

C Programming

Lecture 11: make & Makefile



Lecturer: *Dr. Wan-Lei Zhao*
Autumn Semester 2022

- 1 Build Project with Make
- 2 Build Project with CMake

Why make? (1)

```
1 #ifndef MYLIB_H
2 #define MYLIB_H
3 int isodd(int x);
4 float square(float x);
5 #endif
```

mylib.h

```
1 #include "mylib.h"
2 float square(float x){
3     return x*x;
4 }
5
6 int isodd(int x){
7     if(x%2 != 0)
8         return 1;
9     else
10         return 0;
11 }
```

mylib.c

```
1 #include "mylib.h"
2 #include <stdio.h>
3 int main(){
4     float x = 3.4;
5     int a = 5;
6     float y = square(x);
7     if(isodd(a))
8     {
9         printf("%d is odd\n", a);
10    }
11    return 0;
12 }
```

main.c

```
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
```

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 instructions 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

VARIABLE_NAME = 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

`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 and make them if necessary

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 instructions
- “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

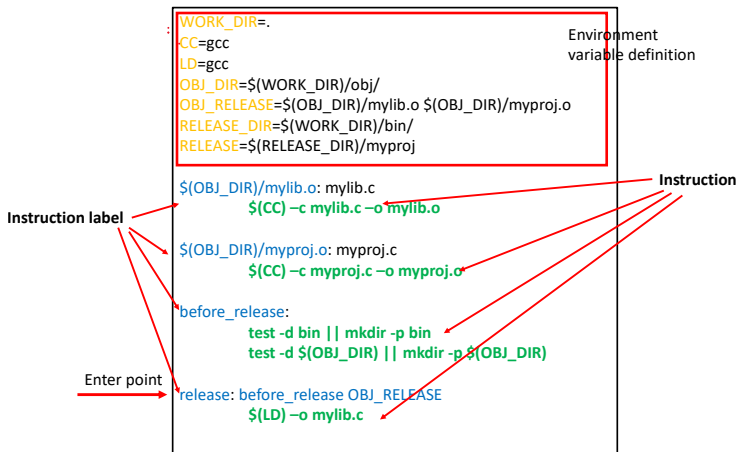
A Complete Makefile

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

Makefile

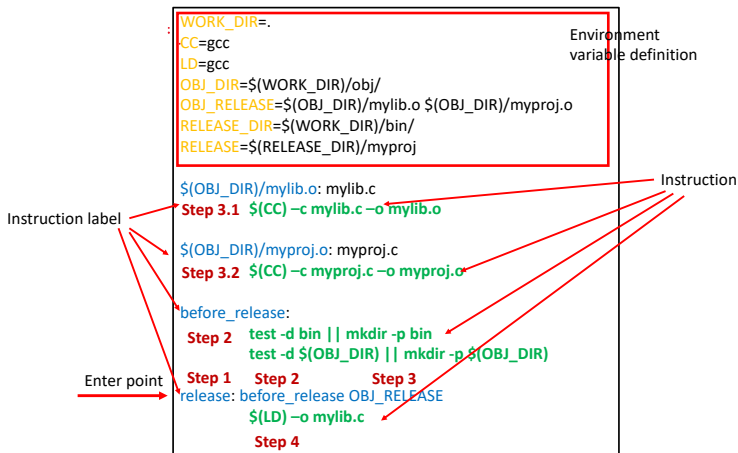
- Five major sections
 - 1 Define the environment variables
 - 2 Prepare directories
 - 3 Instructions of compiling source files to object files
 - 4 Link object files to target binary executable or library
 - 5 Instructions to clean the object files

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

```
1 #include <math.h>
2 #include "mylib.h"
3 #include <stdio.h>
4 int main(){
5     float x = 3.4;
6     int a = 5;
7     float y = square(x);
8     float z = sqrt(x);
9     return 0;
10 }
```

myproj.c

- For this code, we should compile it by

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

Add libraries in Makefile (2)

```
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)
```

- 1 Build Project with Make
- 2 Build Project with CMake

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 cmake²
 - 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/>



- 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)

```
1 #ifndef MYMATH_H
2 #define MYMATH_H
3 float sqrt_nwton(float a);
4 #endif
```

mymath.h

```
1 #include <stdio.h>
2 #include "mymath.h"
3 float sqrt_nwton(float a){
4     float b = 1.2, c = b, err = 1.0;
5     if(a < 0){
6         printf("The input %f must be non-negative!\n", a);
7         return 0;
8     }
9     do{
10         c = b; b = (b + a/b)*0.5;
11         err = b > c?(b-c):(c-b);
12     }while(err > 0.00001);
13     return b;
14 }
```

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