GCC 와 GNU make 를 이용한 C++ 프로젝트 설정

최종 수정 2019/03/04 : 정유철 <ycjung@postech.ac.kr>

개요

이 문서에서는 빌드 자동화 툴을 왜 사용하는지 알아봅니다. 또한, 빌드 자동화 툴 중 하나인 GNU make 를 사용하여 C++ 프로젝트를 설정하는 방법을 설명합니다.

빌드 자동화 툴을 사용하는 이유

일반적으로 소프트웨어 프로젝트가 개발되고 배포될 때는 .c, .cpp, .h 등 소스코드 외에도 각 소스코드를 어떻게 처리하여 최종 결과물을 낼지 자동화하는 빌드 스크립트를 같이 개발합니다. 빌드 스크립트가 없다면 다른 사람이 소프트웨어의 컴파일 과정을 재현할 수 없기 때문에 문제가 발생합니다.

예시 1) CSED232 의 과제를 제출하려는 한 수강생이 .h 파일과 .cpp 파일을 기능별로 폴더를 만들어 구분해 놓았습니다. 조교는 각 소스파일들을 수강생으로부터 전달받았습니다. 그런데 각 .cpp 를 컴파일 할 때, .h 헤더파일들을 어디에서 찾을지 알 수 없습니다. G++에 이 헤더파일들의 위치를 알려줘야 하는데, 그 경로가 알려져 있지 않기 때문입니다. 이 경우 직접 그 경로를 원 작성자로부터 전달받아, g++ 명령어에 수동으로 입력해줘야 할 수도 있습니다.

예시 2) 어떤 프로젝트는 컴파일 중간에 인터넷으로부터 특정 파일을 다운로드 하는 것을 요구하기도 합니다. 각 파일들을 언제 어떻게 받고, 빌드에 어떤 방식으로 적용할지를 자동화하면 매우 편할 것입니다.

예시 3) 어떤 프로젝트는 소스 코드의 양이 매우 많아 빌드를 g++을 직접 실행하여 하는 것이 거의 불가능합니다.

예시 4) 어떤 프로젝트는 외부 라이브러리에 의존하기 때문에, 해당 라이브러리의 코드를 어떤 방식으로 Link 해줄지 명시해줘야 합니다.

소프트웨어 빌드는 복잡한 문제입니다. 이를 해결하기 위한 도구들이 커뮤니티에 존재합니다. 우리가 과제에서 사용할 빌드 자동화 툴은 GNU make 입니다. 이 외에도 현대에는 다양한 기능을 제공하는 CMake, Bazel, Mason 등 툴이 존재하지만, 우리 과목의 과제는 GNU make 를 사용하는 것을 원칙으로 합니다.

이 뒤는 예시를 중심으로, C++ 프로젝트를 GNU make 와 함께 구성하는 방법을 설명하겠습니다.

GNU make 의 워크플로우

1. 프로젝트 폴더에 소스코드 작성
2. 프로젝트 폴더에 빌드 과정을 명시한 특수 문법의 Makefile 생성
3. ‘make’ 명령어로 빌드 수행

Makefile 의 구조

```
target … : prerequisites …
        recipe
        …
        …
```

Target : recipe 들을 수행함으로써 만들고자 하는 최종 target 의 이름. 주로 오브젝트 파일의 이름이나, 실행 파일의 이름이 된다.
Prerequisite : 해당 target 을 만들기 위해 먼저 만들어져야만 하는 파일들의 나열
Recipe : 해당 target 을 만들기 위해서 수행해야 하는 명령어의 나열

예시 1 : CPP 파일 단 하나

프로젝트 내의 소스 코드 구조는 다음과 같습니다.

```
ycjung@DESKTOP-AF2K4BO MINGW64 ~/example1
$ ls -al
합계 5
drwxr-xr-x 1 ycjung 없음  0 3월  4 15:39 .
drwxr-xr-x 1 ycjung 없음  0 3월  4 15:39 ..
-rw-r--r-- 1 ycjung 없음 91 3월  4 15:39 main.cpp
```

main.cpp

```cpp
#include <iostream>

int main()
{
        std::cout << "Hello, World!" << std::endl;
        return 0;
}
```

이 소스코드로부터, ‘example1’이라는 이름의 target 을 만듭니다.

Makefile

```
example1: main.cpp
	g++ -o example main.cpp
~
~
~
~
~
```

example1/ 폴더에 Makefile 을 작성/저장 후, ‘make’ 명령어를 실행하면 example.exe 가 생성되는 것을 확인할 수 있습니다.

```
ycjung@DESKTOP-AF2K4BO MINGW64 ~/example1
$ make
g++ -o example main.cpp

ycjung@DESKTOP-AF2K4BO MINGW64 ~/example1
$ ls -al
합계  374
drwxr-xr-x 1 ycjung 없음      0 3월    4 15:43 .
drwxr-xr-x 1 ycjung 없음      0 3월    4 15:43 ..
-rwxr-xr-x 1 ycjung 없음 369903 3월    4 15:43 example.exe
-rw-r--r-- 1 ycjung 없음     91 3월    4 15:39 main.cpp
-rw-r--r-- 1 ycjung 없음     44 3월    4 15:42 Makefile

ycjung@DESKTOP-AF2K4BO MINGW64 ~/example1
$
```

예시 2 : CPP 파일 두 개

프로젝트 내의 소스 코드 구조는 다음과 같습니다.

```
ycjung@DESKTOP-AF2K4BO MINGW64 ~/example2
$ ls -al
합계 11
drwxr-xr-x 1 ycjung 없음    0 3월    4 15:47 .
drwxr-xr-x 1 ycjung 없음    0 3월    4 15:47 ..
-rw-r--r-- 1 ycjung 없음  109 3월    4 15:46 main.cpp
-rw-r--r-- 1 ycjung 없음   65 3월    4 15:47 my_lib.cpp
-rw-r--r-- 1 ycjung 없음   26 3월    4 15:46 my_lib.h
```

Main.cpp (my_lib.h 에 정의된 함수를 사용합니다.)

```
#include <iostream>
#include "my_lib.h"

int main()
{
        std::cout << my_add(1, 2) << std::endl;
        return 0;
}
```

My_lib.h

```
int my_add(int a, int b);
```

My_lib.cpp

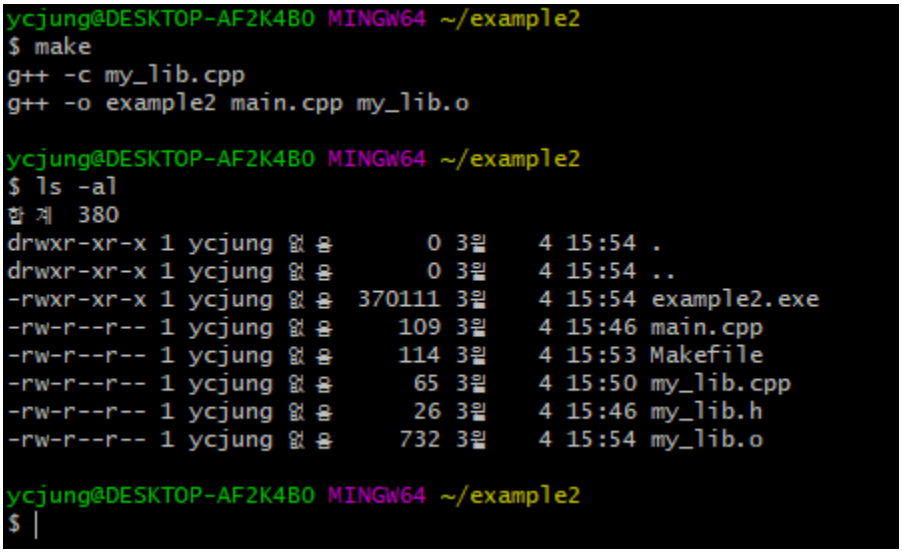
```
#include "my_lib.h"

int my_add(int a, int b)
{
        return a + b;
}
```

my_lib.cpp 로부터 my_lib.o 라는 target 을 먼저 만들고, 이 target 에 의존하는 최종 target 실행파일인 example2 를 만들겠습니다.

Makefile

```
example2: my_lib.o main.cpp
        g++ -o example2 main.cpp my_lib.o

my_lib.o: my_lib.cpp my_lib.h
        g++ -c my_lib.cpp

~
~
```

example2/ 폴더에 Makefile 을 작성/저장 후, ‘make’ 명령어를 실행하면 example.exe 가 생성되는 것을 확인할 수 있습니다. Makefile 에 Rule 이 여러개인 것에 주목하면 좋습니다. Rule 이 여러 개일 때, ‘make’ 명령어는 가장 위에 명시된 Rule 을 수행하려 시도합니다. 만약 my_lib.o 을 타겟으로 하는 Rule 이 맨 위에 있다면, ‘make’는 my_lib.o 만을 컴파일하고 수행을 끝냅니다. My_lib.o 를 컴파일하는데에는 example2 target 이 필요없기 때문입니다.



예시 3 : 복잡한 소스 구조, 더 많은 CPP 파일

파일 구조는 다음과 같이 되어있습니다.

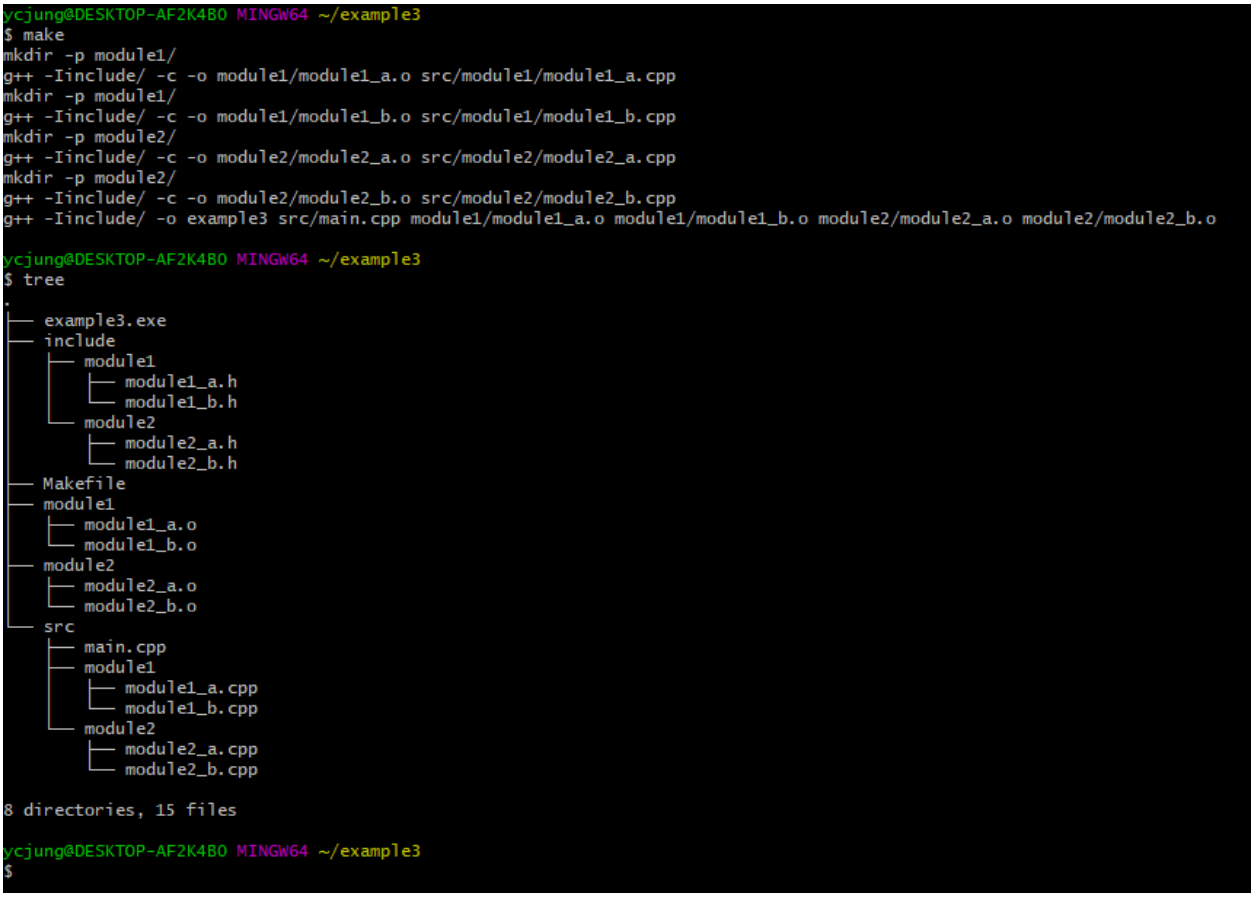
```
ycjung@DESKTOP-AF2K4BO MINGW64 ~/example3
$ tree
.
├── include
│   ├── module1
│   │   ├── module1_a.h
│   │   └── module1_b.h
│   └── module2
│       ├── module2_a.h
│       └── module2_b.h
└── src
    ├── main.cpp
    ├── module1
    │   ├── module1_a.cpp
    │   └── module1_b.cpp
    └── module2
        ├── module2_a.cpp
        └── module2_b.cpp

6 directories, 9 files
```

소스코드의 내용은 생략합니다. Main.cpp 의 main() 에서 module1_a, module1_b, module2_a, module2_b 의 함수를 호출하는 구조입니다.

이 프로젝트를 컴파일하기 위해서 다음과 같은 Makefile 을 만들 수 있습니다.

```
example3: src/main.cpp module1/module1_a.o module1/module1_b.o module2/module2_a.o module2/module2_b.o
	g++ -Iinclude/ -o example3 src/main.cpp module1/module1_a.o module1/module1_b.o module2/module2_a.o module2/module2_b.o

module1/module1_a.o: include/module1/module1_a.h src/module1/module1_a.cpp
	mkdir -p module1/
	g++ -Iinclude/ -c -o module1/module1_a.o src/module1/module1_a.cpp

module1/module1_b.o: include/module1/module1_b.h src/module1/module1_b.cpp
	mkdir -p module1/
	g++ -Iinclude/ -c -o module1/module1_b.o src/module1/module1_b.cpp

module2/module2_a.o: include/module2/module2_a.h src/module2/module2_a.cpp
	mkdir -p module2/
	g++ -Iinclude/ -c -o module2/module2_a.o src/module2/module2_a.cpp

module2/module2_b.o: include/module2/module2_b.h src/module2/module2_b.cpp
	mkdir -p module2/
	g++ -Iinclude/ -c -o module2/module2_b.o src/module2/module2_b.cpp
~
```

작성/저장 후 'make'를 실행합니다.



이렇게 example3 라는 결과물을 성공적으로 얻었지만, 문제가 하나 있습니다. Makefile 에 너무 반복되는 문자열이 많다는 것입니다. 지금은 파일의 수가 적어 그렇게 길지 않지만, 소스 파일의 수가 많아지면 파일 이름을 하나씩 적는 것은 거의 불가능합니다. GNU make 는 이러한 문제를 해결하기 위해 여러 기능을 제공합니다. 해당 기능들을 사용하여, Makefile 을 더 간결하게 만들 수 있습니다. 다음은 와일드카드(wildcard), 문자열 치환(patsubst), 매크로(<심볼이름> = ...) 등 기능을 사용하여 더 보기 좋게 꾸민 Makefile 의 예시입니다. 아래의 예시는 맛보기로, 모든 내용을 이해할 필요는 없습니다.

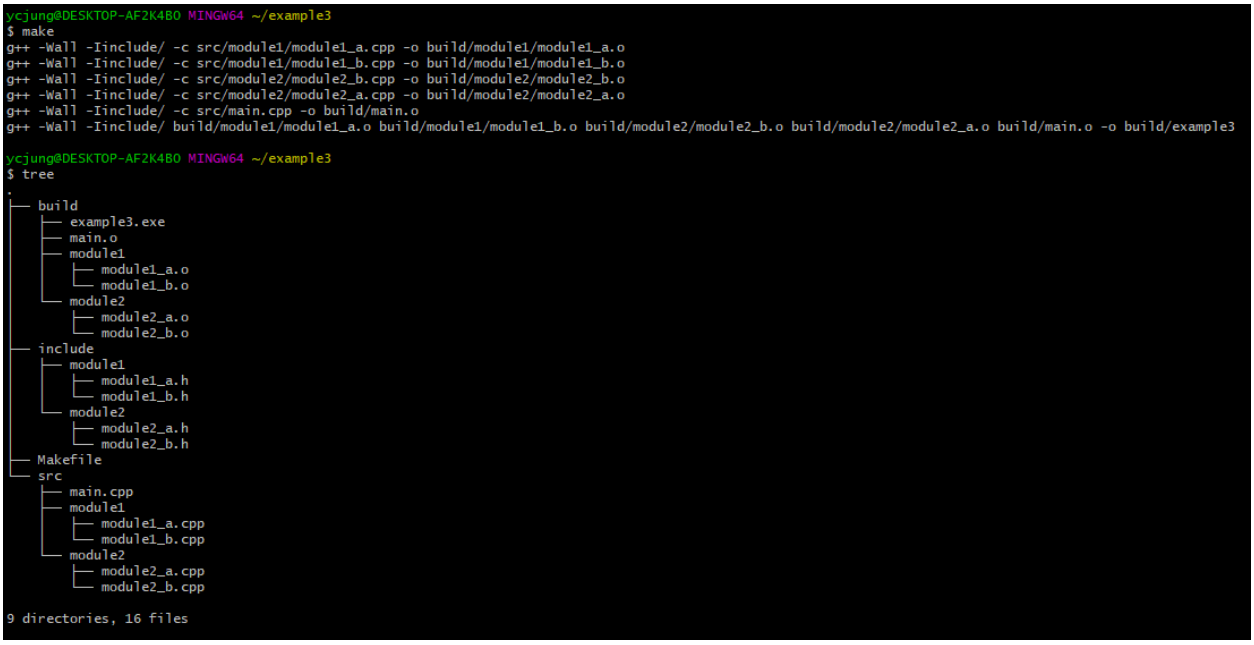
```
SRC_PATH = src/
OUT_PATH = build/
INCLUDE_PATH = include/
MODULE1_REL_PATH = module1/
MODULE2_REL_PATH = module2/
MAIN_SOURCE = $(SRC_PATH)main.cpp
MAIN_OBJECT = $(OUT_PATH)main.o
PROGRAM_NAME= $(OUT_PATH)example3

CXX = g++
CXXFLAGS = -Wall -I$(INCLUDE_PATH)

MODULE1_SOURCES = $(wildcard $(SRC_PATH)$(MODULE1_REL_PATH)*.cpp)
MODULE1_OBJECTS = $(patsubst $(SRC_PATH)%.cpp,$(OUT_PATH)%.o,$(MODULE1_SOURCES))

MODULE2_SOURCES = $(wildcard $(SRC_PATH)$(MODULE2_REL_PATH)*.cpp)
MODULE2_OBJECTS = $(patsubst $(SRC_PATH)%.cpp,$(OUT_PATH)%.o,$(MODULE2_SOURCES))

LIB_SOURCES = $(MODULE1_SOURCES) $(MODULE2_SOURCES)
LIB_OBJECTS = $(MODULE1_OBJECTS) $(MODULE2_OBJECTS)

OBJECTS = $(LIB_OBJECTS) $(MAIN_OBJECT)

$(PROGRAM_NAME): $(OBJECTS)
        @mkdir -p $(@D)
        $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(PROGRAM_NAME)

$(OBJECTS): $(OUT_PATH)%.o: $(SRC_PATH)%.cpp
        @mkdir -p $(@D)
        $(CXX) $(CXXFLAGS) -c $< -o $@
```

위에서 A=b 식으로 정의된 심볼들은, 아래에서 $(A) 형태로 다시 b 형태로 풀어낼 수 있습니다. 이를 이용하여 반복되는 문자열들을 처리했습니다. 이 Makefile 에 사용된 다른 기능인 wildcard, patsubst 등에 대해서는 GNU make 의 공식 매뉴얼을 참조 해주시기 바랍니다.

(https://www.gnu.org/software/make/manual/make.html)

그러나 과제 제출을 할 때는 꼭 이렇게 다채로운 기능을 사용할 필요가 없습니다. 처음 제시한 Makefile 처럼 반복되는 문자열이 많더라도, 최종 .exe 파일을 만들 수 있기만 하면 됩니다.

'make' 실행 결과



결론

Makefile 을 사용하여 다른 사람에게 코드를 배포할 때 빌드를 쉽게 재현하도록 할 수 있습니다. 이 문서에서는 몇 가지 간단한 C++ 프로젝트에서 Makefile 을 작성하는 예시들을 알아보았습니다. 또한 GNU make 의 기능을 잘 활용하면, 더 복잡한 프로젝트에서도 간결한 Makefile 을 작성할 수 있다는 것을 제시했습니다.

주의사항

- Makefile 을 작성할 때, 각 recipe 앞에는 반드시 스페이스가 아닌 "Tab"을 넣어 인덴트를 해야 합니다.
- 'make' 는 별도의 실행인자를 주지 않았을 경우 항상 맨 위의 rule 만을 수행합니다. 따라서 최종 target 을 맨 위에 놓아야 합니다.

GNU make 에 대한 더 자세한 설명은 공식 매뉴얼에서 찾으실 수 있습니다.
(https://www.gnu.org/software/make/manual/make.html)