C Programming

Lecture 11: make & Makefile

Lecturer: *Dr.* Wan-Lei Zhao

Autumn Semester 2022

Email: wlzhao@xmu.edu.cn,

Outline

Why make? (1)

```c
#ifndef MYLIB_H
#define MYLIB_H
int isodd(int x);
float square(float x);
#endif
```

mylib.h

```c
#include "mylib.h"
float square(float x){
    return x*x;
}

int isodd(int x){
    if(x%2 != 0)
      return 1;
    else
      return 0;
}
```

mylib.c

```c
#include "mylib.h"
#include <stdio.h>
int main(){
   float x = 3.4;
   int a = 5;
   float y = square(x);
   if(isodd(a))
   {
        printf("%d is odd\n", a);
   }
   return 0;
}
```

main.c

```
gcc myproj.c -o myproj.o -c
gcc mylib.c -o mylib.o -c

gcc -o myproj myproj.o mylib.o
```

Build the project

Why make? (2)

```
1 gcc myproj.c -o myproj.o -c
2 gcc mylib.c -o mylib.o -c
3
4 gcc -o myproj myproj.o mylib.o
```

Listing 1: "Build the project"

- In practice, we may have many libraries to compile and link
- gcc -o myproj myproj.o mylib.o
- If we do it manually, it is too laborious!!!
- This is where "Makefile" comes to fit in

Makefile

- A script file organize all the compilation things together
- It is responsible for
  1. Compiling the source files (compile from .c to .o)
  2. Linking the files into the final executable software
  3. Installing the software to target directory
- Command make will parse the script
- It fulfills the intructions in the script

Prepare Environment (1)

- Define the variables

```
1 WORK_DIR=.
2 CC=gcc
3 LD=gcc
4 OBJ_DIR=$(WORK_DIR)/obj
5 OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
6 RELEASE=$(WORK_DIR)/bin/myproj
```

- command make supports environment variable definitions

$$\text{VARIABLE\_NAME} = \text{value}$$

- One can specify the file, directory, command, compilation parameters
- They may support the compilation of the project

Prepare Environment (2)

```
1 WORK_DIR=.
2 CC=gcc
3 LD=gcc
```

Makefile

- Command make supports environment variable definitions

$$\text{WORK\_DIR} = .$$

- Specify the project directory where "Makefile" and the project is located
- The variable name is by convention CAPITALIZED

Prepare Environment (3)

```
1 WORK_DIR=.
2 CC=gcc
3 LD=gcc
```

Listing 2: Makefile

- Command make supports environment variable definitions

CC=gcc
LD=gcc

- "CC=gcc" specifies the compiler
- "LD=gcc" specifies the linker

Prepare Environment (4)

```
1 OBJ_DIR=$(WORK_DIR)/obj
2 OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
3 RELEASE=$(WORK_DIR)/bin/myproj
```

Listing 3: Makefile

- Command make supports environment variable definitions

$(WORK_DIR)/obj

- $(VARIABLE) cite the value of the VARIABLE
- Here "$(WORK_DIR)" is replaced by "./"

Prepare Environment (5)

```
1 OBJ_DIR=$(WORK_DIR)/obj
2 OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
3 RELEASE=$(WORK_DIR)/bin/myproj
```

Listing 4: Makefile

- The above instructions indicate
  1. The object files will be put to ./obj/
  2. "OBJ_RELEASE" keeps the lists of all object files
  3. The final target binary software name is "myproj"
  4. It will be put to ./bin/

Prepare Environment (6)

```
1 WORK_DIR=.
2 CC=gcc
3 LD=gcc
4 OBJ_DIR=$(WORK_DIR)/obj
5 OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
6 RELEASE=$(WORK_DIR)/bin/myproj
```

1. We know the working directory
2. We have the compiler and linker
3. We know where we should put the object files
4. We know where we should put the target binary file

Prepare Environment (7)

```
1  WORK_DIR=.
2  CC=gcc
3  LD=gcc
4  OBJ_DIR=$(WORK_DIR)/obj
5  OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
6  RELEASE=$(WORK_DIR)/bin/myproj
7
8  before_release:
9          test -d bin || mkdir -p bin
10         test -d $(OBJ_DIR) || mkdir -p $(OBJ_DIR)
```

1. However, "./obj/" and "./bin/" are not ready
2. We can test whether these directories exist, if not the directories will be created

Compile the source file

```
1 $(OBJ_DIR)/mylib.o: mylib.c
2         $(CC) -c mylib.c -o $(OBJ_DIR)/mylib.o
```

- Instruction "$(OBJ_DIR)/mylib.o" compiles "mylib.c"
- The compilation relies on file "mylib.c"
- The indentation should be by "Tab"
- We can do so for all the source files
- The resulting file is put to "./obj/mylib.o"

```
1 $(OBJ_DIR)/mylib.o: mylib.c
2         $(CC) -c mylib.c -o $(OBJ_DIR)/mylib.o
3
4 $(OBJ_DIR)/myproj.o: myproj.c
5         $(CC) -c myproj.c -o $(OBJ_DIR)/myproj.o
```

Link the source file

```
1 release: $(OBJ_RELEASE)
2         $(LD) -o $(RELEASE) $(OBJ_RELEASE)
```

- The project will be linked with mylib.o and myproj.o
- The list of object files are kept in "$(OBJ_RELEASE)"
- $(LD) calls "gcc"
- The target is specified by "$(RELEASE)"

Build the whole project

```
1 release: before_release $(OBJ_RELEASE)
2         $(LD) -o $(RELEASE) $(OBJ_RELEASE)
```

- The instruction "release" relies on another two intructions
- "before_release" and "$(OBJ_RELEASE)"
- "$(OBJ_RELEASE)" are a list of instructions
  1. Run instruction "before_release"
  2. Run list of instructions in "$(OBJ_RELEASE)"
  3. Run $(LD) -o $(RELEASE) $(OBJ_RELEASE)

Clean the object files

- In some cases, we may want to clean the object files

```
1 clean:
2         rm -rf $(OBJ_DIR)/*.o
3         rm -rf $(RELEASE)
```

- We call command “rm”
- We label the instruction as “clean”

A Complete Makefile

```
4  WORK_DIR=.
5  CC=gcc
6  LD=gcc
7  OBJ_DIR=$(WORK_DIR)/obj
8  OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
9  RELEASE=$(WORK_DIR)/bin/myproj
10
11 $(OBJ_DIR)/mylib.o: mylib.c
12         $(CC) -c mylib.c -o $(OBJ_DIR)/mylib.o
13
14 $(OBJ_DIR)/myproj.o: myproj.c
15         $(CC) -c myproj.c -o $(OBJ_DIR)/myproj.o
16
17 before_release:
18         test -d bin || mkdir -p bin
19         test -d $(OBJ_DIR) || mkdir -p $(OBJ_DIR)
20
21 release: before_release $(OBJ_RELEASE)
22         $(LD) -o $(RELEASE) $(OBJ_RELEASE)
```

Makefile

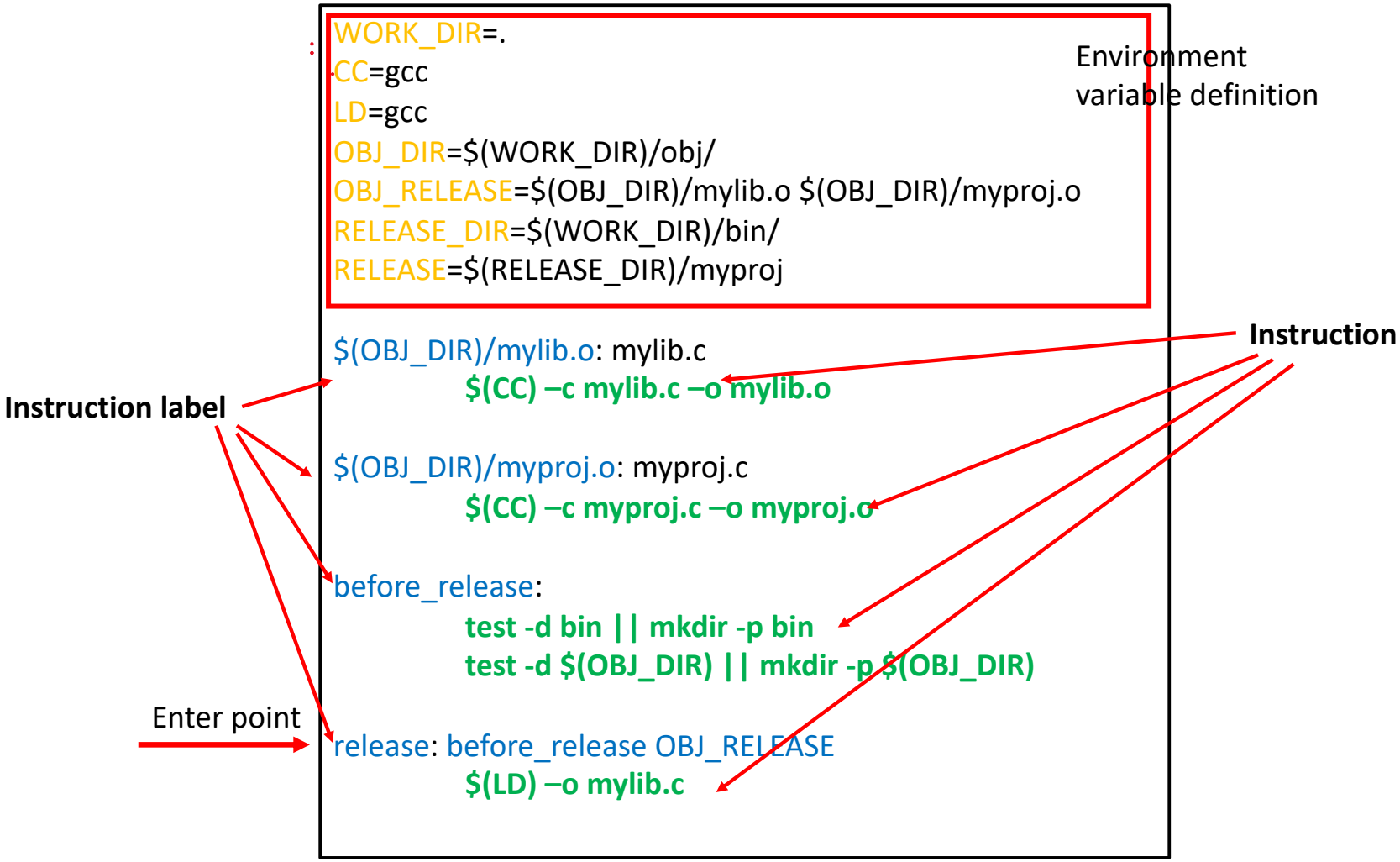
A Complete Makefile

```
23 clean:
24         rm -rf $(OBJ_DIR)/*.o
25         rm -rf $(RELEASE)
```

Makefile

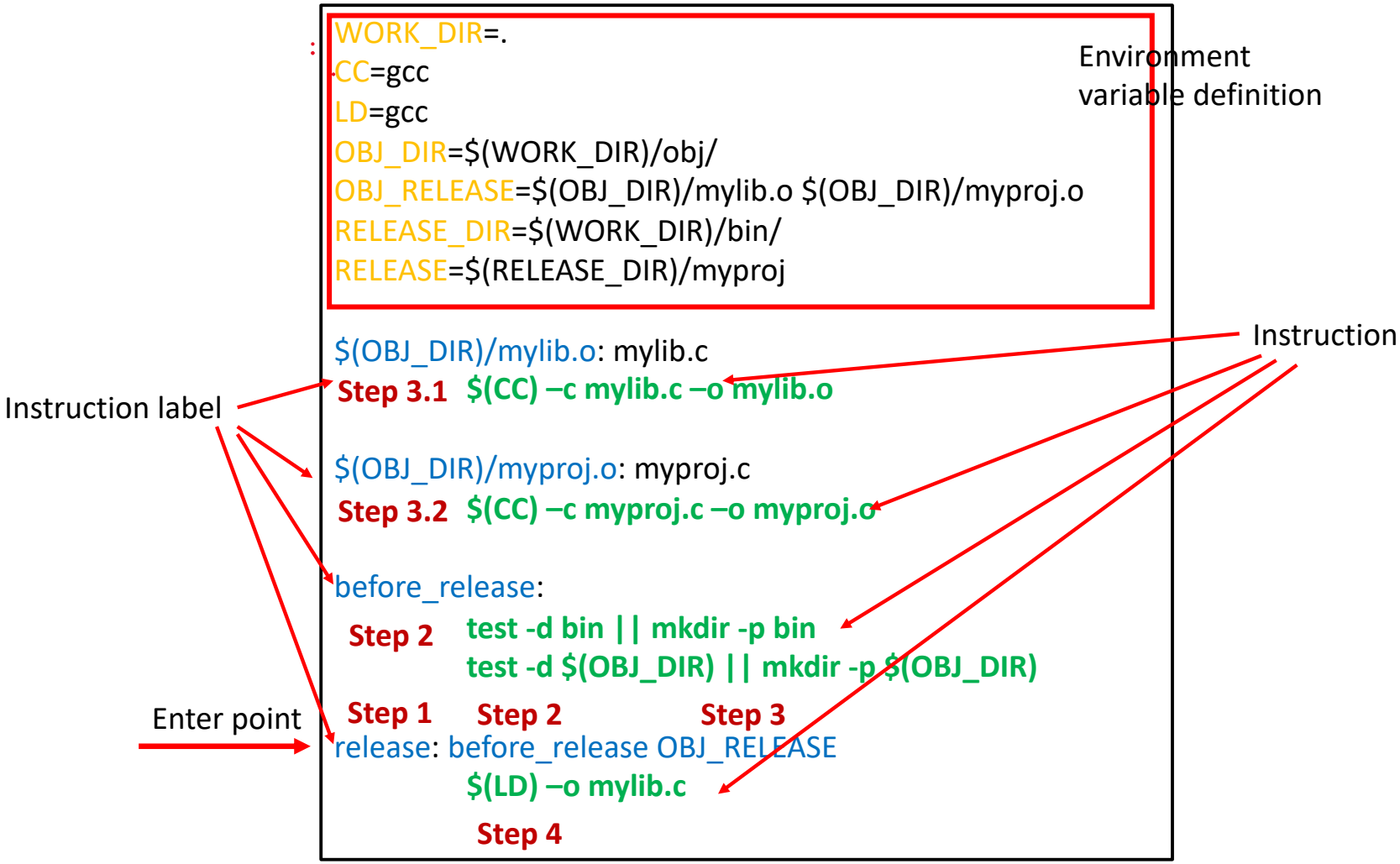
- Five major sections
  1. Define the environment variables
  2. Prepare directories
  3. Instructions of compiling source files to object files
  4. Link object files to target binary executable or library
  5. Instructions to clean the object files

Running Flow inside Makefile (1)



- Run command "make release"

Running Flow inside Makefile (2)



- Run command "make release"

Add libraries in Makefile (1)

- We may need either static or dynamic libraries or both

```c
#include <math.h>
#include "mylib.h"
#include <stdio.h>
int main(){
   float x = 3.4;
   int a = 5;
   float y = square(x);
   float z = sqrt(x);
   return 0;
}
```

myproj.c

- For this code, we should compile it by

```
gcc -o myproj myproj.o mylib.o -lm
```

```
1  WORK_DIR=.
2  CC=gcc
3  LD=gcc
4  LDFLAGS= -lm
5  OBJ_DIR=$(WORK_DIR)/obj
6  OBJ_RELEASE=$(OBJ_DIR)/mylib.o $(OBJ_DIR)/myproj.o
7  RELEASE=$(WORK_DIR)/bin/myproj
8
9  $(OBJ_DIR)/mylib.o: mylib.c
10         $(CC) -c mylib.c -o $(OBJ_DIR)/mylib.o
11
12 $(OBJ_DIR)/myproj.o: myproj.c
13         $(CC) -c myproj.c -o $(OBJ_DIR)/myproj.o
14
15 before_release:
16         test -d bin || mkdir -p bin
17         test -d $(OBJ_DIR) || mkdir -p $(OBJ_DIR)
18
19 release: before_release $(OBJ_RELEASE)
20         $(LD) $(LDFLAGS) -o $(RELEASE) $(OBJ_RELEASE)
```

Makefile

Outline

Why cmake?

- However, writing a Makefile line-by-line is still too sweaty
- There are several convenient ways
  1. "cbp2make"[1]
     - It works with CodeBlocks
     - Command: cbp2make -in project.cbp -out Makefile
  2. cmake[2]
     - It is a powerful cross-platform tool for C/C++ project compilation, test, and installation
     - Based on a "CMakeLists.txt" input file, it produces "Makefile"

[1] https://sourceforge.net/projects/cbp2make/
[2] https://cmake.org/

About cmake

- It is another useful tool
- It helps to produce the “Makefile”
- The cmake requires another simpler script “CMakeLists.txt”
- Compared to “Makefile”, it is a super script and easier to compose

Compose a "CMakeLists.txt" (1)

```
1 cmake_minimum_required (VERSION 2.8)
```

CMakeLists.txt

1. This cmake setting is put in command(value) pattern
2. This is the way set values for environment variables supported by cmake
3. Here we specify the minimum required cmake version is "VERSION 2.8"

Compose a "CMakeLists.txt" (2)

```
1 cmake_minimum_required (VERSION 2.8)
2
3 project (proj1)
```

CMakeLists.txt

1. Here we specify the target project name as "proj1"
2. After compilation, the name of our executable will be "proj1"

Compose a "CMakeLists.txt" (3)

```
1 cmake_minimum_required (VERSION 2.8)
2
3 project (proj1)
4
5 add_executable(proj1 myproj.c mylib.c)
```

CMakeLists.txt

1. "add_executable" allows us to list out all C/C++ source files
2. The leading file name is the target file name "proj1"

Compose a "CMakeLists.txt" (4)

```
1 cmake_minimum_required (VERSION 2.8)
2
3 project (proj1)
4
5 add_executable(proj1 myproj.c mylib.c)
```

CMakeLists.txt

1. We name this text script file as "CMakeLists.txt"
2. Put it to the same folder as the source files
3. Using "mkdir" to make a sub folder "build" under the same folder
4. "cd build"
5. "cmake ../"

- After the above steps, one could see "Makefile" under build folder

Compose a "CMakeLists.txt" (5)

```
1 cmake_minimum_required (VERSION 2.8)
2
3 project (proj1)
4
5 add_executable(proj1 myproj.c mylib.c)
```

CMakeLists.txt

- Under the "build" folder, one will see "CMakeFiles" folder
- Where the object files will be saved
- Run command "make", you get the file compiled

More options in "CMakeLists.txt"

```
1  cmake_minimum_required (VERSION 2.8)
2
3  project (proj1)
4
5  set(CMAKE_BUILD_TYPE "Release")
6  #set(CMAKE_BUILD_TYPE "Debug")
7  set(CMAKE_C_FLAGS_RELEASE "$ENV{CFLAGS} -O3 -Wall")
8  #set(CMAKE_C_FLAGS_DEBUG "$ENV{CFLAGS} -O0 -Wall -g -ggdb")
9
10 add_executable(proj1 myproj.c mylib.c)
```

CMakeLists.txt

- Command "set" is comparable to "=" in a "Makefile"
- Here we set our build type is "Release", otherwise could be "Debug"
- You can also specify the compilation flags

Add SHARED libraries in "CMakeLists.txt"

```
1 cmake_minimum_required (VERSION 2.8)
2
3 project (proj1)
4
5 add_library(libm.so SHARED IMPORTED)
6
7 add_executable(proj1 myproj.c mylib.c)
```

CMakeLists.txt

- Command "set" is comparable to "=" in a "Makefile"
- Here we set our build type is "Release", otherwise could be "Debug"
- You can also specify the compilation flags

Build STATIC library (1)

```
#ifndef MYMATH_H
#define MYMATH_H
float sqrt_nwton(float a);
#endif
```

mymath.h

```
#include <stdio.h>
#include "mymath.h"
float sqrt_nwton(float a){
   float b = 1.2, c = b, err = 1.0;
   if(a < 0){
      printf("The input %f must be non-negative!\n", a);
      return 0;
   }
   do{
      c   = b; b   = (b + a/b)*0.5;
      err = b > c?(b-c):(c-b);
   }while(err > 0.00001);
   return b;
}
```

mymath.c

Build STATIC library (2)

```
1 cmake_minimum_required(VERSION 2.8)
2 project(mymath)
3
4 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} --std=gnu17")
5
6 set(SOURCE_FILES mymath.c mymath.h)
7 add_library(mymath STATIC ${SOURCE_FILES})
```

CMakeLists.txt

- List out all the files to be compiled by "set"
- We actually define a variable "**SOURCE_FILES**"
- The library name is specified by "add_library"
- "STATIC" in command "add_library" tells "static library"
- If we replace "STATIC" with "SHARED", a dynamic/shared library is built

Link with your own STATIC library (1)

```
13 #include <stdio.h>
14 #include "mymath.h"
15 int main(){
16    float a = 4.5;
17    float b = sqrt_nwton(a);
18    printf("sqrt(a) = %.4f\n", b);
19    return 0;
20 }
```

main.c

- The library "libmymath.a" is copied to "libs" under source folder
- The header "mymath.h" is copied to "include" under source folder

Link with your own STATIC library (2)

```
1 cmake_minimum_required(VERSION 2.8)
2 project(proj3)
3
4 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} --std=gnu17")
5 include_directories(${CMAKE_SOURCE_DIR}/include)
6 link_directories(${CMAKE_SOURCE_DIR}/libs)
7 add_executable(proj3 main.c)
8 target_link_libraries(proj3 libmymath.a)
```

CMakeLists.txt

- Specify the directory for header files "include_directories"
- Specify the directory for header files "link_directories"
- Perform linking by "target_link_libraries"
- This works for both static and dynamic library