luncliff / cmake-tutorial.md

CMake 할때 쪼오오금 도움이 되는 문서

cmake-tutorial.md

> CMake를 왜 쓰는거죠?
> 좋은 툴은 Visual Studio 뿐입니다. 그 이외에는 전부 사도(邪道)입니다 사도! - 작성자

주의

- 이 문서는 CMake를 주관적으로 서술합니다
- 이 문서를 통해 CMake를 시작하기엔 적합하지 않습니다
  https://cgold.readthedocs.io/en/latest/ 3.1 챕터까지 따라해본 이후 기본사항들을 속성으로 익히는 것을 돕기위한 보조자료로써 작성되었습니다

참고자료

문헌 / Web

- CGold by ruslo
- CMake Documentation
- https://cognitivewaves.wordpress.com/cmake-and-visual-studio/
- https://github.com/cognitivewaves/CMake-VisualStudio-Example?fbclid=IwAR2lE0Iorx9msmfU4tplmIOyz_NI7z8QzzgY1b8y3PkeTSWRfLbYD-z_0c8
- https://cliutils.gitlab.io/modern-cmake/
- https://github.com/pr0g/cmake-examples
- https://linux.die.net/man/1/cmakevars

영상

- C++ Weekly: Intro to CMake
- C++Now 2017: Daniel Pfeifer "Effective CMake"
- CppCon 2017: Mathieu Ropert "Using Modern CMake Patterns to Enforce a Good Modular Design"

- Florent Castelli: Introduction to CMake

License

tutorial-p1.md

CMake 튜토리얼 Lv.1

프로젝트의 구성

CMake는 빌드 시스템에서 필요로 하는 파일을 생성하는데 그 목적이 있습니다.

CMake 배치

CMake를 사용해 프로젝트를 관리하고자 한다면, 필요/의도에 맞게 CMakeLists.txt 파일을 배치해야 합니다.

일반적으로 Root CMake 파일을 두는 것이 편리합니다.
많은 경우 소스코드들은 폴더를 사용해 조직화되어 있으므로, 모든 프로젝트들이 **재귀적으로** 폴더 내에 `CMakeLists.txt` 를 두고 있으면 관리할 때 `폴더 == 프로젝트` 로 생각할 수 있게 됩니다.

```
$ tree ./small-project/
./small-project          # Project root folder
├── CMakeLists.txt       # <--- Root CMake
├── include              # header files
│   └── ...
├── module1              # sub-project
│   ├── CMakeLists.txt
│   └── ...
├── module2              # sub-project
│   ├── CMakeLists.txt
│   └── ...
└── test                 # sub-project
    ├── CMakeLists.txt
    └── ...
```

project

프로젝트의 이름을 지정할 수 있습니다.
CMakeLists.txt에 다음과 같이 작성하는 것으로 프로젝트에 이름을 부여하게 됩니다.

```
# CMake에서 주석은 #을 사용해 작성합니다
project(my_project)
```

좀 더 상세하게, 언어와 버전을 명시하는 것도 가능합니다.

```
# 한 줄에 모든 것을 적을 수 있습니다
project(my_project  LANGUAGES CXX   VERSION 1.2.3)

# multi-line으로 작성하는 것 또한 가능합니다
project(my_project
        LANGUAGES   CXX
        VERSION     1.2.3
)
```

이렇게 `project` 를 명시하게 되면 Visual Studio 에서는 같은 이름으로 Solution 파일이 생성됩니다. 즉, `project` 만으로는 프로그램을 생성하지 않습니다. 실제로 프로그램을 생성하기 위해서는 `add_executable` , `add_library` 를 사용하여야 합니다.

소스코드 조직화

프로그램(exe, lib, dll, a, so, dylib ...)을 만들기 위해서는 컴파일러에게 제공할 소스코드가 필요합니다. CMake에게 **소스코드 목록**으로부터 생성할 **프로그램의 타입**을 지시하기 위해 사용하는 함수들이 바로 `add_executable` , `add_library` 입니다. `project` 는 하나만 가능하지만, 이 함수들은 CMakeList안에서 여러번 사용되기도 합니다. 빌드 결과 생성되는 프로그램의 **이름만 다르다면** 크게 문제되지 않습니다.

`add_executable` , `add_library`

아래와 같은 구조로 프로젝트가 구성되었다고 해봅시다.

```
$ tree ./project-example/
./project-example
├── CMakeLists.txt
├── include
│   └── ...
└── src
    ├── CMakeLists.txt
    ├── main.cpp
    ├── feature1.cpp
    ├── feature2.cpp
    ├── algorithm3.cpp
    └── data_structure4.cpp
```

우선 Root CMakeList(`project-example/CMakeLists.txt`)가 아니라 src/ 폴더에 있는 CMakeList(`project-example/src/CMakeLists.txt`)부터 살펴보겠습니다.

만약 src 폴더에 있는 모든 .cpp 파일들이 실행 파일(exe)을 만든다면, 아래와 같은 내용이 작성되어야 합니다.

```
# project-example/src/CMakeLists.txt
# case 1

add_executable(my_exe   # 이후에 나오는 .cpp 파일을 사용해 .exe를 생성한다
    main.cpp
                        # 개행을 여러번 하여도 문제되지 않습니다.
    feature1.cpp
    feature2.cpp
    algorithm3.cpp      # 상대 경로로 소스 코드를 찾아냅니다.
                        # 현재 사용중인 CMakeList의 위치를 기준으로
                        # 경로를 지시해야 합니다
    data_structure4.cpp
    # ...
)
```

만약 라이브러리를 만든다면, 아래와 같은 내용이 작성되어야 합니다.

```
# project-example/src/CMakeLists.txt
# case 2
```

```
add_library(my_lib      # 이후에 나오는 .cpp 파일을 사용해 라이브러리를 생성한다
    main.cpp
    feature1.cpp
    # ...
)
```

라이브러리의 링킹의 형태를 명시하지 않는다면 여기서 생성되는 라이브러리는 프로젝트 생성시 `BUILD_SHARED_LIBS` 변수를 따라서 결정됩니다. 물론 직접 명시할 수도 있습니다.

```
# project-example/src/CMakeLists.txt
# case 2.1, 2.2

add_library(my_archive  STATIC  # 정적 링킹 라이브러리(.a, .lib)
    main.cpp
    feature1.cpp
    # ...
)

add_library(my_shared_object  SHARED  # 동적 링킹 라이브러리(.so, .dll)
    main.cpp
    feature1.cpp
    # ...
)
```

`STATIC`, `SHARED` 가 대문자인 점에 주의하십시오. **CMake파일은 대소문자를 신경써서 사용해야만 합니다**. CMake 함수들은 앞서 예시처럼 소문자를 사용해도, `PROJECT`, `ADD_EXECUTABLE` 처럼 대문자로 작성하여도 문제없이 동작합니다. 하지만 `SHARED` 처럼 일부 예약된 단어(sub-command)들은 대문자만을 사용해야 합니다.

가독성을 고려하여 일관성있게 작성하는 것이 좋습니다. CMake기반 프로젝트에서는 개발자들이 서로의 CMakeList를 **주의 깊게** 살펴보는 경우가 많습니다.
보고 배울만한 CMake 사례를 모아두십시오. 특정한 저장소나 프로젝트를 따라해도 좋습니다.

add_subdirectory

이번에는 Root CMakeList(`project-example/CMakeLists.txt`)를 살펴보겠습니다. 이미 어떻게 프로그램을 생성할 것인지는 앞서 src 폴더에서 작성한 CMakeLists가 잘 처리하고 있기 때문에, Root CMakeList에서는 이를 그대로 사용할 것입니다.

```
$ tree ./project-example/
./project-example
├── CMakeLists.txt          # <---- project
├── include
└── src
```

```
    ├── CMakeLists.txt      # <---- add_executable/add_library
    └── ...
```

간단히 `add_subdirectory` 와 폴더 이름을 주는 것으로 이를 달성할 수 있습니다.
많은 경우 CMakeList는 아래와 같이 재귀적으로, 즉 `add_subdirectory` 만으로 프로젝트를 (File Tree를 따라) 조직화 합니다.

```
# project-example/CMakeLists.txt
project(my_project  LANGUAGES CXX   VERSION 1.2.3)

add_subdirectory(src)       # 상대경로
```

외부 경로 가져오기

하지만 경우에 따라선 필요한 CMakeList가 Root CMakeList의 하위에 없을 수도 있습니다. 이런 경우를 Out-of-tree build라고 합니다.

아래와 같은 경우를 생각해봅시다.

```
$ tree ./out-of-tree
./out-of-tree/
├── current-project
│   └── CMakeLists.txt
└── far-far-away
    ├── CMakeLists.txt
    └── algorithm1.cpp
```

여기서 `current-project/CMakeLists.txt` 는 `add_subdirectory` 로 하위 디렉터리가 아닌 다른 곳을 참고 하고 있습니다.

```
# current-project/CMakeLists.txt

cmake_minimum_required(VERSION 3.10)    # CMake 버전을 명시
project(my_project)

add_subdirectory(../far-far-away)       # 상대경로를 사용해 접근
```

이대로 CMake를 실행하면 다음과 같은 오류를 발생시킵니다.

```
# ...
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Error at CMakeLists.txt:6 (add_subdirectory):
  add_subdirectory not given a binary directory but the given source
  directory "X:/Develop/out-of-tree/far-far-away" is not a subdirectory of
  "X:/Develop/out-of-tree/current-project".  When specifying an out-of-tree
```

```
  source a binary directory must be explicitly specified.
-- Configuring incomplete, errors occurred!
```

다행히도, `add_subdirectory` 를 하는 CMakeList에서 경로를 조정하는 방법으로 사용할 수 있습니다. When specifying an 'out-of-tree' source a binary directory must be explicitly specified 문장을 다시 한번 보시기 바랍니다.
binary directory를 명시할 것을 요구하고 있습니다.

```
# current-project/CMakeLists.txt

cmake_minimum_required(VERSION 3.10)    # CMake 버전을 명시
project(my_project)

add_subdirectory(../far-far-away            # CMakeList가 위치한 source dir
                 ./build/far-far-away-dir   # 빌드 결과물을 배치할 binary dir을 지
)
```

CMake 문서의 지시를 따라 위와 같이 수정하면 아래와 같은 결과를 얻습니다.

```
# ...
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: X:/Develop/out-of-tree/current-project
```

오류가 사라진 것을 확인할 수 있습니다. 폴더를 열어 확인해보면 지정한 build directory 경로에 빌드 파일들이 생성되어 있습니다.

```
PS X:\Develop\out-of-tree> tree .
Folder PATH listing for volume Drive_X
Volume serial number is D688-4B7E
X:\DEVELOP\OUT-OF-TREE
├─current-project
│  ├─build
│  │  └─far-far-away-dir    # <----- ./build/far-far-away-dir
│  │      └─CMakeFiles
│  └─CMakeFiles
│      ├─3.10.1
│      ├─a8084ac71a5a995e5212369c2e1624fc
│      └─CMakeTmp
└─far-far-away
```

예시: 단일 실행 파일 / 라이브러리

당연하게도, 간단한 프로젝트들이 모여야 큰 프로젝트를 이루게 됩니다. C++로 하나의 프로그램을 만든다면 아마 아래와 같이 CMake 프로젝트를 구성해야 할 것입니다.

```
$ tree ./small-project/
./small-project
├── CMakeLists.txt
├── include
│   └── ...
└── src
    ├── CMakeLists.txt
    └── ...
```

예시: 라이브러리 + 실행 파일 각각 1개씩

만약 라이브러리를 만들었다면, 테스트와 같이 라이브러리 코드를 실행시켜 볼 수 있는 추가 프로젝트가 필요할 것입니다. 이런 경우 또 다른 sub-project를 추가 하면 됩니다.

```
$ tree ./project-with-test/
./project-with-test
├── CMakeLists.txt
├── include
│   └── ...
├── src
│   ├── CMakeLists.txt
│   └── ...
└── test
    ├── CMakeLists.txt
    └── ...
```

예시: 2개 라이브러리 + 1개 실행 파일

라이브러리를 하나만 사용하라는 법은 없죠.

```
$ tree ./multiple-library-project
./multiple-library-project
├── CMakeLists.txt
├── include
│   └── ...
├── module1
│   ├── CMakeLists.txt
│   └── ...
├── module2
│   ├── CMakeLists.txt
│   └── ...
└── test
    ├── CMakeLists.txt
    └── ...
```

예시: 계층화된 라이브러리 (depth 2) + 1개 실행 파일

OpenCV와 같이 좀 더 복잡하게 구성할 수도 있습니다. 다수의 모듈들을 한번 더 조직화하기도 합니다. 하위 경로에 있기만 하다면 `add_subdirectory` 를 사용하는데 크게 문제될 일이 없습니다. 출처에 따라서 Internal/External로 두기도 하고, 목적에 따라서 특별한 이름을 붙일 수도 있습니다.

```
$ tree ./huge-project
./huge-project
├── CMakeLists.txt
├── cmake
│   └── get-some-package.cmake
├── include
│   └── ...
├── modules
│   ├── guideline_support_library
│   │   ├── CMakeLists.txt
│   │   └── ...
│   ├── fmt
│   │   ├── CMakeLists.txt
│   │   └── ...
│   ├── grpc
│   │   ├── CMakeLists.txt
│   │   └── ...
│   └── ...
├── src
│   ├── CMakeLists.txt
│   └── ...
└── test
    ├── CMakeLists.txt
    └── ...
```

git을 사용하는 프로젝트라면 submodule을 추가하면서 함께 조정하면 될 것입니다.

실행 해보기

앞서까지는 CMake를 실행하는 방법을 언급하지 않았습니다.
CMake가 빌드 시스템에 맞는 파일들(sln, vcxproj, MakeFiles, ninja ...)을 생성하기 위해선 당연하게도 "생성기"를 명시해야 합니다. 이 생성기는 플랫폼에 맞는 개발 툴을 지정하게 됩니다. 이 과정은 Configure / Generate 라는 2 단계로 구성됩니다.

- Configuration : `CMakeLists.txt` 파일 해석
- Generation: 해석 결과를 바탕으로 Project 파일 생성

아래는 커맨드라인으로 각 플랫폼에서 주로 사용되는 IDE에 맞는 파일을 생성하는 것을 보여줍니다

Windows

PowerShell에서는 ""를 사용해 문자열을 명시한 점에 주의하십시오

```
# Generate for VS
cmake ./Path/to/Root/ -G "Visual Studio 15 2017 Win64"
```

Unix/Linux

```
# Generate for Ninja
cmake ./Path/to/Root/ -G Ninja
```

MacOS

```
# Generate for XCode
cmake ./Path/to/Root/ -G XCode
```

빌드 시스템/툴체인 지정

거듭 강조하자면, CMake의 목적은 파일을 생성하는데 있으며, 프로그램 빌드를 하지는 않습니다.
다만 아래와 같이 커맨드라인에서 `-G` 옵션으로 지정한 빌드시스템을 호출하도록 명령할 수는 있습니다.

Powershell/Bash 모두 동일합니다.

```
cmake --build . # ... 지정된 툴을 사용해 빌드를 진행한다 ...
cmake --build . --parallel          # 병렬 빌드
cmake --build . --target install    # 빌드에 성공하면 설치까지 진행한다
cmake --build . --config debug      # Debug로 빌드
```

툴체인 파일은 CMake가 지원하는 다양한 기능들을 사용해 빌드를 수행할 수 있도록 미리 지정된 파일을 의미합니다.

대표적으로 Android NDK에서는 여러 아키텍처로의 크로스 컴파일에 필요한 설정들이 작성된 android.toolchain.cmake 파일이 함께 제공되며, CMake를 사용한 빌드를 수행시에 Gradle에 의해서 자동으로 지정됩니다.
iphone을 대상으로 하는 경우에는 https://github.com/leetal/ios-cmake 를 사용해 XCode 프로젝트를 생성하기도 합니다.

다르게는 https://github.com/Microsoft/vcpkg 와 같이 라이브러리 탐색에 특화된 툴체인 파일을 사용하는 경우도 있습니다.

```
# ... Vcpkg에서 제공하는 cmake 툴체인의 경로로 제공한다 ...
cmake ../ -G "Visual Studio 15 2017 Win64" -DCMAKE_TOOLCHAIN_FILE="X:\Develop\vcp
```

빌드 관계 설정

의존성

프로젝트 내에서 여러 라이브러리를 빌드한다면, 그리고 서로 의존성이 있다면 이를 제어하는 것도 가능합니다. 이 과정은 보통 해당 sub-project들을 모두 확인할 수 있는 CMakeList에서 수행하게 됩니다.

`add_dependencies`

의존성은 `add_dependencies` 함수를 사용해 명시하게 됩니다. 이름에서 알 수 있듯이 다수를 지정할 수 있습니다.

아래와 같은 형태로 프로젝트가 구성되었다고 가정하겠습니다.

```
$ tree ./multiple-library-project
./multiple-library-project
├── CMakeLists.txt      # <--- Root CMakeList
├── include
│   └── ...
├── module1
│   ├── CMakeLists.txt
│   └── ...
├── module2
│   ├── CMakeLists.txt
│   └── ...
└── test
    ├── CMakeLists.txt
    └── ...
```

`module2` 가 `module1` 의 기능을 사용하고, `test` 는 이 둘의 기능을 모두 사용한다면 이는 Root CMake에서 다음과 같이 명시할 수 있습니다.

```
# multiple-library-project/CMakeLists.txt

add_subdirectory(module1)
add_subdirectory(module2)
add_subdirectory(test)

# dependency: from -> { to }
add_dependencies(module1 module2)            # `module1` requires `module2`
add_dependencies(test    module1 module2)    # `test` requires `module1` & `modul
```

이를 바탕으로 CMake에서 프로젝트가 생성되면, 그 프로젝트는 의존성을 고려하여 `module1` , `module2` , `test` 순서로 빌드를 진행하게 됩니다.

별명 붙이기

라이브러리들의 수가 많아지거나 경우에 따라 달라지면 별명을 붙여서 상위 CMakeList 에서 좀 더 편하게 사용할 수 있도록 할수도 있습니다.

```
# sub-project CMakeList
add_library(my_custom_logger_lib
    # ...
)

add_library(module::logger ALIAS my_custom_logger_lib)
```

디버그 모드와 릴리즈 모드에서 라이브러리 이름이 바뀌는 경우 이는 굉장히 유용한 기능이 됩니다.

```
# higher-project CMakeList

add_subdirectory(some_exe)
add_subdirectory(custom_logger)

add_dependencies(some_exe   module::logger) # Use with alias
```

Linking

빌드 순서를 제어하기 위해서 `add_dependencies` 를 사용했다면, 프로그램 생성시에(링킹에) 필요한 (Symbol Table과 같은) 정보들이 공유되도록 하기 위해서는 `target_link_libraries` 를 사용하게 됩니다.

이 문서에서 Target이라는 용어가 처음 사용되었는데, 이는 `add_executable` 혹은 `add_library` 를 사용해 생성한 대상을 의미합니다. 빌드 시스템 파일 생성의 단위이기도 합니다

target_link_libraries

```
add_library(my_custom_logger_lib
    # ...
)

target_link_libraries(my_custom_logger_lib #
PUBLIC
    spdlog fmt
PRIVATE
    utf8proc
)
```

특히 이 함수는 Target의 의존성을 전파시키는 역할도 수행합니다. 위와 같은 경우, `PRIVATE` 와 `PUBLIC` 을 사용해 이를 제어하고 있습니다. C++ class의 멤버 접근 한정자(Access Qualifier)와 유사하게 생각할 수 있습니다.
위와 같이 작성하면 다른 프로젝트에서 `my_custom_logger_lib` 을 link하는 경우, `spdlog` 와 `fmt` 에 있는 **헤더파일을 include하기 위한 경로**와 **빌드 결과 생성되는 라이브러리**에 접근할 수 있게 됩니다. 하지만 `PRIVATE` 로 선언된 `utf8proc` 은 이와 같은 정보를 차단합니다.

```
add_executable(some_test_program)
    # ...
)

target_link_libraries(some_test_program
PUBLIC
    my_custom_logger_lib  # spdlog 와 fmt 이 적혀있지 않지만 자동으로 추가된다
)
```

의존성이 공유되어야 하는 경우, 혹은 기타 라이브러리와 함께 사용하는 라이브러리라면 `PUBLIC` , 내부 구현에만 사용되고 공개되지 않는 경우라면 `PRIVATE` 에 배치하는 것이 적합합니다.

플랫폼 대응

하지만 크로스 플랫폼은 아주아주 힘든 일입니다. CMake에서도 이 문제에 대응해보겠습니다

CMake 변수

프로그래머라면 변수에 익숙할 것입니다. CMake에서도 변수가 있으며, 단순한 Boolean, 문자열, 파일 경로, 리스트 등을 표현할 수 있습니다. 이 튜토리얼에서는 이 중 빈번하게 사용되는 변수들을 짚고 넘어가겠습니다

`set` / `unset`

변수는 `set` 을 사용해서 생성하고, `unset` 을 사용해서 제거할 수 있습니다. 변수는 문자열 기반이며, bash 쉘 프로그래밍처럼 사용할 수 있습니다. 다만 **문자열에 대한 참조**처럼 사용된다는 특이점이 있습니다.

```
set(CMAKE_BUILD_TYPE Debug)

message(STATUS CMAKE_BUILD_TYPE)                    # -- CMAKE_BUILD_TYPE
message(STATUS ${CMAKE_BUILD_TYPE})                 # -- Debug
message(STATUS "Configuration: ${CMAKE_BUILD_TYPE}") # -- Configuration: Debug
```

변수의 값을 사용하기 위해 `${}` 를 사용한 점을 주의깊게 보시길 바랍니다. `message` 는 문자열을 출력하므로 변수를 그대로 주지 않고 한번 참조하여 문자열로 변환한 것입니다. 그대로 변수 이름만을 제공할 경우 변수 이름을 문자열로 바꿔 출력하는 것을 확인할 수 있습니다

조건부 처리

`if / elseif / else / endif`

플랫폼이 다르면 System API, 혹은 같은 API더라도 구현 형태나 지원 범위가 상이할 수 있습니다. 조건부 분기문을 사용해 플랫폼에 맞게 미리 작성된 라이브러리를 선택하도록 하면 상대적으로 부담이 줄어들 것입니다.

CMake는 플랫폼 관련 변수들을 제공하고 있으며, Android 혹은 iOS로 크로스 컴파일을 하기 위해 CMake Toolchain을 사용한 경우 그 값을 참고해 처리를 다르게 할 수 있습니다.

```
# *현재* CMake가 실행되는 시스템을 알려진 변수들로 확인하는 방법

# wrapper::system 같은 별명을 붙이면 상대적으로 편해진다
if(WIN32)
    add_subdirectory(external/winrt)
    add_subdirectory(impl/win32)
elseif(APPLE)
    add_subdirectory(impl/posix)
    # additional implementation for MacOS
    add_subdirectory(impl/macos)
elseif(UNIX)
    add_subdirectory(impl/posix)
    # additional implementation with Linux API
    if(${CMAKE_SYSTEM} MATCHES Linux)
        add_subdirectory(impl/linux)
    endif()
else()
    # 지원하지 않음.
    # android.toolchain.cmake 혹은 ios.toolchain.cmake 에서 지정하는 변수들
    if(ANDROID OR IOS)
        message(FATAL_ERROR "No implementation for the platform")
    endif()
    # ...
endif()
```

가장 윗 줄에 주석으로 적은 것처럼 **현재 시스템**을 변수로 알려주는 것이라는 점에 주의하시기 바랍니다. 크로스 컴파일을 위한 라이브러리라면 변수를 검사하는 순서, 조건문에 주의를 기울여야 합니다.

접근법

구현이 다르더라고, 공통된 인터페이스(헤더파일)가 있다면 이들을 한곳에 모아놓는 것이 타당할 것입니다. 문서 초반부에 프로젝트 예시로 include 폴더가 꾸준히 나타났다는 것을 기억하십니까?

target_include_directories

헤더파일들이 위치한 폴더는 제각기 다를 수 있습니다. C 혹은 C++ 프로젝트에서 외부에 공개되는 헤더파일과 공개되지 않는 헤더파일이 다른 위치에 배치되는 것은 흔한 일입니다.

`target_include_directories` 는 Target에서 헤더파일을 찾기 위해 사용하는 폴더를 지정하는 함수이며, 이곳에 위치한 헤더파일들은 `#include <my_interface.h>` 와 같이 `<>` 형태로 include하는 것이 가능합니다.

아래와 같은 프로젝트 구조를 생각해봅시다.

```
$ tree ./some-huge-project
./some-huge-project
├── CMakeLists.txt
├── include                 # <--- 공용 헤더 파일
│   ├── system_wrapper.h
│   └── ...
├── impl
│   ├── win32
│   │   ├── CMakeLists.txt
│   │   ├── include         # <--- 구현 헤더 파일
│   │   │   ├── pch.h
│   │   │   └── my_win32.h
│   │   └── ...
│   ├── posix
│   │   ├── CMakeLists.txt
│   │   ├── include         # <--- 구현 헤더 파일
│   │   │   └── my_posix.h
│   │   └── ...
│   └── ...
├── src
│   ├── CMakeLists.txt      # <--- system 라이브러리를 사용하는 프로젝트
│   └── ...
├── ...
```

우선 Win32 하위 프로젝트에 있는 CMakeLists 부터 살펴보겠습니다.

```
# some-huge-project/impl/win32/CMakeLists.txt

add_library(my_win32_wrapper
    src/i-love-win32.cpp
)
add_library(wrapper::system ALIAS my_win32_wrapper)

target_include_directories(my_win32_wrapper
```

```
    PUBLIC
        ${CMAKE_SOURCE_DIR}/include
                    # CMAKE_SOURCE_DIR 는 최상위 CMakeLists.txt가 위치한 폴더를 의미한(
                    # 이 프로젝트에서는 some-huge-project/
    PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}/include
                    # include 폴더를 절대경로를 사용해 접근
                    # CMAKE_CURRENT_SOURCE_DIR 는 현재 해석 중인 CMakeLists.txt가 위치
                    # 즉, 경로는 some-huge-project/impl/win32
    )
```

POSIX API에 맞춘 프로젝트의 CMakeList는 이렇게 작성할 수 있습니다.

```
    # some-huge-project/impl/posix/CMakeLists.txt

    add_library(my_posix_wrapper
        src/i-love-posix.cpp
    )
    add_library(wrapper::system ALIAS my_posix_wrapper)

    target_include_directories(my_posix_wrapper
    PUBLIC
        ${CMAKE_SOURCE_DIR}/include
                    # win32와 header 파일들을 공유한다.
                    # 예컨대 some-huge-project/include에 위치한 system_wrapper.h
    PRIVATE
        include     # include 폴더를 상대경로 접근
                    # some-huge-project/impl/posix/include를 의미한다
    )
```

예시에서 본것과 같이 이 함수는 `target_link_libraries` 처럼 `PUBLIC` , `PRIVATE` 을 지정할 수 있으며, PUBLIC에 위치한 폴더들은 경로가 자동으로 전파됩니다.

```
    # some-huge-project/src/CMakeLists.txt

    add_executable(my_system_utility
        some.cpp
        source.cpp
        files.cpp
    )

    target_link_libraries(my_system_utility
    PRIVATE
        wrapper::system # wrapper 라이브러리들의 ALIAS
                        # target_include_directories에 PUBLIC으로 명시된
                        # ${CMAKE_SOURCE_DIR}/include 폴더를 자동으로 접근할 수 있게 돈
    )
```

target_sources

`add_executable` 과 `add_library` 의 한계는 소스 파일 목록을 한번에 결정해서 전달해야 한다는 점입니다. 이는 Target들이 CMakeList 파일의 끝부분에 나타나게 만들며, 2.x 버전 CMake들이 사용했던 방법처럼 List 변수를 사용해 소스파일 목록을 만들어야 하는 불편함이 있습니다.

CMake 3.x에서는 Target에 소스파일을 '추가'할 수 있도록 `target_sources` 를 사용할 수 있습니다.

```
# preview 가 구현된 소스파일을 추가할지 결정하는 변수
set(USE_PREVIEW true)

# target: my_program
add_executable(my_program
    main.cpp
    feature1.cpp
)

# ...
# add_dependencies
# set_target_properties
# target_include_directories
# target_link_libraries
# ...

target_sources(my_program
PRIVATE
    feature2.cpp
)
if(USE_PREVIEW)
    target_sources(my_program
    PRIVATE
        feature3_preview.cpp
    )
else()
```

컴파일러 대응

플랫폼이 달라지면 컴파일러도 달라질 수 있습니다. 컴파일러가 달라지면 프로그램 생성에 사용할 수 있는 컴파일 옵션들이 달라지게 됩니다. 이 튜토리얼에서는 간단히 Warning, Optimization를 다르게 적용하는 예시를 보이겠습니다

컴파일러 검사

앞서서 Platform을 검사한 것처럼, Compiler와 관련해 미리 지정된 변수들이 존재합니다.

관련 CMake 변수들

아래의 내용을 CMakeList에 추가한 이후 실행해보시기 바랍니다.

- `CMAKE_CXX_COMPILER_ID` : 컴파일러의 이름
- `CMAKE_CXX_COMPILER_VERSION` : 컴파일러의 버전
- `CMAKE_CXX_COMPILER` : 컴파일러 실행파일의 경로

```
message(STATUS "Compiler")
message(STATUS " - ID       \t: ${CMAKE_CXX_COMPILER_ID}")
message(STATUS " - Version  \t: ${CMAKE_CXX_COMPILER_VERSION}")
message(STATUS " - Path     \t: ${CMAKE_CXX_COMPILER}")
```

Windows에서 별다른 조작 없이 Visual Studio 15 2017 Win64를 Generator로 지정하면, 아래와 같이 출력될 것입니다. (Community 버전 기준)

```
-- Compiler
--   - ID         : MSVC
--   - Version    : 19.16.27024.1
--   - Path       : C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/V
```

컴파일러를 Clang-cl을 명시하면 아래와 같은 출력을 확인할 수 있습니다.

```
-- Compiler
--   - ID         : Clang
--   - Version    : 7.0.0
--   - Path       : C:/Program Files/LLVM/bin/clang-cl.exe
```

Mac OS에서는 아래와 AppleClang에 대한 정보를 보여줍니다.

```
-- Compiler
--   - ID         : AppleClang
--   - Version    : 9.1.0.9020039
--   - Path       : /Applications/Xcode-9.4.1.app/Contents/Developer/Toolchains/Xcc
```

아래와 같이 CMake 내에서 컴파일러에 따라서 처리를 다르게 할 수 있습니다.

```
if(MSVC)        # Microsoft Visual C++ Compiler
    # ...
elseif(${CMAKE_CXX_COMPILER_ID} MATCHES Clang)  # Clang + AppleClang
    # ...
elseif(${CMAKE_CXX_COMPILER_ID} MATCHES GNU)    # GNU C Compiler
    # ...
endif()
```

선술한 플랫폼 변수들을 함께 고려하면 조합이 많이 발생할 수 있기 때문에, CMake 파일들을 간결히 유지하려면 플랫폼에 따라서 특정 컴파일러만을 지원하는 것이 타당할 것입니다.

컴파일 옵션 사용

`include(CheckCXXCompilerFlag)`

컴파일러를 식별할 수 있게 되었으니 이제 컴파일러 옵션을 지정해줄 차례입니다. 하지만 그 전에 컴파일러가 해당 옵션을 지원하는지 검사가 필요한 경우도 있습니다. CMake의 기본 모듈들 중에는 이를 지원하는 `CheckCXXCompilerFlag` 라는 모듈이 있습니다.

CMake Module은 간단히 말하자면 미리 작성된 CMake 파일이라고 생각할 수 있습니다. 이런 파일들은 각 `project` 들마다 각각 cmake 폴더를 만들어 배치하는 경우가 많습니다.

```
tree -L 2 .
.
├── CMakeLists.txt
├── cmake             # <---- 이 프로젝트에서 필요한 cmake 파일들을 모아둔다
│   ├── display-compiler-info.cmake
│   └── test-cxx-flags.cmake
├── include
│   └── ...
├── modules
│   ├── ...
│   └── ...
├── scripts
│   └── ...
├── src
└── test
```

test-cxx-flag.cmake 파일의 내용은 아래와 같습니다. CMake Module에 대한 내용은 후술할 것이므로, CMake를 처음 접하였다면 **같은 내용을 CMakeList에 붙여넣기하는 것과 같은 효과가 생긴다**고 생각하면 되겠습니다.

```
# test-cxx-flags.cmake
#
# `include(cmake/check-compiler-flags.cmake)` from the root CMakeList
#
include(CheckCXXCompilerFlag)

# Test latest C++ Standard and High warning level to prevent mistakes
if(MSVC)
    check_cxx_compiler_flag(/std:c++latest  cxx_latest          )
    check_cxx_compiler_flag(/W4             high_warning_level  )
elseif(${CMAKE_CXX_COMPILER_ID} MATCHES Clang)
    check_cxx_compiler_flag(-std=c++2a      cxx_latest          )
    check_cxx_compiler_flag(-Wall           high_warning_level  )
```

```
elseif(${CMAKE_CXX_COMPILER_ID} MATCHES GNU)
    check_cxx_compiler_flag(-std=gnu++2a    cxx_latest          )
    check_cxx_compiler_flag(-Wextra         high_warning_level  )
endif()
```

이같은 내용을 추가하여 CMake를 실행하면 다음과 같은 내용이 나타날 것입니다. `check_cxx_compiler_flag` 는 해당 컴파일 옵션을 사용할 수 있으면 두번째 인자에 사용한 이름으로 변수를 생성합니다.

```
# ...
-- Compiler
--   - ID        : Clang
--   - Version   : 6.0.0
--   - Path      : /usr/bin/clang-6.0
# ...
-- Performing Test cxx_latest
-- Performing Test cxx_latest - Success
-- Performing Test high_warning_level
-- Performing Test high_warning_level - Success
# ...
```

위와 같은 경우, clang-6.0은 `-std=c++2a` , `-Wall` 를 모두 사용할 수 있으므로 `cxx_latest` , `high_warning_level` 변수는 모두 true(1)값을 가질 것입니다. 특정 컴파일 옵션을 사용할 수 없는 경우 경고하거나 우회하는 CMakeList를 상상해보시기 바랍니다.

target_compile_options

이제 컴파일 옵션을 사용할 준비과 되었으므로, 간단히 Target에서 사용할 컴파일 옵션을 지정하는 예시를 보이겠습니다.

```
if(MSVC)
    target_compile_options(my_modern_cpp_lib
    PUBLIC
        /std:c++latest /W4  # MSVC 가 식별 가능한 옵션을 지정
    )
else() # Clang + GCC
    target_compile_options(my_modern_cpp_lib
    PUBLIC
        -std=c++2a -Wall    # GCC/Clang이 식별 가능한 옵션을 지정
    PRIVATE
        -fPIC
        -fno-rtti
    )
endif()
```

이번에도 `PUBLIC`, `PRIVATE`으로 컴파일 옵션을 전파시킬 수 있습니다. 즉, my_modern_cpp_lib를 `target_link_libraries`로 사용하는 모든 Target들은 C++ latest, Warn All 옵션으로 빌드가 수행 됩니다.
하지만 옵션의 중복을 걱정할 필요는 없습니다. **중복되는 옵션은 CMake에서 자동으로 하나로 합쳐서 적용하게 됩니다.**

매크로 선언

`target_compile_definitions`

최신 C++에서는 Macro를 대체할 방법으로 `enum class`, `constexpr` 등이 있습니다만, 여전히 Macro에 의존하고 있는 코드가 많은 것 또한 사실입니다. 하지만 수십, 혹은 수백개의 소스 파일에 Macro를 선언하려면 시간이 많이 들뿐만 아니라 이후에 수정하기도 번거로울 것입니다.

```
if(MSVC)
    # 묵시적으로 #define을 추가합니다 (컴파일 시간에 적용)
    target_compile_definitions(my_modern_cpp_lib
    PRIVATE
        WIN32_LEAN_AND_MEAN
        NOMINMAX    # numeric_limits를 사용할때 방해가 되는
                    # max(), min() Macro를 제거합니다
        _CRT_SECURE_NO_WARNINGS
                    # Visual Studio로 C++에 입문했다면 한번쯤 만나본 녀석일 겁니다
                    # 오래된 코드를 위한 프로젝트라면 선택의 여지가 없을 수도 있겠죠
    )
endif()
```

물론 여기에도 `PUBLIC`, `PRIVATE`가 있습니다. 아마 처음부터 읽으셨다면 어떤 기능을 하는지 더는 설명이 필요 없을 것입니다. **다만 이 방법으로 추가하는 Macro는 소스코드에 보이지 않기 때문에 프로젝트를 가져다 쓰는 사람이 찾아내기 어려울 수 있습니다.** 평시에 신중하게, 주의하면서 사용하는게 좋을 것입니다.

CMake 파일 작성

변수와 조건문에 대해서 배우고 나면 보통 함수에 대해서 배우게 됩니다. **안타깝게도 이 튜토리얼 CMake Macro와 CMake Function에 대해서는 생략할 것입니다.** 대신, 앞서 CheckCXXCompilerFlag와 같은 형태의 CMake Module에 대해서는 짚고 넘어가겠습니다.

CMake Module 폴더 만들기

`.cmake` & `include`

CMake는 나름의 문법과 처리방식이 있기 때문에, 전용 확장자가 없는게 더 이상할 것입니다. 최초의 Root CMakeList 호출이나 `add_subdirectory` 는 CMakeLists.txt를 사용하지만, 그렇지 않은 경우라면 보통 `.cmake` 파일을 사용하게 됩니다.
앞서 잠깐 언급했던 vcpkg.cmake와 같이 Toolchain파일들이 이런 확장자를 가지고 있었던 것을 기억해주시기 바랍니다.

아래와 같은 프로젝트를 가정해봅시다.

```
tree -L 2 .
.
├── CMakeLists.txt  # <---- Root
├── cmake
│   ├── display-compiler-info.cmake # <---- going to `include`
│   └── test-cxx-flags.cmake
├── include
│   └── ...
├── src
│   ├── CMakeLists.txt  # <---- going to `add_subdirectory`
│   └── ...
└── test
    ├── CMakeLists.txt  # <---- going to `add_subdirectory`
    └── ...
```

`add_subdirectory` 는 기본적으로 함수(서브루틴)처럼 동작한다고 할 수 있습니다. **독립적으로 CMake 변수들을(유효범위) 가지고**, 별도로 지시하지 않는 한 상위 CMakeList의 변수를 변경하지 않습니다. sub-project에서 벗어나면 그 변수들은 사라집니다.
반면 `include` 는 C++ 코드에서 `inline` 을 지정하는 것과 유사합니다. `include` 를 통해 실행되는 cmake는 **현재 CMakeLists의 변수들에 그대로 접근**할 수 있습니다. 물론 새로운 변수를 추가할 수도 있습니다.

```
# Root/CMakeLists.txt
project(my_new_project)

# .cmake의 내용을 복사-붙여넣기 한 것처럼 동작한다
include(cmake/check-compiler-flags.cmake)
#
# include(CheckCXXCompilerFlag) # 또다른 CMake 기본 모듈을 가져온다
#
# if(MSVC)
#    check_cxx_compiler_flag(/std:c++latest  cxx_latest          )
#    check_cxx_compiler_flag(/W4             high_warning_level  )
# elseif(${CMAKE_CXX_COMPILER_ID} MATCHES Clang)
#    check_cxx_compiler_flag(-std=c++2a      cxx_latest          )
#    check_cxx_compiler_flag(-Wall           high_warning_level  )
# elseif(${CMAKE_CXX_COMPILER_ID} MATCHES GNU)
#    check_cxx_compiler_flag(-std=gnu++2a    cxx_latest          )
#    check_cxx_compiler_flag(-Wextra         high_warning_level  )
# endif()
#
```

```
if(cxx_latest)  # include 파일 내에서 설정한 변수를 사용 가능하다
    target_compile_options(...)
endif()

message(STATUS ${CMAKE_SOURCE_DIR})         # -- Root
message(STATUS ${CMAKE_CURRENT_SOURCE_DIR}) # -- Root

add_subdirectory(src)
    # src/CMakeLists.txt를 실행하기 전에 일부 변수들이 새로 설정된다
    #
    #   message(STATUS ${CMAKE_SOURCE_DIR})         # -- Root
    #   message(STATUS ${CMAKE_CURRENT_SOURCE_DIR}) # -- Root/src
    #

add_subdirectory(test)
    #
    #   message(STATUS ${CMAKE_SOURCE_DIR})         # -- Root
    #   message(STATUS ${CMAKE_CURRENT_SOURCE_DIR}) # -- Root/test
    #

# ...
```

CMAKE_MODULE_PATH

앞서서는 `include` 를 위해 아래와 같이 cmake 모듈의 상대 경로를 사용 했었습니다.

```
# ...
include(cmake/check-compiler-flags.cmake) # 경로를 자세하게 제공한 경우
# ...
```

경로를 참조할 때 특정 경로를 참고하도록 지시할 수도 있습니다. `CMAKE_MODULE_PATH` 를 사용하면, 파일 이름만으로 `include` 하는 것이 가능합니다. 시스템 환경변수 `PATH` 와 유사하다고 생각하셨다면 정답입니다.

```
# 현재 프로젝트를 기준으로 cmake 폴더를 CMAKE_MODULE_PATH에 추가한다
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake)

include(check-compiler-flags) # 위치한 폴더, .cmake 확장자를 생략해도 문제없다
```

CMake에서 미리 제공하는 모듈들은 https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html 에서 확인할 수 있습니다.

설치

설치 경로 지정

실행파일은 exe와 dll (혹은 elf와 so)와 같은 binary만 있으면 되지만, 라이브러리는 좀 다릅니다. 엄밀히 말해 빌드된 라이브러리 파일(lib, dll, a, so, dylib ...)과 함께 **링킹을 위한 symbol 정보**가 함께 제공되어야 하기 때문입니다.
원래라면 exp(export) 파일을 사용하는게 맞겠지만, 개발자의 편의를 생각하면 .h, .hpp 파일을 넘어서기 어려울 것입니다.

`install`

CMake에서는 `install` 명령으로 헤더파일, CMake Target, 그리고 필요하다면 폴더를 지정된 위치에 '설치(복사)' 하는 방법을 제공합니다.

```
# 단일 파일을 지정된 폴더에 설치
install(FILE            LICENSE     # ReadMe와 같이 라이브러리와 함께 배포되어야 하
        DESTINATION     ./install/
)
# 폴더 전체를 설치
install(DIRECTORY       include     # 헤더 파일들을 통째로 옮긴다
        DESTINATION     ./install/
)
# 빌드 결과물을 설치
install(TARGETS         my_new_library  # add_library, add_executable에 사용했던
        DESTINATION     ./install/
)
```

보통 CMake 프로젝트에서 설치의 대상은 다음 3가지가 있습니다. 엄밀히 말하면 `.cmake` 파일을 설치하는 경우도 있기 때문에 더 많은 종류가 있다고 할 수 있지만, 여러분의 프로젝트가 CMake를 지원하지 않을수도 있으므로 여기서는 설명을 생략하겠습니다. (오직 빌드만을 위해서 CMake를 사용하는 경우)

- 프로그램
- 파일
- 폴더

`CMAKE_INSTALL_PREFIX`

하지만 하위 프로젝트들도 제각기 설치 경로를 가지고 있다면 정리하기 어려울 것입니다. 이를 위해 CMake에서는 지정 설치 경로를 의미하는 `CMAKE_INSTALL_PREFIX` 변수가 있습니다.

하위 프로젝트에서 설치 경로를 지정할 때 이 변수를 사용하도록 하면 상위 프로젝트에서 일괄함께 배포하는데 도움을 줄 수 있습니다.

```
# 설치를 CMakeList를 기준으로 하지 않고 CMAKE_INSTALL_PREFIX를 기준으로 수행한다

# 하나의 파일을 옮기는 경우
install(FILE            LICENSE
```

```
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/install/
)

# 특정 폴더를 옮기는 경우
install(DIRECTORY       ${CMAKE_CURRENT_SOURCE_DIR}/include
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/install/
)

# add_library, add_executable에 사용한 이름을 설치하는 경우
install(TARGETS         my_new_library
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/install/
)
```

이 변수는 특히 커맨드라인에서 자주 지정하는 변수이기도 합니다. 아래와 같이 설정이 다른 경우 설치 폴더를 분리해서 배포, 경로 참조를 쉽게합니다.

```
cmake /path/to/CMakeLists.txt \
    -DCMAKE_INSTALL_PREFIX=~/install/debug/static \    # debug, static
    -DCMAKE_BUILD_TYPE=Debug \
    -DBUILD_SHARED_LIBS=false;
cmake /path/to/CMakeLists.txt \
    -DCMAKE_INSTALL_PREFIX=~/install/debug/dynamic \   # debug, dynamic
    -DCMAKE_BUILD_TYPE=Debug \
    -DBUILD_SHARED_LIBS=true;

cmake /path/to/CMakeLists.txt \
    -DCMAKE_INSTALL_PREFIX=~/install/release/static \  # release, static
    -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_SHARED_LIBS=false;
cmake /path/to/CMakeLists.txt \
    -DCMAKE_INSTALL_PREFIX=~/install/debug/dynamic \   # release, dynamic
    -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_SHARED_LIBS=true;
```

위와 같이 실행하고 나면 ~/install 경로에는 아래와 같이 설치될 것입니다.

```
tree -L 3 ~/install
install
├── LICENSE
├── include
│   └── my_new_lib.h
├── release
│   ├── static
│   │   └── my_new_library.a
│   └── release
│       └── my_new_library.so
└── debug
    ├── static
    │   └── my_new_library.a
    └── release
        └── my_new_library.so
```

빌드 산출물이 여러 종류라면 `install(TARGETS)` 에 아래와 같이 `DESTINATION` 들을 보다 구체적으로 지정하는 것이 좋습니다.

```
install(TARGETS  foo_lib bar_exe
                 header_only_lib
        INCLUDES DESTINATION ${CMAKE_INSTALL_PREFIX}/include
        RUNTIME  DESTINATION ${CMAKE_INSTALL_PREFIX}/bin
        LIBRARY  DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
        ARCHIVE  DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
)
```

tutorial-p2.md

CMake 튜토리얼 Lv.2

CMake의 목적과 기능범위

빌드 시스템(build system)이 하는 일은 연속적인 이벤트(컴파일러/링커 호출)를 통해 최종적으로는 프로그램을 생성하는 것입니다. 빌드 시스템 파일, 혹은 프로젝트 파일은 프로그래머의 의도에 맞게 컴파일러/링커 호출을 조직화한 명령서라고 할 수 있습니다.

1편에서 강조한 대로라면 CMake는 '명령서'를 작성하는 것으로 그 기능이 충분하겠지만, 실제 기능적으로는 좀 더 넓은 지원범위를 가지고 있습니다. CMake가 중심이 되는 프로젝트라면, 이를 활용해 빌드에 필요한 소스 파일까지 생성할 수 있습니다.

> 작성자는 이 기능들을 사용하는 것을 추천하지 않습니다.
> 빌드 시스템 파일을 생성하는것이 CMake의 가장 중요한 부분이며, 오직 그 일에 집중해야 한다고 생각하기 때문입니다

- CMake의 목적: 프로젝트(빌드 시스템) 파일 생성
- CMake의 실제 지원범위

- 프로젝트 생성 및 속성, 의존성 설정
  - `add_library` , `add_executable` , `set_target_properties` , `add_dependencies` …
- 프로그램 호출
  - `add_custom_command`
  - `add_custom_target`

CMake에서 Build로 이어지는 과정

CMake를 사용해 빌드 시스템 파일을 생성하는 과정은 2 pass assembler와 유사하다고 할 수 있습니다.

- Configuration
  - `CMakeLists.txt` 문법 검사
  - Macro, Function 실행
  - Cmake Cache 생성
- Generation
  - **Target**에서 Build System Project 생성

Configuration 단계에서는 CMakeList를 해석하고, 재사용 가능한 값을 토대로 `CMakeCache.txt` 를 생성합니다. CMakeFiles 폴더 역시 이 단계에서 생성됩니다. 이 폴더 안에는 CMake의 log, 변경을 확인하는 stamp 파일들이 보관됩니다.

Generation은 Configuration의 정보를 바탕으로 Build System에서 사용하는 Project 파일을 생성합니다. `TargetDirectories.txt` 와 같은 폴더목록도 이 단계에서 생성됩니다.

Command

파일 생성

앞서 지원범위에서 CMake는 소스 파일을 생성할 수 있다고 설명하였습니다. 여기에 사용되는 것이 바로 command 입니다.

만약 존재하지 않는 파일이 CMakeList에 명시되면, CMake는 이를 오류로 처리합니다.

```
$ tree ./wierd-project
./wierd-project
├── CMakeLists.txt
├── include
│   └── simple.h
└── src
    └── main.cpp
```

위의 프로젝트는 src폴더에 main.cpp 밖에 없습니다.
이때 CMakeList의 내용이 아래와 같다면

```
cmake_minimum_required(VERSION 3.8)

add_executable(wierd_exe
    include/simple.h

    src/main.cpp
    src/simple.cpp  # <-- ???
)

target_include_directories(wierd_exe
PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/include
)
```

다음과 같은 오류를 출력할 것입니다.

```
CMake Error at CMakeLists.txt:3 (add_executable):
  Cannot find source file:

    src/simple.cpp

  Tried extensions .c .C .c++ .cc .cpp .cxx .m .M .mm .h .hh .h++ .hm .hpp
  .hxx .in .txx

CMake Error: CMake can not determine linker language for target: wierd_exe
```

요컨대 Build System 파일을 만들기 전에 CMake에서 소스 파일이 존재하지 않는다고 경고하는 내용입니다. 이 튜토리얼에서는 스크립트 파일을 호출해서 simple.cpp를 만들어 빌드에 사용해 보겠습니다.

add_custom_command

먼저 CMakeList에 새로운 내용이 추가되어야 합니다.

```
cmake_minimum_required(VERSION 3.8)

# we will generate the file with the given command
add_custom_command(
    OUTPUT      src/simple.cpp              # <-- output path
    COMMAND     echo    "my first command"  # <-- command + args
)

add_executable(wierd_exe
    include/simple.h

    src/main.cpp
```

```
    src/simple.cpp
)

target_include_directories(wierd_exe
PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/include
)
```

다시 CMake를 실행해보면 이번엔 문제없이 Generation 단계까지 마치는 것을 확인할 수 있을 것입니다.

```
PS C:\wierd-project\build> cmake .. -G "Visual Studio 15 2017 Win64"
-- Configuring done
-- Generating done
-- Build files have been written to: C:/wierd-project/build
```

하지만 echo 명령이 실제로 파일을 생성하지는 않기 때문에, 빌드를 시도하면 No such file or directory 메세지와 함께 실패할 것입니다.

`add_custom_command` 함수는 기본적으로 `OUTPUT` 와 `COMMAND` 인자만 제공하면 동작하지만, 보다 정확히 의도를 반영하기 위해서는 여러가지 인자를 제공해야 합니다. 이 함수를 처음 접한다면 천천히 이후의 내용을 읽어본 후, 인자를 바꿔가며 실행해보기를 권합니다.

먼저, 소스파일을 생성하는 스크립트를 작성해서, COMMAND에서 이를 호출하도록 하여 파일이 생성되는 위치를 확인할 수 있습니다.

```
$ tree ./wierd-project
./wierd-project
├── CMakeLists.txt
├── include
│   └── simple.h
├── scripts
│   └── create_simple_cpp.sh # <--- src file generation script
└── src
    └── main.cpp
```

소스 파일을 생성하는 스크립트의 내용은 아주 단순합니다

```
# create_simple_cpp.sh
echo "extern const int version = 3;" > src/simple.cpp;
```

위 내용은 Windows의 CLI에서도 실행될 수 있기 때문에 따로 셔뱅(shebang)을 두지 않았습니다. UNIX 환경이라면 기본 shell 환경(zsh, bash 등)을 따르면 됩니다. 예시에서는 bash로 가정하겠습니다.

```
cmake_minimum_required(VERSION 3.8)

# more detailed src file generation command
if(WIN32)
    add_custom_command(
        OUTPUT      src/simple.cpp
        COMMAND     call ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        COMMENT     "creating simple.cpp"
    )
elseif(UNIX)
    add_custom_command(
        OUTPUT      src/simple.cpp
        COMMAND     bash ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        COMMENT     "creating simple.cpp"
    )
endif()

# ... add_executable ...
```

Windows에서는 Command Prompt를 사용하게 됩니다. (call 을 사용하는 것을 보고 직감하셨나요?) Powershell이 아님에 주의하시기 바랍니다.
이대로 CMake를 실행하면 Comment가 출력되지 않는 것을 볼 수 있습니다. 이는 Configuration/Generation 단계에서는 command가 실행되지 않는다는 의미입니다.

실제로 빌드를 실행했을때 comment가 출력된 것을 볼 수 있습니다.

```
/path/to/wierd-project/build$ make
[ 25%] creating simple.cpp                      # <-----
Scanning dependencies of target wierd_exe
[ 50%] Building CXX object CMakeFiles/wierd_exe.dir/src/main.cpp.o
[ 75%] Building CXX object CMakeFiles/wierd_exe.dir/src/simple.cpp.o
[100%] Linking CXX executable wierd_exe
[100%] Built target wierd_exe
```

이제까지는 상대경로를 사용하면 CMakeList를 기준으로, 즉 SOURCE_DIR를 기준으로 파일을 참조하였습니다. 하지만 `add_custom_command` 의 생성 파일은 빌드 폴더(BINARY_DIR)를 기준으로 합니다.

> 만약 여기서 빌드가 실패했고, **스크립트 실행 중 no such file or directory 오류**가 발생한다면,
> 이는 build 폴더 내에 src 폴더가 없기 때문일 것입니다. (create_simple_cpp.sh 스크립트에는 폴더를 생성하는 내용이 없기 때문)
> 그런 경우 src 폴더를 생성한 뒤 다시 시도해보시기 바랍니다.

프로젝트 폴더를 tree로 조회하면 src 폴더에는 여전히 main.cpp만 존재하는 것을 확인할 수 있습니다.

```
/path/to/wierd-project$ tree -L 3 .
.          # <--------------- CMAKE_CURRENT_SOURCE_DIR
├── CMakeLists.txt
├── build       # <--------------- CMAKE_CURRENT_BINARY_DIR
│   ├── CMakeCache.txt
│   ├── CMakeFiles
│   │   ├── 3.10.2
│   │   ...
│   │   └── wierd_exe.dir
│   ├── Makefile
│   ├── cmake_install.cmake
│   ├── src
│   │   └── simple.cpp  # <-------- generated file
│   └── wierd_exe
├── include
│   └── simple.h
├── scripts
│   └── create_simple_cpp.sh
└── src
    └── main.cpp    # <----- where is my friend?
```

이런 경우, 빌드 과정에서 생성하는 파일을 어떻게 관리할 것인지 먼저 생각해봐야 합니다. 이 튜토리얼에서는 다음의 2가지 방법을 보이겠습니다.

- Working directory를 지정
- Script에 인자를 제공

Working Directory + `add_custom_command`

CLI 환경에서 working directory는 명령을 호출한 위치를 의미합니다. Bash에서는 `pwd`, PowerShell에서는 `Get-Location` 명령으로 확인할 수도 있습니다. 바로 위에서 실행한 tree의 경우, `/path/to/wierd-project` 가 working directory가 됩니다.

`add_custom_command` 는 선택 인자로 `WORKING_DIRECTORY` 를 명시할 수 있습니다. 이 경로를 SOURCE_DIR로 바꿔서 실행시켜 보겠습니다.

```
# ...
if(WIN32)
    add_custom_command(
        OUTPUT      src/simple.cpp
        COMMAND     call ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
        COMMENT     "creating simple.cpp"
    )
elseif(UNIX)
    add_custom_command(
        OUTPUT      src/simple.cpp
        COMMAND     bash ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
        COMMENT     "creating simple.cpp"
```

```
    )
endif()
# ...
```

이렇게 수정하면 아래와 같이 **오류가 발생해야 합니다**.

```
/path/to/wierd-project/build$ make
[ 25%] creating simple.cpp
Scanning dependencies of target wierd_exe
[ 50%] Building CXX object CMakeFiles/wierd_exe.dir/src/main.cpp.o
[ 50%] creating simple.cpp
[ 75%] Building CXX object CMakeFiles/wierd_exe.dir/src/simple.cpp.o
c++: error: /path/to/wierd-project/build/src/simple.cpp: No such file or director
c++: fatal error: no input files
compilation terminated.
CMakeFiles/wierd_exe.dir/build.make:90: recipe for target 'CMakeFiles/wierd_exe.c
make[2]: *** [CMakeFiles/wierd_exe.dir/src/simple.cpp.o] Error 1
CMakeFiles/Makefile2:67: recipe for target 'CMakeFiles/wierd_exe.dir/all' failed
make[1]: *** [CMakeFiles/wierd_exe.dir/all] Error 2
Makefile:83: recipe for target 'all' failed
make: *** [all] Error 2
```

Windows 환경, Generator가 Visual Studio인 경우에도 마찬가지입니다.

```
Build FAILED.

"C:\wierd-project\build\ALL_BUILD.vcxproj" (default target) (1) ->
"C:\wierd-project\build\wierd_exe.vcxproj" (default target) (3) ->
(ClCompile target) ->
  c1xx : fatal error C1083: Cannot open source file: 'C:\wierd-project\build\src\

    0 Warning(s)
    1 Error(s)

Time Elapsed 00:00:02.92
```

오류메세지의 경로를 확인해보면 두 경우 모두 build 폴더에서 src/simple.cpp를 찾으려 했다는 것을 알 수 있습니다. **이는 `add_custom_command` 의 OUTPUT이 묵시적으로 BINARY_DIR를 기준으로 하기 때문입니다.** OUTPUT 인자를 절대경로로 변경하면 빌드가 성공하는 것을 확인할 수 있을 것입니다.

```
cmake_minimum_required(VERSION 3.8)

if(WIN32)
    add_custom_command(
        OUTPUT      ${CMAKE_CURRENT_SOURCE_DIR}/src/simple.cpp
```

```
        COMMAND     call ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
        COMMENT     "creating simple.cpp"
    )
elseif(UNIX)
    add_custom_command(
        OUTPUT      ${CMAKE_CURRENT_SOURCE_DIR}/src/simple.cpp
        COMMAND     bash ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
        COMMENT     "creating simple.cpp"
    )
endif()

add_executable(wierd_exe
    include/simple.h

    src/main.cpp
    src/simple.cpp
)

target_include_directories(wierd_exe
PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/include
)
```

동시에 src 폴더에 main.cpp와 simple.cpp가 함께 위치하는 것도 확인할 수 있습니다

Script with Arguments + `add_custom_command`

다르게 생각해보면, 코드 생성 스크립트가 너무 단순하다는 것도 문제의 원인일 수 있습니다.

스크립트 파일의 위치, 혹은 프로젝트 폴더와 같이 묵시적인 정보를 기반으로 (상대경로를 써서) 내용을 작성했지만, 실제 스크립트는 완전히 다른 경로에서 실행되고 있을 수 있다는 점을 간과한 것이죠. 스크립트가 현재 실행되는 위치와 무관하게 동작해야 할 수도 있습니다. 절대 경로를 인자로 제공받는다면 이 문제를 해결할 수 있을 것입니다.

```
# script can catch argument with $1, $2 ...
PROJECT_DIR=$1
echo "extern const int version = 3;" > $PROJECT_DIR/src/simple.cpp;
```

이제 COMMAND를 통해 스크립트에서 인자를 전달하면 됩니다. 하지만 앞서 working directory에서 확인한 것처럼, CMake가 생성한 프로젝트는 여전히 build/src/simple.cpp를 찾을 것입니다. 따라서 OUTPUT에는 절대 경로가 필요합니다.

```
# more detailed src file generation command
if(WIN32)
    add_custom_command(
```

```
        OUTPUT      ${CMAKE_CURRENT_SOURCE_DIR}/src/simple.cpp
        COMMAND     call ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        COMMENT     "creating simple.cpp"
    )
elseif(UNIX)
    add_custom_command(
        OUTPUT      ${CMAKE_CURRENT_SOURCE_DIR}/src/simple.cpp
        COMMAND     bash ${CMAKE_CURRENT_SOURCE_DIR}/scripts/create_simple_cpp.sh
        COMMENT     "creating simple.cpp"
    )
endif()
```

두 방법의 차이점은 Working Directory 방법이 순수하게 CMakeList의 변경만으로 해결된 반면, Script Argument 방법은 스크립트 파일도 변경해야 했다는 부분에 있습니다. 이는 단점이 될 수 있지만, out-of-tree build가 기본 빌드 시나리오인 경우, 혹은 다수의 스크립트가 함께 사용되는지에 따라 더 타당한 판단이 될 수 있습니다.

Target

추가 설명

1편에서는 Target을 '빌드 시스템 파일 생성의 단위'라고 설명하였습니다. CMake에서 Target은 'Configuration의 대상'을 말하며, 이는 최종적으로는 Build System에서 사용하는 'Project로 Generation'됩니다. 쉽게 말해, **CMake에서의 빌드 단위**라고 요약할 수 있습니다.

초반부에 Project 파일은 '명령서'와 같다고 설명하였습니다. 보통 이 '명령'은 컴파일러/링커 호출을 의미하지만, 좀 더 일반적인 일도 포함될 수 있습니다. 일례로 Unix Makefiles 프로젝트들은 보통 install/uninstall을 지원하며, Visual Studio는 Build event에 사용자 커맨드를 호출할 수 있도록 지원합니다.

Command와 Target은 의존성을 설정할 수 있고, CMake 파일의 작성자가 실행 내용(COMMAND)을 명시한다는 점이 유사합니다. 하지만 Target은 Name, Property를 가진 보다 복잡한 개체를 의미합니다. CMake 공식문서에서는 Target을 다음과 같이 구분합니다.

- Binary Target
  - executable
  - library
- Pseudo Target
  - Imported Target
    - pre-existing dependency
  - Alias Target
    - read-only name

이 중 Binary target은 실제로 빌드 시스템 파일을 생성하는 경우를 의미하며, Pseudo target은 이미 존재하는 파일을 사용하는 경우만을 의미합니다. 이 문서에서는 지금까지 Binary Target 만을 중점적으로 서술했는데, 다른 타입의 Target들을 짚어보겠습니다.

Pseudo Target

Imported Target

지금까지 빌드를 위해 사용해왔던 `add_executable` , `add_library` 모두 IMPORTED 옵션을 지원합니다.

```
add_executable(protoc_exe IMPORTED /usr/bin/protoc)

add_library(xxx IMPORTED SHARED)
add_library(yyy IMPORTED STATIC)
```

`add_executable(IMPORTED)` 는 `add_custom_command` 의 COMMAND로 사용합니다.

이미 빌드 된 라이브러리가 있는 경우 `add_library(IMPORTED)` 를 사용합니다. 해당 라이브러리가 CMake가 아닌 빌드 툴을 사용해서 빌드를 수행했더라도, 별도의 CMake 파일을 작성하여 이를 CMake 프로젝트에 결합시킬 수 있습니다.

다만 이때는 소스 파일에서부터 빌드를 수행하는 것이 아니기 때문에, `CMakeLists.txt` 가 아닌 다른 cmake 설정(config) 파일을 작성하게 됩니다.

`CMakeLists.txt` 가 아닌 다른 파일을 작성한다는 것에 의아할 수 있겠지만, 라이브러리가 미리 빌드되었다는 것은 아래와 같은 정보를 전달받아야 한다는 의미이므로, **빌드와는 의미가 달라지기 때문**이라고 해석할 수 있습니다. (같은 것은 같게, 다른 것은 다르게)

- 헤더 파일을 찾을 폴더(`target_include_directory`)
- 라이브러리 의존성(`target_link_libraries`)
- 빌드할 당시에 사용한 옵션(`target_compile_options`)

이는 `find_package` 부분에서 다룰 것입니다.

Alias Target

이 부분은 OpenMP을 예로 들어 간단하게만 짚고 넘어가겠습니다.

```
cmake_minimum_required(VERSION 3.8)

# create target without source file list
add_library(xyz IMPORTED SHARED)

find_package(OpenMP) # https://cmake.org/cmake/help/latest/module/FindOpenMP.html
if(OpenMP_FOUND)
    set_target_properties(xyz
```

```
    PROPERTIES
        COMPILE_FLAGS "${OpenMP_CXX_FLAGS}"
    )

    if(ANDROID)
        set_target_properties(xyz
        PROPERTIES
            INTERFACE_LINK_LIBRARIES    omp     # <----- ???
        )
    else()
        set_target_properties(xyz
        PROPERTIES
            INTERFACE_LINK_LIBRARIES     OpenMP::OpenMP_CXX # <----- ???
        )
    endif()
endif()
```

만약 이것이 특정한 빌드 시스템에서 사용하는 파일이었다면 `omp` 보다는 좀 더 구체적인 이름을 사용하고 있었을 것입니다. 플랫폼에 따라서 `omp.dll`, `omp.lib`, `libomp.a`, `libomp.so`, `libomp.dylib` 등이 될 것입니다.

이 점을 생각하면 `omp` 가 CMake Target의 이름이라는 것을 알 수 있습니다. CMake 파일들은 특정 플랫폼에서 사용하는 이름, 확장자에 대해 거의 신경을 쓰지 않으며, 가능한 그렇게 되도록 작성합니다. 달리 말해, CMake의 방식을 우선적으로 고려합니다.

```
target_link_libraries(some_exe
PRIVATE
    xyz
)
```

별다른 주석이 없다면, CMake와 그 파일의 사용자는 위 내용에 대해서 다음과 같은 생각을 할 것입니다.

1. xyz라는 이름의 CMake Target이 존재한다
2. 만약 xyz가 Target이 아니라면, xyz는 라이브러리를 의미한다
    i. `lib[xyz].a` 혹은 `lib[xyz].so` 와 같은 이름의 파일이 `link_directories` 혹은 `target_link_directories` (CMake 3.13부터 지원)로 지정한 폴더들 안에 존재한다
    ii. `[xyz].lib` 혹은 `[xyz].dll` 와 같은 이름의 파일이 시스템 라이브러리 폴더에 존재한다

이것이 의미하는 바는, `xyz` 라는 이름만으로는 **혼동의 여지가 있다**는 것입니다. 앞선 OpenMP 예시에서 `::` 을 사용한 부분이 있었습니다. 이를 (C++ 개발자라면 모두가 친숙한) Namespace라고 합니다. 아래와 같이 사용할 수 있습니다.

```
# ...
elseif(NOT WIN32)
    set_target_properties(custom_mp
    PROPERTIES
        INTERFACE_LINK_LIBRARIES    OpenMP::OpenMP_CXX
    )
endif()
```

이런 ALIAS를 발견한다면, CMake와 읽는이 모두 이것이 Target의 이름이라고 확신할 수 있습니다. Alias Target은 아래와 같이 매우 간단한 방법으로 추가할 수 있습니다.

```
# https://github.com/catchorg/Catch2/blob/master/CMakeLists.txt

# ...
add_library(Catch2 INTERFACE)
# ...
add_library(Catch2::Catch2 ALIAS Catch2)
# ...
```

Custom Target

add_custom_target

이 함수와 `add_custom_command` 의 결정적인 차이점은 `add_custom_command` 은 파일을 생성하기 위한(`OUTPUT`) 기능이고, `add_custom_target` 는 여러 명령을 (이름으로) 묶어서 실행하기 위한 기능이라는 점입니다. 그리고 Target이기 때문에 `add_dependencies` 의 대상이 될 수 있다는 점이 있습니다.

예를 들어, html파일을 하나 다운로드 받아, 이를 첨부하여 메일을 보내는 일은 아래처럼 작성할 수 있습니다. 두 기능 모두 cURL을 사용해 진행하겠습니다.

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.8) # <--- Just any of 3.x will be fine

add_custom_target(get_index_html
    COMMAND curl -L "https://cmake.org/cmake/help/latest/"
                 -o "index.html"
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
)

add_custom_target(mail_index_html
    COMMAND curl --url 'smtps://smtp.gmail.com:465'
                 --ssl-reqd  --mail-from 'sender@mail.com'
                 --mail-rcpt 'receiver@mail.com'
                 --upload-file "${CMAKE_CURRENT_SOURCE_DIR}/index.html"
                 --user 'sender@mail.com:$ENV{sender_password}'
    WORKING_DIRECTORY ${CMAKE_INSTALL_PREFIX}
```

```
)
add_dependencies(mail_index_html get_index_html)
```

굉장히 사악한 행동을 하고 있는데, 자세히 보셨다면 `$ENV{sender_password}` 를 사용해 CMake 변수가 아니라 시스템 환경변수를 참조하는 것을 볼 수 있습니다. 아래와 같이 CMake를 호출하기 전에 환경변수의 값을 정해주면 문제가 없습니다.

예를들어, 여러분이 Bash 사용자라면:

```
export sender_password="the_real_password"
```

만약 PowerShell 사용자라면:

```
$env:sender_password="the_real_password"
```

처럼 환경변수를 설정할 수 있습니다.

위 예시는 UNIX/Linux cURL을 사용하므로, CMake를 바로 다음에 호출하면 아마 아래와 같은 내용을 보실 수 있을 것입니다.

> **주의**: Powershell에서 curl명령은 `Invoke-WebRequest` 의 Alias입니다.
> 빌드를 수행하지 않는 예시이기 때문에 간단히 WSL(Windows Subsystem for Linux)를 사용하면 Linux cURL을 실행할 수 있습니다.

```
$ cmake .
-- The C compiler identification is GNU 9.1.0
-- The CXX compiler identification is GNU 9.1.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
```

`mail_index_html` 은 Target의 이름이므로, 이를 실행하려면 `--target` 파라미터를 명시하는 것으로 충분합니다.

```
cmake --build . --target mail_index_html
```

```
$ cmake --build . --target mail_index_html
Scanning dependencies of target get_index_html
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  8942  100  8942    0     0   7675      0  0:00:01  0:00:01 --:--:--  7675
Built target get_index_html
Scanning dependencies of target mail_index_html
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  8942    0     0  100  8942      0   2888  0:00:03  0:00:03 --:--:--  2888
Built target mail_index_html
```

sender@mail.com와 receiver@mail.com에 자신의 메일주소와 비밀번호를 입력해서 실행해보시기 바랍니다.

빌드를 마친 뒤 `${CMAKE_INSTALL_PREFIX}` 를 Zip으로 압축해서 Maintainer에게 메일로 보낼 수 있다면 멋질 것 같군요.

```
cmake --build . --target install
cmake --build . --target mail_build_outputs
```

tutorial-p3.md

CMake 튜토리얼 Lv.3

Package

Package의 구성

보통 패키지라고 하면 Chocolaty, NuGet, RPM, Brew처럼 관리 소프트웨어를 통해 다운로드/설치/업데이트해서 사용하는 프로그램들(+ 문서)을 말하는데, C++ 프로그래머들에게 패키지란 개발에 필요한 Library + Manifest에 가까운 것 같습니다.

- 일반적인 패키지:
  - 실행 프로그램(executable)
  - 문서 파일(license, manual, readme 등)
- 프로그래밍 패키지: 일반 패키지 + 개발에 필요한 요소들
  - 서브 프로그램(library)
  - 실행 프로그램(test tools, script 등)
  - 소스 코드(include, example 등)

C++ 에서는 미리 빌드된 서브 프로그램 뿐만 아니라 소스 코드가 포함된다는 점(include)이 특이하다고 할 수 있습니다. 비단 템플릿 프로그래밍의 비중이 늘어난 것 뿐만 아니라 크로스 컴파일과 링킹에 손이 많이 가기 때문이기도 할 것입니다.

지금은 많은 C++ 프로젝트들이 Unix Filesystem에서 표준 C 라이브러리를 배치할때 사용하던 파일트리 구조를 적용하고 있습니다. 굳이 이런 배치에 어떤 의미가 부여되어있다기 보다는, "CMake의 초창기부터 Unix 시스템에 빌드 된 라이브러리을 설치하면서 관례를 따르던 것이 이어지고 있다"정도로 생각하면 될 것 같습니다.

- bin : 실행 프로그램(executable)
- lib : 미리 빌드된 라이브러리(so, lib 등)
- include : 소스 코드(헤더)
- share : 기타 필요한 파일들. 주로 빌드 지원 파일
  - docs : 문서가 (많이) 있는 경우 따로 두기도

CMake의 Package 찾기

`find_package`

이미 설치된 패키지를 찾는 기능으로 CMake는 `find_package` 를 제공하고 있습니다. CMake의 패키지를 어떻게 만드는지에 앞서서, 어떻게 사용하는지부터 짚고 넘어가겠습니다.

잠시 미리 적자면, 여러분이 사용하는 라이브러리가 CMake를 지원하는 경우, `find_package` 가 매끄럽게 사용되지 않을 때는 `add_subdirectory` 를 사용하는 것이 '정확한' 해결책이 될 수 있습니다. Package export에 문제가 있는 경우 이를 찾아내기에도, **Import하는 쪽에서 수정하기에도 어렵기 때문**입니다.

이 함수는 일반적으로는 아래와 같이 이름과 버전을 인자로 사용합니다. 탐색에 성공하면 `name_FOUND` 변수가 생성됩니다. 아래 예시처럼 이름으로 `OpenCV` 를 사용했다면, 성공여부는 `OpenCV_FOUND` 로 확인할 수 있습니다.

```
# optional import
find_package(OpenCV 3.3)
if(OpenCV_FOUND)
    # ...
    # target_source:  Add OpenCV related source codes ...
    # target_compile_options:  Enable RTTI for OpenCV ...
    # ...
endif()

# mandatory import
find_package(OpenCV 3.3 REQUIRED)
```

좀더 상세하게 패키지 탐색을 위한 정보를 제공하는 경우, `CONFIG` 를 사용해 Config Mode로 호출하게 됩니다.

PATHS를 수정하여도 제대로 찾지 못한다면, CMake Cache의 문제일 가능성이 높습니다. 그런경우 CMakeCache.txt 를 제거하고 다시 CMake를 실행시켜보시기 바랍니다

```
# cmake might find multiple packages.
# In the case it will peek the first one
find_package(fmt  5.3
CONFIG
    REQUIRED
    PATHS       C:/vcpkg/installed/x64-windows
                /mnt/vcpkg/installed/x64-linux
)
```

수많은 컴포넌트를 가진 Boost에서 필요한 모듈만 가져다 쓴다면 아래처럼 작성하면 될 것입니다. 분명히 설치 되었음에도 CMake에서 찾지 못한다면 CONFIG를 지우고 다시 시도해보시면 찾을수도 있습니다.

> 작성자도 아직 `CONFIG` 의 유무가 탐색에 미치는 영향을 명확하게 알아내지는 못했습니다.
> 대부분의 패키지들은 CONFIG를 함께 쓰면 별 문제없이 탐색에 성공하는 것을 확인했습니다.
> 아마도 CMake로 export 했는지 여부가 영향을 미치는 것 같다고 짐작할 뿐입니다.

```
find_package(Boost  1.59
CONFIG                      # <--- try without CONFIG if the function fails !
    REQUIRED
    COMPONENTS system thread timer
)
```

CMake에서 `find_package` 를 호출하면, 해당 함수는 Package를 찾고, 그 안에 있는 Target들을 가져옵니다(`add_library(IMPORTED)`).
물론 executable과 링킹을 하지는 않기 때문에, 가져온 Target들을 `add_library(INTERFACE)` 혹은 `add_library(SHARED)` 로 만들어진 결과물들입니다. 따라서 이들을 소비하는 함수는 `target_link_libraries` 입니다.

```
find_package(gRPC CONFIG REQUIRED)
# ...
target_link_libraries(main
PRIVATE
    gRPC::gpr gRPC::grpc gRPC::grpc++ gRPC::grpc_cronet
)
```

물론 여기에는 하나의 전제가 있습니다. 해당 라이브러리가 CMake에서 Import할 수 있도록 적절하게 Manifest를 작성해 놓았거나, CMake의 `export` 함수를 사용해 CMake를 위한 Manifest를 생성해놓은 것입니다.

> Manifest라는 표현은 **이 문서 상에서만** "Package를 내보낸것과 같이 가져오기 위한 목록"이라는 의미로 사용하기에 웹에서 CMake관련 검색할 때 사용하면 오히려 방해가 될 수 있습니다.

어떤 파일을 제공해야 하는지 알아보기 위해 아래와 같이 CMakeLists.txt를 작성해 실행해보겠습니다.

```
cmake_minimum_required(VERSION 3.8)

find_package(TBB REQUIRED)
```

Intel TBB가 설치되지 않은 환경에서 `find_package` 가 실패하면서 아래와 같은 오류를 출력할 것입니다.

```
$ cmake .
...
-- Detecting CXX compile features - done
CMake Error at CMakeLists.txt:3 (find_package):
  By not providing "FindTBB.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "TBB", but
  CMake did not find one.

  Could not find a package configuration file provided by "TBB" with any of
  the following names:

    TBBConfig.cmake
    tbb-config.cmake

  Add the installation prefix of "TBB" to CMAKE_PREFIX_PATH or set "TBB_DIR"
  to a directory containing one of the above files.  If "TBB" provides a
  separate development package or SDK, be sure it has been installed.
```

```
-- Configuring incomplete, errors occurred!
```

이를 통해 `find_package` 에서 TBB라는 이름을 가지고 대소문자가 혼합된 경우 (`TBBConfig.cmake`)와 소문자만 사용된 경우(`tbb-config.cmake`)를 고려하여 Manifest파일을 찾으려 했다는 것을 알 수 있습니다.

`-config.cmake`

이전까지는 Manifest파일이라고 하였으나, 이후로는 `-config.cmake` 파일이라고 하겠습니다.

TBB를 설치하면 TBBConfig.cmake가 생성된 것을 확인할 수 있습니다. 다행히 TBB의 `-config.cmake` 파일은 비교적 짧은 편에 속합니다. Details를 열어 한번 읽어보시기 바랍니다.

▶ TBBConfig.cmake <---------- click me !!!!

다소 정리되지 않았다는 느낌이 있지만, 크게 3가지 정도를 눈여겨 볼 수 있습니다.

1. `add_library(IMPORTED)` 를 사용해서 CMake Target을 생성합니다. 이름으로는 `TBB::${_tbb_component}` 를 사용해서 이것이 CMake Target이라는 점을 분명히 드러내고 있습니다.
2. `set_property` 함수를 사용해서 DEBUG/RELEASE 설정으로 빌드되었다는 정보를 추가하는 것을 볼 수 있습니다.
3. `set_target_properties` 함수에서 IMPORTED_LOCATION를 사용해 .lib파일의 위치를 지정하거나, INTERFACE_LINK_LIBRARIES를 사용해 `TBB::tbbmalloc_proxy` 에서 `TBB::tbbmalloc` 를 링킹하도록(의존하도록) 만들고 있습니다.

요약하자면 `find_package` 가 하는 일은 `target_link_libraries` 에서 적합한 정보(Property)을 받아서 실제 Build System에서 필요로 하는 Linking 정보를 생성할 수 있도록 하는 Target Builder라고 할 수 있겠습니다.

Property

CMake에서는 굉장히 많은 Property를 정의하고 있습니다. 특히 이들을 사용하기 어렵게 만드는 것은, Target의 타입에 따라서 사용할 수 있는 property가 달라진다는 것입니다.

한때 작성자는 QMake를 볼때처럼 좀 읽다보면 이해하게 되겠거니 하였지만 아주 멍청하고 어리석은 생각이었습니다. 여러분은 직접적으로 Property를 조작하는 일을 멀리하고 참고영상에 나온 용례들을 보시면서 따라하시는게 낫습니다.

`set_property` / `get_property`

솔직히 적건대 작성자는 `define_property`, `set_property`, `get_property` 를 쓰는 경우는 `-config.cmake` 를 제외하고 아직 보지 못했습니다. 여기서는 단순히 함수의 시그니처만 확인할 수 있도록 실행가능한 예시를 적어놓겠습니다.

```
cmake_minimum_required(VERSION 3.8)
add_library(xyz UNKNOWN IMPORTED)

set_property(TARGET xyz APPEND PROPERTY
    IMPORTED_CONFIGURATIONS RELEASE
)
get_property(xyz_import_config TARGET xyz PROPERTY
    IMPORTED_CONFIGURATIONS
)

message(STATUS ${xyz_import_config})
```

set_target_properties

3.x 버전의 CMake에서 export 된 `-config.cmake` 파일들은 대부분 아래와 같은 Property 들을 설정합니다.

- `INTERFACE_INCLUDE_DIRECTORIES` : 헤더 파일이 위치한 폴더들
  `/usr/local/include;/usr/include` 형태로 ';'을 써서 여러 폴더를 지정할 수 있습니다.

- `INTERFACE_LINK_LIBRARIES` : 현재 Target의 의존성을 보여주는 부분입니다.
  `target_link_libraries` 에서 필요로 하는 인자, 즉 다른 CMake Target들의 이름을 ';'로 구분되는 목록을 사용해서 지정합니다.
  (`INTERFACE_INCLUDE_DIRECTORIES` 와 동일)

- `IMPORTED_LOCATION` : 서브 프로그램의 위치를 '절대경로'로 지정합니다.
  지금까지는 대부분 상대경로로 해결할 수 있었으나, 여기서 절대경로만을 허용하는 이유는 지금 `find_package` 하는 대상이 이미 **설치**되었기 때문일 것입니다.

- `IMPORTED_IMPLIB` : Windows의 경우 링킹을 위해 .lib파일이 필요하기도 합니다.
  다른 플랫폼에서는 사용되는 것을 보지 못했습니다.

실제 사용하는 모습은 다음과 같습니다.

```
add_library(xyz UNKNOWN IMPORTED)

set_target_properties(xyz
PROPERTIES
    INTERFACE_INCLUDE_DIRECTORIES   ${INTERFACE_DIR}
    INTERFACE_LINK_LIBRARIES        "OpenMP::OpenMP_CXX"
)

set_target_properties(xyz
```

```
  PROPERTIES
      IMPORTED_LOCATION   ${LIBS_DIR}/iphone/libxyz.a
  )
  set_target_properties(xyz
  PROPERTIES
      IMPORTED_IMPLIB     ${LIBS_DIR}/windows/xyz.lib
      IMPORTED_LOCATION   ${LIBS_DIR}/windows/xyz.dll
  )
```

덧붙여, Build Target을 작성할때 작성자는 언제나 `CXX_STANDARD` 를 명시합니다. 이는 `target_compile_options` 함수로 `/std:c++latest` 혹은 `gnu++2a` 를 추가하지 않아도 자동으로 추가하도록 해줍니다. 이 Property의 최대 값은 CMake 버전에 따라서 결정됩니다.

```
  cmake_minimum_required(VERSION 3.8)

  add_library(my_modern_cpp_lib
      src/libmain.cpp
  )

  set_target_properties(my_modern_cpp_lib
  PROPERTIES
      CXX_STANDARD 17
  )
```

작성자가 지금까지 항상 3.8 버전을 사용한 이유가 여기에 있습니다. CMake 3.14부터는 C++ 20을 명시할 수 있습니다.

```
  cmake_minimum_required(VERSION 3.14)

  add_library(my_modern_cpp_lib
      src/libmain.cpp
  )

  set_target_properties(my_modern_cpp_lib
  PROPERTIES
      CXX_STANDARD 20
  )
```

CMAKE_CURRENT_LIST_FILE

절대 경로를 지정해야 하는 경우, `/usr/local` 과 같이 잘 알려진 경로면 좋겠지만 그렇지 못한 경우 해당 `-config.cmake` 를 기준으로 탐색을 해야 할수도 있습니다. 여기에는 보통 `CMAKE_CURRENT_LIST_FILE` 변수가 사용됩니다.
이 변수는 `include` 되는 `.cmake` 파일의 위치를 저장하고 있습니다. 물론 `CMakeLists.txt` 도 예외가 아닙니다.

아래와 같이 파일이 배치되었다고 가정해보겠습니다.

```
$ tree $(pwd)
/path/to
├── CMakeLists.txt
└── cmake
    ├── print-current-path.cmake
    └── print-parent-path.cmake

1 directory, 3 files
```

각각의 내용이 아래와 같다면...

```
# cmake/print-current-path.cmake
message(STATUS "cmake    : ${CMAKE_CURRENT_LIST_FILE}")

# CMakeLists.txt
cmake_minimum_required(VERSION 3.8)

include(cmake/print-current-path.cmake)
message(STATUS "cmakelist: ${CMAKE_CURRENT_LIST_FILE}")
```

이런 결과가 출력될 것입니다.

```
$ cmake .
...
-- cmake    : /path/to/cmake/print-filepath.cmake
-- cmakelist: /path/to/CMakeLists.txt
...
-- Configuring done
-- Generating done
```

get_filename_component

보통 특정 경로 하나만으로는 문제를 해결할 수 없기 때문에 여기서는 경로를 다루는 방법 중 두가지를 짚고 넘어가겠습니다.

기본적으로 CMake에서 파일의 경로 생성할때는 `get_filename_component` 를 사용합니다. 앞서 `TBBConfig.cmake` 에서도 이 함수가 사용되었었는데, 코드를 보면 의도를 파악하기가 어렵습니다.

```
# ...
get_filename_component(_tbb_root "${CMAKE_CURRENT_LIST_FILE}" PATH)
get_filename_component(_tbb_root "${_tbb_root}" PATH)
get_filename_component(_tbb_root "${_tbb_root}" PATH)
# ...
```

Parent Path (Removing Base Name)

CMake 문서에 따르면 가장 마지막에 사용된 인자 `PATH` 는 2.8 버전들의 하위호환을 위한 것으로, 그 의미는 `DIRECTORY` 를 사용하는 것과 동일합니다.

```
DIRECTORY = Directory without file name
PATH      = Legacy alias for DIRECTORY (use for CMake <= 2.8.11)
```

따라서 현재 기술하고 있는 3.8 이후 버전을 기준으로 작성한다면 아래와 같을 것입니다.

```
# ...
get_filename_component(_tbb_root "${CMAKE_CURRENT_LIST_FILE}" DIRECTORY)
get_filename_component(_tbb_root "${_tbb_root}" DIRECTORY)
get_filename_component(_tbb_root "${_tbb_root}" DIRECTORY)
# ...
```

이미 설계된 TBB 빌드 결과물의 배치를 고려해서 부모 폴더를 여러번 타고 올라가는 코드라는 것을 쉽게 알 수 있습니다. 이를 `CMAKE_CURRENT_LIST_FILE` 에 적용해보면 어떻게 될까요?

```
# cmake/print-parent-path.cmake

get_filename_component(PARENT_DIR ${CMAKE_CURRENT_LIST_FILE} DIRECTORY)
message(STATUS "parent   : ${PARENT_DIR}")
```

조금 전에 `CMAKE_CURRENT_LIST_FILE` 에서 사용한 CMakeLists.txt를 아래처럼 수정해 실행하면:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.8)

include(cmake/print-current-path.cmake)
include(cmake/print-parent-path.cmake)  # <--- new !
message(STATUS "cmakelist: ${CMAKE_CURRENT_LIST_FILE}")
```

출력 결과는 아래와 같을 것입니다.

```
$ cmake .
-- cmake    : /path/to/cmake/print-current-path.cmake
-- parent   : /path/to/cmake
-- cmakelist: /path/to/CMakeLists.txt
...
-- Configuring done
-- Generating done
```

Path Join

경로를 처리할때 접합(concat)을 수행하는 코드를 흔히 볼 수 있습니다. 이런 코드들은 절대 경로(Absolute Path)와 상대 경로(Relative Path)가 고르게 사용되는 반면, CMake에서 파일 경로는 특별한 처리가 필요하지 않는 한 절대 경로를 사용합니다.

> 작성자의 생각으로는, `PROJECT_SOURCE_DIR`, `CMAKE_CURRENT_SOURCE_DIR` 등 파일경로를 만들때 가장 기초가 되는 경로가 모두 절대경로로 반환되기 때문인 것 같습니다.

이미 존재하는 **폴더** 경로에 새로운 이름을 붙이는 것은 보통의 문자열 생성 방법과 같습니다. Windows에서는 Command Prompt를 실행하는 경우라면 `\\` 를 구분자로 사용해야 하지만, 단순히 CMake 내에서 경로만 처리한다면 `/` 를 사용해도 별다른 문제가 없습니다.

```
# Ok for Windows and the others
get_filename_component(CURRENT_MODULE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/cmake ABSOL
message(STATUS "modules  : ${CURRENT_MODULE_DIR}")
```

여기서 `get_filename_component` 의 역할은 `CURRENT_MODULE_DIR` 변수의 타입을 파일경로로 설정하는 것 뿐입니다. Windows, PowerShell 환경에서 이를 실행해보면 CMake에서 구분자로 `/` 를 사용하는 것을 확인할 수 있습니다.

```
PS > cmake .
...
-- modules  : D:/path/to/cmake
...
```

WSL Bash에서는 아래와 같습니다.

```
$ cmake .
...
-- modules  : /mnt/d/cmake_tutorial/path/cmake
...
```

get_target_property

`set_target_properties` 가 여러 Property를 한번에 설정할 수 있는데 반해, `get_target_property` 는 한번에 하나의 변수를 생성합니다. 사용법 또한 굉장히 단순합니다.

```
cmake_minimum_required(VERSION 3.8)

add_library(my_modern_cpp_lib
    libmain.cpp
)
```

```
set_target_properties(my_modern_cpp_lib
PROPERTIES
    CXX_STANDARD 17
)
get_target_property(specified_cxx_version
                    my_modern_cpp_lib CXX_STANDARD
)

# -- cxx_version: 17
message(STATUS "cxx_version: ${specified_cxx_version}")
```

이제 `find_package` 는 Target을 선언하고 Property들을 설정한다는 것과 Property를 설정하고 확인하는 함수들을 다루었으므로 빌드 이후 Manifest파일을 생성하는 방법에 대해 다뤄보겠습니다.

CMake Export

사실 CMake에서 Export하는 방법은 튜토리얼마다 설명이 조금씩 다른데, 근본적인 차이점은 CMake를 위한 템플릿 파일을 사용하는지에 달려 있습니다. 어떤 프로젝트에서는 CMake 모듈들이 배치된 폴더에 `package-targets.cmake.in` 과 같에 `.in` 으로 끝나는 파일들이 있는 것을 볼 수 있는데, 이런 인라인 파일들은 어디선가 CMake에서 제공하는 `configure_file` 혹은 `configure_package_config_file` 함수를 사용하기 때문일 가능성이 높습니다.

이 함수는 CMake파일 생성 뿐만 아니라 사용자 환경에 맞는 헤더 파일(.h)을 만들거나, Linux플랫폼에서 에서 pkg-config를 위한 파일을 만드는데 사용되기도 합니다.

> 작성자는 이 기능을 사용하는 것을 추천하지 않습니다.
> 빌드 시스템 파일을 생성하는것이 CMake의 가장 중요한 부분이며, 오직 그 일에 집중해야 한다고 생각하기 때문입니다

CMake Manifest 파일의 배치

지금까지 `find_package` 에 어떤 인자를 사용하는지, 해당 함수에서 사용하는 Manifest 파일에 어떤 내용이 들어가는지는 살펴보았으나, **어디에서** 해당 파일을 찾는지는 설명하지 않았습니다.

CMake의 Manifest 탐색 과정

CMake 문서의 설명에 따르면 플랫폼마다 탐색 경로가 다르지만, 공통되는 경로가 있다는 것을 알 수 있습니다. 앞서 이 문서에서는 `install` 을 사용할 때 `CMAKE_INSTALL_PREFIX` 를 기준으로 설치경로를 지정하는 것을 권했었는데, 아마 아래처럼 경로에 프로젝트 이름이 들어가는 것이 다른 프로젝트와의 충돌의 가능성을 낮춰줄 것입니다.

```
cmake_minimum_required(VERSION 3.8)
project(my_modern_cpp_lib LANGUAGES CXX)
# ...

install(FILES           ${VERSION_FILE_PATH}
                        ${LICENSE_FILE_PATH}
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/share/${PROJECT_NAME}
)
```

CMake Manifest 만들기

write_basic_package_version_file

버전 정보를 추가하는 것은 이미 CMake에서 제공하는 모듈을 사용하면 쉽게 작성할 수 있습니다.

```
include(CMakePackageConfigHelpers)
set(VERSION_FILE_PATH   ${CMAKE_BINARY_DIR}/cmake/${PROJECT_NAME}-config-version.
write_basic_package_version_file(${VERSION_FILE_PATH}
    VERSION             ${PROJECT_VERSION} # x.y.z
    COMPATIBILITY       SameMajorVersion
)

# ...

install(FILES           ${VERSION_FILE_PATH}
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/share/${PROJECT_NAME}
)
```

install(EXPORT)

파일 혹은 폴더의 설치는 단순히 복사/갱신으로 끝날 수 있지만, 결정적으로 `-config.cmake` 에는 프로젝트에서 빌드할 Target에 대한 정보가 들어가야 합니다. 여기에는 `install(TARGETS)` 와 `install(EXPORT)` 가 함께 사용됩니다.

간단한 예시로, 아래와 같은 구조의 프로젝트를 만들어보겠습니다.

```
$ tree $(pwd)
/mnt/d/example
├── CMakeLists.txt
└── src
    ├── CMakeLists.txt
    └── libmain.cpp

1 directory, 3 files
```

우선 Root CMakeLists.txt에서는 `EXPORT_NAME` 변수를 만들고 `add_subdirectory` 로 하위 모듈들을 빌드하도록 합니다. 최종적으로는 `install(EXPORT)` 를 사용해 설치까지 수행합니다.

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.8)
project(stones LANGUAGES CXX)

set(EXPORT_NAME ${PROJECT_NAME}-config) # or ${PROJECT_NAME}Config

add_subdirectory(src) # <--- uses EXPORT_NAME

install(EXPORT      ${EXPORT_NAME}
        NAMESPACE   stones::
        DESTINATION ${CMAKE_INSTALL_PREFIX}/share/${PROJECT_NAME}
)
```

src의 CMakeLists.txt는 `add_library` 로 CMake Target을 생성하고, `install(TARGETS)` 에서 `EXPORT` 인자를 사용해 해당 라이브러리를 일종의 Export Group에 추가합니다. 단순히 추가하기만 할 뿐, `install(EXPORT)` 를 사용하기 전까지 실제 설치는 이루어지지 않습니다.
특이하게도 `EXPORT` 는 반드시 다른 인자보다 먼저 사용되어야 한다고 명시하고 있습니다(must appear before).

```
# src/CMakeLists.txt

add_library(stone1 SHARED
    libmain.cpp
)
set_target_properties(stone1
PROPERTIES
    CXX_STANDARD 17
)
target_include_directories(stone1
PUBLIC
    $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include>
)

install(TARGETS         stone1
        EXPORT          ${EXPORT_NAME}  # <---- new!
    RUNTIME DESTINATION ${CMAKE_INSTALL_PREFIX}/bin
    LIBRARY DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
    ARCHIVE DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
)
```

마지막으로 libmain.cpp는 간단히 함수 하나를 동적 링킹(Dynamic Linking)이 가능하도록 정의합니다.

```
#pragma once
// clang-format off
#if defined(_MSC_VER) // MSVC or clang-cl
#   define _HIDDEN_
#   ifdef _WINDLL
#       define _INTERFACE_ __declspec(dllexport)
#   else
#       define _INTERFACE_ __declspec(dllimport)
#   endif
#elif defined(__GNUC__) || defined(__clang__)
#   define _INTERFACE_ __attribute__((visibility("default")))
#   define _HIDDEN_ __attribute__((visibility("hidden")))
#else
#   error "unexpected linking configuration"
#endif
// clang-format on

#include <cstdint>

constexpr auto version_code = 0x0102;

_INTERFACE_ uint32_t get_version() noexcept;

uint32_t get_version() noexcept{
    return version_code;
}
```

Export 결과 확인

Windows에서 설치를 수행하면 아래와 같이 `stones-config.cmake` 파일이 설치되는 것을 볼 수 있습니다.

```
PS D:\examples\build> cmake --build . --config debug --target install
PS D:\install> Tree /f .
Folder PATH listing for volume keep
Volume serial number is B47E-DE87
D:\INSTALL
├─bin
│      stone1.dll
│
├─lib
│      stone1.lib
│
└─share
    └─stones
            stones-config-debug.cmake
            stones-config.cmake
```

여기서 설치된 파일의 이름인 `stones-config` 는 앞서 `EXPORT_NAME` 변수의 값을 따른 것입니다.

```
set(EXPORT_NAME ${PROJECT_NAME}-config) # or ${PROJECT_NAME}Config
```

불필요한 부분을 제외하고 해당 파일의 내용을 살펴보면 `stones::stone1` 와 같이 Target을 가져오는 내용이라는 것을 알 수 있습니다. 이런 파일들은 `stones-targets.cmake` 로 따로 만들고 `-config.cmake` 는 `configure_package_config_file` 을 사용해서 만드는 방법을 사용하기도 합니다. 하지만 이 예시에서는 Import측에 전달할 정보가 없기에 CMake 템플릿 파일을 작성하지 않았고, 따라서 바로 `-config.cmake` 를 생성해도 무방합니다.

```
# stones-config.cmake
# ...

# The installation prefix configured by this project.
set(_IMPORT_PREFIX "D:/install")

# Create imported target stones::stone1
add_library(stones::stone1 SHARED IMPORTED)

set_target_properties(stones::stone1 PROPERTIES
  INTERFACE_INCLUDE_DIRECTORIES "D:/install/include"
)

# Load information for each installed configuration.
get_filename_component(_DIR "${CMAKE_CURRENT_LIST_FILE}" PATH)
file(GLOB CONFIG_FILES "${_DIR}/stones-config-*.cmake")
foreach(f ${CONFIG_FILES})
  include(${f})
endforeach()

# ...
```

특히 패턴매칭(`stones-config-*.cmake`)을 사용해 `-config-debug.cmake` 혹은 `-config-release.cmake` 를 `include` 할 수 있도록 되어있는 점에 주목하시길 바랍니다. 앞서 `write_basic_package_version_file` 에서 Version 파일의 설치 위치를 비롯해 이름을 `${PROJECT_NAME}-config-version.cmake` 로 만들도록 한 것은 이를 고려한 것입니다.

```
set(EXPORT_NAME ${PROJECT_NAME}-config)

add_subdirectory(src) # <--- uses EXPORT_NAME

install(EXPORT          ${EXPORT_NAME}
        NAMESPACE       stones::
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/share/${PROJECT_NAME}
)

include(CMakePackageConfigHelpers)
set(VERSION_FILE_PATH   ${CMAKE_BINARY_DIR}/cmake/${PROJECT_NAME}-config-version.
write_basic_package_version_file(${VERSION_FILE_PATH}
    VERSION             ${PROJECT_VERSION} # x.y.z
    COMPATIBILITY       SameMajorVersion
```

```
)
install(FILES           ${VERSION_FILE_PATH}
        DESTINATION     ${CMAKE_INSTALL_PREFIX}/share/${PROJECT_NAME}
)
```

`target_include_directories` : Build >> Install

좀 전의 예시에서 처음으로 보인 `BUILD_INTERFACE` 와 `INSTALL_INTERFACE` 의 사용을 한마디로 정리하자면,
"빌드할 때 사용하는 include 폴더와 설치 후 사용하는 include 폴더가 다르다" 라는 것입니다.

```
target_include_directories(stone1
PUBLIC
    $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include>
)
```

위와 같이 작성하는 것을 CMake에서는 Generator Expression이라 하는데, 보통 플랫폼에 따라 IF/ELSE/AND/OR이 뒤섞여 가독성을 심하게 해치는 경향이 있습니다.

라이브러리가 설치된 이후에는 Build에 사용한 폴더가 삭제될 가능성이 높기에, Import 할 때 소스코드가 배치된 폴더를 사용하도록 한다면 파일을 못찾는 문제가 발생할 것입니다. 이를 막기 위해 빌드시에는 `PROJECT_SOURCE_DIR` 기준으로 include를 수행하지만, 설치 이후에는 `CMAKE_INSTALL_PREFIX` 를 기준으로 include를 수행합니다.

아마 인터페이스 파일들은 이미 `CMAKE_INSTALL_PREFIX/include` 로 `install(FILES)` 혹은 `install(DIRECTORIES)` 를 통해서 복사되었을 것이기에 설치가 완료된 시점부터 해당 폴더는 사용가능한 경로가 될 것입니다.

`tutorial-pend.md`

맺음말

당초 저는 간단히 Part 1만으로 끝을 내고 더 갱신할 생각이 없었습니다만, 많은 분들의 격려로 마지막에 `find_package` 를 지원할 수 있도록 Export하는 부분까지 작성할 수 있었습니다.

CMake가 하는 일은 이미 Ruslo가 CGold에서 설명한 것처럼, 각 플랫폼마다 사용되고 있는 빌드 시스템 파일을 생성하는 것이 핵심 기능입니다. 다만 CMake에서 지원하는 기능이 늘어남에 따라 Protobuf Compiler와 같은 소스코드를 생성하는 프로그램을 사용하거나, `configure_file` 과 같은 함수로 CMake를 통해야만 빌드/설치를 마칠 수 있도록 맞춰진 프로젝트도 발견할 수 있습니다. 테스트 코드를 위해 gtest를 다운로드하거나 어셈블리 코드를 미리 빌드 하기도 합니다. 빌드 과정이 보다 넓어졌다고 할 수 있을 것 같습니다.

생각해보면 특히 CMake를 익혀나가면서 느낀 어려움의 근본적인 원인은 "어느 누구도 확신을 가지고 CMake를 가르쳐주지 않더라"라는 것이었습니다. 보통 이런 문제는 공식 문서와 검색을 통해 해결할 수 있는 속성의 것들이지만, CMake는 2.x 버전을 대상으로 작성된 수많은 블로그 포스트들과, CMake 3.x 초반에서도 그들로부터 영향받은 설명들로 인해 마음놓고 Ctrl+C,V 하는 것마저 쉽지 않았습니다. 다행히 많은 개발자들이 자신만의 예제 저장소를 공개하고 커뮤니티에 글을 기고해준 덕분에 어떻게든 배워나갈 수 있었지만, 빌드를 자동화 해놓은 상황에서도 "CMake 어렵다"라는 말을 자주 하고, 들을 수 있었습니다.

어쩌면 이 모든 것은 "CMake 책"을 읽지 않았기 때문인지도 모릅니다. 그정도 노력까지는 아니더라도 제가 CGold를 만났을 때처럼, 이 문서를 읽고 따라함으로써 모쪼록 시간을 절약하는 분들이 많아지기를 바랄 따름입니다.

꽤나 오랫동안 방치해놓았는데 이 문서를 인용해주신 분들께 감사드립니다.

naddu77 commented on 4 Dec 2018

다르게는 https://github.com/Microsoft/vcpkg 와 같이 라이브러리 탐색에 특화된 툴체일 파일을 사용하는 경우도 있습니다.

툴체일 오타났어염~~

sgju-lano commented on 25 Dec 2018

도움 정말 많이되고있습니다 cmake 짤 때 필수적으로 켜놓고 작업하고있어요. 감사합니다!

oranke commented on 8 Feb 2019

좋은 글 감사합니다. ^^

Cloud73 commented on 8 Mar 2019

요긴하게 봤습니다.

그렇지않아도, 시스템에 포팅해야하는 놈이 CMake로 되있어서 골이 좀 아팠는데 이해가 가네요.

hgkmail commented on 8 Apr 2019

is there an english version?

luncliff commented on 10 Apr 2019 • edited

@hgkmail I wish these links are enough for you.

- https://cliutils.gitlab.io/modern-cmake/
- https://github.com/pr0g/cmake-examples

hyeonchang commented on 15 May 2019

많은 도움이 되었습니다!

kisuya commented on 23 Jul 2019

개인 플젝때문에 CMake 복기 중인데 여기까지 흘러올줄이야 :)

dongbum commented on 26 Sep 2019

좋은 자료 감사합니다.

KUflower commented on 16 Jan

좋은 자료 감사합니다!😃

dbwodlf3 commented on 5 Mar

you are good.
good person.
ㅇㅇㅇㅇㅇㅇ응아ㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴㄴ

choiwonseok-david commented on 12 May

좋은 자료 감사합니다

Cyp9715 commented on 18 Jun • edited

단언컨데 한국어 Cmake 자료중 최고