크로스 플랫폼_CMake

2024.05.12_노하람

크로스 플랫폼

1. 네이티브
2. 크로스 플랫폼
3. 리눅스 크로스 플랫폼

네이티브

1. 네이티브란
2. 네이티브 장단점

네이티브란

- Native
  : 해당 운영체제에 맞는 프로그래밍 언어

- Native App
  : 운영체제에 맞는 프로그래밍 언어를 사용하여 개발하는 앱
  ex) 코틀린-안드로이드 / 스위프트-Xcode

- 해당 운영체제에 맞지 않는 언어를 사용하기 위해서는
  소스 코드를 각각 운영체제에 맞는 컴파일러로 컴파일해야 한다.
  즉, 윈도우/맥/리눅스 버전을 위해선 각각 3번의 컴파일 필요

네이티브의 장단점

- 장점
  - OS에서 제공하는 다양한 API를 효과적으로 활용 가능

- 단점
  - OS 호환이 불가능
  : 같은 기능 개발이여도, 각 OS에 맞춰 재개발 해야함

크로스 플랫폼

1. 크로스 플랫폼이란
2. 크로스 플랫폼 장단점
3. 크로스 플랫폼 지원 도구

크로스 플랫폼이란

- Cross Platform
  : "교차"를 뜻하는 "Corss"와 "Platform"의 합성어로
  다양한 플랫폼에서 사용할 수 있다 라는 뜻

- 크로스 플랫폼앱
  : 하나의 개발 언어로 여러 OS에서 호환이 가능하게 만드는 개발 도구
  ex) 플러터, 리액트 네이티브, 자마린 등

- Java의 경우 크로스플랫폼을 지원
  JVM(Java Virtual Machine)에서 Java 컴파일러가 "바이트 코드 " 로 생성. 원하는 OS에 맞게 설치된 JVM위에서 실행가능

크로스 플랫폼의 장단점

- 장점
  - 하나의 개발 언어와 도구로 여러 OS와 호환이 가능하다
  - 이에 따라 시간절약, 자원절약 가능

- 단점
  - 네이티브 앱만큼 높은 성능 도출 불가
  - OS에서 제공하는 API 활용에 어려움
  - 크로스 플랫폼에 지나친 의존 가능성 있음
    OS에서 새로운 API 업데이트 시, 크로스플랫폼에서 지원해줄때까지 즉시 사용 불가능

크로스 플랫폼 지원 도구

- Unity 3D
  C#기반의 게임엔진 도구. iOS, Android 등 지원

- Unreal Engine
  C++기반의 게임엔진 도구. iOS, Android 등 지원

- 자마린
  C#기반의 웹/모바일 어플리케이션. iOS, Android, Windows 등 지원

- 리액트 네이티브
  자바스크립트 기반의 웹/모바일 어플리케이션. iOS, Android, UWP(Universal Windows Platform) 지원

- 플러터
  웹/모바일,데스크톱 크로스플랫폼. iOS,Android, Windows, Linux 등 지원

리눅스 크로스 플랫폼

1. 리눅스 빌드 시스템
2. gcc
3. make
4. Cmake
5. Make, Cmake의 장단점

리눅스 빌드 시스템

Gcc: GNU Compiler Collection의 약자. 리눅스의 기본 설치된 컴파일러
G++: g++ 컴파일러. 리눅스의 기본 설치된 컴파일러

Make: 여러 단계의 gcc 명령을 Makefile이라는 스크립트로 만들어 한번에 실행.
Incremental Build 지원

Cmake: 중간 단계를 일일이 지정해줘야 하는 Makefile을 좀더 편리하게 만들어줌.
이를 위해 CMakeLists.txt라는 스크립트를 작성

* Incremental Build : 변경된 소스코드만 다시 빌드하는 기능

리눅스 빌드 시스템_gcc

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

=> c++로 만든 main.cpp 파일을 main.o로 만든다

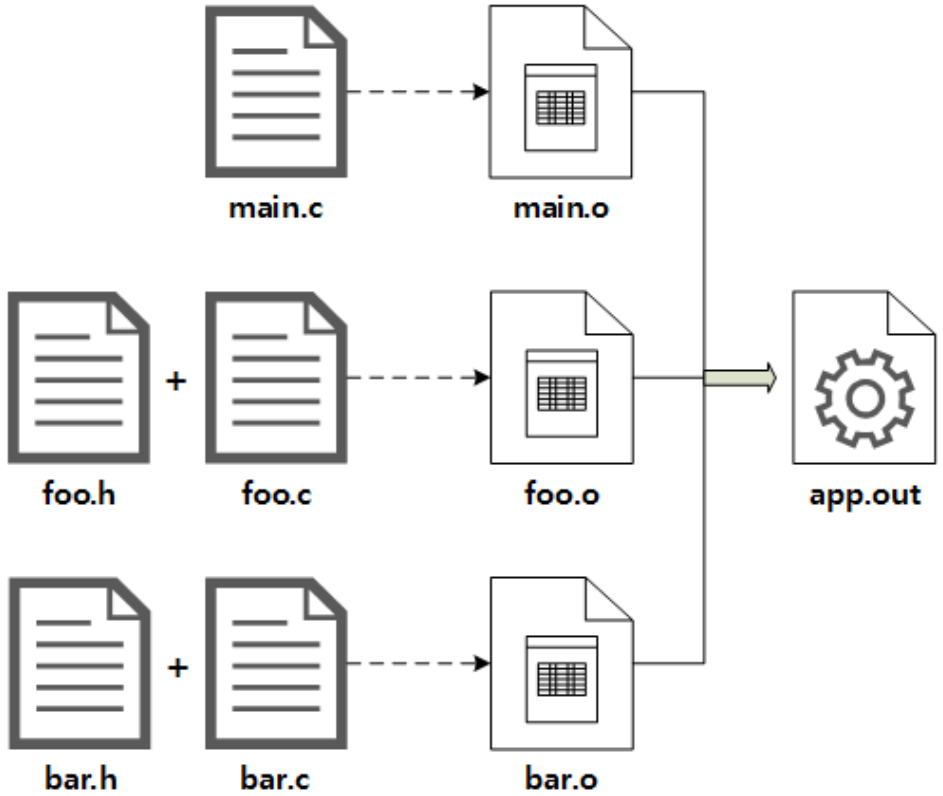
$ g++ -c -o foo.o foo.cpp
=> c++로 만든 foo.cpp 파일을 foo.o로 만든다

$ g++ -c -o bar.o bar.cpp
=> c++로 만든 bar.cpp 파일을 bar.o로 만든다

$ g++ -o myapp main.o foo.o bar.o
=> c++로 만든 main.o foo.o bar.o 파일을 myapp으로 만든다

$ ./myapp

=> myapp 실행



리눅스 빌드 시스템

빌드할 때 마다 단계 수행
해당 작업이 불편하여 makefile 탄생
Makefile에 해당 단계를 미리 작업해놓음

리눅스 빌드 시스템_make

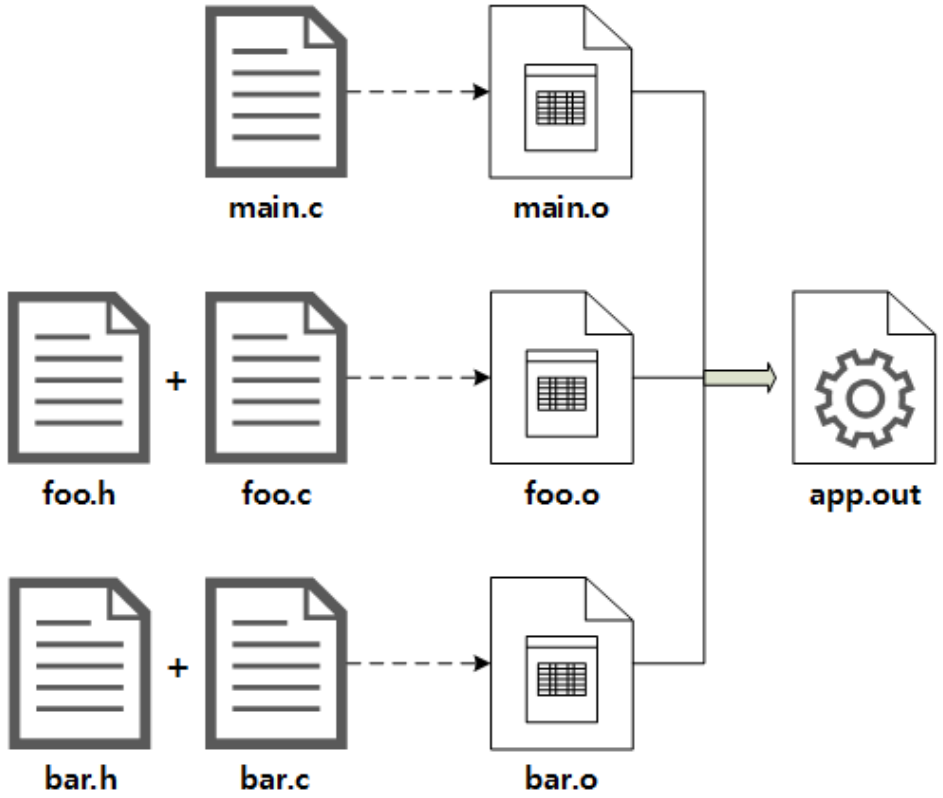
```
## makefile 내용
app.out: main.o foo.o bar.o
		g++ -o app.out main.o foo.o bar.o
main.o: foo.h bar.h main.cpp
foo.o: foo.h foo.cpp
bar.o: bar.h bar.cpp
## makefile 내용
```

$make

=> 메이크 파일을 이용하여 빌드

$./myapp

=> myapp 실행



리눅스 빌드 시스템_make

```
# 컴파일러 및 플래그 설정
CC = g++
CFLAGS = -Wall -Wextra -std=c++11

# 소스 파일 및 실행 파일 이름 설정
SRC = main.cpp
EXECUTABLE = my_program

# 기본 대상 설정
all: $(EXECUTABLE)

# 실행 파일 빌드 규칙
$(EXECUTABLE): $(SRC)
    $(CC) $(CFLAGS) -o $(EXECUTABLE) $(SRC)

# clean 규칙: 생성된 파일 삭제
clean:
    rm -f $(EXECUTABLE)
```

Makefile 예시

리눅스 빌드 시스템

프로젝트 규모가 커지면서
Makefile 손수 작업 불편해짐
Makefile을 만들어주는 툴 탄생

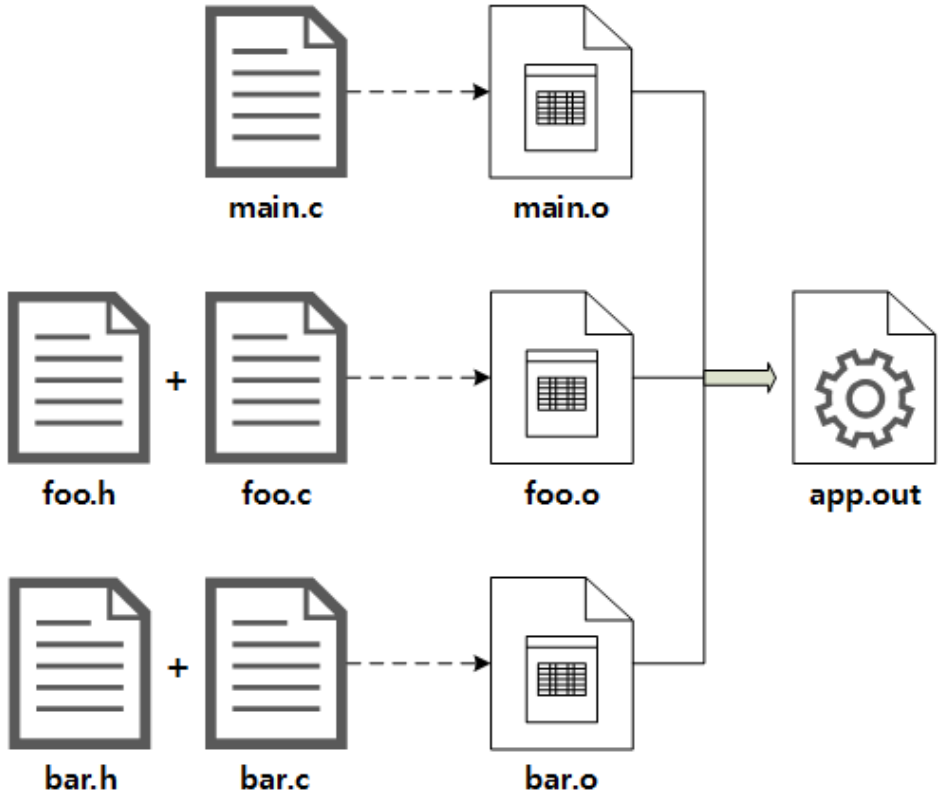
리눅스 빌드 시스템_CMake

CMakeLists..txt 내용
ADD_EXECUTABLE(myapp main.cpp foo.cpp bar.cpp)
CMakeLists..txt 내용

$cmake .
=> 현재 패스에 있는 CMakeLists.txt 를 이용해 cmake 실행
⇒ 결과 [다음과 같은 파일 및 디렉토리 생성]
- CMakeCache.txt
- cmake_install.cmake
- CMakeFiles/Makefile

$make
=> 메이크 파일을 이용하여 빌드

$./myapp
=> myapp 실행



Make

장점

- 유연성: 간단하고 유연한 문법으로 사용자가 원하는 방식으로 구성 가능
- 간단함: 많은 운영체제에서 기본적으로 제공 가능. 익숙한 사용자 많음
- 직관성: 프로젝트의 의존성을 관리하고, 명시적으로 정의하기 때문에 코드의 흐름을 이해하기 쉽다.

단점

- 플랫폼 의존성: 특정 운영체제에 의존적이며, 여러 운영체제에서 동작하기 어려움
- 복잡성: 대규모 프로젝트의 경우 Makefile 손수 작성이 어려움

CMake

장점

- 크로스 플랫폼: 프로젝트를 여러 운영 체제에서 쉽게 빌드 할 수 있다.
  : 각 OS에 맞는 빌드 시스템 파일을 생성하여, 특정 환경에서 빌드 수행할 수 있도록 함

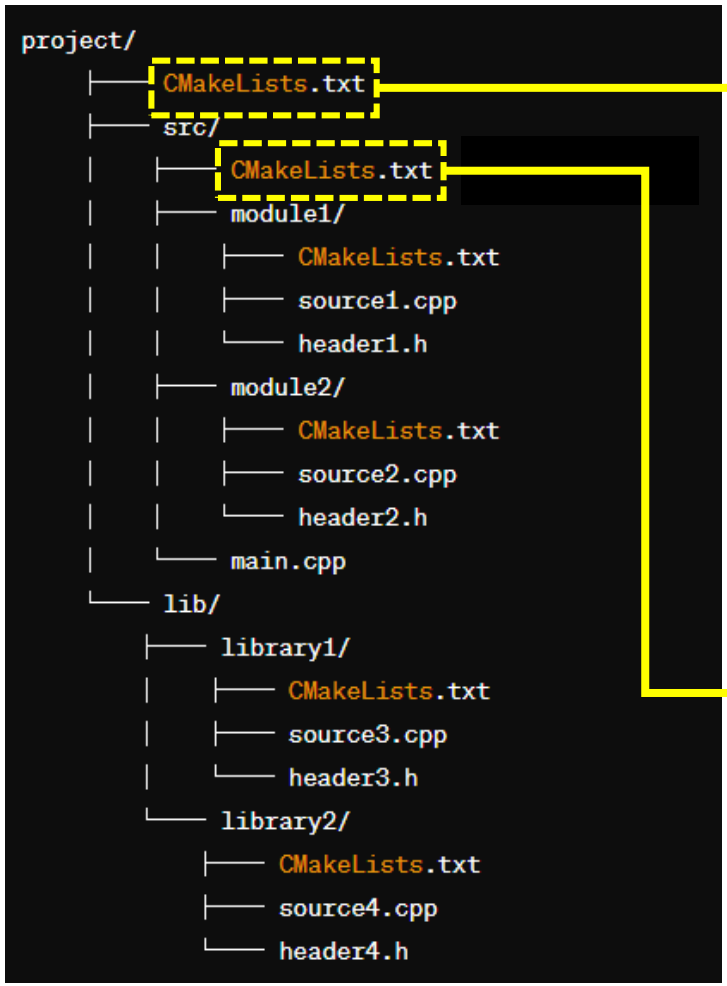
- 모듈화: 모듈화된 구조를 가지고 있어 다음과 같은 장점이 있다
  ① 코드의 재사용성: 각 모듈은 독립적으로 개발될 수 있으며,
  다른 프로젝트에서 필요한 기능을 가져올 수 있다.
  ② 유연성: 프로젝트 일부분을 독립적으로 개발하고 테스트할 수 있게 해줌
  ③ 의존성 관리: 서로 의존성이 있는 경우에도, 의존성을 관리하고 각 모듈이 필요한
  라이브러리 또는 외부 의존성을 자동으로 해결
  ④ 빌드 시스템 간소화: 각 모듈은 독립적으로 컴파일되며,
  변경사항이 발생할 때 해당 모듈만 다시 빌드 되도록 설정 가능
  ⑤ 유지보수 용이

CMake

단점

- 성능: 프로젝트 구성 및 생성에 시간 소요가 크다.
- 학습: Cmake 학습이 추가적으로 필요하다

CMake_모듈화



1. `project/CMakeLists.txt`:

cmake — Copy code

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

add_subdirectory(src)
add_subdirectory(lib)
```

2. `project/src/CMakeLists.txt`:

cmake — Copy code

```
add_subdirectory(module1)
add_subdirectory(module2)

add_executable(my_executable main.cpp)
target_link_libraries(my_executable module1 module2)
```

CMake_모듈화

```
project/
    ├── CMakeLists.txt
    ├── src/
    │   ├── CMakeLists.txt
    │   ├── module1/
    │   │   ├── CMakeLists.txt
    │   │   ├── source1.cpp
    │   │   └── header1.h
    │   ├── module2/
    │   │   ├── CMakeLists.txt
    │   │   ├── source2.cpp
    │   │   └── header2.h
    │   └── main.cpp
    └── lib/
        ├── library1/
        │   ├── CMakeLists.txt
        │   ├── source3.cpp
        │   └── header3.h
        └── library2/
            ├── CMakeLists.txt
            ├── source4.cpp
            └── header4.h
```

3. `project/src/module1/CMakeLists.txt`:

cmake — Copy code

```
add_library(module1 source1.cpp)
target_include_directories(module1 PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

4. `project/src/module2/CMakeLists.txt`:

cmake — Copy code

```
add_library(module2 source2.cpp)
target_include_directories(module2 PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

CMake_모듈화

```
project/
    ├── CMakeLists.txt
    ├── src/
    |   ├── CMakeLists.txt
    |   ├── module1/
    |   |   ├── CMakeLists.txt
    |   |   ├── source1.cpp
    |   |   └── header1.h
    |   ├── module2/
    |   |   ├── CMakeLists.txt
    |   |   ├── source2.cpp
    |   |   └── header2.h
    |   └── main.cpp
    └── lib/
        ├── library1/
        |   ├── CMakeLists.txt
        |   ├── source3.cpp
        |   └── header3.h
        └── library2/
            ├── CMakeLists.txt
            ├── source4.cpp
            └── header4.h
```

5. `project/lib/library1/CMakeLists.txt`:

cmake — Copy code

```
add_library(library1 source3.cpp)
target_include_directories(library1 PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}
```

6. `project/lib/library2/CMakeLists.txt`:

cmake — Copy code

```
add_library(library2 source4.cpp)
target_include_directories(library2 PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}
```

CMakeLists.txt 함수

cmake_minimum_required(VERSION 3.0.1)
project를 build하기 위해 필요한 최소한의 CMake의 버젼 명시

project(name)
project의 이름을 명시, 이 함수 실행 후 CMake내부 매크로로 ${PROJECT_NAME}으로 project이름을 호출할 수 있다.

add_library(library_name lib.cpp lib.h)
lib.cpp 와 lib.h로 정의된 library를 library_name으로 생성할 때 사용한다. 이때, 생성만 되고 프로젝트의 소스 코드를 빌드할 때는 자동 추가 되지 않으며,
뒤에 target_link_libraries를 통해 빌드에 추가 해 주어야 한다.

include_directories(dir ...)
.cpp파일에서 include의 기본 경로들을 추가해 준다.

add_executable(result src)
Makefile을 작성할 때 gcc -o result src와 같다. result라는 실행 파일을 src를 빌드하여 만드는 걸 명시 한다.

add_dependenciese(target depend1 ...)
target의 빌드 의존성을 추가해 준다.

target_link_libraries(target lib1 ...)
linker가 실행파일을 생성할 때 필요한 target에 대한 libraries를 명시해 준다.

message(msg)
CMake가 실행 될때 msg에 해당하는 문자들을 출력해 준다.

file(GLOB dst <type>)
<type>에는 모든 경로 & 파일을 의미하는 * 또는 특정 확장자를 명시하는 *.cpp 와 같은 구문이 들어갈 수 있으며, 이에 해당하는 파일경로 또는 파일 이름들을
dst에 저장하게 되며, 항목이 여러개일 경우 ;을 통해 구분되어 들어간다.

foreach(src srcs) / endforeach()
여러개의 항목이 정의 되고 ;으로 구분된 srcs를 하나 씩 src에 넣어서 foreach() 와 endforeach()사이의 구문을 반복하게 된다.

if(IS_DIRECTORY src) / endif()
간단한 if 구문에 옵션을 추가한 것으로 src에 오는 문자열이 파일경로를 의미하는지 아닌지 판단해 준다.

add_subdirectory(dir)
CMake를 하는 경로를 추가해 준다. 추가된 경로에 있는 CMakeLists.txt의 내용들도 반영하여 추가적인 Makefile을 생성한다.

```
cmake_minimum_required(VERSION 3.0.2)
project(robot_core)
add_compile_options(-std=c++11)
file(GLOB FILE_LISTS include/*)
set(INCLUDE_DIR "")
set(LIB_SRCS "")
set(LIB_HEADERS "")
foreach(FILE_LIST ${FILE_LISTS})
                        if(IS_DIRECTORY ${FILE_LIST})
                                                list(APPEND INCLUDE_DIR ${FILE_LIST})
                                                message("ADD INCLUDE DIRECTORY : ${FILE_LIST}")
                                                file(GLOB LIB_SRC ${FILE_LIST}/*.cpp )
                                                list(APPEND LIB_SRCS ${LIB_SRC})
                                                file(GLOB LIB_HEADER ${FILE_LIST}/*.h )
                                                list(APPEND LIB_HEADERS ${LIB_HEADER})
                        endif()
endforeach()
include_directories(
                        include
                        ${INCLUDE_DIR}
                        )

message("LIB_SRCS : ${LIB_SRCS}")
message("LIB_HEADERS : ${LIB_HEADERS}")
add_library(my_lib ${LIB_SRCS} ${LIB_HEADERS})
target_include_directories(my_lib
PRIVATE include ${INCLUDE_DIR}
)
add_executable(robot_core src/main.cpp)
target_link_libraries(robot_core
                        ${catkin_LIBRARIES} my_lib
)
```