

# 현실적인 (Practical) CMake

---

박 동 하

C++ Korea Facebook Group, **LINE+**

<https://github.com/luncliff>

# 참고자료

---

## 영상(YouTube)

- C++Now 2017: **“Effective CMake”** – Daniel Pfefier
- CppCon 2017: **“Using Modern CMake Patterns to Enforce a Good Modular Design”** – Mathieu Ropert
- JFrog Webinar: **“Introduction to C/C++ Package Management with Conan”** – Diego Rodriguez-Losada
- CppCon 2018: **“Git, CMake, Conan – How to ship and reuse our C++ projects?”** – Mateusz Pusz
- C++ Weekly: Ep 118 **“Trying Out The vcpkg Package Manager”** – Jason Turner
- CppCon 2019: **“Deep CMake for Library Authors”** – Craig Scott

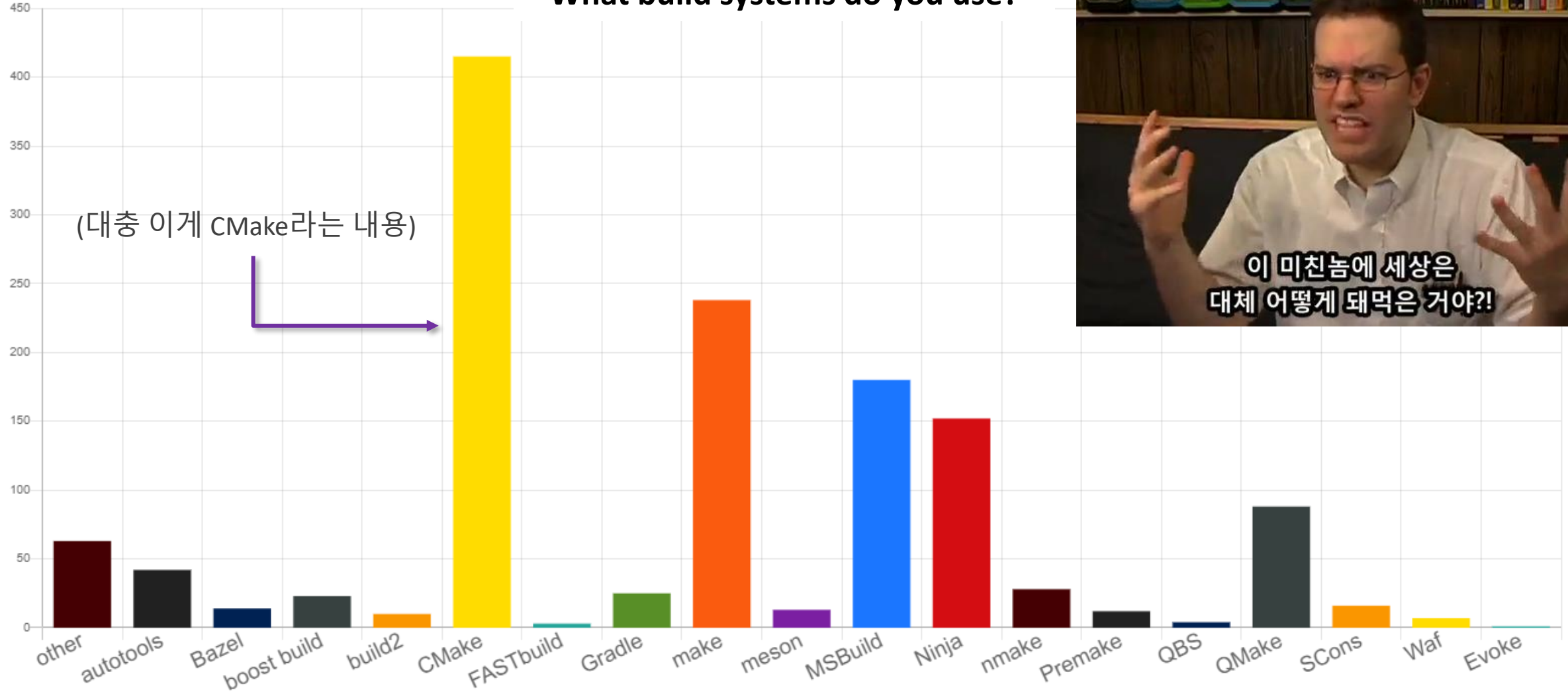
## Web

- An Introduction to Modern CMake <https://gitlab.com/CLIUtils/modern-cmake>
- CGold: The Hitchhiker’s Guide to the CMake <https://cgold.readthedocs.io/en/latest/>

위의 자료들을 기반으로 작성된 것이 “CMake할때 쪼오오금 도움이 되는 문서”(2018.11)  
<https://gist.github.com/luncliff/6e2d4eb7ca29a0afd5b592f72b80cb5c>

## Meeting C++ Community Survey

### What build systems do you use?





---

## Cross-Platform Make

- 빌드 과정을 플랫폼/툴체인(컴파일러)에 종속적이지 않은 방법으로 작성

## CMake is a Buildsystem Generator

- Native Buildsystem과 함께 동작하고, 이들이 사용하는 파일을 생성(Generate)해주는 역할
- pcmaker, configure등의 영향을 받음

생성 이후의 단계를 간단히 실행할 수 있도록 지원하기도...

- Test (CTest)
- Package (CPack)

# Buildsystem이 뭔가요? 먹는건가요?

---

빌드를 정의하고, 수행하기 위한 도구들과 전용 파일들

- Source Code, Resource
- Program Type
  - Static? DLL? Exe?
- Precompiled Header
- Cross Compile Support
- Language
  - 컴파일러가 지원하는 기능들의 On/Off (SIMD, Lanugage Extension ...)
- Testing
  - Pass/Fail Report (XML, HTML...)
  - Coverage, Sanitizer Report
- Code Analysis
  - C++ Core Guidelines, Microsoft Rules

# CMake CLI 사용법

---

# Configure + Unix Makefiles 사용법과 매우 비슷함

```
$ mkdir build && pushd build
$ cmake .. -DCMAKE_INSTALL_PREFIX="/usr/local"
...
-- Configuring done
-- Generating done
-- Build files have been written to: ...
```

```
$ cmake --build .
```

```
$ cmake --build . --target install
...
-- Install configuration: "Debug"
-- Installing: /usr/local/include/ssf.h
-- Installing: /usr/local/lib/libssf.so
```

빌드 위치로 이동 + 파일 생성

- `CMAKE_`, `BUILD_` 와 같이 이후 파일 생성할에 영향을 미치는 변수들이 있음

생성한 파일로 빌드를 수행

- CMake는 빌드를 수행하는 능력이 없음
- 하위 프로세스만 실행할 뿐

빌드 이후 프로그램 설치

# CMAKE\_ 변수들

```
$ cmake .. \  
> -DCMAKE_MODULE_PATH="../../../cmake;../scripts" \  
> -DCMAKE_INSTALL_PREFIX="/usr/local" \  
> -DCMAKE_BUILD_TYPE="Debug" \  
> ...
```

```
$ cmake .. \  
> -DCMAKE_CXX_COMPILER="/tmp/bin/clang++-10" \  
> -DCMAKE_CXX_FLAGS="-O0 --coverage"
```

```
$ cmake .. \  
> -DCMAKE_TOOLCHAIN_FILE="../../../toolchain.cmake"
```

빈번하게 사용되는 변수는 많지 않음

- 널리 사용되는 toolchain.cmake 들에서 대부분 기본값을 설정
- 전역에 영향을 미치기 때문에, 거의 작성하지 않거나, 매우 상세하게 작성하는 경우가 대부분

특정 컴파일러 사용 or 컴파일러 옵션이 특수한 경우

- 많은 경우 비권장
- 참고자료들에서는 그냥 쓰지 마세요 하는 기능

toolchain.cmake

- 최상위 CMakeLists.txt를 처리하기 전에 우선적으로 적용
- 보통 Cross Compile을 위한 빌드 설정들을 작성
- 이를 응용한 경우(Vcpkg)도 있음

[https://cmake.org/cmake/help/latest/search.html?q=CMAKE\\_](https://cmake.org/cmake/help/latest/search.html?q=CMAKE_)



# BUILD\_ 변수들

```
$ cmake .. \  
> -DBUILD_SHARED_LIBS=true
```

```
$ cmake .. \  
> -DBUILD_TESTING=ON \  
> -DBUILD_SAMPLES=OFF \  
> -DBUILD_WITH_CURL=OFF \  
> ...
```

공식문서에 나오는 변수는 하나 뿐

- 프로그램 타입이 명시되지 않은 경우, 생성할때 static/share를 결정

If/Else 처리가 가능하도록 True/False, ON/OFF를 사용

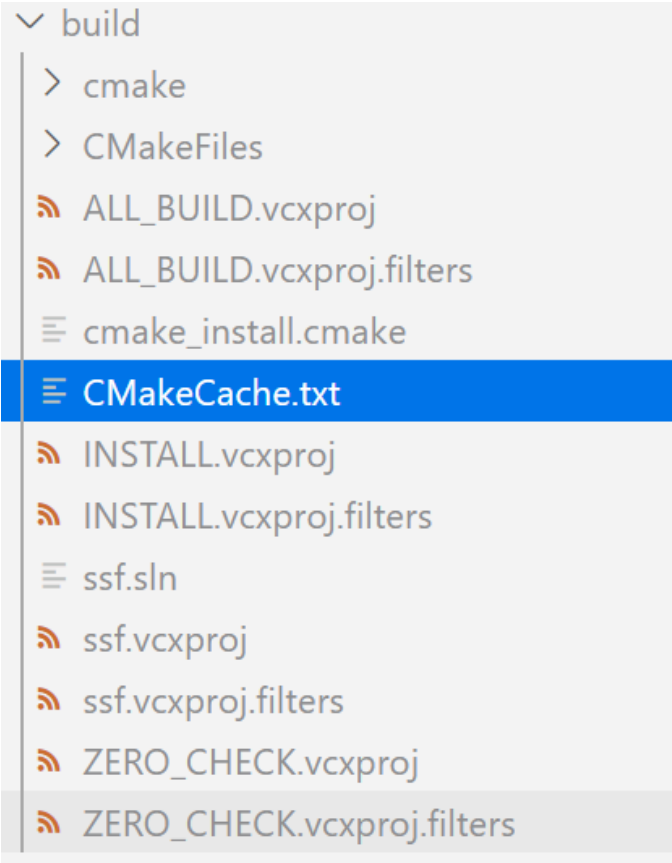
프로젝트의 옵션을 이와 유사하게 명명하는 경우를 볼 수 있음

- 사용하려는 프로젝트의 CMakeLists.txt 에서 BUILD\_를 검색
- 충돌 방지를 원한다면 **HELLO\_BUILD\_SAMPLES**처럼 **Prefix**를 붙이기도

OpenCV의 작명법이 참고할만함

- <https://github.com/opencv/opencv/blob/master/CMakeLists.txt>
- BUILD\_XXX: ...
- WITH\_XXX: ...
- ENABLE\_XXX: ...
- INSTALL\_XXX: ...

# CMakeCache.txt



CMake는 처음 configure 단계에서 지정한 값들을 Cache 한다

- 스크립트 내에서 값을 확인할 때 Cache된 값이 사용되는 경우가 있음
- 2.8 버전 CMake 들은 대부분 변수를 중심으로 작성되었기 때문에 이런 현상이 빈번

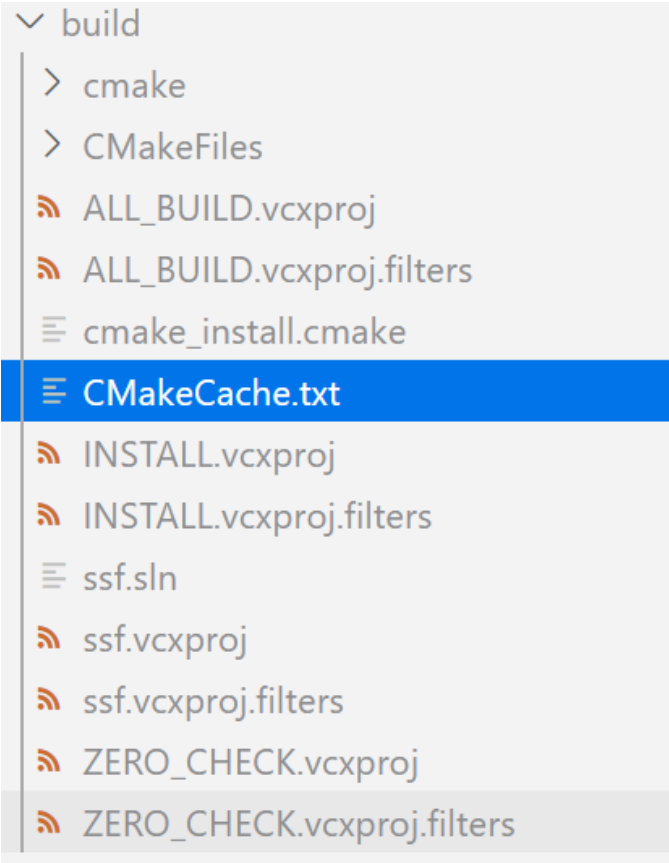
CLI에서 Cache/Uncache(-C, -U)옵션으로 추가,삭제 가능

- 그런데 쓰는 것을 한번도 못봄...

올바른 설정을 입력했지만, 오류가 발생한다?

- 많은 경우 Cache에서 값을 읽어서 처리하면서 발생
- build 폴더를 지우고 다시 생성하는 것이 오히려 편리...

# CMakeCache.txt



```
49 //Flags used by the linker during RELEASE builds.
50 CMAKE_EXE_LINKER_FLAGS_RELEASE:STRING=/INCREMENTAL:NO
51
52 //Flags used by the linker during RELWITHDEBINFO builds.
53 CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO:STRING=/debug /INCREMENTAL
54
55 //Install path prefix, prepended onto install directories.
56 CMAKE_INSTALL_PREFIX:PATH=C:/Users/luncl/source/ssf/install
57
58
59 CMAKE_LINKER:FILEPATH=C:/Program Files (x86)/Microsoft Visual
60
61 //Flags used by the linker during the creation of modules during
62 // all build types.
63 CMAKE_MODULE_LINKER_FLAGS:STRING=/machine:x64
64
65 //Flags used by the linker during the creation of modules during
```

이름

타입

값

# CMake 3.x 작성방법

---

CMakeLists.txt 라는 파일이름을 많이 보긴 했는데...

# CMake 3.x에서 알아야 할 용어들

---

## Target

- CMake에서 빌드를 정의하는 기본 단위

## Property

- 보통 Target에 사용되지만, 개별 File들도 지닐 수 있음

## Variable, Statement, Command, Function, Macro

- CMake를 작성하는 것은 Shell Script를 작성하는 것과 동일

## Module

- 미리 작성된 CMake 스크립트(.cmake)

## Generator Expression

- Configure이후 Generation 시점에 값이 결정되는 '표현식'

# CMake 3.x에서는 Target 단위로 빌드의 절차를 설계

```
add_library(ssf
    # ... source code ...
)

add_executable(ssf_test
    # ... test source codes ...
)
```

```
add_dependencies(ssf_test
    ssf # ... multiple targets ...
)
```

Target 을 선언하는 Command

- `add_library`
- `add_executable`
- `add_custom_target`

**CMAKE\_ 변수들을 사용해 기본값들이 설정됨(암묵적)**

- `CMAKE_CXX_STANDARD`
- `CMAKE_CXX_FLAGS`
- ...

Target 사이의 의존성

- 빌드 명령 실행시 순서를 결정

# Target들은 속성(Property)을 지닌다

## Binary Target

- Native Buildsystem 파일을 생성할 때 세부사항을 설정
- 만약 -G "Xcode"를 사용한다면?  
WINDOWS\_EXPORT\_ALL\_SYMBOLS는 무시된다.
- 어떤 속성들은 CMAKE\_변수를 확인하여 기본값을 설정

```
set_target_properties(ssf
PROPERTIES
    WINDOWS_EXPORT_ALL_SYMBOLS true
    MACOSX_BUNDLE true
    # ...
)
```

```
set(CMAKE_CXX_STANDARD 14)

add_executable(ssf_test
    # ...
)

# set_target_properties(ssf
# PROPERTIES
#     CXX_STANDARD 14
# )
```

# Target들은 속성(Property)을 지닌다

## Imported Target

- 이미 외부에서 빌드가 끝난 라이브러리
- 다른 Target에서 빌드를 수행할 때 필요한 내용
- 언어, 필요한 라이브러리 목록, 파일 경로 등

```
add_library(ssf SHARED IMPORTED)
set_target_properties(ssf
PROPERTIES
    INTERFACE_INCLUDE_DIRECTORIES "C:/installed/include"
    INTERFACE_LINK_LIBRARIES "ws2_32;mswsock"
    IMPORTED_LINK_INTERFACE_LANGUAGES "CXX"
    IMPORTED_IMPLIB "C:/installed/lib/ssf.lib"
    IMPORTED_LOCATION "C:/installed/bin/ssf.dll"
)
```



# Command를 사용해 빌드를 정의: Binary Target

프로그램을 만들기 위해 필요한 내용을 기술

- Program Type
- Source Files

관련 명령들은 Public/Private로 제어 가능

전처리기

- **Macro Definitions**
- **Header Search Path**

컴파일러

- **Compiler Options**

링커

- **Dependent Libraries**
- Library Search Path

```
add_library(ssf SHARED
    src/libmain.cpp
    # ... shared source codes ...
)
```

```
if(WIN32)
    target_sources(ssf
        PRIVATE
        src/main_windows.cpp
    )
elseif(CMAKE_SYSTEM_NAME MATCHES Linux)
    target_sources(ssf
        PRIVATE
        src/main_linux.cpp
    )
endif()
```

# Binary Target: 전처리기

## Macro Definitions:

- `target_compile_definitions`
- 문자열, 상수 등을 지정 가능
- Undef는 미 지원

## Header Search Path:

- `target_include_directories`
- 현재 CMakeLists.txt를 기준으로 상대경로 사용 가능
- PUBLIC에는 Generator Expression을 사용

```
if(WIN32)
    target_compile_definitions(ssf
        PRIVATE
            WIN32_LEAN_AND_MEAN
            VERSION_WSTRING=L"${PROJECT_VERSION}"
    )
endif()
```

```
target_include_directories(ssf
    PRIVATE
        externals/include
        src
    PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
        $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include>
)
```

# Binary Target: 컴파일러

## Compiler Options

- `target_compile_options`
- 컴파일러에 맞게 옵션을 명시하는 방법

```
if(MSVC)
    target_compile_options(ssf
        PRIVATE
            /EHc /JMC-
    )
endif()
```

## Compiler Feature

- `target_compile_features`
- 필요한 표준 C++ 언어 기능들을 기술하는 용도  
(11~14, 14~17 과도기 컴파일러들)

```
target_compile_features(ssf
    PUBLIC
        cxx_std_14
    PRIVATE
        cxx_unicode_literals
)
```

[https://cmake.org/cmake/help/latest/prop\\_gbl/CMAKE\\_CXX\\_KNOWN\\_FEATURES.html](https://cmake.org/cmake/help/latest/prop_gbl/CMAKE_CXX_KNOWN_FEATURES.html)

# Binary Target: 링커

## Dependent Libraries

- `target_link_libraries`
- 링킹 과정에서 필요한 라이브러리들을 열거
- 다른 빌드가 선행될 수 있도록 `add_dependencies` 처리
- 다른 Target의 `PUBLIC` 을 전달받음  
(CMake 2.x와 달리 중복 작성이 불필요)

## Library Search Path

- `target_link_directories (3.13+)`
- 링킹 과정에서 라이브러리들을 탐색할 경로를 명시

```
find_package(Threads REQUIRED)
target_link_libraries(ssf
PUBLIC
    ws2_32 mswsock
PRIVATE
    Threads::Threads
)
```

```
target_link_directories(ssf
PUBLIC
    ${CMAKE_INSTALL_PREFIX}/lib
PRIVATE
    external/lib
)
```

# Command를 사용해 빌드를 정의: Command Target

여러 CLI 명령을 묶어서 수행하기 위한 기능

```
add_custom_target(download_libpng
  COMMENT "download & extract source code"
  WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/externals
  COMMAND wget -q "https://github.com/glennrp/libpng/archive/1.6.37.zip"
  COMMAND unzip 1.6.37.zip
)
```

빌드 명령으로 실행하지만 빌드는 하지 않는다...!

```
$ cmake --build . --target download_libpng
...
```

# CMake 작성: set, if

```
if(NOT DEFINED CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release)
endif()

message(STATUS "type: ${CMAKE_BUILD_TYPE}")
unset(CMAKE_BUILD_TYPE)
```

```
$ cmake ..
...
-- build type: Release
...
```

```
$ cmake .. -DCMAKE_BUILD_TYPE=Debug
...
-- build type: Debug
...
```

## Variable

- `set`으로 설정 후, `if/elseif`에서 사용 가능
- `unset`으로 제거
- `${name}` 표현식으로 값을 참조

## stdout 출력

- `message` 명령을 사용

## CMAKE\_ 변수를 사용했으므로 ...

- 실제 생성된 `vcxproj`, `makefile` 파일들이 영향을 받는다
- == 단순히 출력만 달라지는 것이 아니다!

# CMake 작성: list, foreach

```
list(APPEND files
      a.hpp a.cpp
      b.hpp b.cpp
)
message(STATUS "${files}")
```

```
foreach(fname IN LISTS files)
    message(STATUS "${fname}")
endforeach()
```

```
$ cmake ..
...
-- a.hpp;a.cpp;b.hpp;b.cpp
-- a.hpp
-- a.cpp
-- b.hpp
-- b.cpp
...
```

## List 명령

- 지정한 변수 이름의 값을 변경
- `${name}` 표현식으로 값을 참조
- 여러 기능이 있지만 자주 쓰이는 것은 APPEND, FIND 정도

## foreach 명령

- List 내용을 깔끔하게 출력할 때 외에는 쓸 일이...

## 문자열 변수의 값에서;를 구분자로 사용

- CppCon 2019: Jussi Pakkanen “Let's cmakeify the C++ standard library”
- <https://www.youtube.com/watch?v=YxortD9IxSc>

# CMake 모듈(Module)

---

`include(...)` 명령을 통해 실행(load)되는 미리 작성된 스크립트

- 일반적으로는 CMake와 함께 설치되는 .cmake 파일들을 통칭
- `CMAKE_MODULE_PATH` 를 순회하면서 검색

CMakeLists.txt 의 내용이 너무 많아지지 않도록 도와주는 역할

- 단순한 내용과 사용법을 지향
- 직접 작성하기 보다는 다른 공개 저장소에서 찾아서 사용하는 것을 권장



# CheckIncludeFileCXX

```
include(CheckIncludeFileCXX)

check_include_file_cxx("filesystem" has_filesystem /std:c++17)
if(has_filesystem)
    message(STATUS "support C++ 17 <filesystem>")
endif()

check_include_file_cxx("wrl/client.h" has_WRL)
if(has_WRL)
    message(STATUS "support WRL Client")
endif()
```

```
$ cmake ..
...
-- Looking for C++ include filesystem
-- Looking for C++ include filesystem - found
-- support C++ 17 <filesystem>
-- Looking for C++ include wrl/client.h
-- Looking for C++ include wrl/client.h - not found
...
```

check\_include\_file\_cxx

- Header 탐색을 수행
- 컴파일러 옵션들을 적용 가능

# CheckFunctionExists

```
include(CheckFunctionExists)

check_function_exists(CoInitializeEx    has_com)

if(has_com)
    # ... combaseapi.h ...
endif()
```

```
$ cmake ..
...
-- Looking for CoInitializeEx
-- Looking for CoInitializeEx - found
...
```

check\_function\_exists

- 함수의 탐색가능 여부에 따라 True/False 값을 설정
- 주로 System API 함수 확인에 사용

CMake에서는 CheckSymbolExists를 권장

- 이 모듈은 간편한 사용법 때문에 자주 볼 수 있음

# CheckCXXCompilerFlag

```
include(CheckCXXCompilerFlag)

check_cxx_compiler_flag("-std=c++2a" support_latest)

if(support_latest)
    message(STATUS "Supports C++ latest!")
endif()
```

```
$ cmake ..
...
-- Performing Test support_latest
-- Performing Test support_latest - Success
-- Supports C++ latest!
...
```

check\_cxx\_compiler\_flag

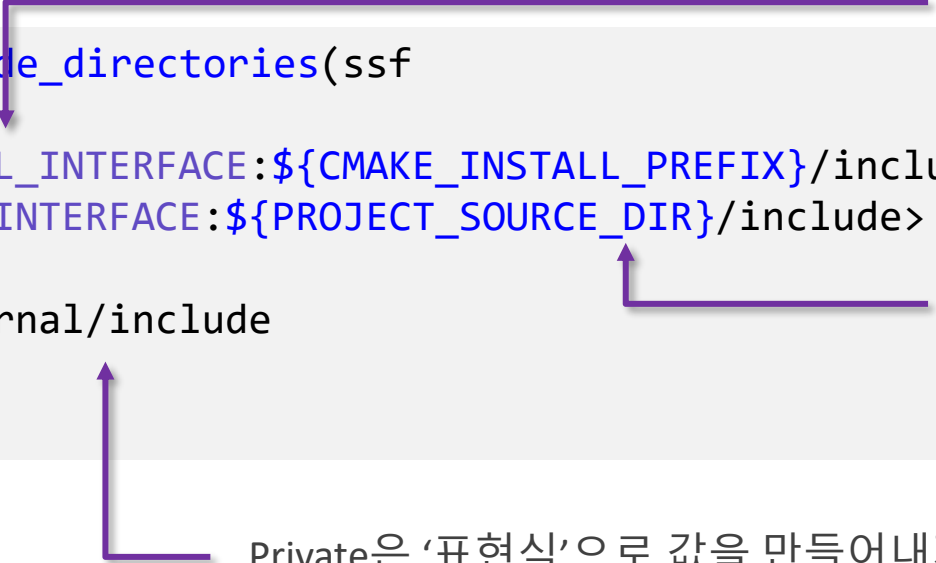
- 지원하는지 여부에 따라 True/False값을 설정

특정 기능 지원 여부를 알기 위해 사용하는 경우

- Coverage
- Sanitizer 활성화 등...

# Generator Expression

```
target_include_directories(ssf
PUBLIC
    $<INSTALL_INTERFACE:${CMAKE_INSTALL_PREFIX}/include>
    $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/include>
PRIVATE
    src external/include
)
```



설치 위치는 고정되어 있음

- 이후에는 소스폴더를 기준으로 접근할 필요가 없다
- 따라서 설치 위치(Destination)를 절대 경로 생성하여 사용

- 현재 빌드 중이라면 소스 폴더를 사용해도 OK

Private은 '표현식'으로 값을 만들어내지 않아도 OK

단점: IF/ELSE 사용시 심각한 가독성 저하

- 할 수 있다면 Config 단계에서 if/else/endif를 사용하고, 표현식은 제한하는 것을 권장
- 사용할때는 주석을 함께

<https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html>

# 왜 Generator Expression이 필요할까?

---

Configuration과 Generation이 다르기 때문

- Configuration의 정합을 확인하고, 각 Configuration에 맞게 생성(Generation)을 진행

Configuration은 Matrix 형태로 구성된다

- Generation할 때 서로 다른 값을 사용할 필요가 있음
- Toolchain
  - 버전, 환경
- Target Platform/Arch
  - Win32, x64, ARM ...
- Debug, Release, RelWithDebInfo ...
  - Macro, Library Set에 영향을 주기도 함

# CMake 3.x 작성방법

---

실제로 작성하면 어떤 모습이 되는가?

# CMake is a buildsystem generator

---

실제 빌드는 Native Buildsystem을 사용한다

- == 개발자는 해당 Buildsystem 을 고려해서 작성해야한다

필요한 정보들?

- 어느 플랫폼에서 실행할 프로그램인가? // 대상 플랫폼
- 프로그램의 형태, 개수? // 빌드 결과물
- 외부 라이브러리를 사용하는가? // 의존성
- 어느 툴체인으로 빌드를 진행할 것인가? // 빌드 환경
- 빌드 이후의 단계가 있는가? // 테스트/설치(혹은 패키징)

# 최소한의 CMake

```
cmake_minimum_required(VERSION 3.10)
```

```
project(ssf VERSION 1.1.4 LANGUAGES CXX)
```

```
add_library(ssf
    src/libmain.cpp
    src/socket.cpp
)

set_target_properties(ssf
    PROPERTIES
        VERSION      ${PROJECT_VERSION}
        SOVERSION    ${PROJECT_VERSION_MAJOR}.0
)
```

필요한 명령<sup>Command</sup>, 속성<sup>Property</sup>에 따라서 버전 선택

- 3.8: CXX\_STANDARD 17 지원
- 3.13: `target_link_directories`
- 3.16: `target_precompile_headers`

`project`: 이후 수행할 빌드작업들을 그룹화

- `PROJECT_` 변수 초기화
- `PARENT_SCOPE`에는 영향을 미치지 않음

Target 선언과 Property 설정

- 공통으로 사용되는 속성을 앞쪽에 배치



# 최소한의 CMake

```
target_include_directories(ssf
PUBLIC
    # $<INSTALL_INTERFACE:...>
    # $<BUILD_INTERFACE:... >
PRIVATE
    src
)
```

```
if(CMAKE_CXX_COMPILER_ID MATCHES Clang)
    if(WIN32)
        target_compile_options(ssf ...)
    else()
        target_compile_options(ssf ...)
    endif()
elseif(CMAKE_CXX_COMPILER_ID MATCHES GNU)
    target_compile_options(ssf ...)
elseif(MSVC)
    target_compile_options(ssf ...)
endif()
```

## 전처리기 설정

- 현재 Target의 Header 를 사용할 수 있도록 작성 (최대 2개)
- 다른 Target의 경로들은 `target_link_libraries`로 전달받음
- `target_compile_definitions`는 소스 코드에서는 보이지 않으므로, 가능한 사용을 최소화

## 컴파일러 대응

- `CMAKE_CXX_COMPILER_ID`로 컴파일러 종류를 확인
- Clang, AppleClang, Windows Clang-cl
- GCC
- Windows MSVC
- ... 이외에도 상용 컴파일러들은 고유 ID를 가지고 있음

# 최소한의 CMake

```
if(WIN32)
    target_link_libraries(ssf
        PUBLIC
        kernel32
    )
elseif(CMAKE_SYSTEM_NAME MATCHES Linux)
    target_link_libraries(ssf
        PUBLIC
        stdc++ m
    )
elseif(APPLE OR UNIX)
    target_link_libraries(ssf
        PUBLIC
        c++
    )
endif()
```

## Target Platform 대응

- WIN32, APPLE, UNIX 등을 검사하는 것으로 간결하게 작성 가능
- CMAKE\_SYSTEM\_NAME 변수를 검사하는 것도 가능함
- 사용중인 toolchain.cmake가 있다면, 의도한다면 CMAKE\_SYSTEM\_NAME 변수 이외에도 추가로 검사 가능한 변수가 있을 수 있음
- android.toolchain.cmake는 set(ANDROID true)

# 복잡도를 높이지 않는 방법?

---

CMake는 설정에 맞게 생성하는 일만 수행

- 핵심은 빌드 설정
- 빌드환경, 대상, 설정들(**if/else** 조합)이 너무 많으면 생성된 파일을 분석하기 어렵다

toolchain.cmake 을 사용한다면?

- CMake 파일들에 제약이 발생하는 방법
- 특정 변수에 대한 충돌방지 // **toolchain.cmake의 내용을 확인해봐야...**
- Generator가 제한되는 경우도 발생 // **ex. UNIX MakeFiles만 사용가능**

가이드라인

- **project**에서 진행하는 빌드의 총량을 축소시키는 것
  - 새로 빌드하기 보다는 미리 빌드된 파일들을 사용한다 // **Imported Target을 위한 설정이 압도적으로 적기 때문**
- 빌드 설정이 늘어나지 않도록 주기적으로 정리(Refinement)
  - 소스 코드는 빌드설정(경로)에 영향을 적게 주도록 작성하고, 조직화한다
  - 빌드 설정의 개수와 소스 파일의 규모를 함께 고려한다
  - 중복적으로 나타난다면 Target들을 통합하는 것도 고려할 수 있음

# 외부 의존성 해결하기

---

Q. 어떻게 다른 라이브러리를 사용할 수 있을까?

A. CMake 생태계에서는 크게 3가지 방법이 존재

- `find_package`
  - `export(install)`되었거나, `find_package`가 가능하도록 `${name}-config.cmake`이 작성된 경우
  - Imported Target을 위한 설정을 전부 제공해주기 때문에 CMake 사용자에게는 편한 방법
- `find_library`
  - 링킹을 위한 라이브러리 파일 탐색
  - 시스템 라이브러리 혹은 Cmake를 지원하지 않는 경우 사용하는 방법
- `add_subdirectory`
  - CMakeLists.txt를 가지고 있으나, 설치(`install`)를 지원하지 않음

빌드/설치를 직접 수행해야 한다면? (비권장)

- `ExternalProject_Add`
- `add_custom_target`

# find\_package

```
find_package(Qt5 REQUIRED
COMPONENTS
    Core Widgets
)
target_link_libraries(test_with_qt
PRIVATE
    Qt5::Core Qt5::Widgets
)
```

```
find_package(cppcoro CONFIG REQUIRED)
target_link_libraries(test_suite
PRIVATE
    cppcoro
)
```

다소 복잡한 탐색 규칙을 사용

- MODULE 모드: Find\${name}.cmake 사용
- CONFIG 모드: \${name}-config.cmake 사용
- 명시하지 않으면 두 방법을 순차적으로 적용

REQUIRED 를 사용해 탐색 실패를 오류로 처리 가능

`target_link_libraries` 친화적

- Imported Target을 정의
  - Header/Library 경로
  - PUBLIC 의존성 정보를 이미 포함
- == 상위 모듈만 `find_package`해서 사용하면 모두 해결

**VcPkg에서 지원하는 패키지 사용법**

<https://github.com/Microsoft/vcpkg>

[https://cmake.org/cmake/help/latest/command/find\\_package.html#search-procedure](https://cmake.org/cmake/help/latest/command/find_package.html#search-procedure)

# find\_library

```
list(APPEND CMAKE_LIBRARY_PATH
  "C:/vcpkg/installed/x64-windows/lib"
)
find_library(LIB_PATH NAMES libEGL
  # PATHS "C:/vcpkg/installed/x64-windows/lib"
)
if(LIB_PATH)
  message(STATUS "using libEGL: ${LIB_PATH}")
Else()
  message(STATUS "library not found!")
endif()
```

```
$ cmake ..
-- using libEGL: C:/vcpkg/installed/x64-windows/debug/lib/libEGL.lib
...
```

## 경로 + 이름을 사용한 라이브러리 탐색

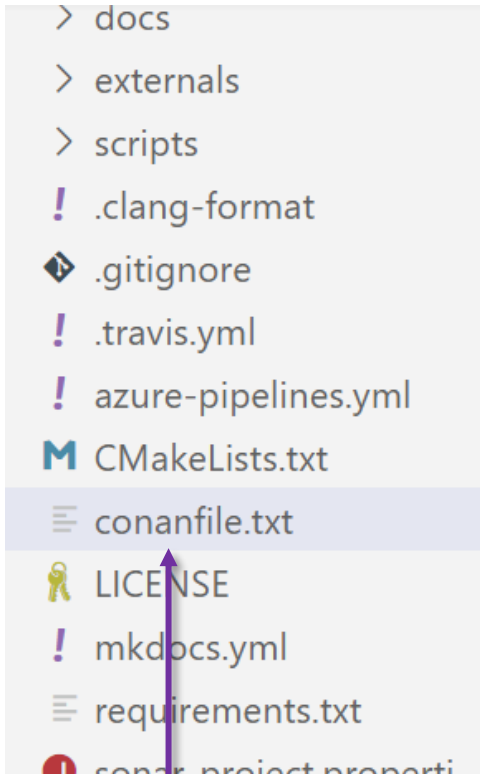
- Header는 찾을 수 없지만...
- include/lib의 설치 위치가 인접한 경우,  
`get_filename_component`를 사용해서 include 폴더 경로를 `set`

## 같은 이름의 파일이 여럿 존재한다면?

- CMAKE\_LIBRARY\_PATH만으로는 잘못 선택될 수 도 있음
- 가능하다면 `find_library`에서 PATHS를 지정하는 것을 권장

[https://cmake.org/cmake/help/latest/command/find\\_library.html](https://cmake.org/cmake/help/latest/command/find_library.html)

# Conan Package Manager



≡ conanfile.txt

```
1 # conan install .. --build missing -s build_type=Debug -s compiler.runtime=MDd
2 [requires]
3 boost/1.72.0
4 icu/66.1
5
6 [generators]
7 cmake
```

설치 시점에 빌드 설정을 지정

설치 폴더는 자동으로 결정

- 같이 설치되는 패키지들이 흩어져 배치됨
- 때문에 정적 라이브러리로 설치하고,  
빌드 결과물은 DLL(Shared Object)로 생성하는 것을 권장

conanfile.txt를 작성해 필요한 패키지 목록을 제공

[https://docs.conan.io/en/latest/getting\\_started.html](https://docs.conan.io/en/latest/getting_started.html), <https://conan.io/center>

[https://github.com/jacking75/examples\\_cpp\\_conan](https://github.com/jacking75/examples_cpp_conan)

# Conan Package Manager: Import

## M CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.13)
2  project(wbcs VERSION 1.0.0 LANGUAGES CXX)
3
4  get_filename_component(CONAN_FILE ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake ABSOLUTE)
5  message(STATUS "using conan: ${CONAN_FILE}")
6  include(${CONAN_FILE})
7  conan_basic_setup()
8
9  target_link_libraries(${PROJECT_NAME}
10 PRIVATE
11     ${CONAN_LIBS_BZIP2} # boost requires bzip2
12     libboost_filesystem
13 )
```

conan install .. 을 통해 생성된 파일

일반적인 Cmake Module 사용법과 동일

필요한 lib/Target만 target\_link\_libraries



# add\_subdirectory

---

## 최후의 수단

이 방법으로도 사용할 수 없다면 CMake가 아닌 다른 방법을 생각해야함

- 알려진 다른 패키지 매니저에서 지원하는지 확인
- 지원하는 Buildsysteem으로 빌드/설치를 진행

원하는 형태로 CMakeLists.txt를 다시 작성하는 것이 오히려 빠른 방법일 수 있음

- 이미 다른 누군가가 작성했을 가능성...
- 빌드 관련사항들을 면밀하게 조사하고 진행해야함
  - 마지막으로 성공한 빌드?
  - 컴파일러, SDK의 버전이 여전히 사용 가능한가?
  - Deprecated된 기능을 사용하고 있지는 않은가?

# ExternalProject\_Add

CMake 모듈로 Custom Target을 만들어 외부 프로젝트의 빌드를 수행하는 방법이 있음

```
include(ExternalProject)
ExternalProject_Add(install_fmt
    TMP_DIR                ${CMAKE_BINARY_DIR}/downloads
    DOWNLOAD_DIR            ${PROJECT_SOURCE_DIR}/externals
    SOURCE_DIR              ${PROJECT_SOURCE_DIR}/externals/fmt
    GIT_REPOSITORY          "https://github.com/fmtlib/fmt.git"
    GIT_TAG                 6.2.1
    GIT_SHALLOW            true
    BUILD_IN_SOURCE         true
    CONFIGURE_COMMAND        cmake . -DCMAKE_INSTALL_PREFIX=${CMAKE_INSTALL_PREFIX}
                           -DFMT_INSTALL=ON -DFMT_TEST=OFF
    BUILD_COMMAND           cmake --build .
    BUILD_ALWAYS            true
    INSTALL_COMMAND         cmake --build . --target install --config ${CMAKE_BUILD_TYPE}
)

```

Build 폴더를 굳이 만들지 않고 사용

CMAKE\_INSTALL\_PREFIX 에 설치 후 결과물을 사용

```
$ cmake --build . --target install_fmt
...
```

# 빌드 후 설치 지원

---

find\_package 할 수 있는 모듈 만들기

# 설치는 무엇이고, 왜 지원하는 걸까?

---

간단하게 생각해 보면...

- 빌드: 프로그램을 만드는 것
- 설치: 프로그램을 사용할 수 있는 위치(Path)에 배치하는 것

빌드 결과물에 대한 사용 편의성

- PATH 혹은 LD\_LIBRARY\_PATH 관리 정책을 따를 수 있음
- 배포/백업을 위한 패키징 가능
- 설치를 하고 나면 사용한 소스코드에 대해 더 이상 신경을 쓰지 않아도 된다

많은 경우, 빌드에는 사용하기 위한 것보다 더 많은 설정이 필요하다

- Private 설정, 구현에만 필요한 외부 라이브러리들 ...
- 빌드에 필요한 설정 Build Requirement >> 사용을 위한 설정 Usage Requirement

# 프로젝트가 설치를 지원하지 않을 이유?

---

어떤 이유가 있을까?

- 배포를 상정하지 않음
  - 단일 저장소에서 진행하는 구성된 소규모 프로젝트
- 관리자/소유자가 해당 생태계에 익숙하지 않음
  - 지정된 Platform/Toolchain만 지원
  - 특정 Buildsystem만 지원
- 설치 과정을 통하지 않고도 패키징이 가능
  - Android NDK with CMake

**사용자 입장에서는 ...**

- Build requirement와 Usage requirement를 직접 구분해야 한다
- 이해하기 전까지는 빌드 결과물을 접할 수 없다
  - 만약 제대로 이해한 것이 아니라면?

# 설치 지원하기

---

## Unix Makefiles 프로젝트들의 사용 방법?

- Make install 이후 ...
  - Manual, .pc, .exp 파일 등이 지정한 prefix 경로 하위에 배치됨
  - 소스 파일이 필요하다면, 이 파일들도 같이 설치 (대표적인 예가 header 들)
- 이후 pkg-config와 같은 도구로 재사용에 필요한 Search Path, dependency를 확인

## CMake에서도 유사한 절차를 따름

- `install` 명령을 통해 빌드 결과물의 설치 경로를 지정
- .pc 대신 `${name}-config.cmake` 파일로 `find_package`를 지원

# install

```
install(TARGETS    ssf
        RUNTIME    DESTINATION ${CMAKE_INSTALL_PREFIX}/bin
        LIBRARY    DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
        ARCHIVE    DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
)
```

플랫폼과 상관없이 사용 가능

- **bin**: 동적 로딩 라이브러리(.dll, .so, .dylib)
- **lib**: 정적 링킹 라이브러리(.lib, .a)

```
$ cmake --build . --target install
...
-- Install configuration: "Debug"
-- Installing: C:/installed/lib/ssf.lib
-- Installing: C:/installed/bin/ssf.dll
```

<https://cmake.org/cmake/help/latest/command/install.html>

## install: Export

`find_package`에서 필요한  
`${name}-config.cmake` 파일을 추가로 생성

```
install(TARGETS      ssf
        EXPORT        ssf-config
        RUNTIME       DESTINATION ${CMAKE_INSTALL_PREFIX}/bin
        LIBRARY        DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
        ARCHIVE        DESTINATION ${CMAKE_INSTALL_PREFIX}/lib
)

install(EXPORT        ssf-config
        DESTINATION ${CMAKE_INSTALL_PREFIX}/share/ssf
)
```

`find_package`의 config 파일 탐색 경로에 배치

```
$ cmake --build . --target install
...
-- Install configuration: "Debug"
-- Installing: C:/installed/share/ssf/ssf-config.cmake
-- Installing: C:/installed/share/ssf/ssf-config-debug.cmake
```



## install: Files

```
# copy headers to $<INSTALL_INTERFACE:...>
install(FILES      ${PROJECT_SOURCE_DIR}/include/socket.hpp
        DESTINATION ${CMAKE_INSTALL_PREFIX}/include/ssf
)
```

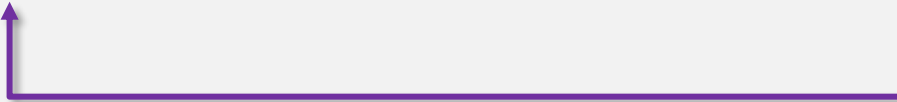
일반 파일의 설치는 파일 목록과 DESTINATION만으로 수행가능

```
$ cmake --build . --target install
...
-- Install configuration: "Debug"
-- Installing: C:/installed/include/ssf/socket.hpp
...
```

## install: config-version.cmake

`find_package`에서 Major/Minor버전을 검사하기 위해서는 버전 정보가 필요

```
include(CMakePackageConfigHelpers)
set(VERSION_FILE_PATH    ${CMAKE_BINARY_DIR}/cmake/ssf-config-version.cmake)
write_basic_package_version_file(${VERSION_FILE_PATH}
    VERSION                ${PROJECT_VERSION} # project(ssf VERSION 1.1.4)
    COMPATIBILITY          SameMajorVersion
)
```



CMake 패키지(Public 설정)의 버전이므로,  
.so 혹은 .dylib에 사용되는 버전에는 영향을 미치지 않음

프로그램의 버전은 `set_target_properties`,  
VERSION 및 SOVERSION 속성을 사용해서 지정해야함

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

```
$ cmake --build . --target install
```

```
...
```

```
-- Installing: C:/installed/share/ssf/ssf-config-version.cmake
```

# 마치면서...

---

이 이상을 CMake로 하려면 힘들어 집니다

## Q. 무엇이 CMake를 어렵게 만드는 걸까?

---

CMake 파일들을 읽는 법에 대해서는 설명이 부족하다

- 공식문서의 설명으로는 다소 부족하고 ...
- CMake 언어에 대해 이해하고 진입해야 부담이 완화됨

자신의 개발에 알맞은 Toolchain.cmake를 찾거나 작성하는데 필요한 노력

- 변화에 따라가면서 내용을 갱신하는 부담

최종적으로는 Native 툴을 사용해야 한다

- 디버깅할 때는 결국 VS, Xcode를 열게 되어있는데...
- Property들은 플랫폼/툴체인마다 달라진다.
  - 내용이 바뀌면 생성된 파일이 어떻게 변화하는지 이해해야 함
  - 빌드 설정의 총량은 거의 줄어들지 않음

# 가이드라인: 문제가 복잡한 경우

---

## 의존성

- 2차, 3차 의존성에 대한 관리
- 더 이상 갱신되지 않는 패키지 사용
- **A. 패키지 매니저를 통해 관리**

## 다수의 빌드 시스템을 지원

- 일반적으로 빌드시스템마다 관례가 다른 경우
- **A. 플랫폼에 따라서 소스코드를 분리하고, Native Buildsystem 파일은 Linker 설정을 공유할 수 있도록 배치**
  - `impl_windows.cpp`, `impl_darwin.cpp`
  - 같은 `/lib /bin` 폴더를 공유

# 가이드라인: 문제를 복잡하게 만든 경우

---

전처리기 설정을 강제하는 소스 코드

- `#include "../common.hpp"`와 같은 상대경로 기반 import는 Header Search Path 설정에 제약을 발생시킴
- A. 상대경로를 금지하고 Header Search Path를 줄이는 방향으로 작성

소스 파일마다 적용해야하는 Macro Set이 다름

- A. 공통 Macro는 Header보다는 빌드 설정으로 유지하고, 나머지는 소스 파일에서 직접 define

빌드와 무관한 일을 CMake에서 수행

- Ex) 현재 빌드 환경에 대한 검사, 파일 내용 변경, 폴더 압축 ...
- 컴파일러/링커 이외의 Tool에 대한 종속성 발생
- A. CMake에서 처리하기보다는 사전 준비 단계에서 스크립트(.sh, .ps1)를 실행 '빌드'를 수행하는데 필요한 내용 이외에는 작성하지 않는다

# 정리

---

**CMake는 참고자료를 정주행하고 사용해야 한다. (제발!)**

CMake는 Buildsystem Generator

- 실제 빌드는 Native Buildsystem이 수행한다

CMake 3.x 는 Target단위로 관리된다

- Property를 사용해 파일을 생성할 때 사용할 값들을 지정
- Public/Private으로 설정을 전파
  - Header/Library 탐색 경로, Library/Target 목록, 컴파일러 옵션

Import 지원

- add\_subdirectory대신 find\_package, find\_library를 권장
- 가능하다면 패키지 매니저를 사용

Export 지원

- Install로 \${name}-config.cmake 생성하기

감사합니다!

---

좀 더 복잡한 내용은 또 언젠가 그룹에 공유하는 걸로...