CMake로 C++ 프로그래밍

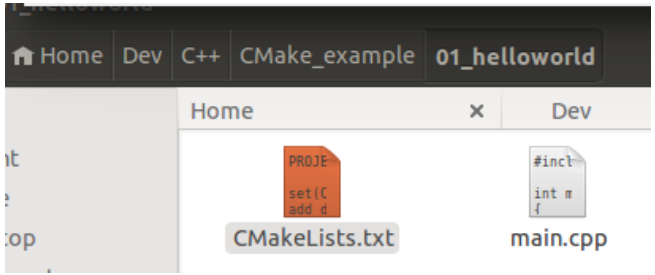
CMake ?

빌드 툴.
make 파일을 만들어준다.
멀티플랫폼에서 사용할 수 있다!!!

```
The following generators are available on this platform:
  Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files (experimental).
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles = Generates CodeBlocks project files.
  Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Unix Makefiles
                               = Generates Eclipse CDT 4.0 project files.
  KDevelop3                    = Generates KDevelop 3 project files.
  KDevelop3 - Unix Makefiles   = Generates KDevelop 3 project files.
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
  Sublime Text 2 - Unix Makefiles
                               = Generates Sublime Text 2 project files.
```

cmake -help

hello world



main.cpp

```
#include <iostream>

int main()
{
    auto name = "jacking";
    std::cout << "hello: " << name << std::endl;
    return 0;
}
```

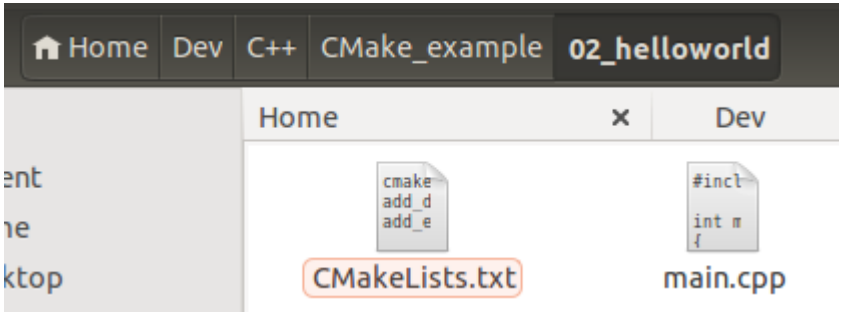
CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_executable(Main main.cpp)
```

cmake_minimum_required
CMakeLists.txt에 기재된 내용을 실행할 때 필요한 CMake 버전을 명시한다.

add_executable
실행 파일 이름과 빌드에 필요한 c/cpp파일 지정.
헤더 파일은 여기에 지정하지 않아도, 갱신하면 자동으로 인식해서 리빌드 대상으로 해준다.

hello world



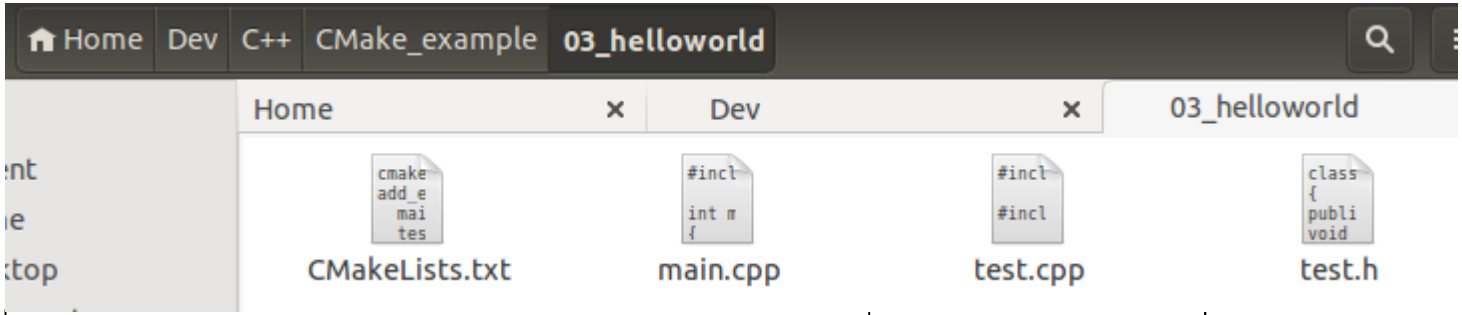
main.cpp

```
#include <iostream>

int main()
{
    auto name = "jacking";
    std::cout << "hello: " << name << std::endl;
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_definitions("-Wall -std=c++14")
add_executable(Main main.cpp)
```

hello world



CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_executable(Main
  main.cpp
  test.cpp
)
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
SET(SRCS
 main.cpp
  test.cpp
)
add_executable(Main ${SRCS})
```

test.h

```
class TEST
{
public:
        void Print();
};
```

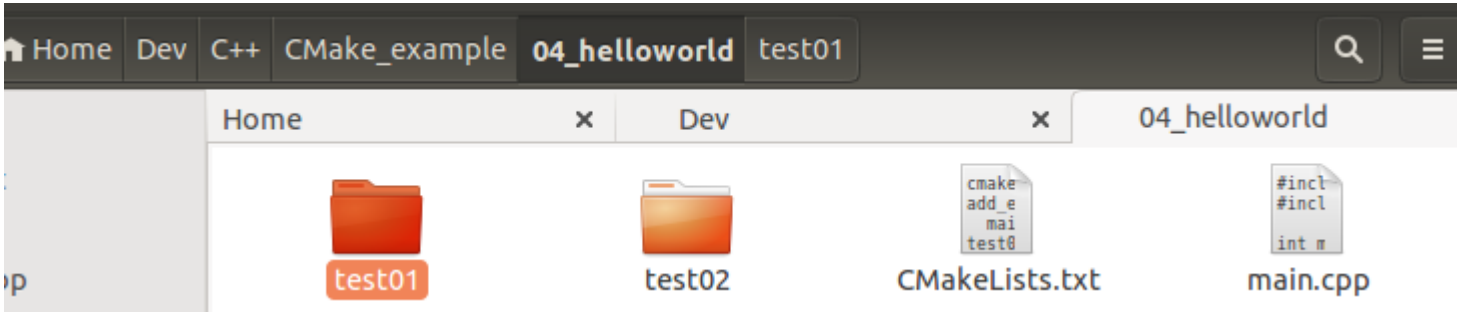
test.cpp

```
#include "test.h"

#include <iostream>

void TEST::Print()
{
        std::cout << "Test::Print" << std::endl;
}
```

main.cpp

```
#include "test.h"

int main()
{
    TEST test;
    test.Print();
    return 0;
}
```

hello world



CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_executable(Main
                main.cpp
                test01/test01.cpp
                test02/test02.cpp
)
```

test01.h

```
class TEST01
{
public:
        void Print();
};
```

test01.cpp

```
#include "test01.h"

#include <iostream>

void TEST01::Print()
{
        std::cout << "Test01::Print"
<< std::endl;
}
```

test02.h

```
class TEST02
{
public:
        void Print();
};
```

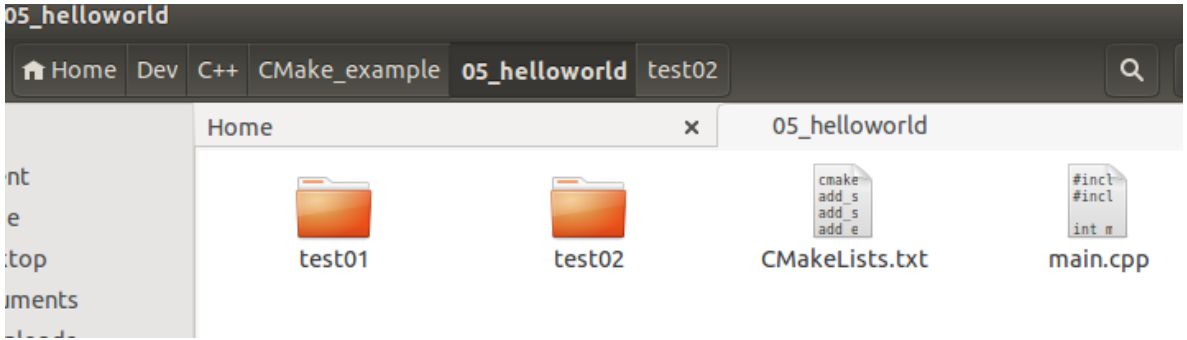
test02.cpp

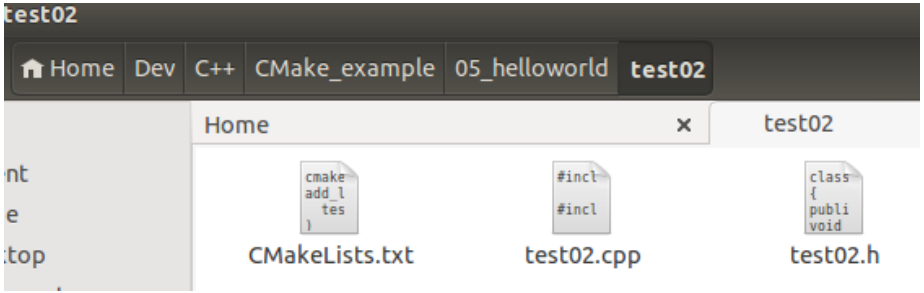
```
#include "test02.h"

#include <iostream>

void TEST02::Print()
{
        std::cout << "Test02::Print" << std::endl;
}
```

main.cpp

```
#include "test01/test01.h"
#include "test02/test02.h"

int main()
{
    TEST01 test01;
    test01.Print();

    TEST02 test02;
    test02.Print();
    return 0;
}
```

hello world





- 각 디렉토리별로 static 라이브러리를 빌드하고, 메인에서 사용한다.
- 메인에서 빌드하면 test01, test02 디렉토리의 것도 빌드한다.

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_subdirectory(test01)
add_subdirectory(test02)
add_executable(Main main.cpp)
target_link_libraries(Main Test01 Test02)
```

test01/CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_library(Test01 STATIC
  test01.cpp
)
```

test02/CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
add_library(Test02 STATIC
  test02.cpp
)
```

add_library
STATIC를 붙이면 정적 라이브러리를 만든다.
test01/libTest01.a, test02/libTest02.a를 만든다.

target_link_libraries
라이브러리를 실행 파일에 링크한다.
구체적으로는 플래그에 `-lTest01-lTest02`가 추가되는 형식.
이름의 해결은 cmake가 하고, cmake는 직접 만든 라이브러리의 이름은 (디렉토리 관계 없이)글로벌 하게 유지하므로 여기서 test01/Test01 처럼 할 필요는 없다.

add_subdirectory
디렉토리를 cmake의 관리에 추가한다. 이 디렉토리에 CMakeLists.txt가 존재하지 않으면 에러가 된다.
add_definitions, set에 의한 변수의 정의는 자식 디렉토리에서 전해지지만, 부모 디렉토리에는 전해지지 않는다.

test01.h

```
class TEST01
{
public:
        void Print();
};
```

test01.cpp

```
#include "test01.h"

#include <iostream>

void TEST01::Print()
{
        std::cout << "Test01::Print"
<< std::endl;
}
```

test02.h

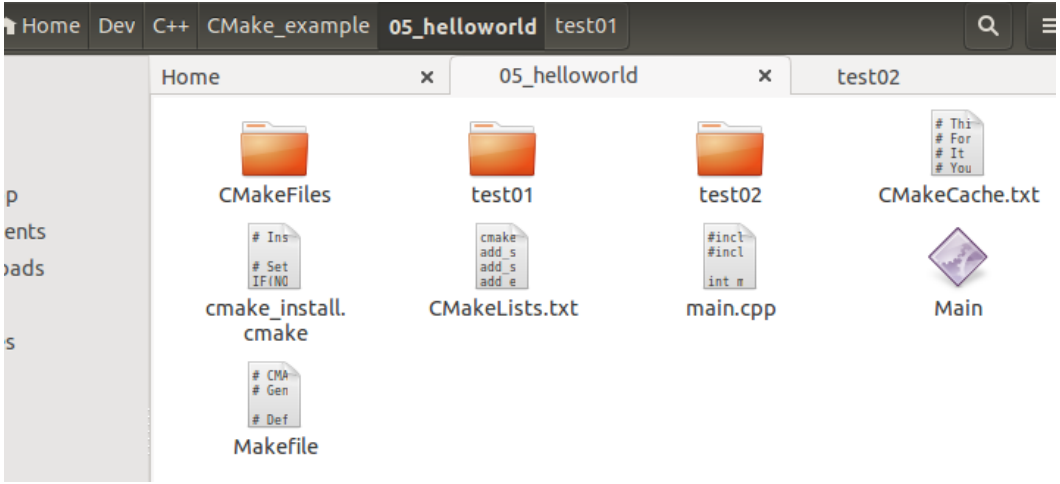
```
class TEST02
{
public:
        void Print();
};
```

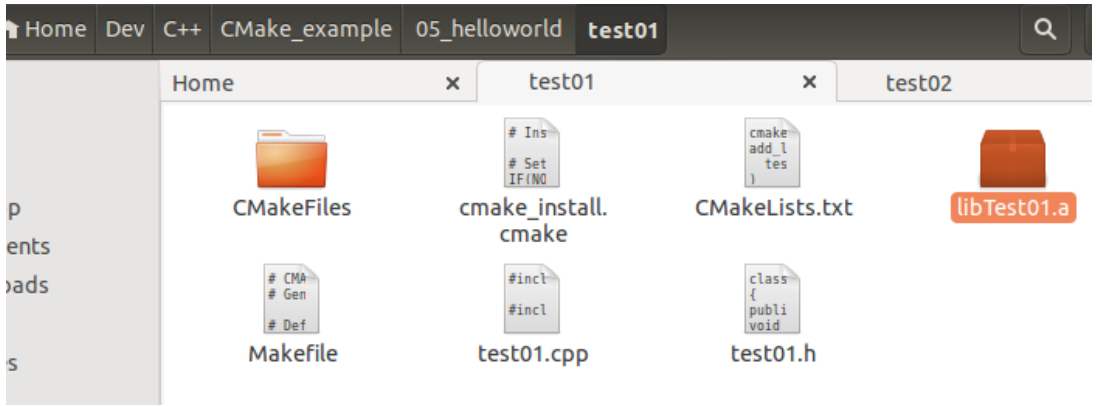
test02.cpp

```
#include "test02.h"

#include <iostream>

void TEST02::Print()
{
        std::cout << "Test02::Print" << std::endl;
}
```

main.cpp

```
#include "test01/test01.h"
#include "test02/test02.h"

int main()
{
    TEST01 test01;
    test01.Print();

    TEST02 test02;
    test02.Print();
    return 0;
}
```

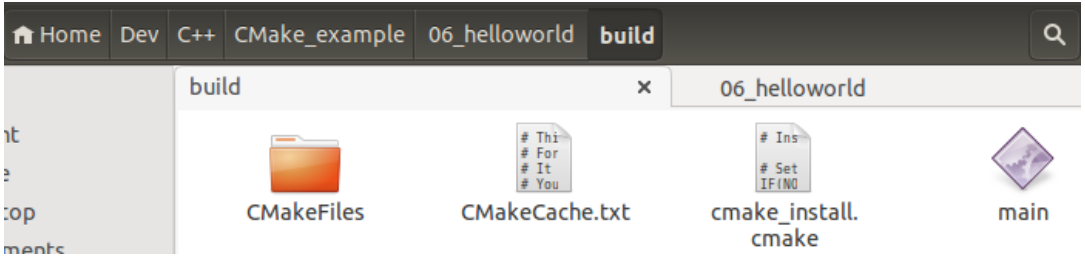

Home
Dev
C++
CMake_example
05_helloworld
test01
Home
05_helloworld
test02
CMakeFiles
test01
test02
CMakeCache.txt
cmake_install.
cmake
CMakeLists.txt
main.cpp
Main
Makefile


Home
Dev
C++
CMake_example
05_helloworld
test01
Home
test01
test02
CMakeFiles
cmake_install.
cmake
CMakeLists.txt
libTest01.a
Makefile
test01.cpp
test01.h

hello world

- cmake에서 만들어지는 빌드 관련 출력물이 소스와 같이 있지 않도록 빌드 디렉토리(build)를 만든다.

```
jacking@ubuntu:~/Dev/C++/CMake_example/06_helloworld$ cd build
jacking@ubuntu:~/Dev/C++/CMake_example/06_helloworld/build$ cmake ..
```



hello world – C++14

hello.cpp

```
#include <iostream>

int main()
{
    auto name = "jacking";
    std::cout << "hello: " << name << std::endl;
    return 0;
}
```

CMakeLists.txt

```
PROJECT(hello)

add_definitions("-Wall -std=c++14")

ADD_EXECUTABLE(hello hello.cpp)
```

Clang

hello.cpp

```
#include <iostream>

int main()
{
    auto name = "jacking";
    std::cout << "hello: " << name << std::endl;
    return 0;
}
```

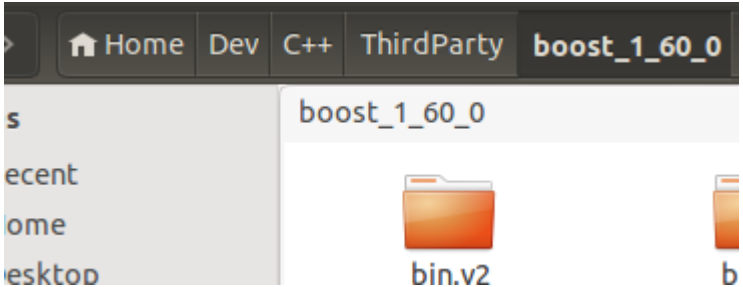
CMakeLists.txt

```
PROJECT(hello)

set(CMAKE_CXX_COMPILER clang++)

ADD_EXECUTABLE(hello hello.cpp)
```

Boost

hello-boost.cpp

```
#include <boost/thread.hpp>
#include <iostream>

int main()
{
    std::cout << "Boost.Thread !!!" << std::endl;
    boost::thread Thread1( [] ()
    {
        for( int i = 0; i < 5; ++i )
        {
            std::cout << "Thread Num : " << i << std::endl;
        }
    } );

    Thread1.join();
    return 0;
}
```



CMakeLists.txt

```
PROJECT(hello-boost)

set(CMAKE_CXX_FLAGS "-m64")
add_definitions("-Wall -std=c++11")

INCLUDE_DIRECTORIES(/$ENV{HOME}/Dev/CPP/ThirdParty/boost_1_56_0)
LINK_DIRECTORIES(/$ENV{HOME}/Dev/CPP/ThirdParty/boost_1_56_0/stage/lib)

ADD_EXECUTABLE(hello-boost hello-boost.cpp)

TARGET_LINK_LIBRARIES(hello-boost pthread boost_thread boost_system boost_chrono)
```

사용자가 LINUX/UNIX에 로그인한 때에 기점이 되는 디렉토리를 홈 디렉토리라고 하는데, 이는 $HOME 라는 쉘 변수의 값으로 이미 설정 되어 있다.

각 유저마다 값이 다르지만 $HOME 으로 제시하면 끝.

예)

```
[root@master home]# echo $HOME
/root

[hopu@master~]$echo $HOME
/home/hopu
```

cmake에서의 변수 다루기

set(HOMHOM "hom")
처럼 지정할 수 있습니다.
이렇게 지정한 변수는 이보다 아래 디렉토리 아래로 전해지지만, 위에는 전달되지 않는다.
위로 전달하는 경우는 CACHE 지정자를 사용한다.

set(HOMHOM "hom“ CACHE STRING "homhom is hommm")
이 경우 ccmake에 의한 변수 덮어 쓰기가 가능하다.
이 사양을 이용하여 가장 부모의 디렉토리로
set(CMAKE_BUILD_TYPE Release)
로 하고, 테스트용 디렉토리에서는
set(CMAKE_BUILD_TYPE Debug)
로 하는 것이 가능하다.

컴파일 옵션

컴파일러에 건네주고 싶은 옵션을 지정할 경우 변수 CMAKE_C_FLAGS/CMAKE_CXX_FLAGS를 사용한다.

CMAKE_C_FLAGS가 C용, CMAKE_CXX_FLAGS가 C++용.

```
#gcc-Wall 옵션을 지정
set(CMAKE_CXX_FLAGS "-Wall")
```

모드 지정

Debug나 Release등 모드를 바꿀 경우 cmake 명령 실행 시 –D 옵션으로 CMAKE_BUILD_TYPE 심볼을 바꾸 쓰는 것으로 실현한다.

```
#Release 모드로 빌드
cmake-DCMAKE_BUILD_TYPE=Release ..
```

선택 가능한 모드의 목록
- 지정 없이

초기 상태로 CMAKE_BUILD_TYPE 심볼을 바꿔 쓰지 않으면 이 상태
한번이라도 다른 값으로 쓰면 쭉 기억하므로 다시 지정 없이 하고자 할 경우
  -DCMAKE_BUILD_TYPE= 로 한다

-Debug
CMAKE_C_FLAGS/CMAKE_CXX_FLAGS 와 함께 변수
CMAKE_C_FLAGS_DEBUG/CMAKE_CXX_FLAGS_DEBUG 값도 사용된다

Release
CMAKE_C_FLAGS/CMAKE_CXX_FLAGS 와 함께 변수
CMAKE_C_FLAGS_RELEASE/CMAKE_CXX_FLAGS_RELEASE 의 값도 사용된다

RelWithDebInfo
최적화 하면서 디버깅용 정보도 부가하기 위한 모드
◦CMAKE_C_FLAGS/CMAKE_CXX_FLAGS 와 함께 변수
CMAKE_C_FLAGS_RELWITHDEBINFO/CMAKE_CXX_FLAGS_RELWITHDEBINFO
값도 사용된다

MinSizeRel
실행 파일의 크기를 가장 작게 하기 위한 모드
CMAKE_C_FLAGS/CMAKE_CXX_FLAGS 와 함께 변수
CMAKE_C_FLAGS_MINSIZEREL/CMAKE_CXX_FLAGS_MINSIZEREL 의 값도
사용된다

CMAKE_C_FLAGS/CMAKE_CXX_FLAGS 에 공통하는 옵션, 기타에 목적 별
옵션을 지정하면 아주 좋다.

프로젝트 이름

http://www.cmake.org/cmake/help/v3.0/command/project.html

project() 로 프로젝트 이름과 사용 언어를 지정할 수 있다.

프로젝트 이름을 지정함으로써 다른 프로젝트에서 읽을 수 있다.

사용 언어는 default로는 C와 CXX 이다.
이것을 명시하지 않으면 C++ 만 사용할 수 있으므로 C 언어용 명령어를 탐색하거나 약간 쓸데없는 것이 많아진다.

샘플 예

```
cmake_minimum_required(VERSION  2.8)

project(MyProject CXX)

set(CMAKE_CXX_FLAGS "-std=c++11 -Wall -Wextra -pedantic -Wcast-align -Wcast-qual -
Wconversion -Wdisabled-optimization -Wendif-labels -Wfloat-equal -Winit-self -Winline -Wlogical-op -
Wmissing-include-dirs -Wnon-virtual-dtor -Wold-style-cast -Woverloaded-virtual -Wpacked -Wpointer-
arith -Wredundant-decls -Wshadow -Wsign-promo -Wswitch-default -Wswitch-enum -Wunsafe-loop-
optimizations -Wvariadic-macros -Wwrite-strings")
set(CMAKE_CXX_FLAGS_DEBUG "-g3 -O0 -pg")
set(CMAKE_CXX_FLAGS_RELEASE "-O2 -s -DNDEBUG -march=native")
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "-g3 -Og -pg")
set(CMAKE_CXX_FLAGS_MINSIZEREL "-Os -s -DNDEBUG -march=native")

add_executable(my-command
    main.cpp
    my-class.cpp)

target_link_libraries(my-command
    my-lib)
```

파일 다루기

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
FILE(WRITE "bada" "first message\n")
FILE(APPEND "bada" "second message\n")
FILE(APPEND "android" "third message\n")
```

file에 메시지를 쓴다. 만약 파일이 없으면 파일을 생성한다. 파일이 생기는 위치는 소스 디렉토리를 기준으로 한다.
위 예제를 실행하면 bada라는 파일과 android라는 파일이 생기고 위에 지정한 메시지가 출력된다.

```
file(WRITE filename "message to write"... )
file(APPEND filename "message to write"... )
file(READ filename variable [LIMIT numBytes] [OFFSET offset] [HEX])
file(STRINGS filename variable [LIMIT_COUNT num]
      [LIMIT_INPUT numBytes] [LIMIT_OUTPUT numBytes]
      [LENGTH_MINIMUM numBytes] [LENGTH_MAXIMUM numBytes]
      [NEWLINE_CONSUME] [REGEX regex]
      [NO_HEX_CONVERSION])
```

Visual Studio 에서

CMakeList.txt를 Visual Studio의 cmd에서 빌드한다.
VS의 프로젝트 파일을 만들어 준다.

빌드 시에 소스 파일 만들기

```
cmake_minimum_required (VERSION 2.6)

# The version number.
set (Tutorial_VERSION_MAJOR 1)
set (Tutorial_VERSION_MINOR 0)

configure_file (
  "${PROJECT_SOURCE_DIR}/TutorialConfig.h.in"
  "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  )

include_directories("${PROJECT_BINARY_DIR}")

add_executable(Tutorial tutorial.cxx)
```

TutorialConfig.h.in 파일은 자동 생성이 아니고 직접 만들어야 한다.

CMake가 TutorialConfig.h 를 자동으로 만들면서 Tutorial_VERSION_MAJOR 와 Tutorial_VERSION_MINOR 를 선언해준다

```
// TutorialConfig.h.in
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

자주 사용하는 command

PROJECT(프로젝트 명)
프로젝트 이름 지정

include_directories(INCLUDE_PATH)
include 디렉터리의 패스를 지정(-I 옵션에 쓰임)

link_directories(LIBRARY_PATH)
lib 디렉터리의 패스를 지정(-L 옵션에 쓰임)

add_definitions(definitions1 definition2 ...)
definition 지정. "add_definitions(-DDEBUG)" 등

set_target_properties(타겟 속성 값)
타겟에 속성을 설정한다. "set_target_properties(hellodemo LINK_FLAGS-lc)" 등

ADD_EXECUTABLE(타겟 소스 코드)
실행 파일을 생성.
"ADD_EXECUTABLE(hellodemo hello.c)“

target_link_libraries(타겟 라이브러리 이름)
링크 라이브러리를 지정
"target_link_libraries(hellodemo hello_lib)"

add_library(타겟 IMPORTED)
라이브러리 생성.
"ADD_LIBRARY(hello_lib SHARED libhello.c) " 라고 하면 "libhello_lib.dylib"이 생성된다.

OPTION(플래그 이름 "간단한 설명” 디폴트)
자작의 옵션을 작성.
"OPTION(FLAG_ONE"flag sample one with no value"OFF) " 라고 사용할 때는 컴파일 시 "-DFLAG_ONE=ON"으로 준다

SUBDIRS(PATH)
함께 컴파일하고 싶은 디렉토리 지정.
"SUBDIRS(demos)“
서브 디렉토리도 CMakeLists.txt가 필요하다.

MESSAGE("문자열")
문자열을 출력

FIND_PACKAGE(패키지 이름)
패키지를 검색해서 여러 설정을 읽어 주는 편리한 놈.
예를 들면 "FIND_PACKAGE(OpenGL) " 라고 하면 cache "CMakeCache.txt"에
OpenGL 경로(OPENGL_INCLUDE_DIR)를 출력해준다.

IF(플래그 이름)~ELSE(플래그 이름)~ENDIF(플래그 이름)
if문으로 제어 가능.

소스 트리와 빌드 트리에 관련하는 변수 목록

변수 이름	참조처의 패스
CMAKE_SOURCE_DIR	소스 트리 top 디렉토리
CMAKE_BINARY_DIR	위에 대응하는 빌드 디렉터리
CMAKE_CURRENT_SOURCE_DIR	현재 처리 중인 CMakeLists.txt의 배치 디렉토리
CMAKE_CURRENT_BINARY_DIR	위에 대응하는 빌드 디렉터리
PROJECT_SOURCE_DIR	현재의 프로젝트의 top 디렉토리
PROJECT_BINARY_DIR	위에 대응하는 빌드 디렉터리
<name>_SOURCE_DIR	프로젝트 name top 디렉토리
<name>_BINARY_DIR	위에 대응하는 빌드 디렉터리
CMAKE_CURRENT_LIST_DIR	현재 처리 중의 cmake 파일의 배치 디렉토리