[KISTI 기술문서] ISBN 978-89-294-1734-5

CMake 빌드툴 튜토리얼

2024. 11. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

곽재혁

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
jhkwak@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

- 소프트웨어 빌드 도구(build tool)는 소스 코드를 실행 가능한 바이너리로 변환하는 과정을 자동화하고 관리하는 시스템임. 소프트웨어의 버전관리, 의존성관리, 컴파일, 테스트, 패키징에 필요할 작업을 자동화 하는 것이 목적이다. 프로젝트에 적절한 빌드 도구를 활용함으로써 소프트웨어 개발 프로세스를 간소화, 코드 품질 향상, 개발 팀의 생산성을 향상시킬 수 있다.
- CMake는 2000년에 Kitware와 Insight Consortium에 의해 개발됨. 기존의 Make 빌드 시스템이 유닉스 계열 OS에 중심을 두고 있었던 것과 달리, CMake는 크로스 플랫폼 지원을 주요 목표로 함. 한번 작성하면 윈도우 계열의 환경도 지원함. 직접 빌드 과정을 수행하지 않고, 지정한 운영 체제에 맞는 Makefile 이나 솔루션 파일을 생성 및 관리함. 소스트리과 빌드 트리가 분리되어 있는것이 특징.
- CMake는 초보자에게는 문법과 개념을 익히는데 시간이 걸릴 수 있어 초기 학습에 노력이 필요하다. 대규모 프로젝트에서는 설정 파일이 복잡해 질 수 있다는 단점이 존재함. CMake 버전에 따라서 일부 기능이 달라질 수 있어 주의가 필요하다. 이러한 단점에도 불구하고 크로스 플랫폼 지원과 사용 편의성 등의 장점으로 인해 많은 오픈소스 프로젝트에서 널리 사용되고 있다.
- 빌드시스템이 필요한 이유:
  * 하드코딩 경로를 피하고 싶을 때
  * 여러 대의 컴퓨터에서 패키지를 빌드해야 할 때
  * CI(continuous integration)을 적용해야 할 때
  * 다양한 OS와 여러 컴파일러를 지원하려고 할 때
  * IDE와 통합하여 개발할 때
  * 대규모 프로젝트를 진행할 때
- 본 문서에서는 CMake의 사용법을 익히기 위한 튜토리얼과 실제 프로젝트에서 활용하기 위한 예제를 소개한다. CTest와 Googletest를 이용한 테스팅에 대한 내용도 다루고 있다. CMake외에 최근 사용되고 있는 다른 빌드 도구 몇가지에 대한 소개도 포함한다.

- 본 문서에서 사용된 예제코드는 다음 깃허브 주소에서 찾을 수 있다:

```
git clone https://github.com/swkim85/linux-drill
cd linux-drill/cmake
```

- CMake 3.0은 2014년에 발표되었으며, 이전 버전과의 차이점은 다음과 같다:
  * 문서화가 개선되어 reStructuredText(.rst)형식으로 변환되어 man페이지와 html로 변환된다.
  * Lua에서 영감을 받은 Bracket Argument와 Bracket Comment 구문이 추가됨
  * 새로운 생성기(CodeLite, Kate)를 지원함
  * "INTERFACE" 라이브러리 타입이 추가되어 헤더 전용 라이브러리 생성이 용이해 짐
  * project() 명령어가 VERSION 옵션을 통해 버전 변수를 설정할 수 있게 됨
  * export() 명령어가 새로운 EXPORT 모드를 지원
  * Visual Studio 생성기 이름에 제품 연도가 포함
  * CMake 2.4 이전 버전을 위한 호환성 옵션들이 제거됨

2. CMake 튜토리얼

- 예제 코드를 다운받기. 본 문서는 CMake 튜토리얼[1]를 참고함.

```
$ wget https://cmake.org/cmake/help/latest/_downloads/bc2a2d94a75e2d8ef62c1e24a0b5281c/cmake-
      3.30.3-tutorial-source.zip
$ unzip cmake-3.30.3-tutorial-source.zip
$ cd  cmake-3.30.3-tutorial-source
```

- CMake 빌드하기

```
$ sudo apt-get install  build-essential  libssl-dev
$ wget https://github.com/Kitware/CMake/releases/download/v3.30.3/cmake-3.30.3.tar.gz
$ tar zxf cmake-3.30.3.tar.gz
$ cd cmake-3.30.3/
$ ./configure --prefix=/usr/local/cmake ; make  ; make install
$ cd Help/guide/tutorial  # 튜토리얼 소스 코드
```

2.1. Step1: 기본 프로젝트 설정

Ex1. 기본 프로젝트 구축

- CMakeLists.txt 파일은 CMake 빌드 시스템의 핵심 설정 파일로 프로젝트 정의 및 설정, 빌드할 타겟(실행파일, 라이브러리 등) 지정, 소스파일 및 헤더 파일 지정, 컴파일 옵션, 링크할 라이브러리 지정, 설치 규칙 지정 등에 관한 내용을 포함한다.
- project() 명령[2]은 프로젝트의 이름을 지정한다.
  형식: project(<PROJECT_NAME> [VERSION <major>.<minor>.<patch>] ...)
- cmake_minimum_required() 명령[3]은 프로젝트에서 요구하는 cmake의 최소 버전을 명시함.
- add_executable() 명령[4]은 소스코드를 빌드하여 만들어져야 하는 실행파일을 정의한다.

```
// Step1/tutorial.cxx
// A simple program that computes the square root of a number
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " number" << std::endl; // 숫자를 입력받음
    return 1;
  }
  const double inputValue = atof(argv[1]); // 숫자를 double 로 변환
  // calculate square root
  const double outputValue = sqrt(inputValue);    // 숫자의 sqrt 값을 구함
  std::cout << "The square root of " << inputValue << " is " << outputValue
            << std::endl;
  return 0;
}
$ vi Step1/CMakeLists.txt
```

1) https://cmake.org/cmake/help/latest/guide/tutorial/index.html
2) https://cmake.org/cmake/help/latest/command/project.html#command:project
3) https://cmake.org/cmake/help/latest/command/cmake_minimum_required.html
4) https://cmake.org/cmake/help/latest/command/add_executable.html

```
[...]
cmake_minimum_required(VERSION 3.10)
project(Tutorial)
add_executable(tutorial tutorial.cxx)

#   빌드 디렉터리를 만들기
$ mkdir Step1_build ;  cd Step1_build
$ rm -rf CMakeCache.txt CMakeFiles   기존 내용을 무시하고 새로 구성하려면 CMake* 를 삭제한다
   cmake 사용법: cmake -S <path-to-source> -B <path-to-build>
   cmake 프로젝트를 구성하고 네이티브 빌드 시스템을 생성하기
$ cmake ../Step1
-- The C compiler identification is GNU 11.4.0
-- The CXX compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /home/ubuntu/linux-drill/cmake/Step1_build
  빌드 시스템을 호출하여 실제로 프로젝트를 컴파일(compile+link) 한다
$ cmake --build  .    # -v 옵션 사용가능
[ 50%] Building CXX object CMakeFiles/tutorial.dir/tutorial.cxx.o
[100%] Linking CXX executable tutorial
[100%] Built target tutorial
   생성된 실행파일
$ file Tutorial
Tutorial: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpre
ter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=4bb949fabd3d002f187a13ab02b42275fda4e9fa, for GN
U/Linux 3.2.0, not stripped
$ ./Tutorial
Usage: ./Tutorial number
$ ./Tutorial 100
The square root of 100 is 10
```

Ex2: C++ 표준 정의하기

- CMake의 특수 변수는 "CMAKE_"으로 시작하기 때문에 프로젝트에 대한 변수를 만들때 이러한 명명 규칙은 피하는 것이 좋다. 2가지 특별한 변수인 CMAKE_CXX_STANDARD와 CMAKE_CXX_STANDARD_REQUIRED 는 프로젝트를 빌드하기 위한 C++ 표준에 대한 요구사항을 정의한다. 변수를 정의하기 위해 set() 명령[5] 을 사용함

```
$ vi Step1/CMakeLists.txt
[...]
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

$ vi Step1/tutorial.cxx
```

5) https://cmake.org/cmake/help/latest/command/set.html

```
[...]
  // convert input to double
  //const double inputValue = atof(argv[1]);      // stdlib.h, cstdlib
  const double inputValue = std::stod(argv[1]);  // std::stod[6] (string to double)
```

Ex3: 프로젝트 버전 추가

- 프로젝트에 버전을 정의하기 위해서는 헤더파일을 만들고, 헤더 파일안에 버전 매크로 변수를 정의함.

```
$ vi Step1/CMakeLists.txt
[...]
project(Tutorial VERSION 1.0)       # 프로젝트 버전이 변경될 경우 이곳에서만 수정하면됨
configure_file(TutorialConfig.h.in  TutorialConfig.h) # .in 파일을 configure_file 입력으로 지정
  # 헤더파일을 include 에서 찾을 수 있도록 설정.
    PROJECT_BINARY_DIR 변수는 현재 프로젝트의 빌드 디렉터리를 의미
target_include_directories(Tutorial PUBLIC "${PROJECT_BINARY_DIR}")
$ vi Step1/TutorialConfig.h.in
   .in 파일에 @변수명@ 을 사용하면 cmake가 자동으로 치환해 준다.
#define Tutorial_VERSION_MAJOR  @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR  @Tutorial_VERSION_MINOR@
$ vi Step1/tutorial.cxx
[...]
#include "TutorialConfig.h"  // 헤더파일을 include 해줌
[...]
  if (argc < 2) {
    // report version  버전 정보는 매크로 변수에 정의됨
    std::cout << argv[0] << " Version " << Tutorial_VERSION_MAJOR << "."
              << Tutorial_VERSION_MINOR << std::endl;
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }
$ cmake --build  .
$ ./Tutorial
./Tutorial Version 1.0   <------ 버전 정보가 표시됨
Usage: ./Tutorial number

$ more TutorialConfig.h
#define Tutorial_VERSION_MAJOR  1    // @Tutorial_VERSION_MAJOR@ 가 1로 치환됨
#define Tutorial_VERSION_MINOR 0     // @Tutorial_VERSION_MINOR@ 가 0으로 치환됨

  CMakeFiles/프로젝트.dir 폴더에는 빌드과정의 임시 파일들이 들어있다.
$ ls CMakeFiles/Tutorial.dir/
DependInfo.cmake   compiler_depend.internal  depend.make  progress.make
build.make         compiler_depend.make      flags.make   tutorial.cxx.o
cmake_clean.cmake  compiler_depend.ts        link.txt     tutorial.cxx.o.d
```

※ 빌드 경로에는 Makefile이 생성된다. 내부적으로 cmake가 무엇을 하는지 확인하려면 -n (--dry-run) 옵션을 이용할 수 있다. 또는 VERBOSE=1을 설정하여 실행할 수 있다.

```
$ make -n
/usr/bin/cmake -S/home/ubuntu/linux-drill/cmake/Step1 -B/home/ubuntu/linux-drill/cmake/Step1_bu
ild --check-build-system CMakeFiles/Makefile.cmake 0
/usr/bin/cmake -E cmake_progress_start /home/ubuntu/linux-drill/cmake/Step1_build/CMakeFiles /h
```

6) https://en.cppreference.com/w/cpp/string/basic_string/stof

```
ome/ubuntu/linux-drill/cmake/Step1_build//CMakeFiles/progress.marks
make -s -f CMakeFiles/Makefile2 all
make -s -f CMakeFiles/Tutorial.dir/build.make CMakeFiles/Tutorial.dir/depend
cd /home/ubuntu/linux-drill/cmake/Step1_build && /usr/bin/cmake -E cmake_depends "Unix Makefile
s" /home/ubuntu/linux-drill/cmake/Step1 /home/ubuntu/linux-drill/cmake/Step1 /home/ubuntu/linux
-drill/cmake/Step1_build /home/ubuntu/linux-drill/cmake/Step1_build /home/ubuntu/linux-drill/cm
ake/Step1_build/CMakeFiles/Tutorial.dir/DependInfo.cmake "--color="
make -s -f CMakeFiles/Tutorial.dir/build.make CMakeFiles/Tutorial.dir/build
/usr/bin/cmake -E cmake_echo_color "--switch=" --progress-dir=/home/ubuntu/linux-drill/cmake/St
ep1_build/CMakeFiles --progress-num=1,2 "Built target Tutorial"
/usr/bin/cmake -E cmake_progress_start /home/ubuntu/linux-drill/cmake/Step1_build/CMakeFiles 0
$ make clean ; make VERBOSE=1
$ cmake --build . -v
```

※ Ninja[7] 를 이용하여 빌드하기. Ninja는 속도와 효율성에 중점을 둔 소형 빌드 시스템으로 빠른 빌드 속도와 효율적인 병렬 처리, 멀티 플랫폼을 지원하는 것이 특징으로 기본의 Make를 대체하는 것을 목표로 개발됨. 닌자 빌드 시스템은 2012년 Google의 Evan Martin이 개발함. 조건문이나 환경변수를 사용할 수 없다는 제한된 기능은 단점이라고 할 수 있으며, 대규모 프로젝트에서 빌드 속도를 향상시키기 위해 CMake와 같은 상위 레벨 빌드 시스템과 함께 사용하는 것이 일반적임.

```
$ sudo apt install ninja-build       ninja를 설치함
$ ninja --version
1.11.1
$ cmake -G Ninja -B build            -G <generator-name>는 빌드 시스템을 지정
$ cd build
$ file build.ninja                   build.ninja 파일이 생성됨
build.ninja: ASCII text, with very long lines (6414)
$ ninja
[2/2] Linking CXX executable Tutorial
```

※ 지원되는 제너레이터 목록은 cmake --help 에서 확인가능하다

```
$ cmake --help
[...]
Generators

The following generators are available on this platform (* marks default):
  Green Hills MULTI            = Generates Green Hills MULTI files
                                 (experimental, work-in-progress).
* Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files.
  Ninja Multi-Config           = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
[...]
```

※ CMakeList.txt 파일에서 주석(comment)문 처리는 다음과 같다:

```
# 이것은 한 줄 주석입니다
add_executable(main main.cpp)   # 이것은 주석입니다.
#[[
 여러 줄에
 걸친 주석  (Bracket Comment)
```

7) https://ninja-build.org/

```
 입니다.
#]]
 # 주석문은 아니지만 if 명령을 써서 명령을 무력화 할 수 있다.
 # VERBOSE 가 참일때만 메시지를 출력함. cmake -DVERBOSE=1  <dir>
if(VERBOSE)
  message(STATUS "hello cmake step1")
endif()
```

※ cmake --trace는 trace mode를 이용하여 cmake 프로젝트 빌드과정에서 상세 추적이 가능함.
--trace-source=CMakeLists.txt 옵션은 해당 소스파일만(하위디렉터리 포함) 추적한다.

```
$ cmake --trace <path-to-source>
/home/ubuntu/linux-drill/cmake/Step1/CMakeLists.txt(2):  cmake_minimum_required(VERSION 3.10 )P
ut cmake in trace mode.
/home/ubuntu/linux-drill/cmake/Step1/CMakeLists.txt(5):  project(Tutorial VERSION 1.0 )
/usr/share/cmake-3.28/Modules/CMakeDetermineSystem.cmake(35):  if(CMAKE_HOST_UNIX )
[...]
```

※ CMakeFiles 디렉터리에 생성되는 파일들은 다음과 같다.

```
CMakeDirectoryInformation.cmake   현재 디렉터리에 대한 CMake 정보를 포함
Makefile.cmake                  최상위 Makefile 생성에 사용되는 정보
Makefile2                       재귀적 make 호출을 위한 Makefile
TargetDirectories.txt       모든 타겟의 절대 경로 목록
cmake.check_cache           CMake 캐시 확인용 파일
progress.marks              빌드 진행 상황 표시에 사용
<target_name>.dir/          각 타겟별 중간 파일들이 저장되는 디렉터리
├── DependInfo.cmake      의존성 정보
├── build.make            타겟 빌드 규칙
├── cmake_clean.cmake     clean 타겟용 스크립트
├── depend.make           파일간 의존성
├── flags.make            컴파일러 플래그
├── link.txt              링크 명령어
└── progress.make         진행 상황 표시 정보
```

2.2. Step2: 라이브러리 추가하기

Ex1: 라이브러리를 생성하기

- 본 절에서는 MathFunctions 라이브러리를 생성하고 프로젝트에 포함시킨다.

```
   라이브러리 소스코드가 들어있는 폴더
$ ls Step2/MathFunctions/
CMakeLists.txt  MathFunctions.cxx  MathFunctions.h  mysqrt.cxx  mysqrt.h
```

- add_library() 명령[8]은 빌드할 라이브러리가 어떤 소스 파일로부터 빌드되는지 알려준다

```
형식: add_library(<name> [<type>] <sources>...)
   <type> : STTIC, SHARED, MODULE
```

- target_link_libraries() 명령[9]은 타겟에 라이브러리를 연결한다.

```
형식: target_link_libraries(<target> [item1 [item2 [...]]])
  여기서 <target>은 add_executable() 또는 add_library()로 생성된 타겟이어야 함
  item 들은 연결할 라이브러리 이름이나 플래그를 지정함
```

8) https://cmake.org/cmake/help/latest/command/add_library.html
9) https://cmake.org/cmake/help/latest/command/target_link_libraries.html

```
$ vi Step2/MathFunctions/CMakeLists.txt
[...]
  라이브러리를 추가해준다.
add_library(MathFunctions MathFunctions.cxx mysqrt.cxx)
```

```
$ vi Step2/CMakeLists.txt
 # 하위 디렉터리를 이 프로젝트에 포함시켜 줌
add_subdirectory(MathFunctions)
 # MathFunctions 라이브러리를 Tutorial 프로젝트에 연결시켜준다.
target_link_libraries(Tutorial PUBLIC MathFunctions)
 # include 디렉터리에 포함시켜 준다. list(APPEND <list> <var>)는 해당 리스트에 값을 추가함
list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MathFunctions")
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                            ${EXTRA_INCLUDES}
                           )
```

```
$ vi Step2/tutorial.cxx
[...]
#include "MathFunctions.h"    // 라이브러리 헤더 파일을 소스코드에서 include 해줌
[...]      sqrt 를 라이브러리에서 제공하는 sqrt 로 변경
  //const double outputValue = sqrt(inputValue);
  const double outputValue = mathfunctions::sqrt(inputValue);
```

```
// Step2/MathFunctions/MathFunctions.h
#pragma once
namespace mathfunctions {
double sqrt(double x);
}
```

```
// Step2/MathFunctions/MathFunctions.cxx
#include "mysqrt.h"
namespace mathfunctions {
double sqrt(double x) {
  return detail::mysqrt(x);      // mathfunctions::sqrt 에서는 detail::mysqrt 를 호출
}
}
```

```
$ mkdir Step2_build ;  cd Step2_build
$ cmake ../Step2 ; cmake --build .
$ ./Tutorial  10000
Computing sqrt of 10000 to be 5000.5
Computing sqrt of 10000 to be 2501.25
Computing sqrt of 10000 to be 1252.62
Computing sqrt of 10000 to be 630.304
Computing sqrt of 10000 to be 323.084
Computing sqrt of 10000 to be 177.018
Computing sqrt of 10000 to be 116.755
Computing sqrt of 10000 to be 101.202
Computing sqrt of 10000 to be 100.007
Computing sqrt of 10000 to be 100
The square root of 10000 is 100
```

Ex2: 옵션을 추가하기

- 자체적으로 정의한 라이브러리를 사용할지 여부를 옵션으로 선택할 있게 한다. 다음과 같이 변수를 지정하면 라이브러 사용을 끌수 있어야 한다.

```
$ cmake ../Step2  -DUSE_MYMATH=OFF
```

- MathFunctions 라이브러리를 수정함. option() 명령[10]은 이진(boolean) 옵션을 제공한다.

형식 : option(<variable> "<help_text>" [value])

```
$ vi Step2/MathFunctions/CMakeLists.txt
[...]
# USE_MATH 변수를 이용할 수 있게 한다. 디폴트 값은 ON임
option(USE_MYMATH "Use tutorial provided math implementation" ON)
# mysqrt.cxx 는 MathFunctions 에 직접 포함시키지 않고, 하위라이브러리인 SqrtLibrary에 포함되므로 여기서는 제거한다.
# add_library(MathFunctions MathFunctions.cxx mysqrt.cxx)
add_library(MathFunctions MathFunctions.cxx)

# USE_MATH 가 ON 이면 pre-compile 과정에 USE_MYMATH 매크로를 전달함
if (USE_MYMATH)
  target_compile_definitions(MathFunctions PRIVATE "USE_MYMATH")
  # SqrtLibrary 라이브러리를 정의. 정적(static) 방식으로 mysqrt.cxx 소스코드를 포함함
  add_library(SqrtLibrary STATIC
              mysqrt.cxx
              )
  # SqrtLibrary 를 MathFunctions 라이브러리에 포함시킨다.
  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
endif()
```

```
$ vi Step2/MathFunctions/MathFunctions.cxx
[...]
#include <cmath>
#ifdef USE_MYMATH     // 이 매크로가 정의될때만 헤더파일이 필요함
#  include "mysqrt.h"
#endif
namespace mathfunctions {
double sqrt(double x)
{
#ifdef USE_MYMATH    // 이 매크로가 정의되면 자체 라이브러리 mysqrt 를 이용한다.
  return detail::mysqrt(x);
#else                // 그렇지 않으면 std::sqrt를 이용
  return std::sqrt(x);
#endif
}
}
```

- 테스트. 캐시(CMakeCache.txt 파일, CMakeFiles 디렉터리)를 삭제하거나, **--fresh** 옵션(CMake 3.24 이상)을 사용하여 빌드한다. **USE_MYMATH** 옵션에 따라서 다른 라이브러리가 사용된다.

```
$ mkdir Step2_build ; cd Step2_build
$ cmake --fresh ../Step2 ; cmake --build .
[...]
[ 16%] Building CXX object MathFunctions/CMakeFiles/SqrtLibrary.dir/mysqrt.cxx.o
[ 33%] Linking CXX static library libSqrtLibrary.a
[ 33%] Built target SqrtLibrary
[ 50%] Building CXX object MathFunctions/CMakeFiles/MathFunctions.dir/MathFunctions.cxx.o
[ 66%] Linking CXX static library libMathFunctions.a
[ 66%] Built target MathFunctions
[ 83%] Building CXX object CMakeFiles/Tutorial.dir/tutorial.cxx.o
[100%] Linking CXX executable Tutorial
[100%] Built target Tutorial
```

10) https://cmake.org/cmake/help/latest/command/option.html

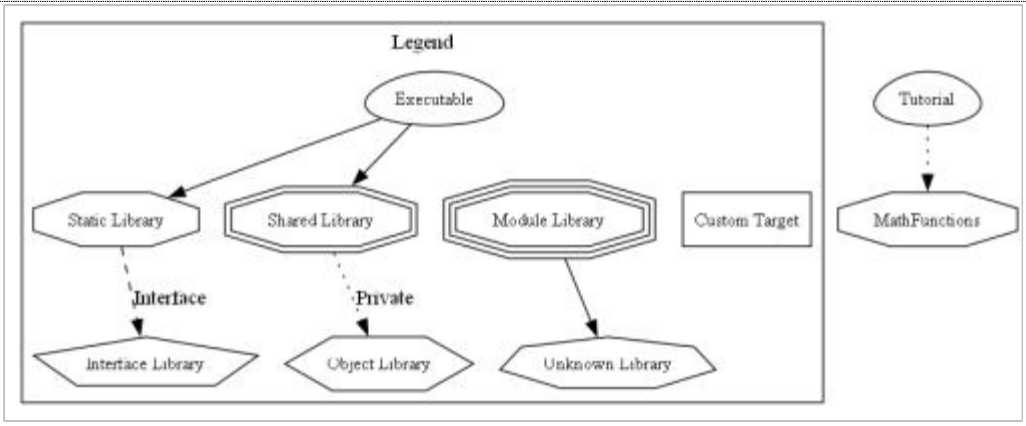
```
$ nm Tutorial | egrep "sqrt|mysqrt"
00000000000015d0 T _ZN13mathfunctions4sqrtEd
00000000000015f0 T _ZN13mathfunctions6detail6mysqrtEd   자체 라이브러리 mysqrt 가 포함
$ cmake --fresh ../Step2 -DUSE_MYMATH=OFF ; cmake --build .    자체 mysqrt 를 사용하지 않음
$ ./Tutorial 10000
The square root of 10000 is 100
$ nm Tutorial | egrep "sqrt|mysqrt"
00000000000015e0 T _ZN13mathfunctions4sqrtEd
                 U sqrt@GLIBC_2.2.5           glibc sqrt 가 동적 링크됨
```

- 캐시 변수를 조회하려면 -L 옵션을 이용. -LA 옵션은 고급(advanced) 캐시 변수도 포함하여 표시함.

```
$ cmake -L .          # cmake -L <path-to-source>
[...]
-- Cache values
CMAKE_BUILD_TYPE:STRING=
CMAKE_INSTALL_PREFIX:PATH=/usr/local
USE_MYMATH:BOOL=ON    <------- 이 프로젝트의 option 명령 에서 정의한 변수
$ cmake -LA .
CMAKE_ADDR2LINE:FILEPATH=/usr/bin/addr2line
CMAKE_AR:FILEPATH=/usr/bin/ar
CMAKE_BUILD_TYPE:STRING=
CMAKE_COLOR_MAKEFILE:BOOL=ON
CMAKE_CXX_COMPILER:FILEPATH=/usr/bin/c++
[...]
CMAKE_VERBOSE_MAKEFILE:BOOL=FALSE
USE_MYMATH:BOOL=ON
$ cmake -LH ..               -LH 는 <help_text> 를 출력한다.
[...]
// Install path prefix, prepended onto install directories.
CMAKE_INSTALL_PREFIX:PATH=/usr/local

// Use tutorial provided math implementation
USE_MYMATH:BOOL=ON
```

※ cmake --graphviz 옵션을 사용하면 라이브러리간의 의존성을 도식화 할 수 있음

```
$ cmake --graphviz=deps.dot .
[...]
Generate graphviz: /home/ubuntu/linux-drill/cmake/Step2_build/deps.dot
$ sudo apt install graphviz
$ dot -Tpng  deps.dot -o deps.png
```



2.3. Step3: 라이브러리에 대한 사용 요구사항 추가

Ex1: 라이브러리에 대한 사용 요구사항 추가

- target_include_directories() 명령[11]에서 범위 지정 PRIVATE, PUBLIC, INTERFACE 차이점은 다음과 같다: • PRIVATE: 타겟에만 적용, 의존성에 전파되지 않음. • PUBLIC: 타겟과 의존성에 모두 적용, •INTERFACE: 의존성에만 적용되고 타겟 자체에는 적용되지 않음

```
$ vi Step3/CMakeLists.txt
[...]  최상위 레벨 소스코드인 tutorial.cxx 에서 MathFunctions.h를 include 하기 위해서  target_include_directories 에서 MathFunctions 폴더를 포함해 주었으나, 이렇게 하는 것은 라이브러리 내부의 구현 세부사항이 외부에 노출되어야 하므로, 유연하지 못한 방법이므로 수정한다:
#delete# list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MathFunctions")  # 주석처리
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"    # ${EXTRA_INCLUDES} 는 제거한다.
                           )

$ vi Step3/MathFunctions/CMakeLists.txt
[...]  대신 MathFunctions 라이브러리 내부에서 다음과 같이 설정하여 라이브러리 자체를 추상화함으로써 유연한 설계가 가능하다. MathFunctions 은 대상 라이브러리의 이름이고, INTERFACE 키워드는 include 디렉터리가 MathFunctions 라이브러리를 사용하는 다른 타겟들에게만 적용되며, MathFunctions 자체를 빌드할 때는 사용되지 않음을 나타낸다.
message(STATUS "CMAKE_CURRENT_SOURCE_DIR -->: ${CMAKE_CURRENT_SOURCE_DIR}")
target_include_directories(MathFunctions
                           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
                           )
[...]
```

Ex2: 인터페이스 라이브러리를 사용하여 C++ 표준 설정

```
$ vi Step3/CMakeLists.txt
[...]
 # tutorial_compiler_flags 라는 이름의 인터페이스 라이브러리를 생성함. INTERFACE 라이브러리는 헤더 파일만 포함하고 컴파일되는 소스코드는 없음.
add_library(tutorial_compiler_flags  INTERFACE)
 #  INTERFACE 라이브러리를 사용하여 C++11 표준을 프로젝트 전체에 적용할 수 있음
target_compile_features(tutorial_compiler_flags INTERFACE cxx_std_11)
 # PUBLIC 키워드는 'MathFunctions'와 'tutorial_compiler_flags'가 'Tutorial' 타겟에 링크될 뿐만 아니라, 'Tutorial'을 사용하는 다른 타겟에도 이 의존성이 전파됨을 의미.
target_link_libraries(Tutorial PUBLIC MathFunctions tutorial_compiler_flags)

$ vi Step3/MathFunctions/CMakeLists.txt
[...]
  #  'SqrtLibrary'는 'tutorial_compiler_flags'에 정의된 컴파일러 설정을 상속받음
  target_link_libraries(SqrtLibrary PUBLIC tutorial_compiler_flags)
  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
endif()

target_link_libraries(MathFunctions PUBLIC tutorial_compiler_flags)
$ mkdir Step3_build ; cd Step3_build
$ cmake --fresh ../Step3 -DUSE_MYMATH=ON ; cmake --build .
[ 33%] Built target SqrtLibrary
[ 66%] Built target MathFunctions
```

11) https://cmake.org/cmake/help/latest/command/target_include_directories.html

```
[ 83%] Building CXX object CMakeFiles/Tutorial.dir/tutorial.cxx.o
[100%] Linking CXX executable Tutorial
[100%] Built target Tutorial
```

2.4. Step4: 제너레이터 표현식

Ex1: 제너레이터 표현식을 이용하여 컴파일러 플래그 추가

- 제너레이터 표현식(generator expression)[12]은 빌드 시스템 생성 중에 평가되어 각 빌드 구성에 대한 특정 정보를 생성한다. 형식은 $<...> 이다. 제너레이터 표현식을 사용하는 이유는 다음과 같다: ①조건부 설정: 빌드 구성, 플랫폼, 컴파일러 등에 따라 다른 설정을 적용할 수 있음. ② 지연 평가: 구성 단계가 아닌 생성 단계(Makefile 생성)에서 평가되므로, 구성 시점에 알 수 없는 정보를 사용할 수 있음 ③ 멀티 구성 지원: Visual Studio와 같은 멀티 구성 생성기에서 여러 빌드 타입을 동시에 지원할 수 있음 ④ 속성 설정의 유연성: 타겟 속성을 설정할 때 더 유연하고 동적인 방식으로 값을 지정할 수 있음 ⑤ 크로스 플랫폼 지원: 서로 다른 플랫폼에 대해 동일한 CMakeLists.txt 파일을 사용하면서 플랫폼별 설정을 적용할 수 있다. ⑥코드 중복 감소: 조건문을 사용하는 대신 제너레이터 표현식을 사용하여 코드를 더 간결하게 만들 수 있다.

```
$ vi  Step4/CMakeLists.txt
cmake_minimum_required(VERSION 3.15)
[...]
  # gcc_like_cxx 라는 변수를 정의한다. COMPILE_LANG_AND_ID 제너레이터 표현식이 사용됨
   COMPILE_LANG_AND_ID: 컴파일 언어와 컴파일러 ID를 확인
   CXX,ARMClang,AppleClang,Clang,GNU,LCC: 이 부분은 C++ 언어와 GCC 계열 컴파일러들을 나열
   현재 사용 중인 컴파일러가 나열된 컴파일러 중 하나이고 C++를 컴파일하고 있다면 1을 반환.
   그렇지 않으면 0을 반환
set(gcc_like_cxx "$<COMPILE_LANG_AND_ID:CXX,ARMClang,AppleClang,Clang,GNU,LCC>")
set(msvc_cxx "$<COMPILE_LANG_AND_ID:CXX,MSVC>")

  # tutorial_compiler_flags 타겟에 대한 컴파일러 옵션을 설정한다. 앞에서 정의된 결과에 따라 GCC 계열 컴파일러와 MSVC 컴파일러에서 각각 다른 옵션을 설정함
target_compile_options(tutorial_compiler_flags  INTERFACE
  "$<${gcc_like_cxx}:-Wall;-Wextra;-Wshadow;-Wformat=2;-Wunused>"
  "$<${msvc_cxx}:-W3>"
)
  # 빌드 인터페이스에서만 특정 컴파일러 경고 플래그를 적용하도록 한다. 설치된 타겟을 사용할 때는 이 옵션들이 적용되지 않음
target_compile_options(tutorial_compiler_flags  INTERFACE
  "$<${gcc_like_cxx}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-Wformat=2;-Wunused>>"
  "$<${msvc_cxx}:$<BUILD_INTERFACE:-W3>>"
)
[...]
```

```
$ mkdir Step4_build ;  cd Step4_build
$ cmake --fresh  ../Step4
$ cmake --build . -v  ¦ Wall
cd /home/ubuntu/linux-drill/cmake/Build/MathFunctions && /usr/bin/c++   -Wall -Wextra -Wshadow -Wformat=2 -Wunused -MD -MT [...]
cd /home/ubuntu/linux-drill/cmake/Build/MathFunctions && /usr/bin/c++ -DUSE_MYMATH  -Wall -Wextra -Wshadow -Wformat=2 -Wunused -MD -MT [...]
$ grep -r Wall *    # CXX_FLAGS 에 옵션이 적용되었는지 확인
CMakeFiles/Tutorial.dir/flags.make:CXX_FLAGS = -Wall -Wextra -Wshadow -Wformat=2 -Wunused
MathFunctions/CMakeFiles/MathFunctions.dir/flags.make:CXX_FLAGS = -Wall -Wextra -Wshadow -Wformat=2 -Wunused
MathFunctions/CMakeFiles/SqrtLibrary.dir/flags.make:CXX_FLAGS = -Wall -Wextra -Wshadow -Wformat=2 -Wunused
```

12) https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html

※ 제너레이터 표현식 문서 참조

```
$<BUILD_INTERFACE:...>
   Content of ... when the property is exported using export(), or when the target is used by a
   nother target in the same buildsystem. Expands to the empty string otherwise.
```

2.5. Step5: 설치와 테스트

Ex1: 설치 규칙

- CMake는 **install()** 명령[13]을 이용하여 빌드 결과를 설치할 때 규칙을 설정할 수 있음. 설치할 위치와 타겟, 파일 목록을 명시하면 된다.
- 설치를 실행하려면 **cmake --install** 옵션[14]을 사용한다. (cmake 3.15 버전 이후). 설치 경로는 **CMAKE_INSTALL_PREFIX** 변수[15] (또는 **--prefix** 옵션)가 사용된다.

```
$ vi Step5/MathFunctions/CMakeLists.txt
[...]
  # installable_libs 라는 변수를 정의하고, MathFunctions 과 tutorial_compiler_flags 라는 2개의
타겟을 포함시킨다.
set(installable_libs  MathFunctions  tutorial_compiler_flags)
  # TARGET 키워드와 함께 사용되는 if 문은 지정된 이름의 타겟이 현재 CMake 프로젝트에 존재하는지
확인. list 명령어의 APPEND 옵션은 기존 리스트에 새 항목을 추가한다.
if(TARGET SqrtLibrary)
  list(APPEND  installable_libs  SqrtLibrary)
endif()
 # 설치규칙1: installable_libs 변수에 포함된 모든 타겟(라이브러리)을 설치한다. DESTINATION  키
워드는 파일들이 설치될 위치를 지정한다. lib 디렉터리에 설치됨.
install(TARGETS ${installable_libs} DESTINATION lib)
 # 설치규칙2: FILES 키워는 일반 파일을 의미함. DESTINATION  키워드는 파일들이 설치될 위치
install(FILES MathFunctions.h DESTINATION include)
```

```
$ vi Step5/CMakeLists.txt
[...]
  # 설치규칙3: TARGETS 키워드는 설치할 대상이 CMake 타겟임을 나타냄.  설치 접두사(install prefi
x) 아래의 bin 디렉토리에 설치됨
install(TARGETS Tutorial DESTINATION bin)
  # 설치규칙4: TutorialConfig.h 파일도 include 폴더에 설치한다
install(FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  DESTINATION include
  )
```

```
$ mkdir Step5_build ; cd Step5_build
$ cmake --fresh ../Step5  ; cmake --build .
$ cmake --install . --prefix "/home/ubuntu/installdir"
-- Install configuration: ""
-- Installing: /home/ubuntu/installdir/lib/libMathFunctions.a
-- Installing: /home/ubuntu/installdir/lib/libSqrtLibrary.a
-- Installing: /home/ubuntu/installdir/include/MathFunctions.h
```

13) https://cmake.org/cmake/help/latest/command/install.html
14) https://cmake.org/cmake/help/latest/manual/cmake.1.html#install-a-project
15) https://cmake.org/cmake/help/latest/variable/CMAKE_INSTALL_PREFIX.html

```
-- Installing: /home/ubuntu/installdir/bin/Tutorial
-- Installing: /home/ubuntu/installdir/include/TutorialConfig.h
$ tree  ~/installdir/
/home/ubuntu/installdir/
├── bin
│   └── Tutorial
├── include
│   ├── MathFunctions.h
│   └── TutorialConfig.h
└── lib
    ├── libMathFunctions.a
    └── libSqrtLibrary.a
```

Ex2: 테스트 지원

- CTest는 CMake와 함께 제공되는 테스트 실행 및 관리 도구. `add_test()` 명령[16]으로 테스트할 내용을 프로젝트에 추가한다. Google Test Infra(GTest)[17]를 이용한 GoogleTest 모듈[18]도 제공한다. `ctest -N` 와 `ctest -VV` 를 통하여 테스트를 수행[19]한다.

```
$ vi  Step5/CMakeLists.txt
[...]
enable_testing()    # enables testing for this directory and below [20]
add_test(NAME Runs  COMMAND Tutorial 25)  # add a test to the project to be run by ctest [21]
```

```
$ cd Step5 ; mkdir build ; cd build ; cmake ..  ; cmake --build .
$ ctest
Test project /home/ubuntu/linux-drill/cmake/Step5/build
    Start 1: Runs
1/1 Test #1: Runs ............................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1
Total Test time (real) =   0.00 sec

 -N 옵션은 테스트를 실제로 실행하지는 않지만, 테스트의 총 개수와 각 테스트의 레이블을 출력함
$ ctest -N
Test project /home/ubuntu/linux-drill/cmake/Step5/build
  Test #1: Runs
Total Tests: 1

  -V 옵션은 테스트를 실행하고 자세한 출력을 제공한다
$ ctest -V
UpdateCTestConfiguration  from :/home/ubuntu/linux-drill/cmake/Step5/build/DartConfiguration.tcl
Test project /home/ubuntu/linux-drill/cmake/Step5/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 1
    Start 1: Runs
```

16) https://cmake.org/cmake/help/latest/command/add_test.html
17) https://github.com/google/googletest
18) https://cmake.org/cmake/help/latest/module/GoogleTest.html
19) https://cmake.org/cmake/help/latest/manual/ctest.1.html

```
1: Test command: /home/ubuntu/linux-drill/cmake/Step5/build/Tutorial "25"
1: Working Directory: /home/ubuntu/linux-drill/cmake/Step5/build
1: Test timeout computed to be: 10000000  <-- CTest 의 기본 타임아웃 설정(최대 실행 시간, 단위는 밀리초)
1: Computing sqrt of 25 to be 13         <--- Tutorial 25 를 실행한 결과
1: Computing sqrt of 25 to be 7.46154
1: Computing sqrt of 25 to be 5.40603
1: Computing sqrt of 25 to be 5.01525
1: Computing sqrt of 25 to be 5.00002
1: Computing sqrt of 25 to be 5
1: Computing sqrt of 25 to be 5
1: Computing sqrt of 25 to be 5
1: Computing sqrt of 25 to be 5
1: Computing sqrt of 25 to be 5
1: The square root of 25 is 5
1/1 Test #1: Runs ............................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1
Total Test time (real) =   0.00 sec
```

```
$ vi  Step5/CMakeLists.txt
[...]
add_test(NAME Usage COMMAND Tutorial)
  # PASS_REGULAR_EXPRESSION 속성은 테스트의 속성으로 실행파일의 출력내용에서 매치되어야 함
set_tests_properties(Usage              # set a property of the tests [22] [23]
  PROPERTIES PASS_REGULAR_EXPRESSION "Usage:.*number"
  )

add_test(NAME StandardUse COMMAND Tutorial 4)
set_tests_properties(StandardUse
  PROPERTIES PASS_REGULAR_EXPRESSION "4 is 2"
  )
 # do_test 라는 함수를 정의하여 여러가지 테스트 CASE를 정의함
function(do_test  target  arg  result)
  add_test(NAME Comp${arg} COMMAND ${target} ${arg})
  set_tests_properties(Comp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result}
    )
endfunction()
# do a bunch of result based tests
do_test(Tutorial 4 "4 is 2")  # target=Tutorial arg=4  result="4 is 2"
do_test(Tutorial 9 "9 is 3")
do_test(Tutorial 5 "5 is 2.236")
do_test(Tutorial 7 "7 is 2.645")
do_test(Tutorial 25 "25 is 5")
do_test(Tutorial -25 "-25 is (-nan|nan|0)")
do_test(Tutorial 0.0001 "0.0001 is 0.01")
```

2.6. Step6: 테스트 데시보드 지원 추가

Ex1: 테스트 데시보드로 결과 보내기

---

20) https://cmake.org/cmake/help/latest/command/enable_testing.html
21) https://cmake.org/cmake/help/latest/command/add_test.html
22) https://cmake.org/cmake/help/latest/command/set_tests_properties.html
23) https://cmake.org/cmake/help/latest/prop_test/PASS_REGULAR_EXPRESSION.html

- CDash[24]는 CTest와 함께 작동하여 테스트 결과를 표시하는 웹 기반 테스팅 서버임. `ctest -D <dashboard type>` 명령을 통하여 테스트 결과를 데시보드 서버로 전송한. dashboard type은 Nightly, Continuous, Experimental 이 있다. 데시보드 제출에 관련한 사항은 `CTestConfig.cmake` 파일에 설정한다. CDash 서버는 웹서버(Apache 또는 Nginx), PHP, MySQL, Composer를 이용하여 자체적으로 직접 설치하거나, Kitware에서 운영하는 공개 CDash 서버(my.cdash.org) 등을 이용할 수도 있다. CTest를 활용하면 C++ 프로젝트의 테스트 관리와 품질 보증 프로세스를 개선할 수 있음

```
$ vi Step6/CMakeLists.txt
[...]
 # 데시보드에 제출을 하기 위해서는 CTest 모듈을 포함시켜 주어야 한다.
include(CTest)

$ vi Step6/CTestConfig.cmake
set(CTEST_PROJECT_NAME "CMakeTutorial")
set(CTEST_NIGHTLY_START_TIME "00:00:00 EST")

set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "my.cdash.org")   # CDash 서버 주소
set(CTEST_DROP_LOCATION "/submit.php?project=CMakeTutorial")
set(CTEST_DROP_SITE_CDASH TRUE)

  -D 옵션은 dasbboard 테스트를 실행함. -D <dashboard>
$ ctest [-VV] -D Experimental
[...]
100% tests passed, 0 tests failed out of 9
[...]
Submit files
   SubmitURL: http://my.cdash.org/submit.php?project=CMakeTutorial
[...]
   Submission successful
  # 제출된 내용을 Kitware의 데시보드[25]에서 확인할 수 있다.
```

24) https://www.cdash.org/
25) https://my.cdash.org/index.php?project=CMakeTutorial

2.7. Step7: 시스템 내성 추가

- 대상 플랫폼에 없는 기능에 따라 빌드 방법이 달라지는 예제. 본 절에서는 대상 플랫폼에 log 와 exp함수가 있는지 여부에 따라 달라지는 코드를 추가함. check_cxx_source_compiles 모듈[26]은 주어진 C++ 소스코드 조각이 실행파일로 링크되는지 체크함. 코드 조각은 main()함수를 포함해야 함.

  형식: check_cxx_source_compiles(<code> <resultVar> [FAIL_REGEX <regex1> [<regex2>...]])

```
$ vi Step7/MathFunctions/CMakeLists.txt
[...]
option(USE_MYMATH "Use tutorial provided math implementation" ON)
if (USE_MYMATH)
  [...]
  # link SqrtLibrary to tutorial_compiler_flags
  target_link_libraries(SqrtLibrary PUBLIC tutorial_compiler_flags)

  include(CheckCXXSourceCompiles)

  check_cxx_source_compiles("
    #include <cmath>
    int main() {      std::log(1.0);      return 0;    }  " HAVE_LOG)
  check_cxx_source_compiles("
    #include <cmath>
    int main() {      std::exp(1.0);      return 0;    }  " HAVE_EXP)

  if(HAVE_LOG AND HAVE_EXP)     # 위 테스트 결과에 따라 HAVE_LOG, HAVE_EXP 를 정의함
    target_compile_definitions(SqrtLibrary
                               PRIVATE "HAVE_LOG" "HAVE_EXP"
                               )
  endif()

  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
endif()
$ vi Step7/MathFunctions/mysqrt.cxx
[...]
#include <cmath>
[...]
#if defined(HAVE_LOG) && defined(HAVE_EXP)
  double result = std::exp(std::log(x) * 0.5);   // sqrt(x)를 계산, 0.5*log(x)=log(x^0.5)
  std::cout << "Computing sqrt of " << x << " to be " << result
            << " using log and exp" << std::endl;
#else
  double result = x;
  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result << std::endl;
  }
#endif
$ cd Step7 ; cmake -B build
```

26) https://cmake.org/cmake/help/latest/module/CheckCXXSourceCompiles.html

```
[...]
-- Performing Test HAVE_LOG
-- Performing Test HAVE_LOG - Success
-- Performing Test HAVE_EXP
-- Performing Test HAVE_EXP - Success
[...]
$ cd build ; cmake --build .
$ ./Tutorial 2
Computing sqrt of 2 to be 1.41421 using log and exp
The square root of 2 is 1.41421
```

2.8. Step8: 사용자 정의 명령 및 생성된 파일 추가

- log와 exp를 사용하지 않고, 대신 함수에서 미리 계산된 값의 표를 생성하고 싶다고 가정함.

```
$ vi Step8/MathFunctions/MakeTable.cxx
// A simple program that builds a sqrt table
#include <cmath>
#include <fstream>
#include <iostream>
int main(int argc, char* argv[]) {
  // make sure we have enough arguments
  if (argc < 2) {
    return 1;
  }
  std::ofstream fout(argv[1], std::ios_base::out);  // 파일명을 입력받음
  const bool fileOpen = fout.is_open();
  if (fileOpen) {
    fout << "double sqrtTable[] = {" << std::endl;   // sqrtTable을 정의
    for (int i = 0; i < 10; ++i) {    // sqrt 값을 미리 계산하고 저장
      fout << sqrt(static_cast<double>(i)) << "," << std::endl;
    }
    // close the table with a zero
    fout << "0};" << std::endl;
    fout.close();
  }
  return fileOpen ? 0 : 1; // return 0 if wrote the file
}

$ vi Step8/MathFunctions/CMakeLists.txt
[...]
  include(MakeTable.cmake) # generates Table.h

$ vi Step8/MathFunctions/MakeTable.cmake
# first we add the executable that generates the table
add_executable(MakeTable  MakeTable.cxx)
target_link_libraries(MakeTable PRIVATE  tutorial_compiler_flags)

# add the command to generate the source code
# MakeTable 프로그램을 이용하여 소스코드를 생성한다.  27)
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h  # MakeTable 을 실행
  DEPENDS MakeTable
  )
```

```
$ vi Step8/MathFunctions/CMakeLists.txt
[...]
  # library that just does sqrt
  add_library(SqrtLibrary STATIC
              mysqrt.cxx
              ${CMAKE_CURRENT_BINARY_DIR}/Table.h   # Table.h 를 추가해 줌
              )
  # state that we depend on our binary dir to find Table.h
  target_include_directories(SqrtLibrary PRIVATE
                             ${CMAKE_CURRENT_BINARY_DIR}  # include 경로에도 추가함
                             )
$ vi Step8/MathFunctions/mysqrt.cxx
// include the generated table
#include "Table.h"
[...]
  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
    std::cout << "Use the table to help find an initial value " << std::endl;
    result = sqrtTable[static_cast<int>(x)];  // 입력값을 int로 변환하여 테이블에서 참조함
  }
$ cd Step8 ; mkdir build ; cd build
$ cmake ..  ; cmake --build .
[ 11%] Building CXX object MathFunctions/CMakeFiles/MakeTable.dir/MakeTable.cxx.o
[ 22%] Linking CXX executable MakeTable
[ 22%] Built target MakeTable
[ 33%] Generating Table.h <--------
[ 44%] Building CXX object MathFunctions/CMakeFiles/SqrtLibrary.dir/mysqrt.cxx.o
[ 55%] Linking CXX static library libSqrtLibrary.a
[ 55%] Built target SqrtLibrary
[...]
$ ./Tutorial 5
Computing sqrt of 5 to be 3   <------ 초기값은 테이블에 참조한 결과
Computing sqrt of 5 to be 2.33333
Computing sqrt of 5 to be 2.2381
Computing sqrt of 5 to be 2.23607
[...]
The square root of 5 is 2.23607

$  cat MathFunctions/Table.h
double sqrtTable[] = {
0,1,1.41421,1.73205,2,2.23607,2.44949,2.64575,2.82843,3,0};
```

2.9. Step9: 패키징 및 인스톨러

- 본 절에서는 프로젝트를 배포하는 방법에 대해서 알아본다. 다양한 플랫폼에서 바이너리와 소스 배포를 모두 제공하는 것이 목표이다. 설치 패키지 빌드를 위해서 CPack[28]을 사용하여 플랫폼별 설치 프로그램을 만든다.

27) https://cmake.org/cmake/help/v3.0/command/add_custom_command.html
28) https://cmake.org/cmake/help/latest/manual/cpack.1.html

```
$ vi Step9/CMakeLists.txt
[...]

# setup installer
include(InstallRequiredSystemLibraries)
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set(CPACK_PACKAGE_VERSION_MAJOR "${Tutorial_VERSION_MAJOR}")
set(CPACK_PACKAGE_VERSION_MINOR "${Tutorial_VERSION_MINOR}")
set(CPACK_GENERATOR "TGZ")
set(CPACK_SOURCE_GENERATOR "TGZ")
include(CPack)
```

```
$ rm -rf Build ; cmake -B Build  -S Step9  -DUSE_MYMATH=ON ; cmake --build Build
$ cd Build ; cpack
CPack: Create package using TGZ
CPack: Install projects
CPack: - Run preinstall target for: Tutorial
CPack: - Install project: Tutorial []
CPack: Create package                    .tar.gz 패키지 파일이 만들어짐
CPack: - package: /home/ubuntu/linux-drill/cmake/Build/Tutorial-1.0-Linux.tar.gz generated.
  TAR 파일에 헤더파일, 바이너리, 라이브러리가 포함됨
$ tar ztf Tutorial-1.0-Linux.tar.gz
Tutorial-1.0-Linux/include/
Tutorial-1.0-Linux/include/MathFunctions.h
Tutorial-1.0-Linux/include/TutorialConfig.h
Tutorial-1.0-Linux/bin/
Tutorial-1.0-Linux/bin/Tutorial
Tutorial-1.0-Linux/lib/
Tutorial-1.0-Linux/lib/libMathFunctions.a
Tutorial-1.0-Linux/lib/libSqrtLibrary.a
   소스 패키지 생성을 위해 CPackSourceConfig.cmake 파일이 만들어진다.
$ more CPackSourceConfig.cmake
[...]
set(CPACK_RPM_PACKAGE_SOURCES "ON")
set(CPACK_SOURCE_PACKAGE_FILE_NAME "Tutorial-1.0-Source")
set(CPACK_SOURCE_TOPLEVEL_TAG "Linux-Source")
set(CPACK_TOPLEVEL_TAG "Linux-Source")
[...]
   소스 패키지 생성. -Source.tar.gz 파일이 생성됨
$ cpack --config CPackSourceConfig.cmake
CPack: Create package using TGZ
CPack: Install projects
CPack: - Install directory: /home/ubuntu/linux-drill/cmake/Step9
CPack: Create package
CPack: - package: /home/ubuntu/linux-drill/cmake/Step9/build/Tutorial-1.0-Source.tar.gz generated.
$ tar ztf Tutorial-1.0-Source.tar.gz
Tutorial-1.0-Source/CTestConfig.cmake
Tutorial-1.0-Source/TutorialConfig.h.in
Tutorial-1.0-Source/MathFunctions/
Tutorial-1.0-Source/MathFunctions/MathFunctions.cxx
Tutorial-1.0-Source/MathFunctions/MathFunctions.h
Tutorial-1.0-Source/MathFunctions/CMakeLists.txt
[...]
```

```
$ cpack -G ZIP -C Debug
```

```
CPack: Create package using ZIP
CPack: Install projects
CPack: - Run preinstall target for: Tutorial
CPack: - Install project: Tutorial [Debug]
CPack: Create package
CPack: - package: /home/ubuntu/linux-drill/cmake/Step9/build/Tutorial-1.0-Linux.zip generated.

$ unzip -l Tutorial-1.0-Linux.zip
Archive:  Tutorial-1.0-Linux.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2024-09-30 09:16   Tutorial-1.0-Linux/include/
       65  2024-09-28 01:34   Tutorial-1.0-Linux/include/MathFunctions.h
      118  2024-09-30 09:14   Tutorial-1.0-Linux/include/TutorialConfig.h
        0  2024-09-30 09:16   Tutorial-1.0-Linux/bin/
    25064  2024-09-30 09:14   Tutorial-1.0-Linux/bin/Tutorial
        0  2024-09-30 09:16   Tutorial-1.0-Linux/lib/
     1660  2024-09-30 09:14   Tutorial-1.0-Linux/lib/libMathFunctions.a
     3258  2024-09-30 09:14   Tutorial-1.0-Linux/lib/libSqrtLibrary.a
---------                     -------
    30165                     8 files
```

2.10. Step10: 정적 및 동적 라이브러리 선택

- add_library() 명령에서 BUILD_SHARED_LIBS 변수[29]는 라이브러리의 빌드 방식을 결정할 수 있다.

```
$ vi Step10/CMakeLists.txt
[...]

# control where the static and shared libraries are built so that on windows
# we don't need to tinker with the path to run the executable
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")

  # shared libs 옵션을 MathFunctions 에서 참조하기때문에 라이브러리를 추가하기 전에 option 문을
넣어준다.
option(BUILD_SHARED_LIBS "Build using shared libraries" ON)
add_subdirectory(MathFunctions)

$ vi  Step10/MathFunctions/MathFunctions.h
  윈도우 플랫폼에서 dll 라이브러리 지원을 위해서 헤더 파일을 수정
#if defined(_WIN32)
#  if defined(EXPORTING_MYMATH)
#    define DECLSPEC __declspec(dllexport)
#  else
#    define DECLSPEC __declspec(dllimport)
#  endif
#else    // windows 플랫폼이 아닌 경우는 영향 없음
#  define DECLSPEC
#endif
```

29) https://cmake.org/cmake/help/latest/variable/BUILD_SHARED_LIBS.html

```
namespace mathfunctions {
double DECLSPEC sqrt(double x);
}
```

- POSITION INDEPENDENT CODE [30]속성을 SqrtLibrary 에 부여한다.

```
$ vi Step10/MathFunctions/CMakeLists.txt
[...]

if(USE_MYMATH)

  # state that SqrtLibrary need PIC when the default is shared libraries
  set_target_properties(SqrtLibrary PROPERTIES
                        POSITION_INDEPENDENT_CODE ${BUILD_SHARED_LIBS}
                        )
[...]
endif()
```

- Windows 에서 빌드할때 EXPORTING_MYMATH 매크로를 처리해 줌

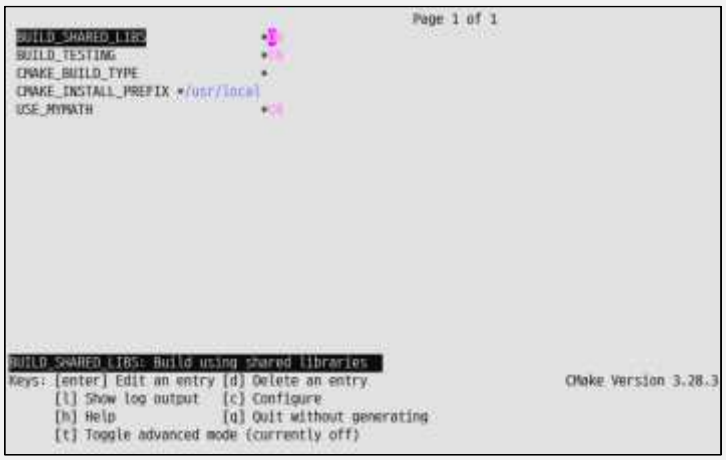
```
# link MathFunctions to tutorial_compiler_flags
target_link_libraries(MathFunctions PUBLIC tutorial_compiler_flags)

# define the symbol stating we are using the declspec(dllexport) when
# building on windows
target_compile_definitions(MathFunctions PRIVATE "EXPORTING_MYMATH")
```

```
$ rm -rf Build ; cmake -B Build  -S Step10  -DUSE_MYMATH=ON ; cmake --build Build
[...]
[ 44%] Building CXX object MathFunctions/CMakeFiles/SqrtLibrary.dir/mysqrt.cxx.o
[ 55%] Linking CXX static library libSqrtLibrary.a
[ 55%] Built target SqrtLibrary
[ 66%] Building CXX object MathFunctions/CMakeFiles/MathFunctions.dir/MathFunctions.cxx.o
[ 77%] Linking CXX shared library libMathFunctions.so  <----- shared 라이브러리
[ 77%] Built target MathFunctions
[ 88%] Building CXX object CMakeFiles/Tutorial.dir/tutorial.cxx.o
[100%] Linking CXX executable Tutorial
[100%] Built target Tutorial
$ ldd Build/Tutorial     # shared object 의존성을 조회
        linux-vdso.so.1 (0x00007fff0e2aa000)
        libMathFunctions.so => /home/ubuntu/linux-drill/cmake/Build/MathFunctions/libMathFunctions.so (0x
000077c9fc7c2000)    절대 경로를 참조함에 유의
        libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x000077c9fc400000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x000077c9fc78d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x000077c9fc000000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x000077c9fc6a4000)
        /lib64/ld-linux-x86-64.so.2 (0x000077c9fc7cf000)
```

※ ccmake는 커서 기반의 인터페이스를 제공하는 CMake 도구이다. 사용법: 't' 키로 고급 옵션 표시/숨김. Enter키로 불리언 옵션 ON/OFF 전환. 'c' 키로 변경사항을 적용. 'g'키로 설정을 저장하고 Makefile을 생성. '/'키로 옵션 검색 모드 진입. 'q'키로 ccmake 종료

30) https://cmake.org/cmake/help/latest/prop_tgt/POSITION_INDEPENDENT_CODE.html

```
$ sudo apt  install cmake-curses-gui
$ ccmake Build
```



2.11. Step11: 내보내기 구성 추가

- 다음 단계는 다른 CMake 프로젝트에서 빌드 디렉터리, 로컬 설치 또는 패키징시 프로젝트를 사용할 수 있도록 정보를 추가하는것. install()명령에 EXPORT 구문을 추가함. 이것은 해당 타겟을 import 하기 위한 CMake 코드를 생성한다.

```
$ vi Step11/MathFunctions/CMakeLists.txt
[...]
# install libs
set(installable_libs MathFunctions tutorial_compiler_flags)
if (TARGET SqrtLibrary)
  list(APPEND  installable_libs SqrtLibrary)
endif()
install(TARGETS ${installable_libs}
        EXPORT MathFunctionsTargets    # MathFunctionsTargets.cmake 가 생성됨
        DESTINATION lib)
# install include headers
install(FILES MathFunctions.h DESTINATION include)
```

```
$ vi  Step11/CMakeLists.txt
[...]
# install the configuration targets
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions   # 이 폴더에 .cmake 파일이 생성됨
)
```

```
$ vi Step11/MathFunctions/CMakeLists.txt
[...]
target_include_directories(MathFunctions
                           INTERFACE
                            $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>
                            $<INSTALL_INTERFACE:include>
                           )
```

```
$ vi  Step11/Config.cmake.in
@PACKAGE_INIT@

include ( "${CMAKE_CURRENT_LIST_DIR}/MathFunctionsTargets.cmake" )
```

```
$ vi  Step11/CMakeLists.txt
```

```
[...]
include(CMakePackageConfigHelpers)
$ rm -rf Build ; cmake -B Build  -S Step11  -DUSE_MYMATH=ON ; cmake --build Build
$ cmake --install Build --prefix "/home/ubuntu/installdir"
/home/ubuntu/installdir
├── bin
│   └── Tutorial
├── include
│   ├── MathFunctions.h
│   └── TutorialConfig.h
└── lib            lib/cmake/MathFunctions 디렉터리
    ├── cmake
    │   └── MathFunctions
    │       ├── MathFunctionsTargets-noconfig.cmake
    │       └── MathFunctionsTargets.cmake
    ├── libMathFunctions.so
    └── libSqrtLibrary.a
```

2.12. Step12: 디버그 및 릴리즈 패키징

- 디버그 및 릴리즈 빌드가 설치될 라이브러리에 대해 다른 이름을 사용하도록 한다. 디버그 라이브러리에 대한 접미사로 dbg 를 사용함.

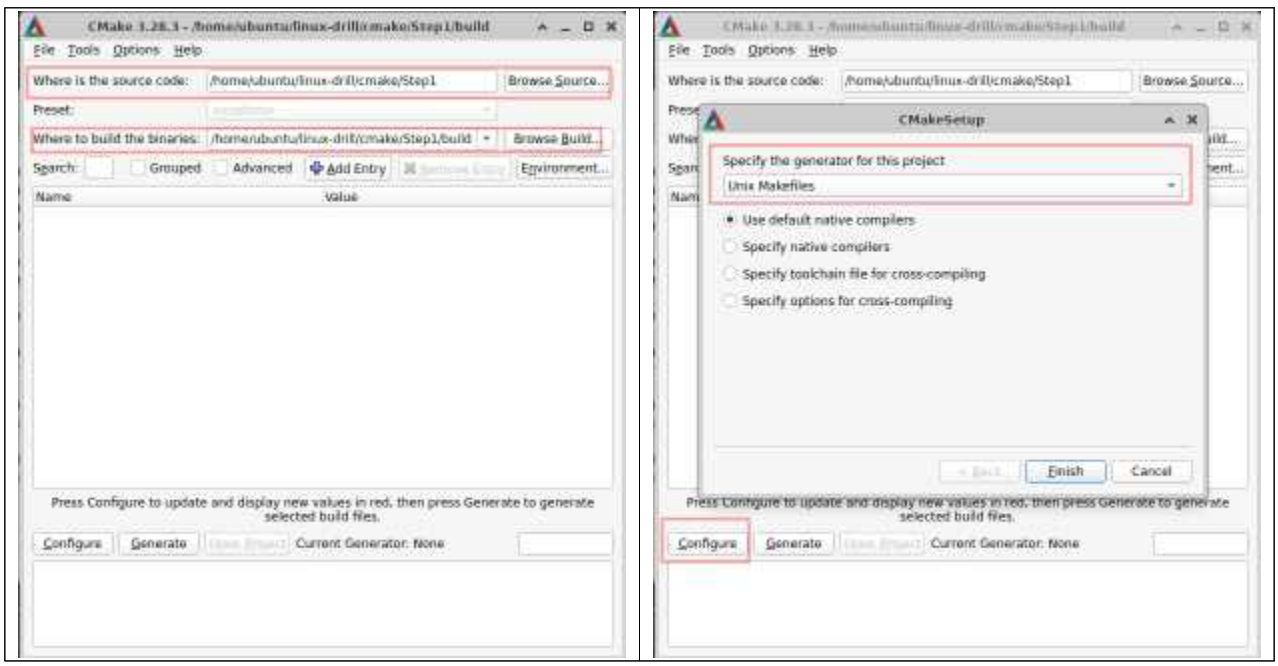
```
$ vi Step12/CMakeLists.txt
[...]
# step12
set(CMAKE_DEBUG_POSTFIX dbg)
set_target_properties(Tutorial PROPERTIES DEBUG_POSTFIX ${CMAKE_DEBUG_POSTFIX})
```

```
$ vi Step12/MathFunctions/CMakeLists.txt

# step12
set_property(TARGET MathFunctions PROPERTY VERSION "1.0.0")
set_property(TARGET MathFunctions PROPERTY SOVERSION "1")
```

```
$ vi Step12/MultiCPackConfig.cmake
include("release/CPackConfig.cmake")

set(CPACK_INSTALL_CMAKE_PROJECTS
    "debug;Tutorial;ALL;/"
    "release;Tutorial;ALL;/"
    )
$ cd Step12
$ rm -rf debug release ; mkdir debug release
$ cd debug
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ cmake --build .
$ cd ../release
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ cmake --build .
$ cd ..
$ cpack --config MultiCPackConfig.cmake
```

```
$ tar ztf Tutorial-1.0-Linux.tar.gz
Tutorial-1.0-Linux/include/
Tutorial-1.0-Linux/include/MathFunctions.h
Tutorial-1.0-Linux/include/TutorialConfig.h
Tutorial-1.0-Linux/bin/
Tutorial-1.0-Linux/bin/Tutorial  <--------------- Release 바이너리
Tutorial-1.0-Linux/bin/Tutorialdbg  <--------------- Debug 바이너리
Tutorial-1.0-Linux/lib/
Tutorial-1.0-Linux/lib/cmake/
Tutorial-1.0-Linux/lib/cmake/MathFunctions/
Tutorial-1.0-Linux/lib/cmake/MathFunctions/MathFunctionsTargets-debug.cmake
Tutorial-1.0-Linux/lib/cmake/MathFunctions/MathFunctionsTargets-release.cmake
Tutorial-1.0-Linux/lib/cmake/MathFunctions/MathFunctionsTargets.cmake
Tutorial-1.0-Linux/lib/libMathFunctions.so    <--------------- Release 동적라이브러리
Tutorial-1.0-Linux/lib/libMathFunctionsdbg.so <--------------- Debug 동적라이브러리
Tutorial-1.0-Linux/lib/libMathFunctions.so.1.0.0
Tutorial-1.0-Linux/lib/libMathFunctionsdbg.so.1
Tutorial-1.0-Linux/lib/libSqrtLibrary.a     <--------------- Release 정적라이브러리
Tutorial-1.0-Linux/lib/libMathFunctionsdbg.so.1.0.0
Tutorial-1.0-Linux/lib/libMathFunctions.so.1
Tutorial-1.0-Linux/lib/libSqrtLibrarydbg.a  <--------------- Debug 정적라이브러리
```

※ CMake GUI는 CMake 프로젝트를 구성하기 위한 그래픽 사용자 인터페이스 임. Configure 버튼에서 프로젝트 옵션을 설정할 수 있음. Generate 버튼을 사용하여 빌드 파일(Makefile) 생성. 캐시 항목 탐색 및 편집 기능 제공,



3. CMake 가이드

3.1. 실행파일 가져오기

- 가져온(INPORTED) 타겟은 프로젝트 외부의 파일을 프로젝트 내부의 논리적 타겟으로 가져와 사용하는데 사용됨. 가져온 타겟은 add_executable() 와 add_library() 명령에서 IMPORTED 옵션을 사용하여 생성됨. 가져온 타겟은 빌드 규칙을 생성하지 않으며, 이미 존재하는 바이너리를 참조하는 용도로만 사용됨. 'IMPORTED_' 접두사로 시작하는 속성 (IMPORTED_LOCATION, IMPORTED_IMPLIB, IMPORTED_CONFIGURATIONS, IMPORTED_LOCATION_<CONFIG>) 들을 사용하여 타겟의 세부 정보를 지정함.
- MyExe라는 프로젝트를 만들고, 바이너리 파일을 지정된 위치에 설치한다.

```
$ cd MyExe
$ more main.cxx
#include <fstream>
#include <iostream>
int main(int argc, char* argv[]) {
  std::ofstream outfile("main.cc");  // main.cc를 생성하고, 간단한 프로그램 소스를 출력함
  outfile << "#include <iostream>" << std::endl;
  outfile << "int main(int argc, char* argv[])" << std::endl;
  outfile << "{" << std::endl;
  outfile << "  // Your code here" << std::endl;
  outfile << "  std::cout << \"Hello World\" << std::endl; " << std::endl;
  outfile << "  return 0;" << std::endl;
  outfile << "}" << std::endl;
  outfile.close();
  return 0;
}
```

```
$ more CMakeLists.txt
cmake_minimum_required(VERSION 3.15)
project(MyExe)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
add_executable(myexe main.cxx)    # myexe 바이너리를 만들고 설치한다.
install(TARGETS myexe)
```

```
$ cd MyExe
$ rm -rf build ; mkdir build ; cd build ; cmake .. ; cmake --build .
$ rm -rf $HOME/install_myexe
$ cmake --install . --prefix $HOME/install_myexe
-- Install configuration: ""
-- Installing: /home/ubuntu/install_myexe/bin/myexe
$ tree /home/ubuntu/install_myexe
/home/ubuntu/install_myexe
└── bin
    └── myexe
```

- 위 바이너리 파일을 이용한 프로젝트.

```
$ cd ../Importing/
$ export myexeloc=/home/ubuntu/install_myexe    # 설치 경로를 환경변수에 설정
$ more CMakeLists.txt
[...]
# Add executable
add_executable(myexe  IMPORTED)    # 가져온 타겟  IMPORTED_LOCATION 에 경로를 명시함
set_property(TARGET myexe PROPERTY IMPORTED_LOCATION "$ENV{myexeloc}/bin/myexe")
```

```
# 커스텀 커맨드를 실행하여 main.cc 소스 코드를 생성함
add_custom_command(OUTPUT main.cc COMMAND myexe)
# Use source file
add_executable(mynewexe main.cc)     # 컴파일하여 mynewexe를 만든다
$ rm -rf build ; mkdir build ; cd build
$ cmake .. --fresh ; cmake --build .
$ ./mynewexe
Hello World
```

3.2. 라이브러리 가져오기

- MathFunctions 라이브러리 만들기. sqrt() 함수는 제곱근을 계산한다.

```
$ cd MathFunctions
$ more MathFunctions.cxx
#include "MathFunctions.h"
#include <cmath>
namespace MathFunctions {
double sqrt(double x) {    // MathFunctions::sqrt
  return std::sqrt(x);
}
}
$ more CMakeLists.txt
[...]
# add include directories
target_include_directories(MathFunctions
     PUBLIC
     "$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>"
     "$<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>"
)

# install the target and create export-set
install(TARGETS MathFunctions
        EXPORT MathFunctionsTargets
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
        INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)

# install header file
install(FILES MathFunctions.h DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})

# generate and install export file
install(EXPORT MathFunctionsTargets
        FILE MathFunctionsTargets.cmake
        NAMESPACE MathFunctions::
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MathFunctions
)

# include CMakePackageConfigHelpers macro
include(CMakePackageConfigHelpers)
```

```
set(version 3.4.1)   # 라이브러리의 버전을 설정
set_property(TARGET MathFunctions PROPERTY VERSION ${version})
set_property(TARGET MathFunctions PROPERTY SOVERSION 3)
set_property(TARGET MathFunctions PROPERTY
  INTERFACE_MathFunctions_MAJOR_VERSION 3)
set_property(TARGET MathFunctions APPEND PROPERTY
  COMPATIBLE_INTERFACE_STRING MathFunctions_MAJOR_VERSION
)

# generate the version file for the config file
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${version}"
  COMPATIBILITY AnyNewerVersion
)
# create config file
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MathFunctions
)
# install config files
install(FILES
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MathFunctions
)
# generate the export targets for the build tree
export(EXPORT MathFunctionsTargets
       FILE "${CMAKE_CURRENT_BINARY_DIR}/cmake/MathFunctionsTargets.cmake"
       NAMESPACE MathFunctions::
)
```

```
$ more Config.cmake.in
@PACKAGE_INIT@       ==> build/MathFunctionsConfig.cmake 와 비교해 볼 것

include("${CMAKE_CURRENT_LIST_DIR}/MathFunctionsTargets.cmake")

check_required_components(MathFunctions)
```

```
$ mkdir build ; cd build
$ cmake ..
$ cmake --build .       # 프로젝트를 빌드하고, 해당 경로에 설치한다
$ cmake --install . --prefix=$HOME/math
-- Install configuration: ""
-- Installing: /home/ubuntu/math/lib/libMathFunctions.a
-- Installing: /home/ubuntu/math/include/MathFunctions.h
-- Installing: /home/ubuntu/math/lib/cmake/MathFunctions/MathFunctionsTargets.cmake
-- Installing: /home/ubuntu/math/lib/cmake/MathFunctions/MathFunctionsTargets-noconfig.cmake
-- Installing: /home/ubuntu/math/lib/cmake/MathFunctions/MathFunctionsConfig.cmake
-- Installing: /home/ubuntu/math/lib/cmake/MathFunctions/MathFunctionsConfigVersion.cmake
$ tree $HOME/math
/home/ubuntu/math
├── include
│   └── MathFunctions.h
└── lib
    ├── cmake
    │   └── MathFunctions
    │       ├── MathFunctionsConfig.cmake
```

```
    │       ├── MathFunctionsConfigVersion.cmake
    │       ├── MathFunctionsTargets-noconfig.cmake
    │       └── MathFunctionsTargets.cmake
    └── libMathFunctions.a
```

- 위 라이브러리를 이용하는 프로그램을 작성

```
$ cd Downstream/
$  more main.cc
// A simple program that outputs the square root of a number
#include <iostream>
#include <string>
#include "MathFunctions.h"    // 라이브러리 헤더파일
int main(int argc, char* argv[]) {
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }
  // convert input to double
  const double inputValue = std::stod(argv[1]);
  // calculate square root
  const double sqrt = MathFunctions::sqrt(inputValue);  // 라이브러리 내부 함수
  std::cout << "The square root of " << inputValue << " is " << sqrt
            << std::endl;
  return 0;
}
$ more CMakeLists.txt
cmake_minimum_required(VERSION 3.15)
project(Downstream)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
find_package(MathFunctions 3.4.1 EXACT)    # 라이브러리 패키지를 검색 find_package()[31]
message(STATUS "MathFunctions_FOUND ==> ${MathFunctions_FOUND}")
message(STATUS "MathFunctions_DIR ==> ${MathFunctions_DIR}")
message(STATUS "MathFunctions_VERSION ==> ${MathFunctions_VERSION}")
message(STATUS "MathFunctions_CONFIG ==> ${MathFunctions_CONFIG}")
message(STATUS "MathFunctions_CONSIDERED_CONFIG ==> ${MathFunctions_CONSIDERED_CONFIG}")
add_executable(myexe main.cc)
  # 라이브러리를 바이너리에 링크 시킨다
target_link_libraries(myexe PRIVATE  MathFunctions::MathFunctions)
$ mkdir build ; cd build
$ export CMAKE_PREFIX_PATH=/home/ubuntu/math   #  이 경로의 lib/cmake/<lib> 폴더에서 찾는다
$ cmake ..
[...]
-- MathFunctions_FOUND ==> 1
-- MathFunctions_DIR ==> /home/ubuntu/math/lib/cmake/MathFunctions
-- MathFunctions_VERSION ==> 3.4.1
-- MathFunctions_CONFIG ==> /home/ubuntu/math/lib/cmake/MathFunctions/MathFunctionsConfig.cmake
-- MathFunctions_CONSIDERED_CONFIG ==>
[...]
$ cmake --build . -v
$ ./myexe
Usage: ./myexe number
$ ./myexe 100
The square root of 100 is 10
```

3.3. 패키지 생성 및 가져오기

- 패키지 내부에서 상대적인 경로를 갖도록 라이브러리를 배치할 수 있다. MathFunctions 라는 패키지 내부에 구성요소로 Addition과 SquareRoot를 포함한다.

```
$ cd  MathFunctionsComponents
$ more Addition/Addition.cxx
#include "Addition.h"
namespace MathFunctions {   // MathFunctions::add
double add(double x, double y) {
  return x + y;
}
}
$ more SquareRoot/SquareRoot.cxx
#include "SquareRoot.h"
#include <cmath>
namespace MathFunctions {     // MathFunctions::sqrt
double sqrt(double x) {
  return std::sqrt(x);
}
}
$ more CMakeLists.txt
[...]
# make cache variables for install destinations
include(GNUInstallDirs)    # 이 모듈에서 CMAKE_INSTALL_PREFIX 변수를 참고함
[...]
add_subdirectory(Addition)    # 하위 라이브러리들
add_subdirectory(SquareRoot)

# include CMakePackageConfigHelpers macro
include(CMakePackageConfigHelpers)  # config 파일을 생성하기 위한 헬퍼 모듈
set(version 3.4.1)
# generate the version file for the config file
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${version}"
  COMPATIBILITY AnyNewerVersion
)

# create config file
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MathFunctions
  NO_CHECK_REQUIRED_COMPONENTS_MACRO
)
# install config files
install(FILES
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
          "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MathFunctions
)
$ more Addition/CMakeLists.txt
# create library
```

31) https://cmake.org/cmake/help/latest/command/find_package.html

```
add_library(Addition STATIC Addition.cxx)
add_library(MathFunctions::Addition ALIAS Addition)

# add include directories
target_include_directories(Addition
                           PUBLIC
                           "$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>"
                           $<INSTALL_INTERFACE:${CMAKE_INSTALL_INCLUDEDIR}>
)
 # install 관련 설정
# install the target and create export-set
install(TARGETS Addition
        EXPORT AdditionTargets
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
        INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
# install header file
install(FILES Addition.h DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})
# generate and install export file
install(EXPORT AdditionTargets
        FILE MathFunctionsAdditionTargets.cmake
        NAMESPACE MathFunctions::
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/MathFunctions
)
```

```
$ more Config.cmake.in
@PACKAGE_INIT@
set(_MathFunctions_supported_components Addition SquareRoot)
foreach(_comp ${MathFunctions_FIND_COMPONENTS})
  if (NOT _comp IN_LIST _MathFunctions_supported_components)
    set(MathFunctions_FOUND False)
    set(MathFunctions_NOT_FOUND_MESSAGE "Unsupported component: ${_comp}")
  endif()
  include("${CMAKE_CURRENT_LIST_DIR}/MathFunctions${_comp}Targets.cmake")
endforeach()
```

```
$ mkdir build ; cd build ; cmake ..
$ cmake --build .
$ cmake --install . --prefix=$HOME/mathcomp
-- Install configuration: ""
-- Installing: /home/ubuntu/mathcomp/lib/libAddition.a
-- Installing: /home/ubuntu/mathcomp/include/Addition.h
-- Installing: /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsAdditionTargets.cmake
-- Installing: /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsAdditionTargets-noconfig.cmake
-- Installing: /home/ubuntu/mathcomp/lib/libSquareRoot.a
-- Installing: /home/ubuntu/mathcomp/include/SquareRoot.h
-- Installing: /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsSquareRootTargets.cmake
-- Installing: /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsSquareRootTargets-noconfig.cmake
-- Installing: /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsConfig.cmake
-- Installing: /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsConfigVersion.cmake
$ tree $HOME/mathcomp
/home/ubuntu/mathcomp
├── include  <---------- 헤더파일
│   ├── Addition.h
│   └── SquareRoot.h
└── lib
```

```
    ├── cmake
    │   └── MathFunctions
    │       ├── MathFunctionsAdditionTargets-noconfig.cmake
    │       ├── MathFunctionsAdditionTargets.cmake
    │       ├── MathFunctionsConfig.cmake
    │       ├── MathFunctionsConfigVersion.cmake
    │       ├── MathFunctionsSquareRootTargets-noconfig.cmake
    │       └── MathFunctionsSquareRootTargets.cmake
    ├── libAddition.a     <------- Addition.cxx.o
    └── libSquareRoot.a   <------- SquareRoot.cxx.o
```

- 위에서 만들어 설치되어 있는 패키지를 이용하는 프로그램

```
$ cd  DownstreamComponents/
$ more main.cc
// A simple program that outputs the square root of a number
#include <iostream>
#include <string>
#include "Addition.h"  # 라이브러리 헤더파일
#include "SquareRoot.h"
int main(int argc, char* argv[]) {
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
  }
  // convert input to double
  const double inputValue = std::stod(argv[1]);
  // calculate square root
  const double sqrt = MathFunctions::sqrt(inputValue);  # 라이브러리 호출
  std::cout << "The square root of " << inputValue << " is " << sqrt
            << std::endl;
  // calculate sum
  const double sum = MathFunctions::add(inputValue, inputValue);  # 라이브러리 호출
  std::cout << inputValue << " + " << inputValue << " = " << sum << std::endl;
  return 0;
}
$ more CMakeLists.txt
[...]
# find MathFunctions
find_package(MathFunctions 3.4 COMPONENTS Addition SquareRoot)
# create executable
add_executable(myexe main.cc)
# use MathFunctions library
target_link_libraries(myexe PRIVATE  MathFunctions::Addition  MathFunctions::SquareRoot)
# Workaround for GCC on AIX to avoid -isystem, not needed in general.
set_property(TARGET myexe PROPERTY NO_SYSTEM_FROM_IMPORTED 1)
$ mkdir build ; cd build
$ export CMAKE_PREFIX_PATH=/home/ubuntu/mathcomp
$ cmake ..
[...]
-- MathFunctions_FOUND ==> 1
-- MathFunctions_DIR ==> /home/ubuntu/mathcomp/lib/cmake/MathFunctions
-- MathFunctions_VERSION ==> 3.4.1
-- MathFunctions_CONFIG ==> /home/ubuntu/mathcomp/lib/cmake/MathFunctions/MathFunctionsConfig.cmake
-- MathFunctions_CONSIDERED_CONFIG ==>
-- _MathFunctions_supported_components ==> Addition;SquareRoot
```

```
[...]
$ cmake --build .
$ ./myexe 100
The square root of 100 is 10
100 + 100 = 200
```

4. CMake 활용

4.1. GoogleTest

- 구글은 2005년 구글 웹서버(GWS)의 복잡성이 증가하면서 생산성이 급격히 떨어지는 경험을 하고, 이를 계기로 엔지니어 주도의 자동 테스트를 정책적으로 도입함. C++ 코드 테스트를 위해 Googletest [32]라는 자체 프레임워크를 개발함: Googletest의 주요 특징은 다음과 같다: ① 독립적이고 반복 가능한 테스트 실행 ② 테스트 스위트를 통한 관련 테스트 그룹화 ③ 플랫폼 중립적이고 재상용 가능한 테스트 코드 ④ 실패시 상세한 정보 제공 ⑤ 자동화된 테스트 추적 및 실행
- Google Test 설치하기

```
$ wget https://github.com/google/googletest/releases/download/v1.15.2/googletest-1.15.2.tar.gz
$ tar zxvf googletest-1.15.2.tar.gz
$ cd googletest-1.15.2
$ mkdir build ; cd build
$ cmake ..
$ cmake --build .
$ sudo cmake --install .
$ ls /usr/local/lib/libgtest* -a
/usr/local/lib/libgtest.a  /usr/local/lib/libgtest_main.a
$ ls /usr/local/include/gtest/
gtest-assertion-result.h  gtest-message.h     gtest-spi.h         gtest.h            internal
gtest-death-test.h        gtest-param-test.h  gtest-test-part.h   gtest_pred_impl.h
gtest-matchers.h          gtest-printers.h    gtest-typed-test.h  gtest_prod.h
$ ls /usr/local/include/gmock/
gmock-actions.h          gmock-matchers.h       gmock-nice-strict.h    internal
gmock-cardinalities.h    gmock-more-actions.h   gmock-spec-builders.h
gmock-function-mocker.h  gmock-more-matchers.h  gmock.h
```

- 간단한 함수를 만들고, 이를 테스트하기 위한 코드

```
$ cd gtest
// sum.h
int sum(int a, int b);
```

```
// sum.c
#include "sum.h"
int sum(int a, int b) {
  return 0;  // 이 코드에는 오류가 있음  ==> return a+b
}
```

```
// main.c
#include <stdio.h>
#include "sum.h"
int main() {
  printf("sum : %d\n", sum(1, 1));
  return 0;
}
```

```
// sum_test.c
#include <stdio.h>
#include <gtest/gtest.h>   # googletest 헤더파일
#include "sum.h"
//  TEST(TestSuiteName, TestName) {
//    ... test body ...
//  }
```

32) https://github.com/google/googletest

```
TEST(sum_test1, test1) {
  EXPECT_EQ(2, sum(1,1));
  EXPECT_EQ(0, sum(1,-1));
}
TEST(sum_test2, test2) {
  EXPECT_EQ(100, sum(50,50));
}
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv); # 초기화후 RUN_ALL_TESTS()를 호출한다
  return RUN_ALL_TESTS();
}
```

```
$ more Makefile
all:
    g++ -o main main.c sum.c
test:
    g++ -o sum_test sum_test.c sum.c -isystem /usr/local/include -L/usr/local/lib -pthread -lgt
est
    ./sum_test
```

```
$ make test
[...]
[==========] Running 2 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from sum_test1
[ RUN      ] sum_test1.test1
sum_test.c:10: Failure
Expected equality of these values:
  2
  sum(1,1)
    Which is: 1

sum_test.c:11: Failure
Expected equality of these values:
  0
  sum(1,-1)
    Which is: -1

[  FAILED  ] sum_test1.test1 (0 ms)   <-------- 첫번째 그룹
[----------] 1 test from sum_test1 (0 ms total)

[----------] 1 test from sum_test2
[ RUN      ] sum_test2.test2
sum_test.c:14: Failure
Expected equality of these values:
  100
  sum(50,50)
    Which is: 2500

[  FAILED  ] sum_test2.test2 (0 ms)  <-------- 두번째 그룹
[----------] 1 test from sum_test2 (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] sum_test1.test1
[  FAILED  ] sum_test2.test2

 2 FAILED TESTS
```

```
make: *** [Makefile:5: test] Error 1

  # sum.c 에서 오류를 수정하고 다시 테스트함
$ make test
g++ -o sum_test sum_test.c sum.c -isystem /usr/local/include -L/usr/local/lib -pthread -lgtest
./sum_test
[==========] Running 2 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 1 test from sum_test1
[ RUN      ] sum_test1.test1
[       OK ] sum_test1.test1 (0 ms)   <------ 첫번째 그룹의 모든 테스트가 성공
[----------] 1 test from sum_test1 (0 ms total)

[----------] 1 test from sum_test2
[ RUN      ] sum_test2.test2
[       OK ] sum_test2.test2 (0 ms)
[----------] 1 test from sum_test2 (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 2 tests.
```

- Googletest Samples[33] 예제 참조

```
$ git clone https://github.com/google/googletest.git
$ cd  googletest
$ more googletest/samples/sample1.cc
#include "sample1.h"
// Returns n! (the factorial of n).  For negative n, n! is defined to be 1.
int Factorial(int n) {   // n이 음수이면 1을 리턴함
  int result = 1;
  for (int i = 1; i <= n; i++) {
    result *= i;
  }
  return result;
}
// Returns true if and only if n is a prime number.
bool IsPrime(int n) {
  // Trivial case 1: small numbers   1보다 작으면 소수가 아님
  if (n <= 1) return false;
  // Trivial case 2: even numbers    짝수이면 n=2일때만 소수이고, 그이외에는 아님
  if (n % 2 == 0) return n == 2;
  // Now, we have that n is odd and n >= 3.
  // Try to divide n by every odd number i, starting from 3
  for (int i = 3;; i += 2) {
    // We only have to try i up to the square root of n
    if (i > n / i) break;
    // Now, we have i <= n/i < n.
    // If n is divisible by i, n is not prime.
    if (n % i == 0) return false;   // 나누어 지면 prime number가 아님
  }
  // n has no integer factor in the range (1, n), and thus is prime.
  return true;
}
$ vi googletest/samples/sample1_unittest.cc
```

33) https://google.github.io/googletest/samples.html

```
#include "sample1.h"
#include <limits.h>
#include "gtest/gtest.h"
namespace {
// Tests factorial of negative numbers.
TEST(FactorialTest, Negative) {  // 음수에 대해서 테스트
  EXPECT_EQ(1, Factorial(-5));   // EQ : equal
  EXPECT_EQ(1, Factorial(-1));
  EXPECT_GT(Factorial(-10), 0);  // GT: greater than
}
// Tests factorial of 0.
TEST(FactorialTest, Zero) { EXPECT_EQ(1, Factorial(0)); }
// Tests factorial of positive numbers.
TEST(FactorialTest, Positive) {
  EXPECT_EQ(1, Factorial(1));
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
  EXPECT_EQ(40320, Factorial(8));
}
// Tests negative input.
TEST(IsPrimeTest, Negative) {
  // This test belongs to the IsPrimeTest test case.
  EXPECT_FALSE(IsPrime(-1));  // FALSE : 거짓 Prime(-1) 은 false를 리턴
  EXPECT_FALSE(IsPrime(-2));
  EXPECT_FALSE(IsPrime(INT_MIN)); // INT_MIN는 음수 이므로 거짓
}
// Tests some trivial cases.
TEST(IsPrimeTest, Trivial) {
  EXPECT_FALSE(IsPrime(0));
  EXPECT_FALSE(IsPrime(1));
  EXPECT_TRUE(IsPrime(2));
  EXPECT_TRUE(IsPrime(3));
}
// Tests positive input.
TEST(IsPrimeTest, Positive) {
  EXPECT_FALSE(IsPrime(4)); // IsPrime(4) => false
  EXPECT_TRUE(IsPrime(5));  // IsPrime(5) => true
  EXPECT_FALSE(IsPrime(6));
  EXPECT_TRUE(IsPrime(23));
}
}  // namespace
   # googletest 를 빌드하고 테스트
$ cmake -B Build ; cmake --build Build
$ ./googletest/sample1_unittest
Running main() from /home/ubuntu/googletest/googletest/src/gtest_main.cc
[==========] Running 6 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 3 tests from FactorialTest
[ RUN      ] FactorialTest.Negative
[       OK ] FactorialTest.Negative (0 ms)
[ RUN      ] FactorialTest.Zero
[       OK ] FactorialTest.Zero (0 ms)
[ RUN      ] FactorialTest.Positive
[       OK ] FactorialTest.Positive (0 ms)
[----------] 3 tests from FactorialTest (0 ms total)
```

```
[----------] 3 tests from IsPrimeTest
[ RUN      ] IsPrimeTest.Negative
[       OK ] IsPrimeTest.Negative (0 ms)
[ RUN      ] IsPrimeTest.Trivial
[       OK ] IsPrimeTest.Trivial (0 ms)
[ RUN      ] IsPrimeTest.Positive
[       OK ] IsPrimeTest.Positive (0 ms)
[----------] 3 tests from IsPrimeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 2 test suites ran. (0 ms total)
[  PASSED  ] 6 tests.
```

- Googletest를 프로젝트에서 활용하기

```
$ mkdir my_project && cd my_project
$ wget https://github.com/google/googletest/archive/03597a01ee50ed33e9dfd640b249b4be3799d395.zip
$ more CMakeLists.txt
cmake_minimum_required(VERSION 3.14)
project(my_project)
# GoogleTest requires at least C++14
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
include(FetchContent)  # FetchContent 모듈을 이용하여 URL에서 가지고 옴
FetchContent_Declare(
  googletest
  URL "file://${PROJECT_SOURCE_DIR}/03597a01ee50ed33e9dfd640b249b4be3799d395.zip"
)
# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

include(CTest)
enable_testing()
include(GoogleTest)   # GoogleTest 모듈을 포함해줌
add_executable( sort_test sort_test.cc sort.cc )
target_link_libraries( sort_test  GTest::gtest_main )
gtest_discover_tests(sort_test DISCOVERY_MODE POST_BUILD)
// sort_test.cc         힙소트 알고리즘을 테스트
#include <gtest/gtest.h>
#include <iostream>
#include <random>
#include "sort.h"

TEST(TSuit1, Basic) {
  int data[] = {9,8,7,6,5,4,3,2,1,0};  // 정렬되어 있지 않은 상태
  int size = sizeof(data) / sizeof(data[0]);
  EXPECT_FALSE( is_sorted(data, size) ); // is_sorted()는 false를 기대함
  heapSort(data, size);                 // heapSort 를 실행
  EXPECT_TRUE( is_sorted(data, size) );  // is_sorted()는 true를 기대함
}
TEST(TSuit2, Random) {
  int data[1000];
  int size = sizeof(data) / sizeof(data[0]);
  initialize_vector(data, size) ;   // 1000 개의 데이터를 랜덤으로 초기화
```

```
  heapSort(data, size);  // heapSort 를 실행
  print_vec("random array", data, size);
  EXPECT_TRUE( is_sorted(data, size) );  // is_sorted()는 true를 기대함
}
```

```
$ rm -rf build ; cmake -S . -B build ; cd build ; cmake --build .
$ ctest
Test project /home/ubuntu/linux-drill/cmake/my_project/build
    Start 1: TSuit1.Basic
1/2 Test #1: TSuit1.Basic .....................   Passed    0.00 sec
    Start 2: TSuit2.Random
2/2 Test #2: TSuit2.Random .....................   Passed    0.08 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.09 sec
```

4.2. Boost 라이브러리

- Boost[34]는 C++ 프로그래밍 언어를 위한 무료 오픈 소스 라이브러리임. 오픈소스이며 무료로 사용, 수정, 배포가 가능. C++표준에 포함되었거나 영향을 미쳤음. 선형 대수, 멀티스레딩, 이미지 처리, 정규 표현식 등 164개 이상의 개별 라이브러리를 포함. Getting Started [35]를 참고한다.

- GCC 14를 설치하고 기본 빌드툴로 설정하기

```
$ sudo apt install gcc-14 g++-14
$ gcc-14 --version
gcc-14 (Ubuntu 14-20240412-0ubuntu1) 14.0.1 20240412 (experimental) [master r14-9935-g67e1433a94f]
[...]
  GCC 14를 기본 컴파일러로 설정하기
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-14 140 \
   --slave /usr/bin/g++ g++ /usr/bin/g++-14
$ sudo update-alternatives --config gcc
$ gcc --version
gcc (Ubuntu 14-20240412-0ubuntu1) 14.0.1 20240412 (experimental) [master r14-9935-g67e1433a94f]
```

- Boost 설치 하기

```
$ wget  https://archives.boost.io/release/1.86.0/source/boost_1_86_0.tar.gz  \
  && tar zxf boost_1_86_0.tar.gz  && cd boost_1_86_0
$ ./bootstrap.sh
Building B2 engine..
[...]
tools/build/src/engine/b2
Unicode/ICU support for Boost.Regex?... not found.
Generating B2 configuration in project-config.jam for gcc...

Bootstrapping is done. To build, run:
    ./b2
[...]
$ ./b2 --help   # 도움말
```

34) https://www.boost.org/
35) https://www.boost.org/doc/libs/1_86_0/more/getting_started/

```
$ ./b2          # 빌드하기
[...]
The Boost C++ Libraries were successfully built!
$ ./b2  --prefix=$HOME/boost   install    # 빌드하고 해당 위치에 설치
$ ls $HOME/boost/lib
cmake                        libboost_math_tr1l.a
libboost_atomic.a            libboost_math_tr1l.so
libboost_atomic.so           libboost_math_tr1l.so.1.86.0
libboost_atomic.so.1.86.0    libboost_nowide.a
libboost_charconv.a          libboost_nowide.so  [...]
```

- Boost 사용하기. Date_Time 라이브러리[36)]

```
$ more today.cpp           # boost::date_time 를 이용하여 날짜 계산을 하는 프로그램
#include <iostream>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
int main() {
  // 현재 날짜 가져오기
  boost::gregorian::date today = boost::gregorian::day_clock::local_day();
  // 기본 형식으로 출력
  std::cout << "오늘 날짜 (기본 형식): " << today << std::endl;
  // ISO 확장 형식으로 출력
  std::cout << "오늘 날짜 (ISO 확장 형식): " << boost::gregorian::to_iso_extended_string(today)
<< std::endl;
  // 간단한 형식으로 출력
  std::cout << "오늘 날짜 (간단한 형식): " << boost::gregorian::to_simple_string(today) << st
d::endl;
  // 사용자 정의 형식으로 출력
  boost::gregorian::date_facet* facet = new boost::gregorian::date_facet("%Y년 %m월 %d일");
  //std::cout.imbue(std::locale(std::cout.getloc(), facet));
  std::cout << "오늘 날짜 (사용자 정의 형식): " << today << std::endl;
  // 현재 날짜와 시간 출력
  boost::posix_time::ptime now = boost::posix_time::second_clock::local_time();
  std::cout << "현재 날짜와 시간: " << now << std::endl;
  return 0;
}
$ more  CMakeLists.txt
cmake_minimum_required(VERSION 3.10)
project(today)
set(BOOST_ROOT "/home/ubuntu/boost")  # boost 가 설치된 경로
set(Boost_NO_SYSTEM_PATHS ON)
find_package(Boost REQUIRED COMPONENTS date_time)
include_directories(${Boost_INCLUDE_DIRS})
add_executable(today today.cpp)
target_link_libraries(today ${Boost_LIBRARIES})
$ rm -rf build ; mkdir build ; cd build
$ cmake -G Ninja .. ; cmake --build .
[ 50%] Building CXX object CMakeFiles/today.dir/today.cpp.o
[100%] Linking CXX executable today
[100%] Built target today
$ ./today
오늘 날짜 (기본 형식): 2024-Oct-04
```

36) https://www.boost.org/doc/libs/1_58_0/doc/html/date_time.html

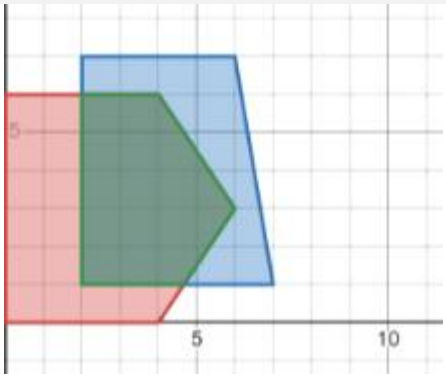
```
오늘 날짜 (ISO 확장 형식): 2024-10-04
오늘 날짜 (간단한 형식): 2024-Oct-04
오늘 날짜 (사용자 정의 형식): 2024-Oct-04
현재 날짜와 시간: 2024-Oct-04 07:10:28
```

- Boost.Geometry[37]를 이용하여 다각형의 교차 영역을 구하기

```
$ more intersection2.cpp
#include <iostream>
#include <deque>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/foreach.hpp>

namespace bg = boost::geometry;
typedef bg::model::d2::point_xy<double> Point;
typedef bg::model::polygon<Point> Polygon;

void print_point(const Point& p) {
  std::cout << "(" << bg::get<0>(p) << ", " << bg::get<1>(p) << ") ";
}
void print_poly(const Polygon& poly) {
  std::cout << "Polygon coordinates: ";
  bg::for_each_point(poly, print_point);
  std::cout << std::endl;
}
int main() {
  Polygon poly1, poly2;
  bg::read_wkt("POLYGON((0 0, 0 6, 4 6, 6 3, 4 0, 0 0))", poly1); // clockwise
  bg::read_wkt("POLYGON((2 1, 2 7, 6 7, 7 1, 2 1))", poly2); // clockwise
  print_poly(poly1);
  print_poly(poly1);

  std::deque<Polygon> output;
  boost::geometry::intersection(poly1, poly2, output);  // 겹치는(intersect) 영역 구하기
  std::cout << "output.size(): " << output.size() << std::endl;

  int i = 0;
  std::cout << "intersections:" << std::endl;
  BOOST_FOREACH(Polygon const& p, output) {
    print_poly(p);
    std::cout << i++ << ": " << boost::geometry::area(p) << std::endl; // 면적(area)
  }
  return 0;
}
$ cmake .. ; cmake --build .
$ ./intersection2
Polygon coordinates: (0, 0) (0, 6) (4, 6) (6, 3) (4, 0) (0, 0)    poly1 (빨간색)
Polygon coordinates: (2, 1) (2, 7) (6, 7) (7, 1) (2, 1)           poly2 (파란색)
output.size(): 1
intersections:
Polygon coordinates: (2, 6) (4, 6) (6, 3) (4.66667, 1) (2, 1) (2, 6)   intersection(초록색)
```

37) https://www.boost.org/doc/libs/1_86_0/libs/geometry/doc/html/index.html

```
0: 15.6667  <---- 교차된 영역의 면적 (area)
```



- Boost Serialization[38]을 이용한 객체 직렬화 예제

```
$ more serial.cpp
[...]
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>

class Person {
private:
  friend class boost::serialization::access;
  std::string name;
  int age;
  template<class Archive>
  void serialize(Archive & ar, const unsigned int version) {
    ar & name;
    ar & age;
  }
public:
  Person() {}
  Person(const std::string& n, int a) : name(n), age(a) {}
  void display() const {
    std::cout << "Name: " << name << ", Age: " << age << std::endl;
  }
};
[...]
$ vi CMakeLists.txt
[...]
find_package(Boost REQUIRED COMPONENTS serialization filesystem)
add_executable(serial serial.cpp)
target_link_libraries(serial  Boost::serialization)
$ ./serial
Enter name: 홍길동
Enter age: 30
Loaded person: Name: 홍길동, Age: 30     객체를 직렬화 하여 person.txt 파일에 저장
$ file person.txt
person.txt: Unicode text, UTF-8 text
$ ./serial
Enter name: .        person.txt 파일에 저장된 내용을 객체로 로딩하여 출력
Loaded person: Name: 홍길동, Age: 30
```

38) https://www.boost.org/doc/libs/1_86_0/libs/serialization/doc/tutorial.html

5. 다른 빌드 도구

5.1 Meson

- Meson[39]은 CMake의 대안으로 주목받는 빌드 시스템으로 2013년 최초 릴리즈됨. 개발자는 Jussi Pakkanen. 2018년 GNOME 프로젝트에서 공식 채택되었다. Meson의 특징은 다음과 같다: 파이썬 기반의 간결한 문법, 빠른 빌드 속도, 크로스 컴파일 지원, 다양한 언어 및 플랫폼 지원,
- meson 설치

```
$ sudo apt install meson
$ meson --version
1.3.2
$ file /usr/bin/meson
/usr/bin/meson: Python script, ASCII text executable
$ meson --help       # 도움말
$ meson setup --help  # setup 도움말
```

- 테스트 프로젝트

```
$ cd meson/hello
$ more main.c
#include <stdio.h>
int main(int argc, char **argv) {
  printf("Hello there.\n");
  return 0;
}
$ more meson.build
project('tutorial', 'c')
executable('demo', 'main.c')
$ meson setup builddir
The Meson build system
Version: 1.3.2
Source dir: /home/ubuntu/linux-drill/meson/hello
Build dir: /home/ubuntu/linux-drill/meson/hello/builddir
Build type: native build
Project name: tutorial
Project version: undefined
C compiler for the host machine: cc (gcc 11.4.0 "cc (Ubuntu 11.4.0-9ubuntu1) 11.4.0")
C linker for the host machine: cc ld.bfd 2.42
Host machine cpu family: x86_64
Host machine cpu: x86_64
Build targets in project: 1
Found ninja-1.11.1 at /usr/bin/ninja
$ cd builddir
$ ninja -v
[1/2] cc -Idemo.p -I. -I.. -fdiagnostics-color=always -D_FILE_OFFSET_BITS=64 -Wall -Winvalid-pc
h -O0 -g -MD -MQ demo.p/main.c.o -MF demo.p/main.c.o.d -o demo.p/main.c.o -c ../main.c
[2/2] cc  -o demo demo.p/main.c.o -Wl,--as-needed -Wl,--no-undefined
$ ./demo
Hello there.
```

39) https://mesonbuild.com/

※ clang 컴파일러를 이용하기

```
$ sudo apt install lld
$ CC=clang CC_LD=lld  meson setup buildclang
The Meson build system
Version: 1.3.2
Source dir: /home/ubuntu/linux-drill/meson/hello
Build dir: /home/ubuntu/linux-drill/meson/hello/buildclang
Build type: native build
Project name: tutorial
Project version: undefined
C compiler for the host machine: clang (clang 18.1.3 "Ubuntu clang version 18.1.3 (1ubuntu1)")
C linker for the host machine: clang ld.lld 18.1.3
[...]
$ cd buildclang/
$ ninja -v
[1/2] clang -Idemo.p -I. -I.. -fdiagnostics-color=always -D_FILE_OFFSET_BITS=64 -Wall -Winvalid
-pch -O0 -g -MD -MQ demo.p/main.c.o -MF demo.p/main.c.o.d -o demo.p/main.c.o -c ../main.c
[2/2] clang  -o demo demo.p/main.c.o -Wl,--as-needed -Wl,--no-undefined -fuse-ld=lld
```

- 의존성 라이브러리 추가하기

```
$ cd hellogtk
$ more main.c         # GTK 프로그램
#include <gtk/gtk.h>
static void activate(GtkApplication* app, gpointer user_data) {
  GtkWidget *window;
  GtkWidget *label;
  window = gtk_application_window_new (app);
  label = gtk_label_new("Hello GNOME!");
  gtk_container_add (GTK_CONTAINER (window), label);
  gtk_window_set_title(GTK_WINDOW (window), "Welcome to GNOME"); // 윈도우의 제목
  gtk_window_set_default_size(GTK_WINDOW (window), 400, 200); // 윈도우 기본 크기
  gtk_widget_show_all(window);
}
int main(int argc, char **argv) {
  GtkApplication *app;
  int status;
#if GLIB_CHECK_VERSION(2, 74, 0)
  app = gtk_application_new(NULL, G_APPLICATION_DEFAULT_FLAGS);
#else
  app = gtk_application_new(NULL, G_APPLICATION_FLAGS_NONE);
#endif
  g_signal_connect(app, "activate", G_CALLBACK(activate), NULL);
  status = g_application_run(G_APPLICATION(app), argc, argv);
  g_object_unref(app);
  return status;
}
$ more meson.build
project('tutorial', 'c')
gtkdep = dependency('gtk+-3.0')
executable('demo', 'main.c', dependencies : gtkdep)

$ sudo apt install libgtk-3-dev
$ meson setup builddir
```

```
The Meson build system
[...]
Host machine cpu family: x86_64
Host machine cpu: x86_64
Found pkg-config: YES (/usr/bin/pkg-config) 1.8.1
Run-time dependency gtk+-3.0 found: YES 3.24.41
Build targets in project: 1
Found ninja-1.11.1 at /usr/bin/ninja
```

```
$ cd builddir
$ ninja
$ ./demo
```

- 라이브러리 빌드 및 설치하기

```
$ cd mylib
$ tree
.
├── meson.build
└── src
    ├── meson.build
    ├── mylib.c
    └── mylib.h
$ more meson.build
project('mylib', 'c',
  version : '0.1',
  default_options : ['warning_level=3'])
subdir('src')
$ more src/mylib.h
extern int sum(int a, int b);
$ more src/mylib.c
#include "mylib.h"
int sum(int a, int b) {
  return a+b;
}
$ more src/meson.build
mylib_sources = ['mylib.c']
mylib = shared_library('mylib',
  mylib_sources,
  install : true,
  include_directories : include_directories('.'))
install_headers('mylib.h')
$ rm -rf $HOME/mylib
$ meson setup builddir --prefix=$HOME/mylib
$ cd builddir
$ ninja -v
```

```
[1/2] cc -Isrc/libmylib.so.p -Isrc -I../src -fdiagnostics-color=always -D_FILE_OFFSET_BITS=64 -
Wall -Winvalid-pch -Wextra -Wpedantic -O0 -g -fPIC -MD -MQ src/libmylib.so.p/mylib.c.o -MF src/
libmylib.so.p/mylib.c.o.d -o src/libmylib.so.p/mylib.c.o -c ../src/mylib.c
[2/2] cc  -o src/libmylib.so src/libmylib.so.p/mylib.c.o -Wl,--as-needed -Wl,--no-undefined -sh
ared -fPIC -Wl,--start-group -Wl,-soname,libmylib.so -Wl,--end-group
$ ninja -v install
[0/1] /usr/bin/meson install --no-rebuild
Installing src/libmylib.so to /home/ubuntu/mylib/lib/x86_64-linux-gnu
Installing /home/ubuntu/linux-drill/meson/mylib/src/mylib.h to /home/ubuntu/mylib/include
$ tree  $HOME/mylib
/home/ubuntu/mylib
├── include
│   └── mylib.h
└── lib
    └── x86_64-linux-gnu
        └── libmylib.so  <-------- shared library
```

5.2 Bazel

- Bazel[40]은 Google에서 개발한 빌드 시스템으로 다음과 같은 특징이 있음: 대규모 프로젝트에 적합, 재현 가능한 빌드, 캐싱 및 병렬 빌드 지원, 다양한 언어 지원(C++, Java, Kotlin, Python, Go, Rust)
- C++프로젝트 빌드 튜토리얼 참고[41]

```
$ git clone https://github.com/bazelbuild/examples
$ cd cpp-tutorial/stage1
$ touch WORKSPACE    # 프로젝트 루트 디렉토리에 빈 WORKSPACE 파일을 생성
$ tree
.
├── MODULE.bazel
├── README.md
├── WORKSPACE
└── main
    ├── BUILD
    └── hello-world.cc
$ more main/BUILD
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
)
$ bazel build //main:hello-world
INFO: Analyzed target //main:hello-world (15 packages loaded, 57 targets configured).
INFO: Found 1 target...
INFO: From Compiling main/hello-world.cc:
1728106571.601703617: src/main/tools/linux-sandbox.cc:152: calling pipe(2)...
1728106571.601723027: src/main/tools/linux-sandbox.cc:171: calling clone(2)...
[...]
Target //main:hello-world up-to-date:
  bazel-bin/main/hello-world
INFO: Elapsed time: 0.841s, Critical Path: 0.38s
INFO: 6 processes: 4 internal, 2 linux-sandbox.
INFO: Build completed successfully, 6 total actions
```

40) https://bazel.build/?hl=ko
41) https://bazel.build/start/cpp?hl=ko

```
$ file bazel*
bazel-bin:      symbolic link to /home/ubuntu/.cache/bazel/_bazel_ubuntu/b897d1913c8a62139df9c782b7e4165a
           /execroot/__main__/bazel-out/k8-fastbuild/bin
bazel-out:      symbolic link to /home/ubuntu/.cache/bazel/_bazel_ubuntu/b897d1913c8a62139df9c782b7e4165a
           /execroot/__main__/bazel-out
bazel-stage1:   symbolic link to /home/ubuntu/.cache/bazel/_bazel_ubuntu/b897d1913c8a62139df9c782b7e4165a
           /execroot/__main__
bazel-testlogs: symbolic link to /home/ubuntu/.cache/bazel/_bazel_ubuntu/b897d1913c8a62139df9c782b7e4165a
           /execroot/__main__/bazel-out/k8-fastbuild/testlogs
$ bazel-bin/main/hello-world
Hello world
Sat Oct  5 05:41:12 2024
$ bazel clean     # 프로젝트 초기화 하기
```

- 여러 빌드 타겟

```
$ cd cpp-tutorial/state2
$ touch WORKSPACE
$ tree
.
├── MODULE.bazel
├── README.md
├── WORKSPACE
└── main
    ├── BUILD
    ├── hello-greet.cc
    ├── hello-greet.h
    └── hello-world.cc
$ more main/BUILD
cc_library(
    name = "hello-greet",
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
)
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        ":hello-greet",
    ],
)
$ bazel build //main:hello-world
Starting local Bazel server and connecting to it...
INFO: Analyzed target //main:hello-world (15 packages loaded, 60 targets configured).
INFO: Found 1 target...
INFO: From Compiling main/hello-greet.cc:
[...]
$ bazel-bin/main/hello-world
Hello world
Sat Oct  5 05:46:57 2024
```

- 여러 패키지

```
$ cd cpp-tutorial/state3
$ tree
```

```
.
├── MODULE.bazel
├── README.md
├── lib
│   ├── BUILD
│   ├── hello-time.cc
│   └── hello-time.h
└── main
    ├── BUILD
    ├── hello-greet.cc
    ├── hello-greet.h
    └── hello-world.cc
```

```
$ more lib/BUILD
cc_library(
    name = "hello-time",
    srcs = ["hello-time.cc"],
    hdrs = ["hello-time.h"],
    visibility = ["//main:__pkg__"],
)
$ more main/BUILD
cc_library(
    name = "hello-greet",
    srcs = ["hello-greet.cc"],
    hdrs = ["hello-greet.h"],
)
cc_binary(
    name = "hello-world",
    srcs = ["hello-world.cc"],
    deps = [
        ":hello-greet",
        "//lib:hello-time",
    ],
)
$ bazel build //main:hello-world
$ bazel-bin/main/hello-world
Hello world
Sat Oct  5 05:52:39 2024
```

5.3 SCons

- SCons[42][43]는 2000년에 개발이 시작됨. 파이썬 기반. ScCons 라는 빌드 도구 설계에서 시작됨. 기존의 Cons라는 빌드 도구를 기반으로 함. Make 유틸리티를 대체하기 위한 목적으로 개발됨. 이후 지속적으로 발전하여 오픈소스 소프트웨어 빌드 도구로 널리 사용됨

```
$ sudo apt install scons
$ cd scons/hello
$ more hello.c
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
}
```

42) https://scons.org/
43) https://github.com/SCons/scons

```
$ more SConstruct
Program('hello.c')
$ scons    # 빌드하기
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
gcc -o hello.o -c hello.c
gcc -o hello hello.o
scons: done building targets.
$ ./hello
Hello, world!
$ scons -c    # clean
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.o
Removed hello
```

5.4 Premake

- Premake[44]는 Lua 스크립트 기반의 빌드 구성 도구

```
   # premake5 다운로드 및 설치
$ wget https://github.com/premake/premake-core/releases/download/v5.0.0-beta2/premake-5.0.0-bet
      a2-linux.tar.gz
$ sudo cp premake5 /usr/local/bin/
$ premake5 --help # 도움말
Premake 5.0.0-dev, a build script generator
Copyright (C) 2002-2021 Jason Perkins and the Premake Project
Lua 5.3 Lua 5.3.5  Copyright (C) 1994-2018 Lua.org, PUC-Rio

Usage: premake5 [options] action [arguments]
[...]
```

```
$ cd premake/hello
$ more premake5.lua
-- premake5.lua
workspace "HelloWorld"
   configurations { "Debug", "Release" }

project "HelloWorld"
   kind "ConsoleApp"
   language "C"
   targetdir "bin/%{cfg.buildcfg}"

   files { "**.h", "**.c" }

   filter "configurations:Debug"
      defines { "DEBUG" }
      symbols "On"

   filter "configurations:Release"
      defines { "NDEBUG" }
```

44) https://premake.github.io/

```
      optimize "On"
```

```
$ premake5 gmake
Building configurations...
Running action 'gmake'...
Generated Makefile...
Generated HelloWorld.make...
Done (21ms).
$ make config=release
==== Building HelloWorld (release) ====
Creating obj/Release
hello.c
Creating bin/Release
Linking HelloWorld
$ tree
.
├── HelloWorld.make
├── Makefile
├── bin
│   └── Release
│       └── HelloWorld
├── hello.c
├── obj
│   └── Release
│       ├── hello.d
│       └── hello.o
└── premake5.lua
```

5.5 Gradle

- Gradle[45] 최신버전 설치하기

```
# cd /opt
# wget https://services.gradle.org/distributions/gradle-8.10.2-bin.zip
# unzip gradle-8.10.2-bin.zip
# mv gradle-8.10.2/ gradle
$ export PATH=$PATH:/opt/gradle/bin
$ gradle --version
------------------------------------------------------------
Gradle 8.10.2
------------------------------------------------------------

Build time:    2024-09-23 21:28:39 UTC
Revision:      415adb9e06a516c44b391edff552fd42139443f7

Kotlin:        1.9.24
Groovy:        3.0.22
Ant:           Apache Ant(TM) version 1.10.14 compiled on August 16 2023
Launcher JVM:  21.0.4 (Ubuntu 21.0.4+7-Ubuntu-1ubuntu224.04)
Daemon JVM:    /usr/lib/jvm/java-21-openjdk-amd64 (no JDK specified, using current Java home)
OS:            Linux 6.8.0-1016-aws amd64
```

45) https://gradle.org/

- C++ 어플리케이션 시작하기 가이드 참고 [46]

```
$ mkdir demo ; cd demo
$ gradle init
Select type of build to generate:
  1: Application
  2: Library
  3: Gradle plugin
  4: Basic (build structure only)
Enter selection (default: Application) [1..4] 1

Select implementation language:
  1: Java
  2: Kotlin
  3: Groovy
  4: Scala
  5: C++
  6: Swift
Enter selection (default: Java) [1..6] 5

Project name (default: demo):

Select build script DSL:
  1: Kotlin
  2: Groovy
Enter selection (default: Kotlin) [1..2] 1

Generate build using new APIs and behavior (some features may change in the next minor releas
e)? (default: no) [yes, no] no

> Task :init
Learn more about Gradle by exploring our Samples at https://docs.gradle.org/8.10.2/samples/samp
le_building_cpp_applications.html

BUILD SUCCESSFUL in 44s
1 actionable task: 1 executed
```

```
$ tree
.
├── app
│   ├── build.gradle.kts
│   └── src
│       ├── main
│       │   ├── cpp
│       │   │   └── app.cpp
│       │   └── headers
│       │       └── app.h
│       └── test
│           └── cpp
│               └── app_test.cpp
├── gradle
│   ├── libs.versions.toml
│   └── wrapper
```

46) https://docs.gradle.org/current/samples/sample_building_cpp_applications.html

```
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
└── settings.gradle.kts
```

```
$ more settings.gradle.kts
rootProject.name = "demo"
include("app")
```

```
$ more app/build.gradle.kts
plugins {
    `cpp-application`
    `cpp-unit-test`
}
application {
    targetMachines.add(machines.linux.x86_64)
}
```

```
$ more src/main/cpp/app.cpp
#include <iostream>
#include <stdlib.h>
#include "app.h"

std::string demo::Greeter::greeting() {
    return std::string("Hello, World!");
}

int main () {
    demo::Greeter greeter;
    std::cout << greeter.greeting() << std::endl;
    return 0;
}
```

```
$ more src/test/cpp/app_test.cpp
/*
 * This source file was generated by the Gradle 'init' task
 */

#include "app.h"
#include <cassert>

int main() {
    demo::Greeter greeter;
    assert(greeter.greeting().compare("Hello, World!") == 0);
    return 0;
}
```

```
$ ./gradlew build

BUILD SUCCESSFUL in 18s
8 actionable tasks: 8 executed
$ tree app/build/install/
app/build/install/
```

```
├── main
│   └── debug
│       ├── app
│       └── lib
│           └── app
└── test
    ├── appTest
    └── lib
        └── appTest
```

```
$ ./app/build/exe/main/debug/app
Hello, World!
```

[참고자료]

- CMake 튜토리얼 https://cmake.org/cmake/help/latest/guide/tutorial/index.html
- CMake Commands https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html
- Ninja build system document https://ninja-build.org/manual.html
- cmake gui 사용법 https://blog.naver.com/howdy_amigo/220864067385
- CDash https://www.cdash.org/
- https://cmake.org/cmake/help/latest/guide/importing-exporting/index.html
- Google Test 정리 - GMock 활용 https://blog.naver.com/sheld2/222383962273
- Google Test 사용법 https://hiseon.me/c/google-test/
- Google C++ Testing, GTest, GMock Framework https://www.youtube.com/watch?v=nbFXI9SDfbk&list=PL_dsdStdDXbo-zApdWB5XiF2aWpsqzV55
- 구글 엔지니어는 이렇게 일한다 https://velog.io/@shinabeuro/구글-엔지니어는-이렇게-일한다-11장-테스트-개요
- SCons Cookbook https://scons-cookbook.readthedocs.io/en/latest/

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

ISBN 978-89-294-1734-5 (PDF)