Makefile guide by Shachar Tsrafati

For C/C++

Contents

Note:

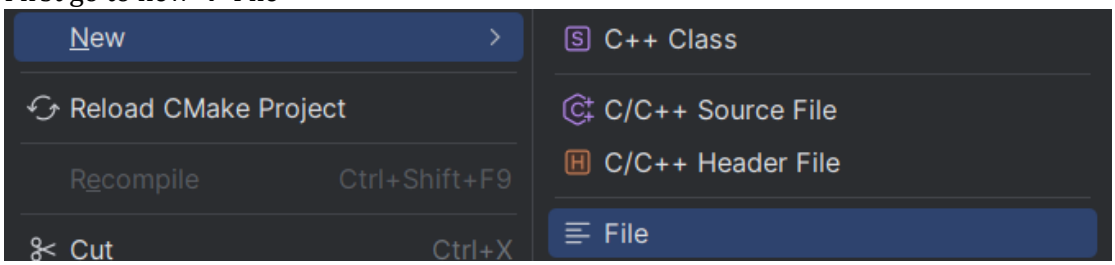
Makefiles are for both C and C++. Most of the commands are for Linux

For C: we will use gcc, Example: gcc -c file.c -o file.o

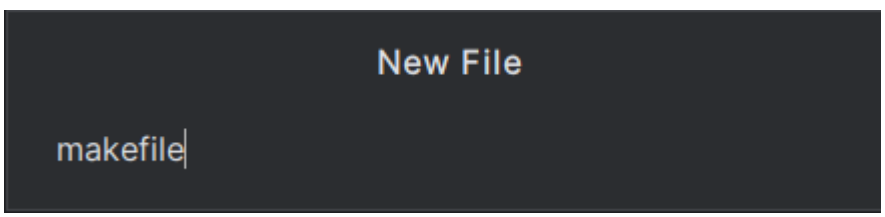
For C++: we will use g++, Example: g++ -c file.cpp -o file.o

How to create Makefile

I will show how to create makefile inside CLION.
First go to new → File



And simply type makefile



Note: typing makefile with uppercase or camelCase can cause problems, best to create file with only lowercase.

Basic Makefile

Let's start with basic Makefile without any shortcuts

First, we will compile each file and then we will link them together and create an executable file with the name output.

```
other.o: other.cpp
          g++ -c other.cpp -o other.o
main.o: main.cpp
          g++ -c main.cpp -o main.o

output: other.o main.o
          g++ other.o main.o -o output
```

Shortcuts

Let's be honest with ourselves, we all love shortcuts. Luckly for us we got shortcuts we can use with Makefiles.

Let's start with simple shortcut, we got 2 files, main.cpp and other.cpp
We want to compile both, so we can do like we did above. But if we got 100 files ? do we really have to write right now 100 lines for each file ?. NO !
We will use shortcut !

To make that kind of shortcut, first we need to declare something. I will call it SRCS,

SRCS will be shortcut for every time we want to do something with both files

```
SRCS = main.cpp other.cpp
```

How do we use the shortcut to compile more than one file ?

We are going to use OTHER SHORTCUT

```
%.o: %.cpp
   g++ -g -c $< -o $@
```

Now you are asking yourself. "Ok but how this helped us ?" and you are 100% correct, We need ANOTHER SHORTCUT to connect everything together.

Before we will connect all together, We never compile any file. Let's add ANOTHER SHORTCUT
We will call it OBJS and tell it to take the files we want and make an .o files

```
OBJS = $(SRCS:.cpp=.o)
```

Now with all the shortcuts we can combined them together and we will get.

```
CXX = g++
CXXFLAGS = -g

OUTPUT = output

SRCS = main.cpp other.cpp

OBJS = $(SRCS:.cpp=.o)

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

$(OUTPUT) : $(OBJS)
	$(CXX) $(CXXFLAGS) -o $(OUTPUT) $^
```

Symbols Shortcuts (Wildcards)

We used some weird symbols for shortcuts before, let's explain some of the most common symbols that we will use in almost every Makefile.

Definition	Usages	Example
CXX = g++ CXXFLAGS = -Wall	$(CXX) $(CXXFLAGS) -c main.cpp -o main.o	Variables for shortcut

Automatic Variables:

$@ → target name

```
main.o: main.cpp
	g++ -c main.cpp -o $@
```

$< → The First Prerequisite
This is the **first dependency listed** after the colon (:).

```
main.o: main.cpp
	g++ -c $< -o $@
// same as
g++ -c main.cpp -o main.o
```

$^ → All Prerequisites
Represents **all dependencies**, space separated. Useful when linking.

```
output: main.o utils.o
	g++ $^ -o $@
// same as
g++ main.o utils.o -o output

$^ becomes main.o utils.o
$@ becomes output
```

$? → Newer Prerequisites

Refers to **dependencies that are newer than the target**.
Useful for **incremental builds**, when only changed files should be recompiled or relinked.

```
output: main.o utils.o
          g++ $? -o $@

//if only utils.o was modified, then:

g++ utils.o -o output

$? becomes only the modified .o files.
```

$* → Target Without Extension

$* is rarely needed directly. Instead, prefer $@ which clearly represents the current target.

Only works in rules like:

```
%.o: %.cpp

//Inside that rule, $* is the base name (file name without extension).

%.o: %.cpp
     g++ -c $< -o $*.o

//For main.cpp, this becomes:

g++ -c main.cpp -o main.o

$* becomes main
```

Clean

When we make the Makefile we will create a lot of .o files.

Do we need to clean it manual each time ?. Nah!

We will make a command that we can call to clean it for us.

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

Phony

Adding .PHONY to a target will prevent Make from confusing the phony target with a file name.
Good example is for clean or output.

```
.PHONY output clean … etc
```

When we do phony then we can use it inside the terminal.

Libraries

A library is a collection of code (functions, global variables, etc.) written (sometimes also compiled) by someone else, that you may want to use.

Examples: C++'s standard libraries ,Math library ,Graphic libraries.

Static libraries:

- linked with your executable at compilation time
- standard unix suffix: .a (windows: .lib)

Shared libraries:

- loaded by the executable at run-time
- standard unix suffix: .so (windows: .dll)

Static libraries:

To compile object files to be static libraries we will use the keyword "ar"

```
ar -rcs libmine.a ctest.o ctest1.o ctest2.o
```

The flag "-rcs" is option to create the files without a warning and it will replace any preexisting object files in the library with the same name. + add "s" in the end of the file.

How to include the library in an executable:

```
gcc prog.cpp -lmine -L. -o myprog
```

The flag "-L" = look in directory for library files

The dot after the L represents the current working directory.
We can give it a path to target:

```
-L./Libs_Folder // for example
```

The “-l” flag tells the linker to link with a library named libmine.a or libmine.so.
The word mine is the **library name**.
The linker automatically adds the lib prefix and .a/.so extension.

```
CXX = g++
CXXFLAGS = -g -Wall -Werror -std=c++17


# Output names
OUTPUT = main
LIB_OUTPUT = lib/libmylib.a

# Source files
MAIN_SRC = main.cpp
LIB_SRCS = libs/foo.cpp libs/bar.cpp

# Object files
MAIN_OBJ = $(MAIN_SRC:.cpp=.o)
LIB_OBJS = $(LIB_SRCS:.cpp=.o)

# Default target
all: $(OUTPUT)

# Build final executable from main.o and the static library

$(OUTPUT): $(MAIN_OBJ) $(LIB_OUTPUT)

          $(CXX) $(CXXFLAGS) -o $@ $^

# Create the static library (archive .o files)

$(LIB_OUTPUT): $(LIB_OBJS)

          ar rcs $@ $^

# Compile .cpp to .o for all files

%.o: %.cpp

          $(CXX) $(CXXFLAGS) -c $< -o $@

# Clean everything

.PHONY: clean

clean:

          rm -f *.o libs/*.o $(LIB_OUTPUT) $(OUTPUT)
```

Dynamic Libraries:

To compile object files for a **shared dynamic library**, use the “-fPIC” flag to produce position-independent code, then use -shared to create the .so file:

```
g++ -Wall -fPIC -c utils.cpp

g++ -shared utils.o -o libutils.so
```

- -fPIC = Generate **position-independent code** (required for shared libraries)
- -shared = Create a .so (shared object) instead of an executable

How to include the library in an executable:

```
g++ main.cpp -L. -lutils -o FinalProg
```

- -L. → Search for libraries in the **current directory**
- -lutils → Link with libutils.so
  (the linker adds the lib prefix and .so suffix automatically)

How to run the program with the shared library:

```
LD_LIBRARY_PATH=. ./FinalProg
```

```
CXX = g++
CXXFLAGS = -g -Wall -Werror -std=c++17 -fPIC
LDFLAGS = -shared

OUTPUT = main
LIB_OUTPUT = lib/libmylib.so

MAIN_SRC = main.cpp
LIB_SRCS = libs/foo.cpp libs/bar.cpp

MAIN_OBJ = $(MAIN_SRC:.cpp=.o)

LIB_OBJS = $(LIB_SRCS:.cpp=.o)

.PHONY: all clean run

all: $(OUTPUT)

$(OUTPUT): $(MAIN_OBJ) $(LIB_OUTPUT)

	$(CXX) $(CXXFLAGS) -Llib -lmylib -o $@ $(MAIN_OBJ)

$(LIB_OUTPUT): $(LIB_OBJS)

	$(CXX) $(CXXFLAGS) $(LDFLAGS) -o $@ $^

%.o: %.cpp

	$(CXX) $(CXXFLAGS) -c $< -o $@

run:

	LD_LIBRARY_PATH=lib ./$(OUTPUT)

clean:

	rm -f *.o libs/*.o $(OUTPUT) $(LIB_OUTPUT)
```