.



add_subdirectory command indicates there is a subdirectory in the project. When running the command, it will execute the CMakeLists.txt in the subdirectory automatically.

Indicates that the project needs link a library named **MyDynamicFun**, MyDynamicFun can be a static library file or a dynamic library file.

project name

library file name

If there are more than one file, list them using space as the separator.


```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo5$ mkdir build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo5$ cd build
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo5/build$ cmake ..
-- The C compiler identification is GNU 9.4.0
-- The CXX compiler identification is GNU 9.4.0
-- 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/CMake/CMakeDemo5/build
```

```
maydlee@LAPTOP-U1MO0N2F:/mnt/d/CMake/CMakeDemo5/build$ make
Scanning dependencies of target MyDynamicFun
[ 25%] Building CXX object lib/CMakeFiles/MyDynamicFun.dir/function.cpp.o
[ 50%] Linking CXX shared library libMyDynamicFun.so
[ 50%] Built target MyDynamicFun
Scanning dependencies of target CMakeDemo5
[ 75%] Building CXX object CMakeFiles/CMakeDemo5.dir/main.cpp.o
[100%] Linking CXX executable CMakeDemo5
[100%] Built target CMakeDemo5
```

Exercise 1

Overload a function **bool vabs(int * p, int n)** which can compute the absolute value for every element of an array, the array can be int, float and double.

Should n be int or size_t? what's the difference? Remember to check whether the pointer is valid.

Create a shared library “libvabs.so” with 3 overloaded vabs() functions in it, and then compile and run your program with this shared library.

Exercise 2

Write a program that uses a function template called ***Compare*** to compare the relationship between the values of the two arguments and return 1 when the first argument is greater than the second one; return -1 when the first argument is smaller than the second one, return 0 when the both values are equal. Test the program using integer, character and floating-point number arguments and print the result of the comparation.

If there is a structure as follows, how to define an explicit specialization of the template function **Compare** and print the result of the comparation?

```
struct stuinfo{
   string name;
    int age;
};
```

The prototype of the Compare:

```
template <typename T>
int Compare(const T &a, const T &b);
```

The output:

```
Compare of the two integers:-1
Compare of the two floats:1
Compare of the two characters:1
Compare of the two structs:1
```