



LINKS

[ABOUT](#) | [ARTICLES](#) | [ECE](#) | [SHOWCASE](#) | [GUESTBOOK](#) | [f FACEBOOK](#)

맞춤검색

Home > [ECE](#)

✓뷰어로 보기

일반

[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리

Posted 2017. 02. 19 Updated 2018. 06. 13 Views 30066 Replies 0

ABOUT

ARTICLES

일반 (38)
건강 (9)
여행 (91)
독서 (5)
영화 (3)
박람회 (5)

ECE

일반 (60)
PSpice (6)
AVR (32)
Android (8)
Nginx (2)
Apache (9)
Linux (49)
XE (11)
Python (11)
Security (3)

SHOWCASE

GUESTBOOK

모든 게시물에 대하여 '링크' 방식의 퍼가기만 허용합니다.



한양대학교

Be Bold of Robotics &
Advanced Micro Intelligence
바라미
Since 1994

826

1104916

DNS Powered by...
dnserver



▶ [Makefile 튜토리얼](#)에서는 C 프로젝트를 빌드하기 위한 Makefile을 작성하는 방법에 대해 다뤘습니다. 이번 글에서는 Makefile을 보다 쉽고 편리하게 작성할 수 있는 툴인 **CMake**에 대해 소개하고, 동작 원리를 소개하도록 하겠습니다.

■ Make의 아킬레스건: 항상 신경써야 하는 Makefile 유지/보수

"소스코드를 수정해서 의존성이 바뀔 때마다 Makefile에 보고해야 한다."

그렇지 않으면 당신의 빌드를 비비 꼬이게 만들어 주겠다."

(Makefile이 툴인지 보스인지, 내가 Makefile을 관리하는건지 Makefile이 나를 관리하는건지.. 다는 ㅇㄹ)

Make는 셸 스크립트 기반 빌드에 비해 편리하지만, 프로젝트의 규모가 거대해지면서 관리해야 할 소스 파일들이 많아지고 의존성 관계가

서로 복잡하게 뒤엉키면 점점 그 한계를 드러내기 시작합니다.

Makefile 자체적으로 의존성을 파악하고 어느 정도 자동화를 해 주긴 하지만, 이는 어디까지나 Makefile에 기술된 의존성 정보가 소스코드의 내용에 부합하게 올바르게 관리되고 있다는 것에 전제합니다.

'**Makefile에 기술된 의존성 정보**'란 다음과 같이 Object파일이 의존하는 소스코드나 헤더 파일을 기술한 구문을 의미합니다.

Makefile의 의존성 정의 구문 예시

```
main.o: foo.h bar.h main.c
foo.o: foo.h foo.c
bar.o: bar.h bar.c
```

Make에서는 의존성 정보를 검사할 때 소스파일까지 일일이 뒤져서 어떤 헤더파일이 포함되어 있는지 조사하지 않으므로, Makefile에 의존성 정보가 잘못 기술되어 있으면 빌드시 변경 사항이 제대로 반영되지 않는 소위 '빌드가 꼬이는' 상황이 발생합니다. 이렇게 빌드가 꼬이면 매번 Clean build를 해야 하므로 결국 Make의 가장 강력한 기능 중 하나인 Incremental build를 활용할 수 없게 됩니다.

■ CMake를 사용하면: 소스코드-결과물 사이를 깔끔하게 추상화

"내가 관심있는건 소스코드와 실행 바이너리뿐,
부산물(Object파일)들은 (생기던 말던 어디 처박혀 있던) 내 알 바가 아니다."



CMake를 사용하면 의존성 정보를 일일이 기술해 주지 않아도 되므로 빌드 스크립트의 관리 측면에서 매우 효율적입니다. 프로젝트를 처음 시작할 때 Build Step만 잘 구성해 놓으면, 이후에는 소스 파일(*.c)을 처음 추가할 때만 CMakeLists.txt 파일을 열어서 등록해 주면 됩니다. (그다지 추천하는 방법은 아니지만, 소스파일을 자동으로 찾아서 추가하도록 구성하는 방법도 있습니다.) 이후에는 소스코드를 어떻게 수정하더라도 빌드에서 제외하지 않는 한 스크립트를 수정하지 않아도 됩니다.

CMake도 Make와 마찬가지로 의존성 검사를 해서 Incremental Build를 수행하지만, 가장 큰 차이점은 **CMake는 소스파일 내부까지 들여다보고 분석해서 의존성 정보를 스스로 파악**한다는 점입니다. 예를 들어, 소스파일에 헤더파일을 추가(#include)하면, 직후 빌드부터 의존성 관계 변화가 자동으로 추적되어 헤더 파일의 변화까지 추적하기 시작합니다.

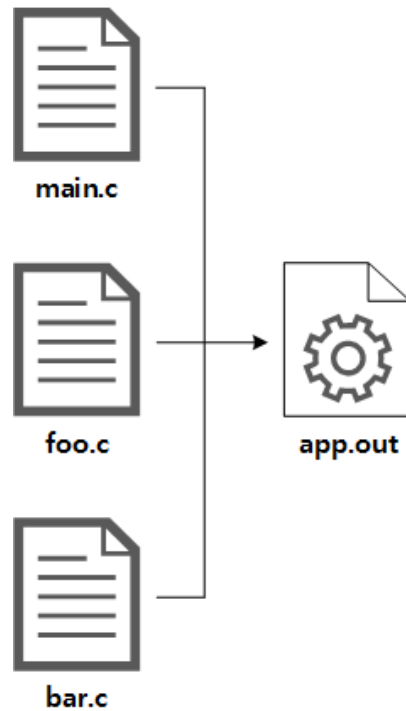
또한, Makefile에서는 빌드 중간생성물인 Object파일들의 이름과 의존성 정보까지 모두 기술해 줘야 하지만, CMake에서는 그럴 필요가 전혀 없습니다. 뒤에서 살펴보겠지만, CMake의 빌드 스크립트인 **CMakeLists.txt에서는 최종 빌드 결과물과 이를 빌드하기 위한 소스 파일들만 명시해 주면 그것으로 끝**입니다. (여기서 최종 빌드 결과물은 실행 바이너리나 라이브러리가 됩니다.)

그렇다고 CMake가 Make보다 훨씬~씬 좋고 편리한 전혀 다른 오버테크놀러지의 새로운 무언가(!)라는게 아니고, Makefile의 다소 지저분한 루틴들을 추상화(Abstraction)해서 보다 직관적으로 빌드 과정을 기술해주는 것입니다. 즉, CMake는 Makefile을 보다 쉽게 기술해 주는 일종의 Meta-Makefile이라고 할 수 있습니다. CMake로 프로젝트를 관리하더라도 결국 최종 빌드는 Make와 마찬가지로 make 명령으로 수행합니다.

그 외에 CMake로 프로젝트를 관리하면 CLion이나 Eclipse와 같은 범용 IDE에서 프로젝트 설정 파일로 사용할 수 있다는 장점도 있습니다. 따라서 협업 프로젝트에서 프로젝트를 체계적으로 관리하면서도 각 개발자마다 선호하는 개발 환경에서 작업을 할 수 있습니다.

■ 빌드 예제

빌드 예제는 직전 글인 Make 튜토리얼에서 다뤘던 내용과 거의 비슷합니다. 다만, Makefile과 달리 CMakeLists.txt에서는 중간 생성물인 Object 파일들은 기술할 필요가 없으므로 생략하였습니다.



세 개의 소스파일로부터 하나의 실행파일을 만드는 예제이며, `main.c`에서는 `foo.c`와 `bar.c`에서 정의된 함수들을 호출하는 의존성이 존재합니다.

■ 전격 비교 분석: Makefile VS CMakeLists.txt

지난 글([Makefile 튜토리얼](#))에서 작성했던 위의 예제를 빌드하기 위한 **Makefile**의 최종 버전은 다음과 같습니다.

Makefile

```
1  OBJS=main.o foo.o bar.o
2  TARGET=app.out
3
4  all: $(TARGET)
5
6  clean:
7      rm -f *.o
8      rm -f $(TARGET)
9
10 $(TARGET): $(OBJS)
11     $(CC) -o $@ $(OBJS)
12
13 main.o: foo.h bar.h main.c
14 foo.o: foo.h foo.c
15 bar.o: bar.h bar.c
```

중간 생성물인 Object 파일명과 빌드 바이너리명을 최상단에서 정의하고, 이어서 각 빌드 Target을 정의했습니다. 끝으로 각 Object 파일들의 의존성을 기술하는 구문이 필요했습니다. 지난 글이야 뭐; 주제가 Makefile 튜토리얼이었으니 별 언급은 하지 않았지만, 여전히 뭔가 좀 긴 것 같지 않나요?^^

위와 같은 기능을 하는 CMake 빌드 스크립트인 **CMakeLists.txt**파일은 다음과 같습니다.

(...?! 달랑 한 줄?? 이걸로 될까 싶긴 하지만 일단 믿어 봅시다...)

CMakeLists.txt

```
1 | ADD_EXECUTABLE( app.out main.c foo.c bar.c )
```

위와 같이 작성하고 이로부터 Makefile을 생성하기 위해 다음 명령을 실행합니다.

```
cmake CMakeLists.txt
```

이렇게 하면 Makefile이 생성되는데, 이제 다음과 같이 make 명령으로 빌드를 할 수 있습니다.

(왜 CMake인데 Makefile이 생성되지?? 라고 머리 위에 물음표 뜨신 분들은 위로 올라가서 글을 다시 읽고 와 주세요..)

```
make
```

```
[16:25:49] ~/Workspace/CMake
tuw@desktop.zip.tuwlab.com:~$ cmake CMakeLists.txt
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/lib/icecc/bin/cc
-- Check for working C compiler: /usr/lib/icecc/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/lib/icecc/bin/c++
-- Check for working CXX compiler: /usr/lib/icecc/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tuw/Workspace/CMake

[16:25:55] ~/Workspace/CMake
tuw@desktop.zip.tuwlab.com:~$ make
Scanning dependencies of target app.out
[ 33%] Building C object CMakeFiles/app.out.dir/main.c.o
[ 66%] Building C object CMakeFiles/app.out.dir/foo.c.o
[100%] Building C object CMakeFiles/app.out.dir/bar.c.o
Linking C executable app.out
[100%] Built target app.out
```

▲ CMake를 활용한 빌드

CMakeLists.txt에 한 줄만 썼다고 해서 app.out만 생성하는 make명령만 되는 게 아닙니다. 'make main.c.o'와 같이 개별 Object파일들을 생

성하는 명령은 물론, 'make all', 'make clean'과 같은 매크로들도 모두 정의되어 있습니다.

이 점에서 빌드 명령이 두 개(cmake + make)가 된 것 같아서 뭔가 손해 보는 것 같기도 하지만, 'cmake CMakeLists.txt' 명령은 자동 생성된 Makefile을 삭제하지 않는 한 최초 한 번만 실행해 주면 됩니다. 생성된 Makefile을 실행할 때 CMakeLists.txt파일의 변경 여부를 검사해서 필요한 경우 Makefile을 자동으로 재생성해 줍니다. 즉, cmake 명령은 최초 한 번만 쓰고, 이후에는 계속 make명령만으로 지지고 볶고 다 할 수 있습니다.

실제로, CMakeLists.txt파일을 수정하고 make명령을 실행하면 다음과 같이 변경 사항을 Makefile에 먼저 반영(--로 시작하는 세 줄)한 뒤 빌드를 수행함을 알 수 있습니다.

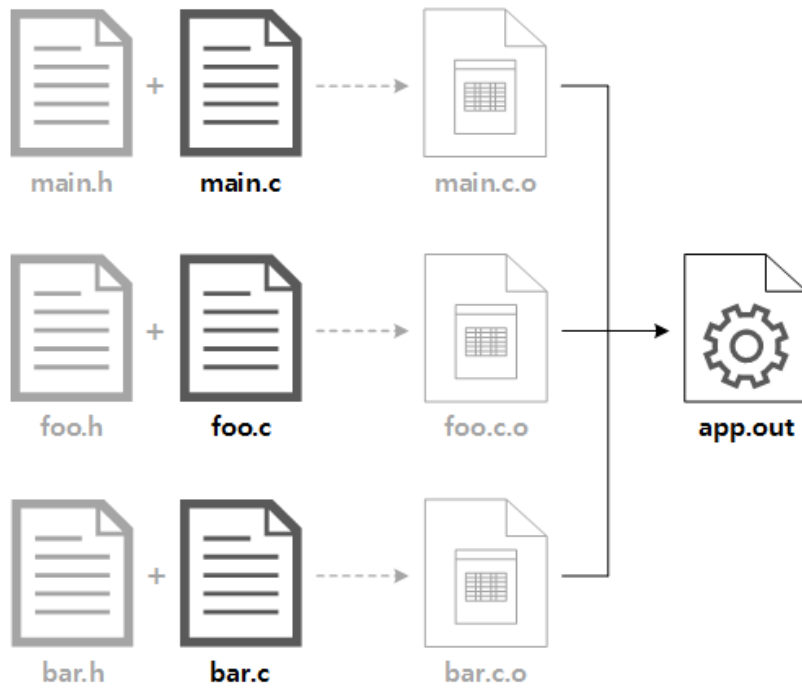
```
tuw@desktop.zip.tuwlab.com:~$ make
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tuw/Workspace/CMake
Scanning dependencies of target app.out
Linking C executable app.out
[100%] Built target app.out
```

▲ CMakeLists.txt 변경 자동 감지

■ 심화: CMake 내부 동작

위의 예제에서 살펴봤듯이, CMake 빌드 스크립트에서 빌드 대상 바이너리 정의 구문(ADD_EXECUTABLE)에 들어가는 내용은 출력 바이너리 이름과 소스 파일 목록이 전부입니다. 그 외의 소스 파일에 포함되는 헤더 파일들과 각 소스 파일을 컴파일한 Object파일들은 명시할 필요가 없으며, CMake 내부적으로 알아서 처리됩니다.

다음 그림에서 흐리게 표시한 부분이 CMake 내부적으로 처리되어서 빌드 스크립트에 명시하지 않아도 되는 부분입니다.



헤더 파일(*.h)은 직접적인 빌드 대상은 아니지만 소스 파일에 포함되어서 해당 Object 파일과 내재적 의존 관계를 만듭니다. CMake는 각 Object파일을 생성하기 전에 소스 파일을 분석하여 어떤 헤더 파일들이 포함되어 있는지 파악하고, 이들 헤더 파일의 변경 여부를 검사하여 필요시 다시 컴파일을 수행합니다.

Object 파일(*.o)은 신경쓰지 않아도 되지만, CMake 내부적으로는 빌드를 수행할 때 자동 생성되는 CMakeFiles 디렉토리 안에 생성됩니다. 빌드 후 지저분하게 소스 파일과 Object 파일이 섞여 있지 않으므로 프로젝트 디렉토리를 깔끔하게 유지할 수 있습니다.

참고로, cmake를 실행하면 프로젝트 디렉토리에 다음과 같은 파일과 디렉토리가 자동으로 생성됩니다.

- Makefile
- CMakeCache.txt
- cmake_install.cmake
- [CMakeFiles]

자동 생성된 파일과 디렉토리들은 언제든지 삭제해도 무방합니다. 즉, GIT으로 버전관리를 한다면, .gitignore 파일에 다음 네 줄을 추가해 주면 됩니다.

```

1 | /CMakeCache.txt
2 | /cmake_install.cmake
3 | /CMakeFiles/
4 | /Makefile
  
```

?

make 명령으로 CMake로 생성한 Makefile을 실행하면 가장 먼저 CMakeLists.txt파일이 변경됐는지 여부를 검사하고, 변경된 경우 Makefile을 다시 생성하여 실행합니다.

다음으로 Makefile에 정의된 각 Target별로 빌드를 수행하는데, 이 때 내부 Build Step에 따라 cmake명령으로 각 Target을 빌드하는 데 필요한 Sub-Makefile을 생성합니다. 이 때 생성되는 Sub-Makefile들도 역시 CMakeFiles 디렉토리 내부에 저장됩니다. 자동 생성되는 Sub-Makefile들도 역시 의존성 검사를 통해 이전에 만들어 뒀던 것을 재활용하거나, 다시 생성합니다.

실제로 CMake로 자동 생성된 Makefile을 뜯어보면, Target 빌드 Recipe에 다음과 같이 Sub-Makefile을 make 명령으로 호출하는 구문 하나만 달랑 써져 있음을 알 수 있습니다.

```
main.c.o:
$(MAKE) -f CMakeFiles/app.out.dir/build.make CMakeFiles/ap
p.out.dir/main.c.o
```

★ 다음 글에서는 본격적으로 CMake 빌드 스크립트인 CMakeLists.txt 파일을 작성하는 방법과, 전형적인 패턴에 대해 다루도록 하겠습니다.

연관글

[CMake 튜토리얼] 3. CMakeLists.txt 기본 패턴 (11278) *4

[CMake 튜토리얼] 2. CMakeLists.txt 주요 명령과 변수 정리 (29178) *1

[Make 튜토리얼] Makefile 예제와 작성 방법 및 기본 패턴 (29680) *2

TAG *

Make, Makefile, CMake, CMakeLists.txt

< Prev

[CMake 튜토리얼] 2. CMakeLists.txt

[Make 튜토리얼] Makefile 예제와 작성 방법

주요 명령과 변수 정리

및 기본 패턴

Next >

FacebookTwitterGooglePinterest

↑↓🖨

이용중인 SNS 버튼을 클릭하여 로그인 해주세요.
SNS 계정을 통해 로그인하면 회원가입 없이 댓글을 남길 수 있습니다.

?

등록

- 일반

PSpice

AVR

Android








Nginx

Apache

Linux
- XE

Python

Security

번호	분류	제목	글쓴이	최근 수정일	조회 수
191	일반	[WSL] Windows Subsystem for Linux - 디스플레이 서버 설정 및 GUI 사용하기 📄	 TUW	2019.01.15	2348
190	일반	[TeraTerm] 명령줄 인수와 공개키 인증으로 간편하게 SSH 접속하기	 TUW	2018.11.06	770
189	일반	[WSL] Windows Subsystem for Linux - SSH 서버 자동 시작 설정하기 📄	 TUW	2018.11.06	1563
188	일반	[WSL] Windows Subsystem for Linux - SSH 서버 세팅하기 📄	 TUW	2018.11.09	2239
187	일반	[WSL] Windows Subsystem for Linux - Bash.exe를 Ubuntu와 유사하게 설정하기 📄	 TUW	2018.11.06	1369
186	일반	[WSL] Windows Subsystem for Linux - 초기 설치와 Ubuntu 배포판 설치 📄	 TUW	2018.11.06	2345
185	일반	[AutoHotkey] 단축키(Hotkey) 스크립트 작성과 자동 시작 등록 📄	 TUW	2018.11.08	1524

번호	분류	제목	글쓴이	최근 수정일	조회 수
184	일반	[AutoHotkey] 소개와 설치 및 기본 설정 - GUI 예시, 기본 에디터 변경 	 TUW	2018.11.11	1355
183	일반	Windows에서 포트 포워딩(Port Forwarding) 설정하기 - Netsh 	TUW	2018.02.03	8105
182	Security	[SSL/HTTPS] Let's Encrypt 무료 SSL 인증서 발급/설치/관리 - certbot 사용법 	 TUW	2019.02.25	11191
181	Security	[SSL/HTTPS] StartSSL/StartCom 상태와 Let's Encrypt로의 이전 1 	 TUW	2018.05.02	3414
180	Linux	[Ubuntu] Windows와 멀티부팅 환경에서 시간이 맞지 않는 현상 해결하기 	TUW	2017.06.08	7016
179	일반	[Windows] 다중 NIC(LAN카드) 환경에서 Routing Table 설정 - route 명령 	 TUW	2018.06.13	15883
178	일반	[CMake 튜토리얼] 3. CMakeLists.txt 기본 패턴 4 	TUW	2019.02.11	11278
177	일반	[CMake 튜토리얼] 2. CMakeLists.txt 주요 명령과 변수 정리 1 	 TUW	2018.05.31	29178
»	일반	[CMake 튜토리얼] 1. CMake 소개와 예제, 내부 동작 원리 	 TUW	2018.06.13	30066

≡ 목록

검색

< Prev **1** 2 3 4 5 6 7 ... 12 Next >