6장. CMake

앞의 기본 개념 내용과 다른 빌드 도구에 관련한 내용이 있지만 나는 C,C++에서 사용할 CMake를 공부를 먼저 할 예정이다. 그리고 GNU Make도 존재를 하지만 빠른 프로젝트 진행을 위해서 CMake를 하고 MakeFile에 관한 공부를 계속해서 진행을 하겠다.

현재 이 장에서 설명할 빌드 도구는 CMake이다. GNU Make등 과 다르게 CMake는 직접 빌드 프로세스를 수행하지 않는다. 대신 CMake는 상위 레벨 빌드 기술을 그 밖의 도구가 해석할 수 있는 하위 레벨 빌드 기술로 변환을 해준다.

이런 방식의 빌드 기법은 빌드 시스템을 구축하는 작업을 단순화하고 크로스플랫폼 개발을 용이하게 한다. CMake는 빌드 프로세스를 기술하는 고급 언어를 제공함으로써 앞서 언급한 문제를 해결한다. 즉, CMake 생성기가 이 빌드 기술을 빌드 프로세스를 직접 수행할 수 있는 네이티브 빌드 도구 언어로 변환해 빌드 시스템의 구성의 복잡성을 줄인다.

CMake는 윈도우, 맥, 리눅스, 그 밖에 다양한 유닉스 계열 머신에서 동작할 수 있는 여러 개의 생성기를 제공해 많은 개발 환경에서 사용할 수 있다.

CMake 프로그래밍 언어

CMake 언어 기초

CMake의 빌드 기술 파일의 구문은 어렵지 않기 때문에 예제를 살펴 보는 것만으로 기초 구문을 습득할 수 있다. 다음 예제는 두 개의 변수를 정의하고 두 개의 소스 파일의 속성 값을 입력하고 메세지를 출력한다.

```
 1 project(basi-syntax C)
 2
 3 cmake_minimum_required(VERSION 2.6)
 4
 5 set (wife Grace)
 6 set (dog Stan)
 7 message("${wife}, please take ${dog} for a walk")
 8
 9 set_property(SOURCE add.c PROPERTY Author Peter)
10 set_property(SOURCE mult.c PROPERTY Author John)
11 get_property(author_name SOURCE add.c PROPERTY Author)
12 message("The author of add.c is ${author_name}")
```

모든 명령은 스페이스로 구분된 인수를 전달받는다.

```
command (arg1 arg2 ...)
```

구문에 다른 인수 규칙은 숫자, 파일, 이름, 문자열, 속성명이 될 수 있으며, 명령에 따라 어떤 인수가 사용될지 결정된다. 두 개의 단어를 하나의 인수로 사용하고 싶을 때는 문장에 따옴표를 사용해 하나의 인수로 사용한다. → 쉘프로그래밍에서 배웠던 내용

1번 행의 project 명령은 빌드 시스템을 식별할 수 있는 고유한 이름을 정의한다. 여기서 정의된 이름은 이클립스와 같은 프로젝트 이름이 필요한 네이티브 빌드 도구에서 그대로 사용된다. 그리고 이 프로젝트에서 사용할 언어를 지정한다.

3번 행의 cmake_minimum_required 명령은 CMake 버전 2.6 이후부터 지원되는 명령을 사용하겠다는 점을 명시하는 것이다. cmake_minimum_required 명령은 두 개의 인수를 취하는데 그 첫 번째 인수는 VERSION 키워드이고 두 번째 인수는 버진 번호이다.

5번과 6번 행에서는 변수를 정의한다. 첫 번째 인수는 변수명이고, 두번째 인수는 값을 나타낸다. 7번 행에서는 변수 대치를 사용하여 변수 값을 참조하고 있다.

9번 행에서는 set_property 명령을 사용해 속성 값을 설정한다. 이 속성을 통해 지정된 물리 디스크 파일에 값을 저장할 수 있다. 빌드 시스템은 파일 이름을 기준으로 속성 값을 관리하고, 설정된 속성은 파일 내용에 영향을 주지 않는다. 또한 이렇게 설정된 속성 값은 다른 명령에서 자유롭게 접근할 수 있다.

예제에서 add.c 소스 파일의 Author 속성에 값을 할당했다. 할당된 값은 add.c의 고유 값이 된다. 즉, 다른 소스 파일에는 다른 Author 값이 할당될 수 있다.

10번 행에서 mult.c의 값 Author 속성에 또 다른 값을 할당한 점을 확인할 수 있다.

11번 행에서 get_property 명령을 사용해 add.c에 할당된 Author 속성 값을 얻어와 author_name 변수에 할당한다.

마지막으로 12번 행에서 author_name의 값을 출력한다.

실행 프로그램과 라이브러리 빌드

빌드 도구의 가장 기본적인 도작은 소스 파일을 컴파일해 실행 프로그램을 생성하는 작업이다. CMake는 SCons에서 살펴본 빌더 메소드와 유사한 많은 명령을 제공한다. 하지만 그 외에도 CMake만이 갖고 있는 흥미로운 기능도 있다.

- 실행 프로그램과 라이브러리 생성

다음 코드는 계산기 프로그램이 네 개의 소스 파일로부터 어떻게 생성되는지 보여준다.

```
add_executable(calculator add sub mult calc)
```

코드는 간단해보이지만, 앞에서 살펴본 다른 빌드 도구를 통해 add_executable이 많은 작업을 수행한다는 사실을 쉽게 눈치챌 수 있을 것이다. 작업에는 컴파일 명령을 생성하고 종속성 그래프에 파일 이름을 추가하는 작업이 포함된다.

또 주의 깊게 살펴봐야 할 점은 파일 확장자가 사용되지 않았다는 사실이다. CMake는 각 빌드 시스템에서 사용될 확장자를 자동으로 붙여준다. 예를 들면 윈도우에서는 최종 실행 프로그램이 .exe파일이 된다.

```
1 add_library(math STATIC add sub mult)
2 add_executable(calculator calc)
3 target_link_libraries(calculator math)
```

1번 행에서 add.c와 sub.c mult.c를 컴파일해 정적 라이브러리를 생성했다. 최종 라이브러리 이름은 빌드 머신의 종류에 따라 결정된다. 에를 들어 유닉스 시스템에서는 libmath.a 가 된다. 2번과 3번 행에서 calc.c 파일을 컴파일한 후 math 라이브러리와 링크 과정을 통해 계산기 프로그램을 생성한다.

add_executable과 add_library 이전에 include_directories를 사용해 C컴파일러에 헤더 파일 경로를 추가할 수 있으며, link_directories를 사용해 라이브러리 경로를 추가할 수 있다. → 버전마다 사용되는 명령이 다를 수 있으므로 조심

이렇게 추가된 명령은 gcc 컴파일러 -I와 -L 같은 네이티브 빌드 시스템의 컴파일 플래그로 변경된다.

- 컴파일 플래그 설정

컴파일 도구에 따라 옵션을 가변적으로 설정할 수 있는 기능은 상당히 유용하다. 플랫폼 종속적인 빌드 도구와 대조적으로 CMake는 직접 컴파일 옵션을 사용하는 것을 지양한다. 대

신 CMake에서는 빌드 기술에 어떤 타입의 결과물을 생성할지 지정하고, 사용할 컴파일 플래그를 설정한다.

예를 들어 CMake가 소스 레벨 디버깅 정보를 포함한 debug 빌드를 생성하게 지정하고 싶다면 다음과 같은 명령을 CMakeLists.txt에 저장하면 된다.

```
set(CMAKE_BUILD_TYPE Debug)
# command line version
$ cmake .. -D CMAKE_BUILD_TYPE=Debug or Release
```

플랫폼에 특화 된 플래그가 추상화 되더라도 CMake는 해당 시스템에 대한 올바른 컴파일 플래그로 네이티브 빌드 시스템을 생성한다. 예를 들어 유닉스 시스템에서 C 컴파일러 옵션어 -g 플래그가 추가된다.

여러 다른 C 컴파일러는 각기 다른 커맨드라인 옵션을 지니기 때문에 C언어의 preprocessor definition에서도 같은 방법이 사용된다. 이때 전체 디렉터리나 파일별로 속성을 정의해줄 수 있다.

```
set_property(DIRECTORY PROPERTY COMPILE_DEFINITIONS TEST=1)
set_property(SOURCE add.c PROPERTY COMPILE_DEFINITIONS QUICKADD=1)
```

위 코드에서의 첫 번째는 빌드 시스템이 현재 디렉터리에 있는 모든 C파일을 컴파일할 때 TEST 심볼을 정의하게 한다. 두 번째는 add.c를 컴파일 할 때 QUICKADD 심볼이 추가되게 한다. 네이티브 빌드 시스템은 위에서 지정한 옵션이 반영되게 커맨드라인 옵션에 추가한다.

- 외부 명령과 타겟 추가

복잡한 빌드 시스템을 구성할 때 새로운 컴파일 도구를 사용할 수도 있는데, 이때 CMake는 네이티브 빌드 시스템이 새로운 컴파일 도구를 사용할 수 있게 지원할 수 있다. add_custom_command는 GNU Make의 규칙과 유사하며 add_custom_target은 GNU Make의 phony target과 개념적으로 유사하다.

다음 예제에서 유닉스 명령을 사용해 data.dat 입력 파일로부터 data.c 출력 파일을 생성하는 작업을 가정해보자, 즉, data.c는 자동으로 생성되는 소스 파일이다.

```
# add_custom_command example
project(custom_command)
cmake_minimum_required(VERSION 2.6)

set(input_data_file ${PROJECT_SOURCE_DIR}/data.dat)
set(output_c_file data.c)
```

```
add_custom_command(
    OUTPUT ${output_c_file}
    COMMAND /tools/bin/make-data-file
            < ${input_data_file}
            > ${output_c_file}
    DEPENDS ${input_data_file}
)

add_executable(print-data ${output_c_file})
```

add_custom_command부분을 살펴보자. 이 부분에서 사용자 도구가 종속성 그래프에 추가된다. OUPUT지시어는 생성될 출력 파일을 지정하고 DEPENDS 지시어는 위의 정의된 유닉스 shell 명령의 입력 파일을 지정한다.

마지막 부분에서 새로운 도구를 호출하는 상위 레벨 타겟인 실행 프로그램이 정의돼 있다. 위와 같이 실행 프로그램을 정의하지 않으면 종속성 그래프가 정상적으로 만들어지지 않기 때문에 data.c파일은 생성되지 않는다. 다른 빌드 도구와는 달리 CMake는 빌드 트리에서 상위 레벨 타겟과 일반 파일을 명확히 구분한다.

add_custom_target 명령은 새로운 상위 레벨 타겟을 정의하고 실행될 순서를 지정한다. 결과물을 생성하지 않고 파일의 갱신 상태에 의존하지 않다는 점에서 GNU Make의 포니 타겟과 유사하다.

```
project(custom_target)
cmake_minimum_required(VERSION 2.6)

add_custom_target(
    print-city ALL
    COMMAND echo "Vancouver is a nice city"
    COMMAND echo "Even when it rains"
)

add_custom_target(
    print-time
    COMMAND echo "It is now 2:17pm"
)

add_custom_target(
    print-day
    COMMAND echo "Today is Monday"
)

add_dependencies(print-city print-time print-day)
```

위 코드에서 주의 깊게 살펴봐야 할 점은 ALL 키워드이다. ALL키워드는 개발자가 타겟을 명시하지 않을 때 print-city가 기본 빌드로 실행되게 정의한다. 그리고 마지막 add_dependencies는 print-city가 print-time print-day에 종속된다는 점을 나타낸다.

흐름 제어

CMake의 조건문, 반복문, 매크로 같은 흐름 제어는 다른 프로그래밍 언어와 유사하다. 하지만 네이티브 빌드 시스템이 아닌 CMake 생성기가 제어 흐름은 조건을 확인하고 실행한다는 점은 명심하기 바란다.

```
# if control flow example
set(my_val 1)
if(${my_val})
    message ("my_var is true")
else()
    message ("my_var is false")
endif()

# we can use NOT, AND, OR operation in cmake text
if(NOT my_var)
    ...
else()
    ...
endif()

# we can compare other value
if(${my_age} EQUAL 40)
    ...
endif()

if(EXISTS file1.txt)
    ...
endif()

# we can check create order between two files
if(file1.txt IS_NEWER_THAN file2.txt)
    ...
endif()

# we can use regular expression to check value
if(${symbol_name} MATCHES "^[a-z][a-z0-9]*$")
    ...
endif()
```

여기서 주의할 점은 파일 존재 여부를 확인 할 수도 있는데, 파일을 확인하는 시점은 네이티브 빌드 시스템이 실행되는 시점이 아닌 생성되는 시점이라는 것을 주의해야 한다.

```
# use macro example
project(macro)
cmake_minimum_required(VERSION 2.6)

macro(my_macro ARG1 ARG2 ARG3)
    message("The my_macro macro was passed the following arguments:")
    message("${ARG1} ${ARG2} ${ARG3}")
endmacro(my_macro)
```

```
my_macro(1 2 3)
my_macro(France Germany Russia)
```

위에 처럼 매크로를 사용해서 함수나 메소드처럼 코드를 재사용할 수도 있다.

```
1 project(foreach)
2 cmake_minimum_required(VERSION 2.6)
3
4 foreach(source_file add.c sub.c mult.c calc.c)
5     message("Calculation cksum for ${source_file}")
6     add_custom_target(cksum-${source_file} ALL
7         COMMAND cksum ${PROJECT_SOURCE_DIR}/${souce_file}
8     )
9 endforeach(source_file)
```

마지막 예제를 좀 더 설명하면 6번 행에서 foreach 구문에 의해 새로운 레벨 타겟이 리스트에 각 소스 파일별로 추가된다. 이렇게 추가된 타겟은 해당 소스 파일과 관련된 cksum 명령을 수행한다. makefile 기반 빌드 시스템을 만든다고 가정하면 다음과 같은 모든 타겟을 한 번에 실행시킬 수도 있고 각 개별로 실행시킬 수도 있다.

크로스플랫폼 지원

크로스 플랫폼을 지원하려면 CMake의 빌드 기술은 플랫폼 중립적이어야 한다. CMake는 특정 도구나 파일의 위치를 찾을 수 있게 지원할 뿐만 아니라 사용되는 컴파일러의 특징을 확인할 수 있는 기능도 제공한다.

빌드 머신에서 파일과 도구 찾기

여러 가지 종류의 빌드 머신에서 동작하는 빌드 시스템을 만들기 위해서는 빌드 프로세스에서 사용되는 도구와 파일의 위치를 상세하게 기술하면 안된다. 단순히 빌드 머신마다 위치는 다르겠지만 필요한 도구와 파일이 파일 시스템 어딘가에 있다고 가정해야만 한다.

CMake는 기본 경로에서 파일과 도구를 찾기 위한 많은 명령을 제공한다. 다음은 ls 프로그램, stdio.h 헤더 파일 , C 표준 math 라이브러리를 찾는 예제이다.

```
#============================================================
 1 project(finding)
 2 cmake_minimum_required(VERSION 3.0)
 3
 4 find_program(LS_PATH ls)
 5 message("The path to the ls program is ${LS_PATH}")
 6
 7 find_file(STDIO_H_PATH stdio.h)
 8 message("The path to the stdio.h file is ${STDIO_H_PATH}")
 9
10 find_library(LIB_MATH_PATH m /usr/local/lib /usr/lib64)
```

```
11 message("The path to the math Library is {LIB_MATH_PATH}")
#=============================================================
```

cmake를 실행시켜 이 빌드 기술을 실행시키면 각 파일의 path를 출력해주는 네이티브 빌드 시스템으로 변환된다.

위의 기술된 결과는 빌드 머신마다 다를 수 있다. 그래서 빌드 기술은 경로를 하드 코딩하기보다는 CMake에 의해서 찾아진 값을 이용해야 한다. 10번행에서 find_library를 사용해서 math 라이브러리를 두 가지의 경로에서 찾게 한다. 결국 CMake가 가진 기본 경로에 추가로 지정한 경로에서 라이브러리를 찾는다.

CMake는 빌드 기술을 쉽게 작성하기 위해 많이 사용되는 도구와 라이브러리를 찾기 위한 코드 모듈을 제공한다. 예를 들면 FindPerl 모듈을 포함해 펄 인터프리터를 쉽게 찾을 수 있다.

```
 1 project(find-perl)
 2 cmake_minimum_required(VERSION 2.6)
 3
 4 include(FindPerl)
 5 if(PERL_FOUND)
 6     execute_process(
 7         COMMAND ${PERL_EXECUTABLE} ${PROJECT_SOURCE_DIR} /
 8         config.pl
 9     )
10 else()
11    message(SEND_ERROR "There is no perl interpreter on this system")
12 endif()
```

4번 행에 있는 FindPerl 모듈은 CMakeLists.txt빌드 기술에서 아주 작은 양을 차지한다. 이 모듈은 빌드 머신이 어떤 타입의 시스템이든 상관없이 펄 인터프리터를 찾아준다. 그리고 펄을 찾으면 PERL_EXECUTABLE 변수에 프로그램의 절대경로를 할당하고 PERL_FOUND 변수는 true가 된다.

6번 행에서는 execute_process를 사용해 펄 인터프리터에 config.pl파일을 넘겨준다. 이 작업은 네이티브 빌드 시스템을 생성하는 과정에서 발생한다. 반면 이전에 살펴본 add_custom_command를 사용하면 이 작업은 네이티브 빌드 시스템에 추가되고 네이티브 빌드 시스템이 동작할 때 호출된다.

소스코드 검증 기능

두 번째 크로스플랫폼을 지원하기 위해 사용될 컴파일러를 확인할 수 있어야한다. 이는 프로그램을 컴파일하기 전에 컴파일러가 필요하는 함수와 헤더 파일을 제공하는지 알아야 하기 때문이다. 제공하지 않는다면 이를 대신할 부분을 직접 구현하거나 빌드 프로세스를 취소해야한다.

CMake에서는 try_complie과 Try_run 명령을 사용해 C,C++ 코드가 정상적으로 컴파일되는지, 그리고 정상적으로 실행되는지 확인할 수 있는 기능을 제공한다. 두 명령을 편리하게 사용할 수 있게 CMake는 많은 매크로를 제공한다.

```
project(try-complie)
cmake_minimum_required(VERSION 2.6)

include(CheckFuctionExists)
include(CheckStructHasMember)

CHECK_FUNCTION_EXISTS(vsnprintf VSNPRINTF_EXISTS)
if(NOT VSNPRINTF_EXISTS)
    message(SEND_ERROR "vsnprintf not available on this build machine")
endif()

CHECK_STRUCT_HAS_MEMBER("struct rusage" ru_stime wait.h HAS_STIME)
if(NOT HAS_STIME)
    message(SEND_ERROR "ru_stime field not available in struct rusage")
endif()
```

7~10번 행에서는 4번 행에 포함된 CheckFunctionExists 모듈 내부에 정의된 CHECK_FUNCTION_EXISTS 매크로 사용법을 보여준다. 이 매크로는 try_complie 명령을 사용해 C컴파일러와 링커가 vsnprintf 함수를 제공하는지 확인하고, 결과를 VSNPRINTF_EXISTS에 설정한다.

12~15번 행은 struct rusage 안에 ru_stime 필드가 있는지 확인하는 동작이다. 필드가 존재하지 않다면 HAS_STIME은 정의되지 않게 되고, 빌드 시스템은 에러 메세지를 출력하고 실패하게 된다.

네이티브 빌드 시스템 생성

앞에서 언급했듯이 CMake는 CMakeLists.txt 파일을 처리해 네이티브 빌드 시스템을 생성하는 단계와 빌드 도구를 사용해 소프트웨어를 실제로 컴파일하는 두 단계로 이뤄진다. 이 중 전자인 다양한 운영체제와 빌드 도구를 지원하기 위해 빌드 시스템을 생성하는 과정이 CMake의 핵심 기능이라고 할 수 있다.

기본 빌드 시스템 생성

가장 손쉽게 빌드 시스템을 생성하는 방법은 기본 설정을 사용하는 방법이다. 개발자는 오브젝트 파일을 위한 임의의 디렉터리를 생성하고 나서 해당 디렉터리에서 cmake 명령을 실행하기만 하면 된다. 소스 파일이 있는 디렉터리에서는 아무것도 변경되지 않는다. 또한 같은 소스 트리를 사용해 여러 오브젝트 디렉터리를 생성할 수도 있다.

CMake는 필요한 개발 도구를 찾으면 해당 도구의 버전을 확인한다. 이때 CMakeLists.txt 파일에 정의된 try_complie, try_run 명령이 실행된다.

cmake를 실행하면 많은 파일을 포함하게 되는데, 주목해서 살펴봐야 할 파일은 다음과 같다.

- makeifle

네이티브 빌드 시스템의 진입점이다.

- CMakeCache.txt

텍스트 기반 설정 파일로 해당 빌드 머신을 위해 자동으로 생성된 기본 설정이 저장돼 있다.

- CMakeFiles/

자동 생성된 프레임워크 파일들이 있는 디렉터리이다. 생성된 파일들은 메인 makefile에서 사용된다.

사용자 선택에 따른 빌드 시스템 형성

CMake의 장점 중 하나는 여러 네이티브 빌드 시스템을 생성할 수 있는 유연성이다. cmake 명령을 실행할 때 -G 옵션을 사용해 기본으로 생성되는 빌드 시스템을 변경할 수 있다.

생성 단계 커스터마이징

CMake의 기본 동작은 빌드 머신의 컴파일 도구를 자동으로 찾아 설정하는 것 이지만, 때로는 자동으로 설정된 값을 변경해야 할 때도 있다. CMake는 cmake 명령외에도 네이티브 빌드 시스템을 더욱 자세히 설정할 수 있는 ccmake명령을 제공한다.

```
$ ccmake <CMakeLists.txt path>
```

만약 실행을 시켜보면 보이는 변수들은 일명 cache라고 불리며, 오브젝트 디렉터리에 있는 CMakeCache.txt에 파일이 저장되어 있다. 각 변수는 기본 값이 할당 돼 잇지만 쉽게 변경할 수 있어 어렵지 않게 빌드 프로세스를 변경할 수 있다. 아래는 일반적으로 사용되는 캐시 변수이다.

- CMAKE_AR, CMAKE_C_COMPILER, CMAKE_LINKER

라이브러리를 묶어주는 아카이브 도구와 C컴파일러, 오브젝트 링커의 절대 경로이다. 이 변수 값을 변경해 빌드 머신의 기본 도구를 사용자 정의 도구로 변경할 수 있다.

- CMAKE_MAKE_PROGRAM

/usr/bin/gmake와 같은 네이티브 빌드 도구의 절대 경로다. 기본 설정 버전을 다른 버전으로 변경할 수 있다.

- CMAKE_BULID_TYPE
    - Debug : 디버깅 정보를 포함하여 빌드

    - Release : 실행 파일을 최적화하고 디버깅 정보를 포함하지 않는다.
    - RelWithDebInfo : 실행 파일도 최적화, 디버깅 정보를 포함한다.
    - MinSizeRel : 실행 파일이 최소한의 메모리를 가지게 한다.
- CMAKE_C_FLAGS_*

앞에서 말한 4가지 빌드 타입에 따라 이 옵션이 지정된다.

- CMAKE_EXE_LINKER_FLAGS_*

앞에서 언급한 C 컴파일 옵션과 유사하게 각 빌드 타입에 따른 링커 옵션을 나타낸다.

CMakeLists.txt에서 네이티브 빌드 시스템으로 변환

CMake 명령은 다음과 같은 두 가지 그룹으로 나눌 수 있다.

1. cmake가 바로 호출될 때 바로 반영되는 명령이다. 이런 명령으로는 if, foreach, macro와 같은 제어 흐름을 위한 명령과 , 변수를 설정하고 변수 값을 화면에 출력하기 위한 명령 등이 있다.
2. 네이티브 빌드 시스템을 만드는 데 사용되는 명령이다. add_executable, add_library, add_custom_command, add_custom_target 등과 같은 명령어이다.

두번째로 언급한 명령들은 네이티브 빌드 시스템의 종속성 그래프를 결정짓는다. 반대로 첫번째 종류의 명령들은 변수를 사용해 추가될 파일을 지정하거나, 반복문을 통해 많은 수의 파일들을 개별적으로 추가할 수 있게 함으로써 어던 파일을 추가할지 결정할 수 있게 해준다. 주의할 점은 종속성 그래프에 영향을 주는 명령은 네이티브 빌드 시스템으로 직접 변환된다는 점이다.

즉, CMake 빌드 기술 파일의 모든 명령어는 일대일 매칭으로 네이티브 빌드 시스템으로 변경되지는 않는다. 하지만 네이티브 빌드 시스템은 다른 명령을 사용해 같은 동작을 하게 만들어준다.

요약

간단하지만 빠른 시간안에 공부를 해야해서 이 정도 이론만 공부를 적어보며 공부를 해봤다. 평소 빌드파일을 직접 만들고 싶었고 아주 흥미로운 시간이 되었던 것 같다. 아직 많은 내용이 남아있기 때문에 대략적인 개념을 이해하면서 좀더 이젠 깊숙히 공부를 해봐야겠다.