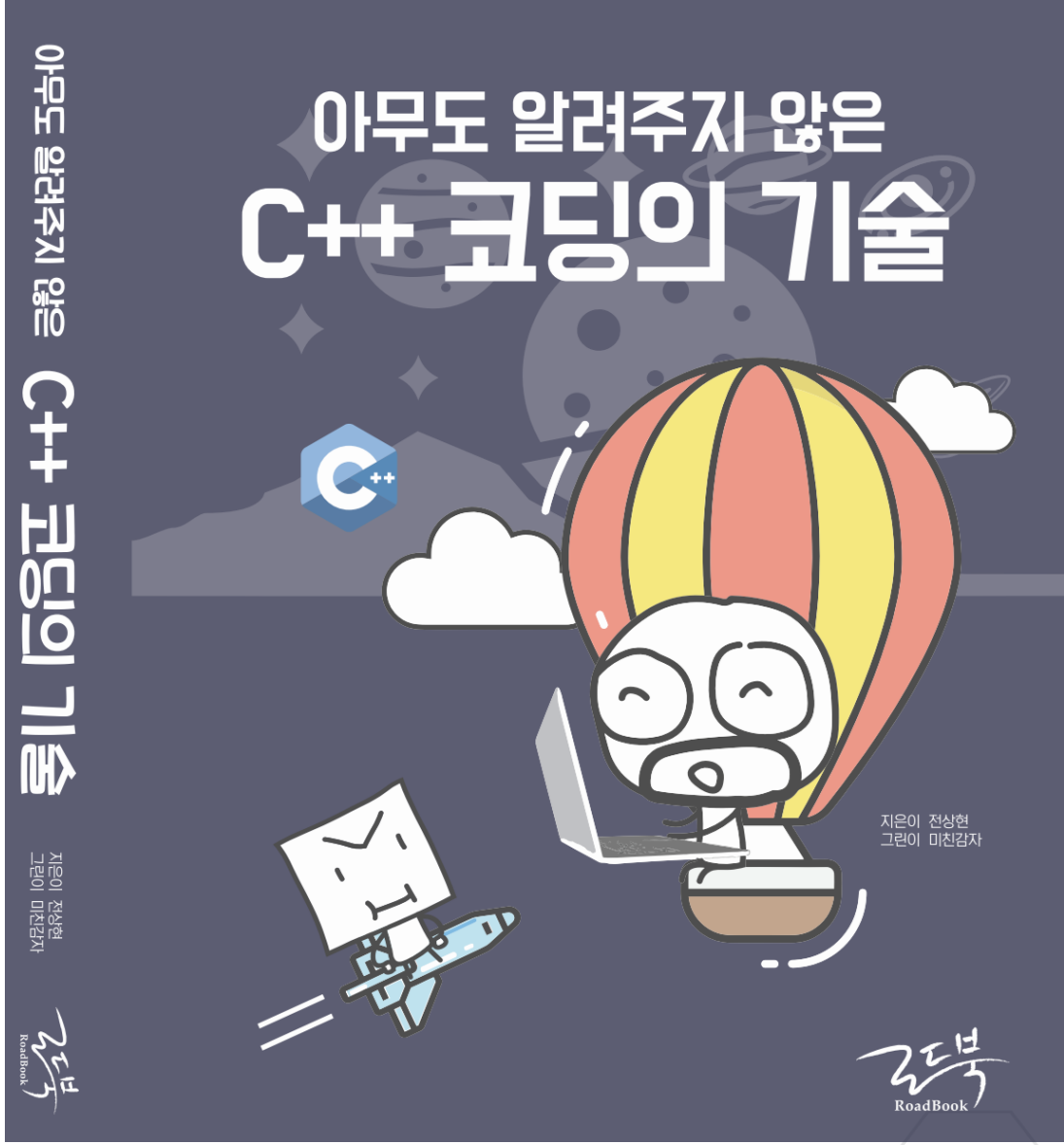
시스템 프로그래밍2

S-개발자 4기 2026-03-10(화)

서울 송파구 동남로 130, 2층 제 4강의실

악성코드검거단 MALWARE ARREST TEAM 대표이사 전상현

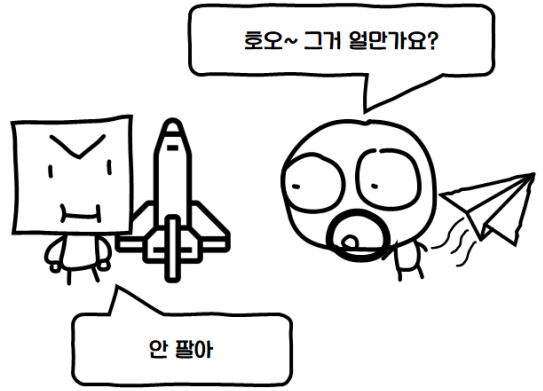


들어가기 전에 수업 개요

과정명	시스템 프로그래밍	강사명	전상현
강의시간	24시간 (4일 x 6시간)		
강의목표	컴퓨터 시스템을 이해한다. 나아가 다양한 시스템을 제어하는 C++ 프로그래밍 기술을 익힌다.		
평가방식	시험 50%, 실습 50%		

강의내용

시간(H)	제목	내용
1H	오리엔테이션	강사 소개 후 간단한 퀴즈와 함께 실행파일의 구조를 이해한다.
2H	변수형과 유니코드	시스템에 존재하는 모든 종류의 변수형을 알아본다. 다국어를 표현하는 문자체계인 유니코드를 이해한다.
3H	개발환경 구축하기	윈도우/리눅스(wsl)에 빌드할 수 있는 환경을 구성한다. 기본적인 개발툴 사용법을 익힌다.
2H	문자열 정복하기	C++과 친해지기 위한 코드를 작성해본다. VisualStudio 디버깅 기술을 배운다.
2H	C++ 컴파일러/링커/디버거 정복하기	유니코드 상호 변환하는 함수를 이해한다. 문자열을 자유자재로 다루는 방법을 배운다.
2H	파일시스템 정복하기	플랫폼별 파일/디렉토리 접근 함수를 이해한다. 파일 시스템을 자유자재로 다루는 방법을 배운다.
2H	데이터 정복하기	STL 자료구조를 이해하고, 실무 응용 기술을 배운다. XML/JSON/INI 등을 자유자재로 다루는 방법을 배운다.
2H	메모리 정복하기	C++의 꽃, 스택 메모리의 장단점을 알아본다. 4가지 영역의 메모리를 자유자재로 활용한다.
2H	소켓 정복하기	UDP/TCP 통신, DATAGRAM/STREAM의 차이를 이해한다. 소켓 통신 시스템을 자유자재로 다루는 방법을 배운다.
6H	**최종실습**	**앞에서 배운 지식과 기술을 활용하여 미니 프로젝트를 진행한다.**



코딩 실력이 곧 우주선이다.

우주같이 광활한 컴퓨터 세계로 여행을 떠나자.

<참고서적>

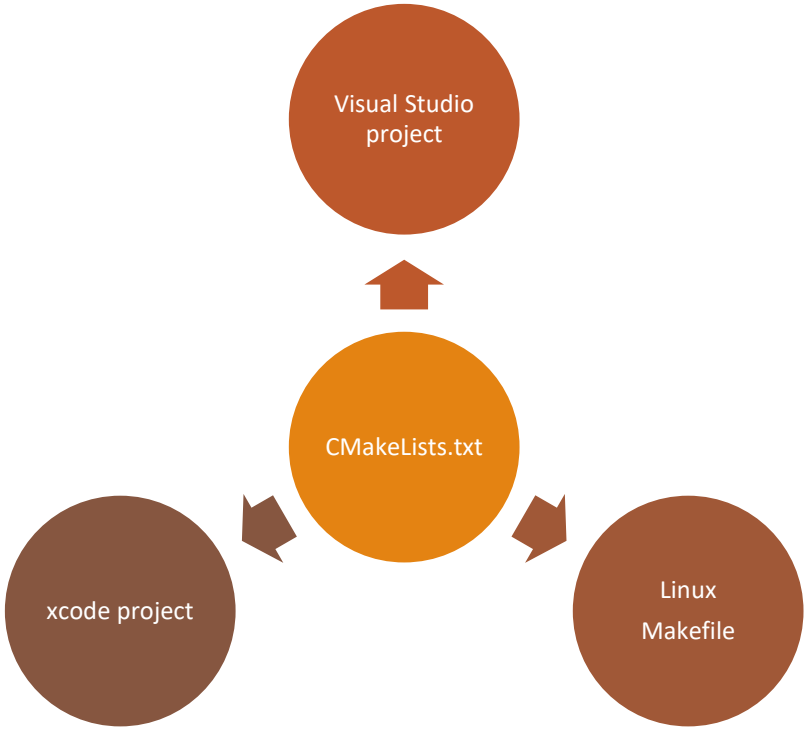
크로스플랫폼 핵심모듈 설계의 기술(2018) – 전상현, 로드북

아무도 알려주지 않은 C++ 코딩의 기술 (2023) – 전상현, 로드북

다양한 플랫폼의 개발환경

개발 플랫폼은 크게 윈도우/유닉스(리눅스)/맥 으로 분류할 수 있습니다.

플랫폼	주요개발툴	특징
윈도우	VisualStudio	명실공히 최강의 디버깅 및 코드 편집도구
리눅스	VI / VisualStudio Code	제한적인 디버깅, 약간 불편한 편집기능
맥	Xcode	안정적인 디버깅 및 코드 편집도구, 약간 느림



Cppcore의 CMakeList.txt 파일을 참고해보자.

CMakeLists.txt

CMakeList.txt 라는 파일명으로 빌드 스크립트를 작성합니다.

cmake <CMakeLists.txt 경로>

실행하면 빌드를 위한 플랫폼별 중간 산출물이 생성됩니다.

이후 동일 디렉토리에서 다음 명령을 수행하면 빌드가 진행된다.

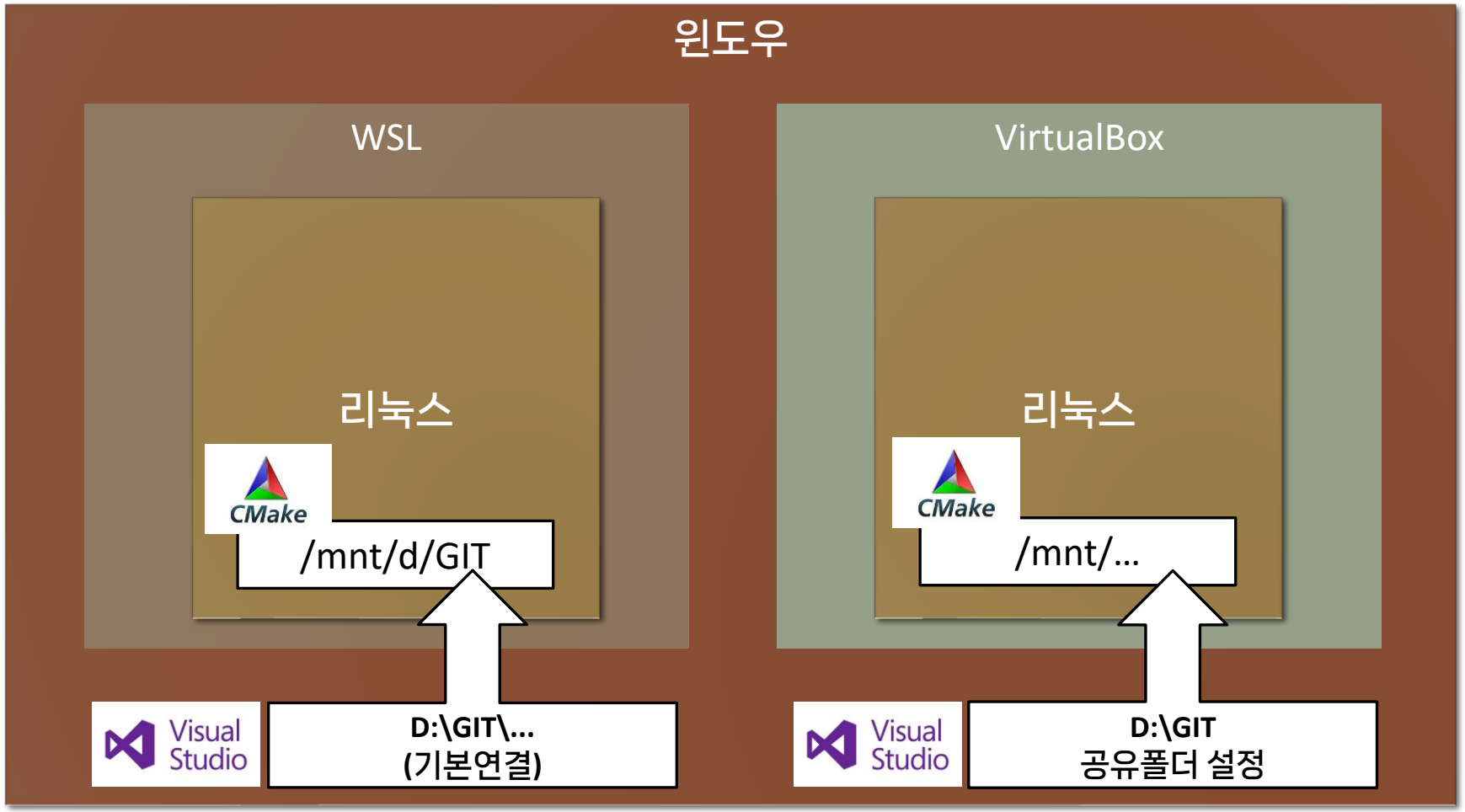
cmake --build <앞의 명령을 수행한 경로>

Ex) 사용예제
cmake ../Src

cmake --build .

게스트 OS의 디스크 공유 후, 공유폴더에서 소스 코드를 작성합니다.

리눅스 계열에서는 코딩과 디버깅을 VS로, 컴파일만 cmake로 해서 윈도우 개발 툴의 장점만 가지면 됩니다.



개발환경 구축하기

앞으로 여러분들은 많은 프로젝트를 관리하게 될 것입니다.
같은 코드를 반복적으로 만들면서 인생을 낭비하고 싶지 않다면, 내가 만든 모든 소스코드들을 소중하게 관리해봅시다.

가장 이상적인 관리 방법은 다음과 같은 구조로 구성하는 것입니다.

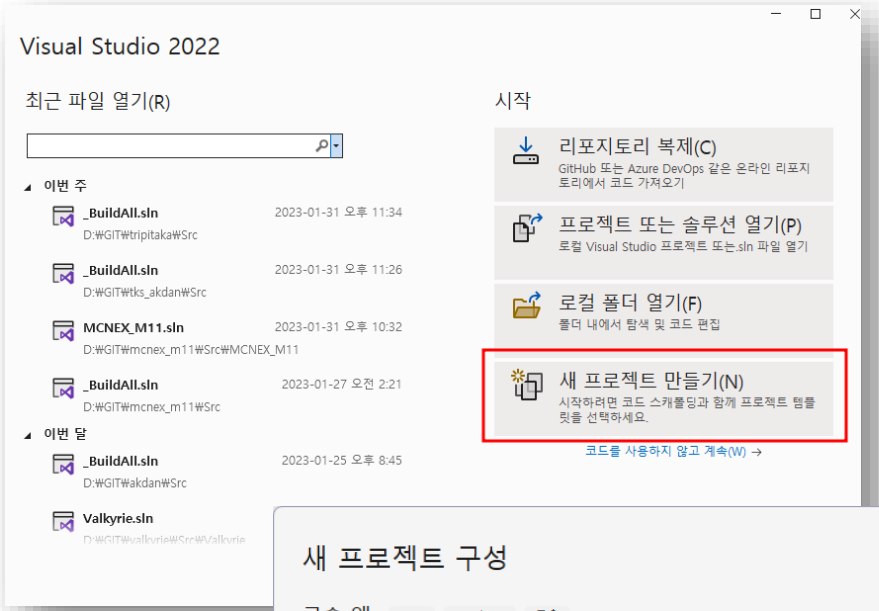
D:		
↘	GIT	
	↘	cppcore
		cryptography
		network
		secure-coding
		system-programming
		....

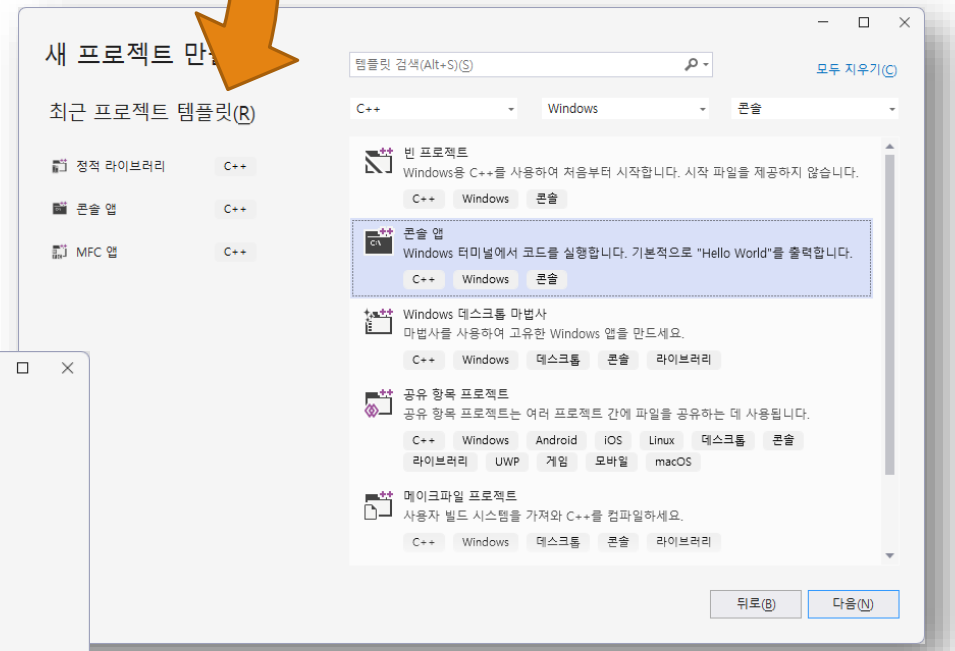
각각의 디렉토리에는 하나 이상의 솔루션과 프로젝트들이 존재합니다.
솔루션은 여러 프로젝트의 집합으로 **VisualStudio**에서는 **sln** 파일과 **vcxproj** 파일로 구분됩니다.
즉 각 디렉토리는 다음과 같이 솔루션과 프로젝트가 들어가게 합니다.

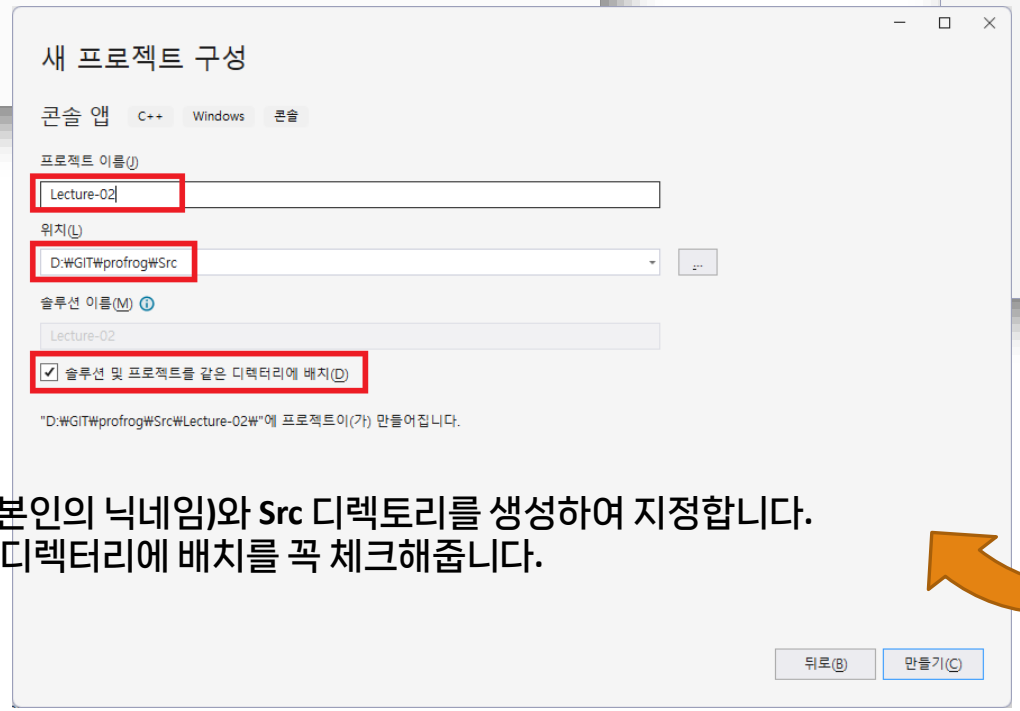
D:\GIT\system-programming			
↘	Build	...	빌드 산출물(*.exe, *.dll, *.pdb 등)
	Output	...	빌드에 사용되는 임시파일들 (*.o)
	Src		솔루션 파일이 들어갈 곳 (_BuildAll.sln)
	↘	Lecture01	프로젝트 파일과 소스코드가 존재
		Lecture02	〃
		Lecture03	〃
		...	

이렇게 하면 아무리 많은 프로젝트를 만들더라도 한눈에 구분할 수 있을 겁니다.
또한 각 디렉토리는 **git** 저장소로도 관리하기가 쉽습니다.

먼저 콘솔 앱 프로젝트를 만들어보겠습니다.

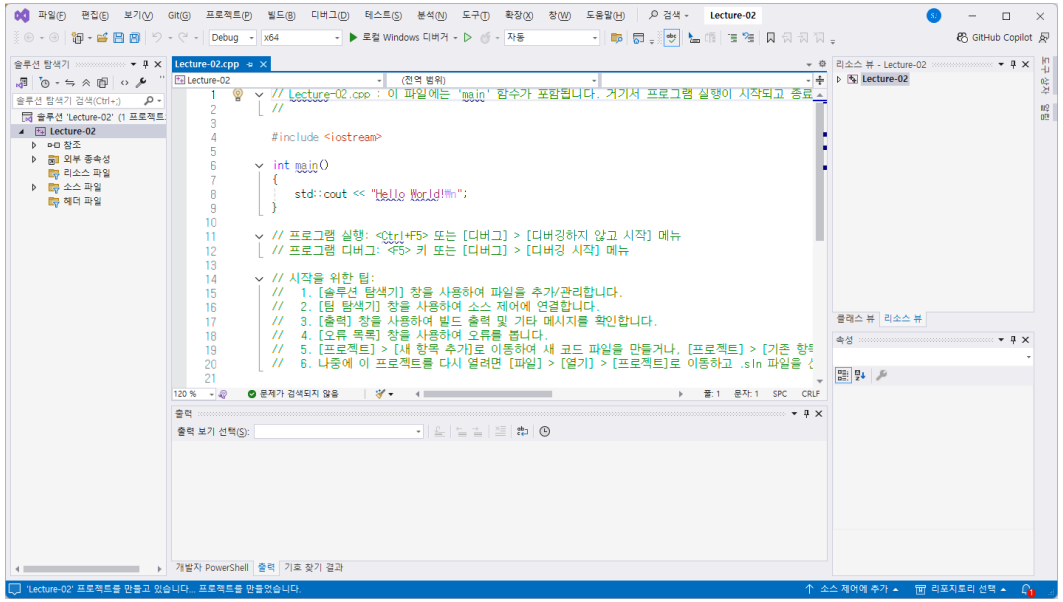


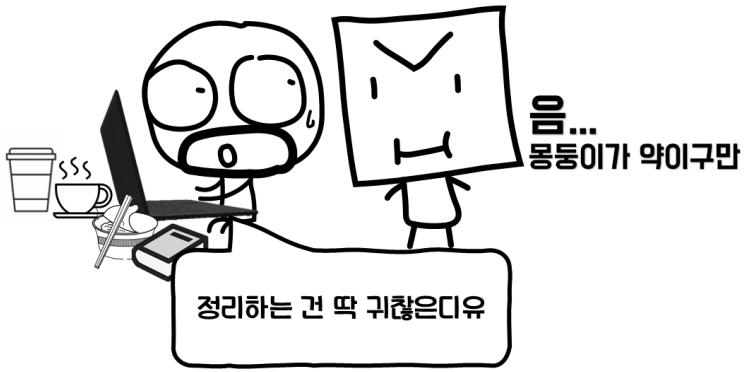




저장소 이름이 될 상위 디렉토리(본인의 닉네임)와 Src 디렉토리를 생성하여 지정합니다.
또한, 솔루션 및 프로젝트를 같은 디렉터리에 배치를 꼭 체크해줍니다.

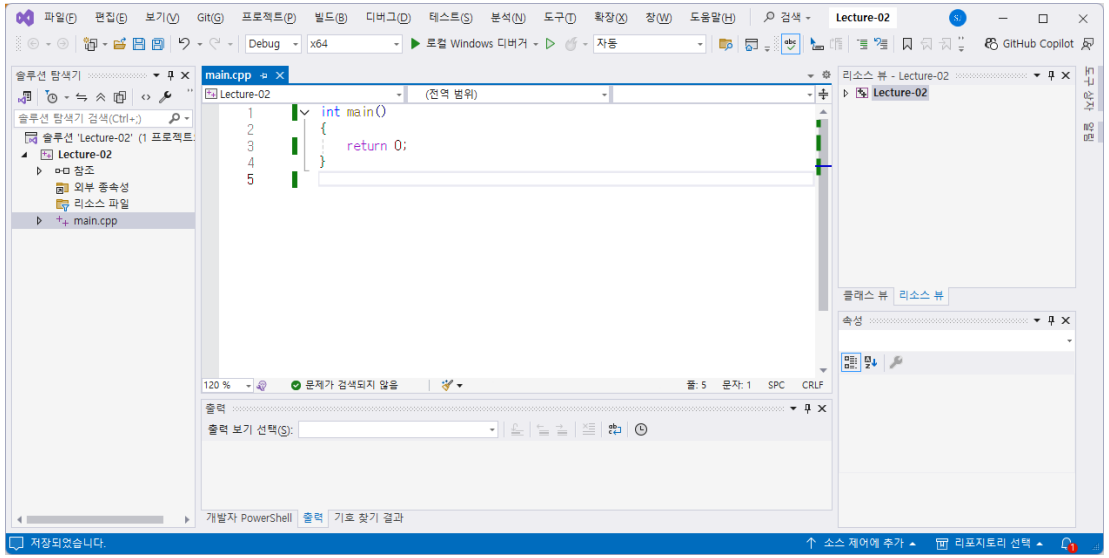
소스 관리 처음 시작은 깨끗하게 비우는 것 부터



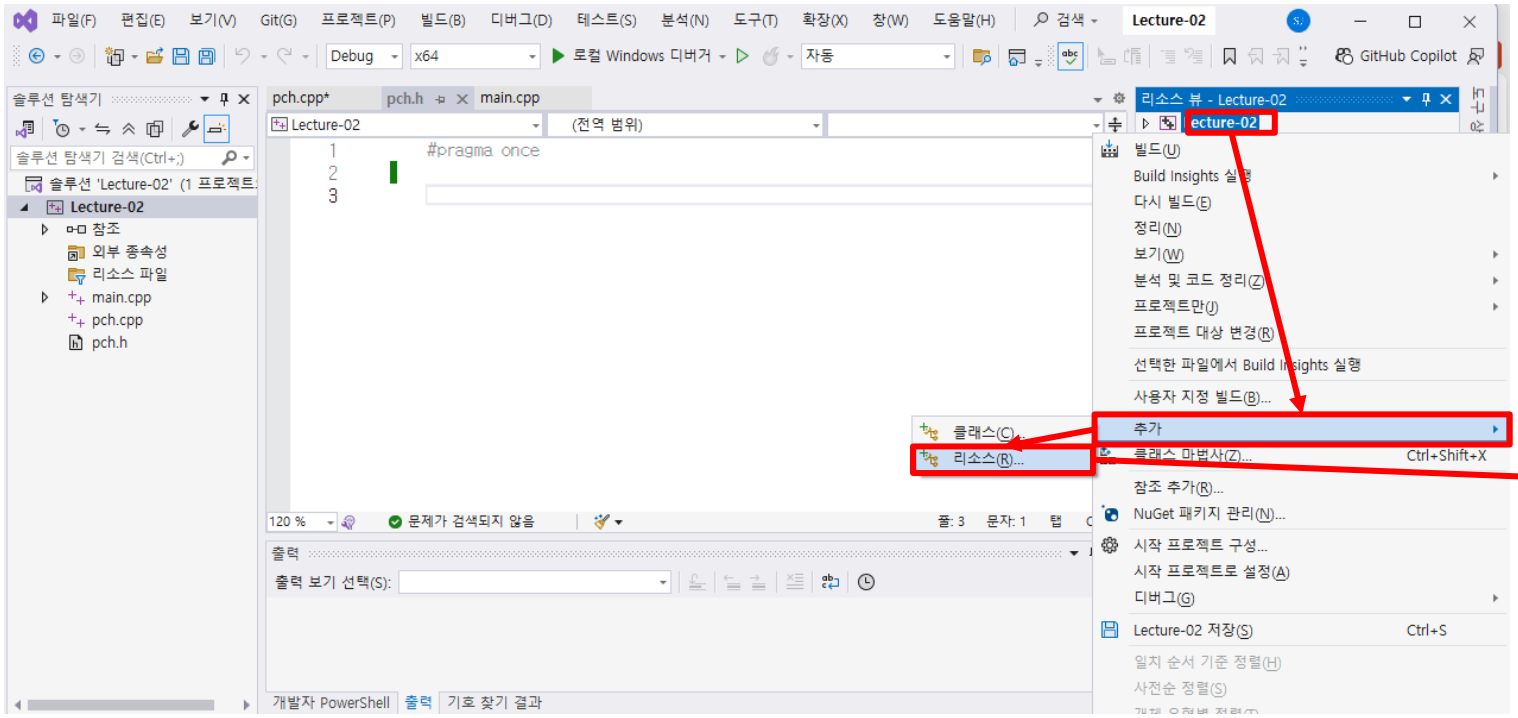


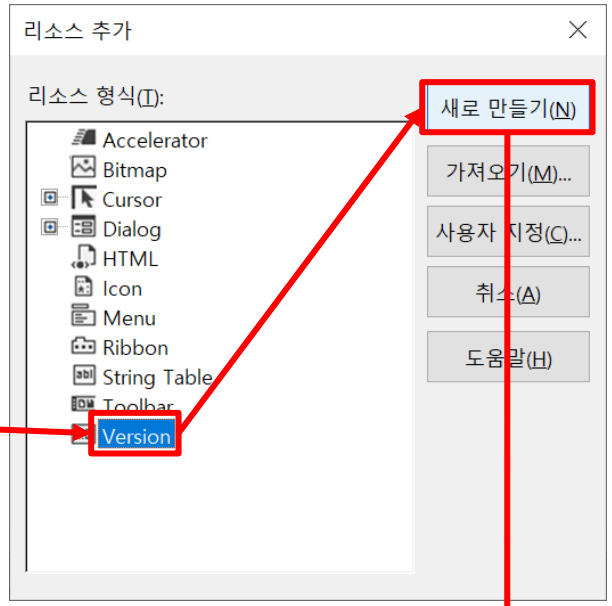
내 프로젝트에 내가 모르는 글자는 있을 수 없다!

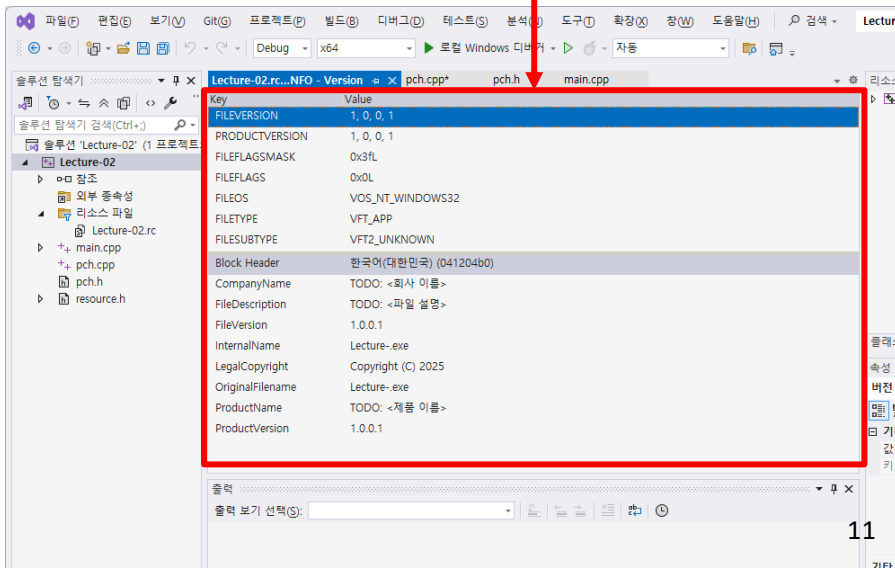
1. 모든 것을 깨끗하게 정리하고 시작하자.
2. main 함수가 포함된 cpp 파일은 main.cpp 으로 이름변경



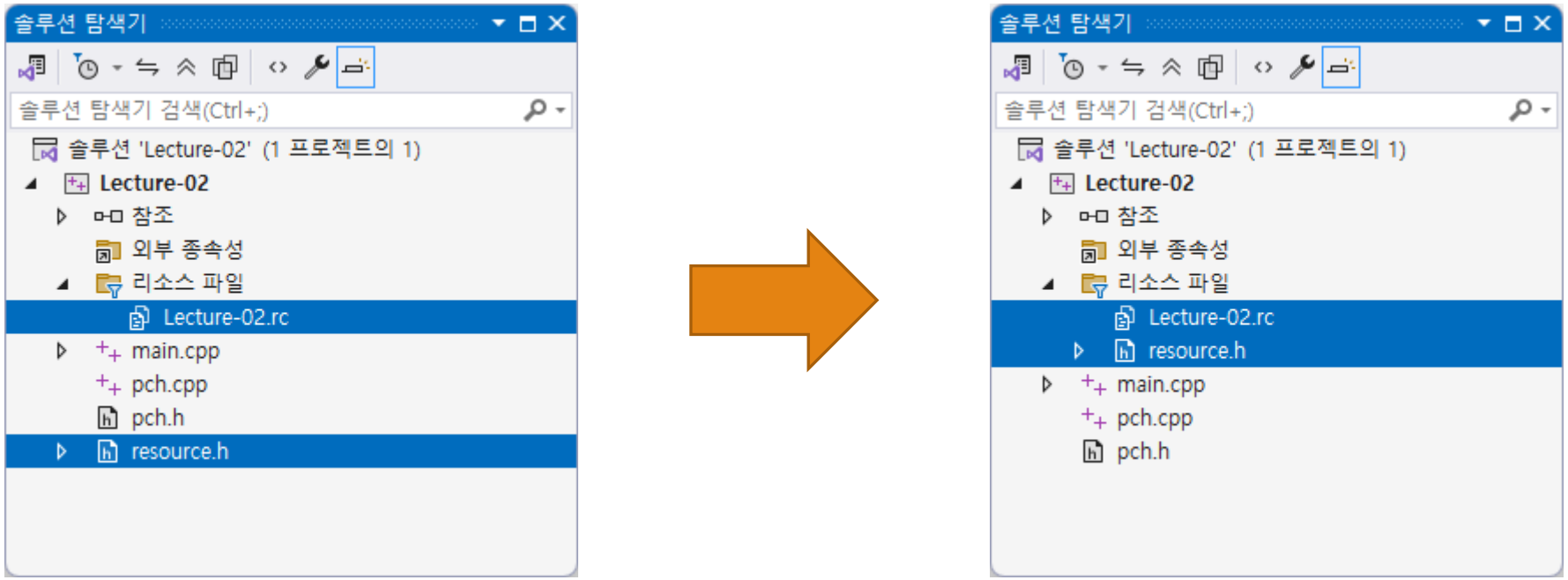
리소스 파일 추가하기







위의 그림처럼 [리소스 뷰] 창에서 리소스를 추가할 수 있습니다. 이어서 Version을 선택하고 [새로 만들기]하면 됩니다. 그럼 버전에 관한 정보를 기입할 수 있는 창이 나타납니다.

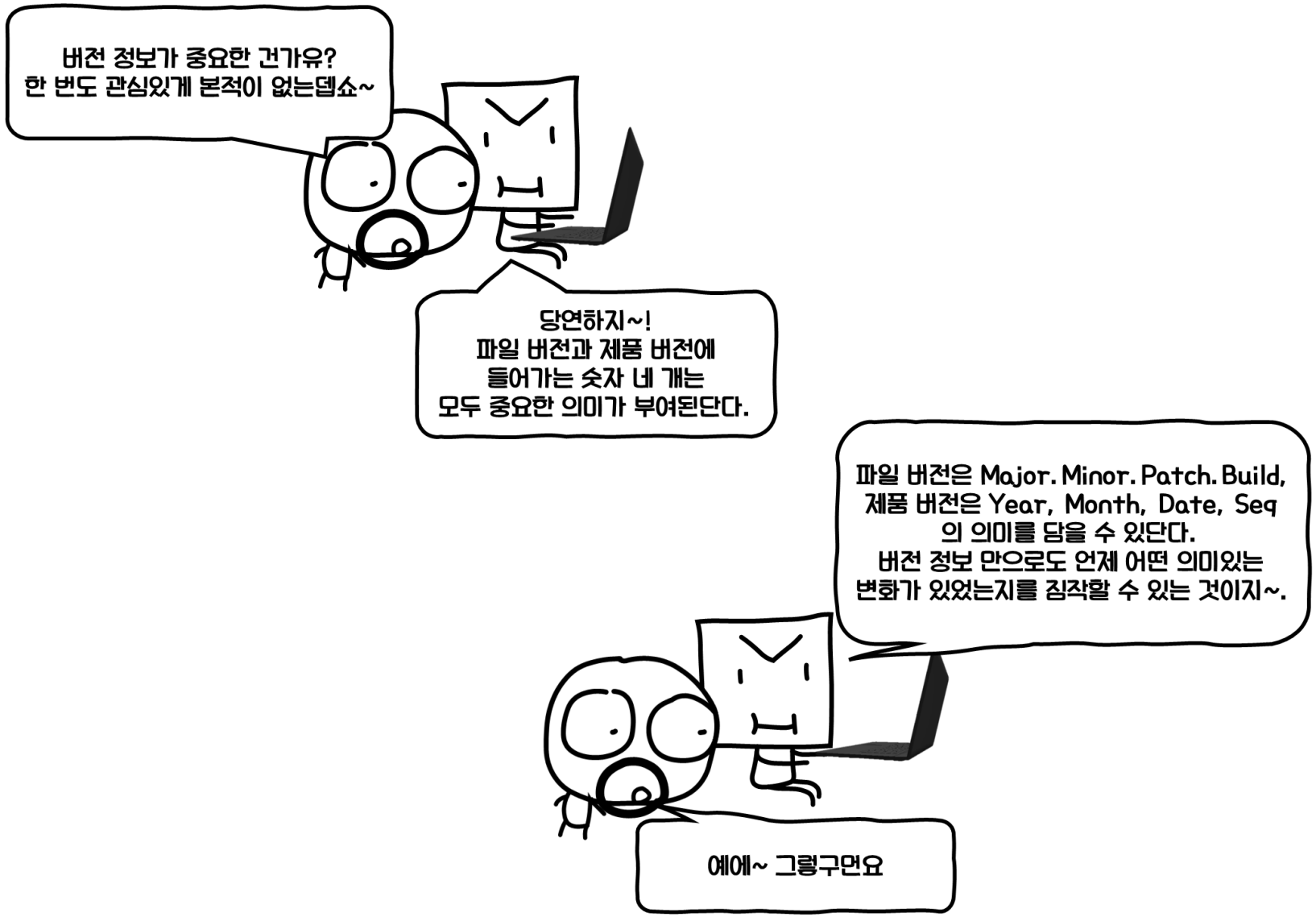


이제 [솔루션 탐색기]를 보면 리소스 관련 파일도 두 개(resource.h, Lecture-02.rc) 추가된 것을 알 수 있습니다.

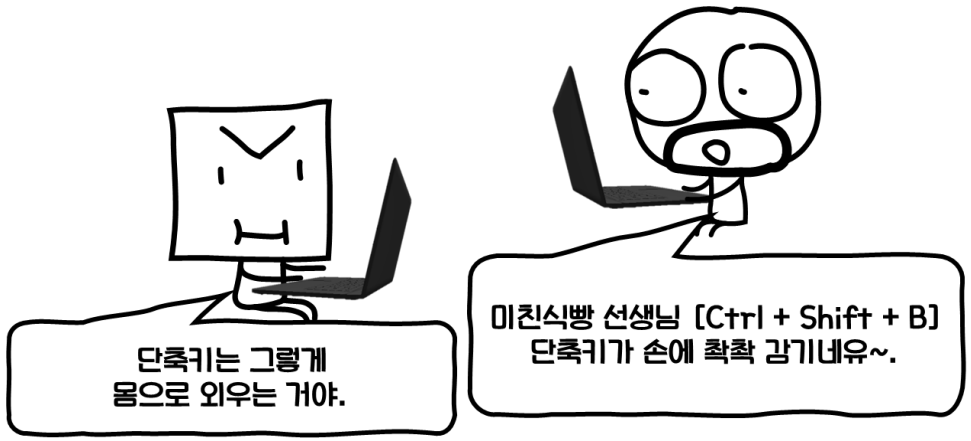
이 파일들은 우리가 직접 손볼 일은 없으므로 리소스 필터 안에 들어가도 상관없습니다.

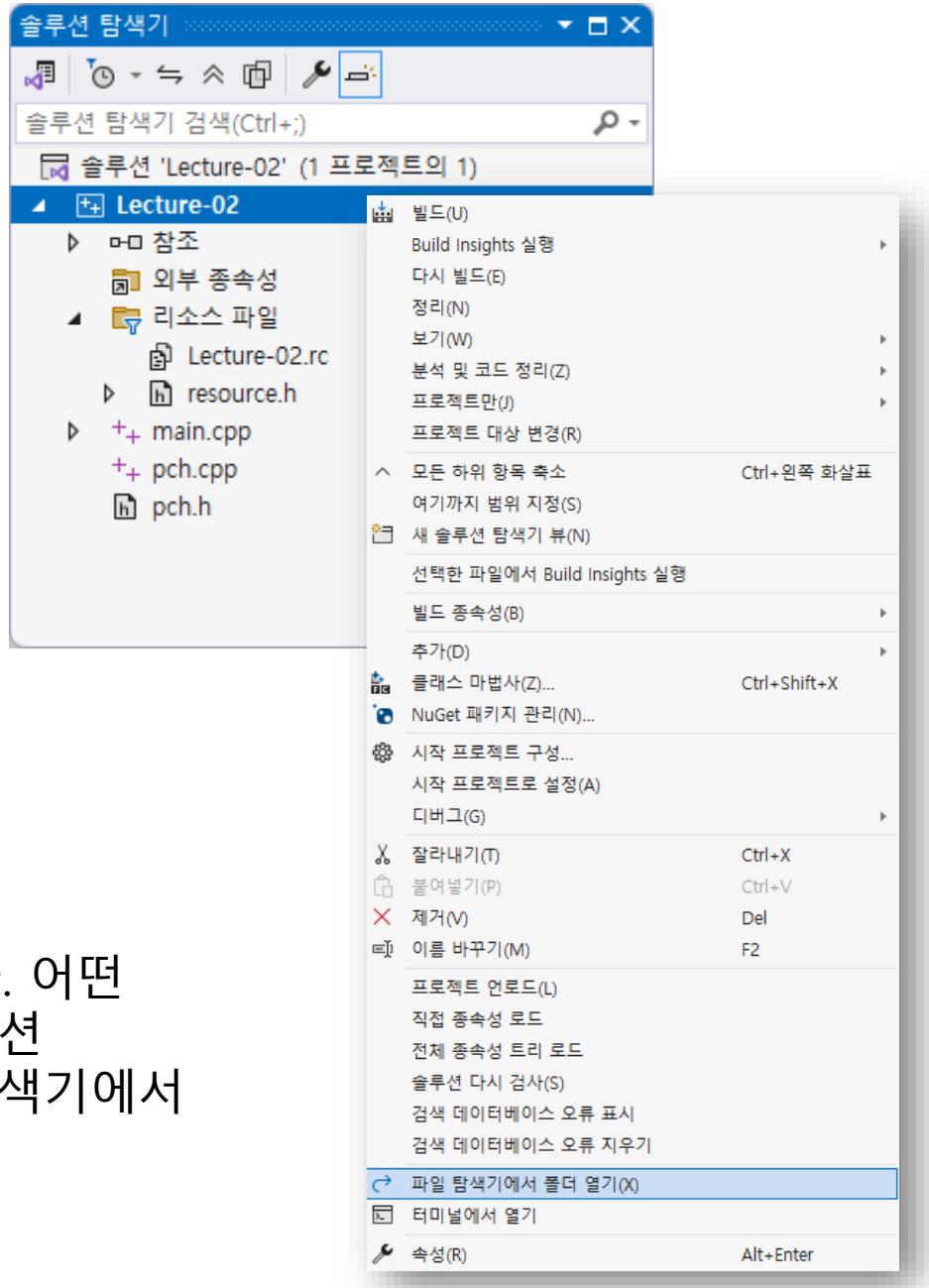
저 같은 경우는 resource.h 파일을 [리소스 파일] 필터에 잡아넣었습니다. 왼쪽 그림에서 오른쪽 그림처럼 된 것이죠.

리소스 파일 추가하기


버전 정보가 중요한 건가유?
한 번도 관심있게 본적이 없는뎁쇼~
당연하지~!
파일 버전과 제품 버전에
들어가는 숫자 네 개는
모두 중요한 의미가 부여된단다.
파일 버전은 Major. Minor. Patch. Build,
제품 버전은 Year, Month, Date, Seq
의 의미를 담을 수 있단다.
버전 정보 만으로도 언제 어떤 의미있는
변화가 있었는지를 짐작할 수 있는 것이지~.
예에~ 그렇구먼요

CTRL + SHIFT + B를 눌러서 빌드를 수행해 봅시다.



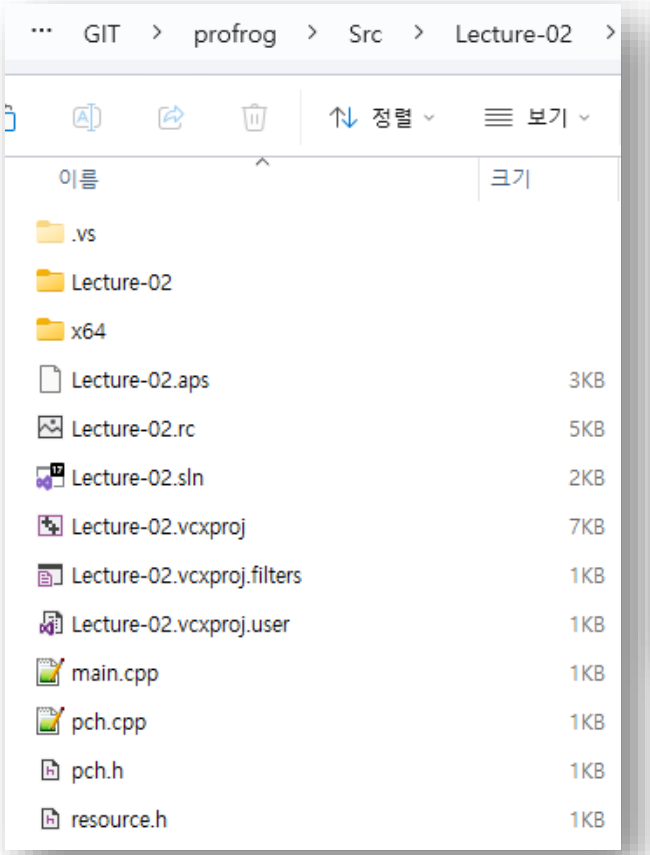


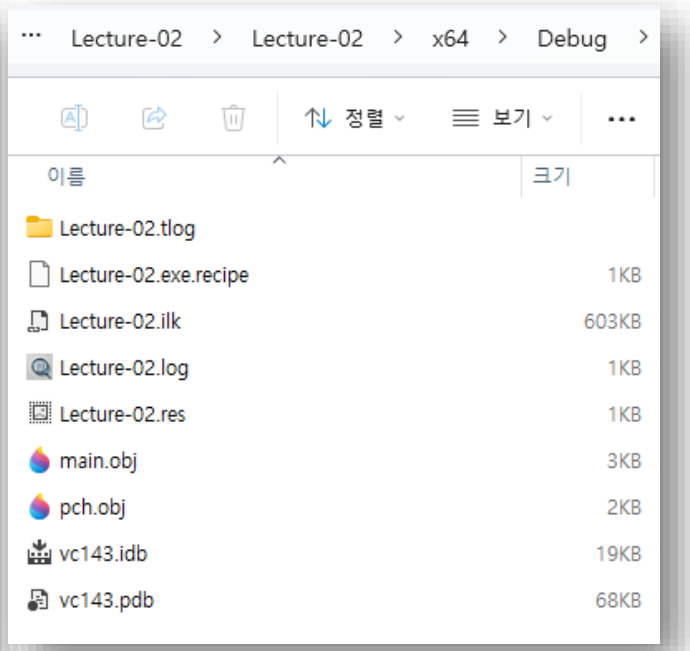
컴파일에 성공하면 프로젝트 디렉토리로 가봅니다. 어떤 파일들이 있는지. 이때 비주얼 스튜디오라면 [솔루션 탐색기] -> [프로젝트 이름] 오른 클릭 -> [파일 탐색기에서 폴더 열기]를 누르면 바로 탐색기가 뜹니다.

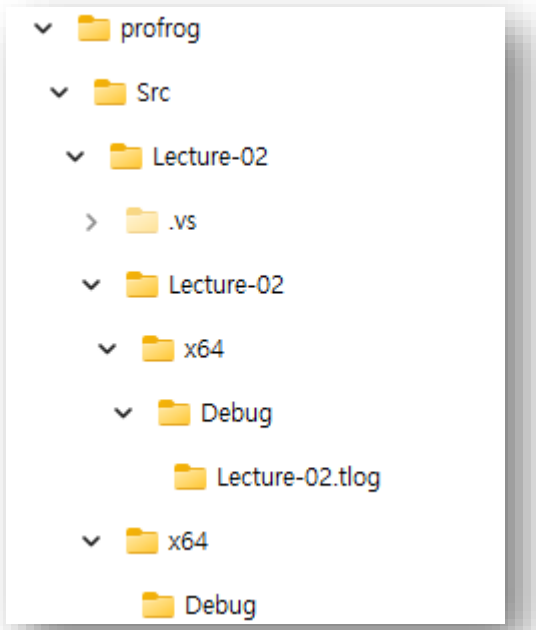
디렉토리를 찾아가보면 뭔 파일들이 잔뜩 있습니다.

우리가 원하는 exe는 어디에 있는지 모르겠고 aps 파일과 vcxproj, filter, user가 생겨났습니다.
하위 디렉토리에 Debug가 있어 들어가보면 더 알 수 없는 파일들이 나옵니다.

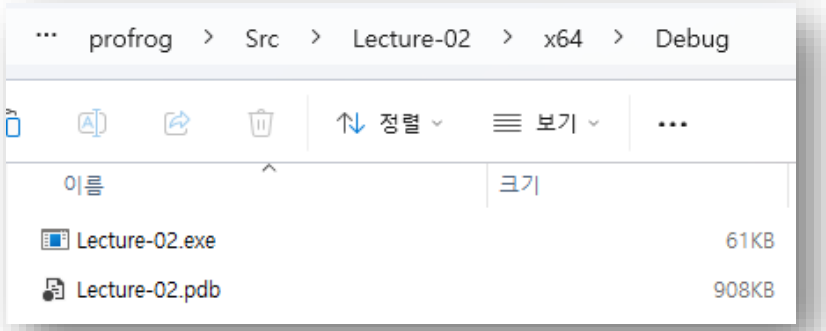
문제는 여기에도 빌드된 exe 파일이 없다는 겁니다.





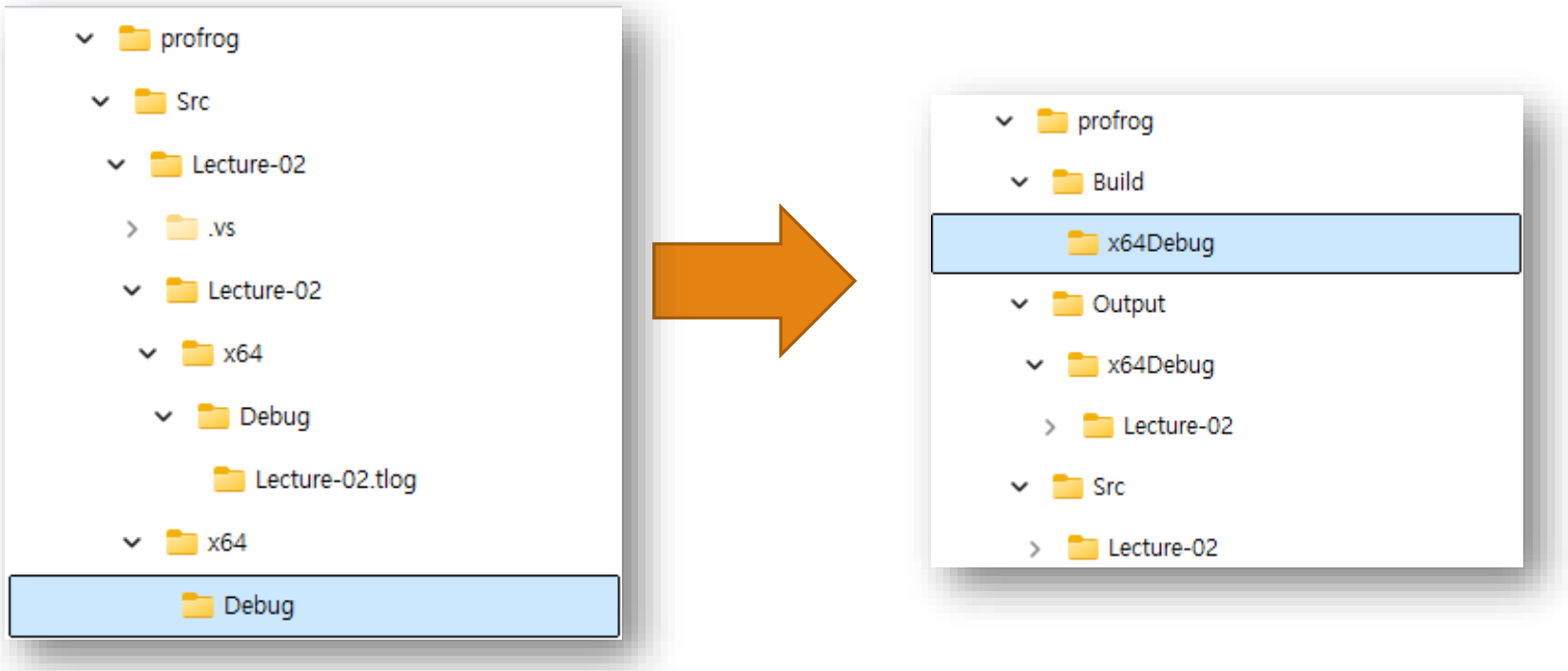


그럼 어디에 있느냐? 바로 여깁니다.



얼핏 보더라도 정리가 안되었다는 것을 알 수 있습니다. 우리가 관심있는 파일은 [소스 코드]이거나 [최종 빌드 산출물]인데 좀 깔끔하게 관리되면 좋겠습니다. 어떤 형태가 좋을까요?

자주 사용해 본 사람들은 알겠지만 오픈소스들은 대체로 코드 관리 구조가 비슷합니다. 우리도 같은 방식으로 정리해봅시다.



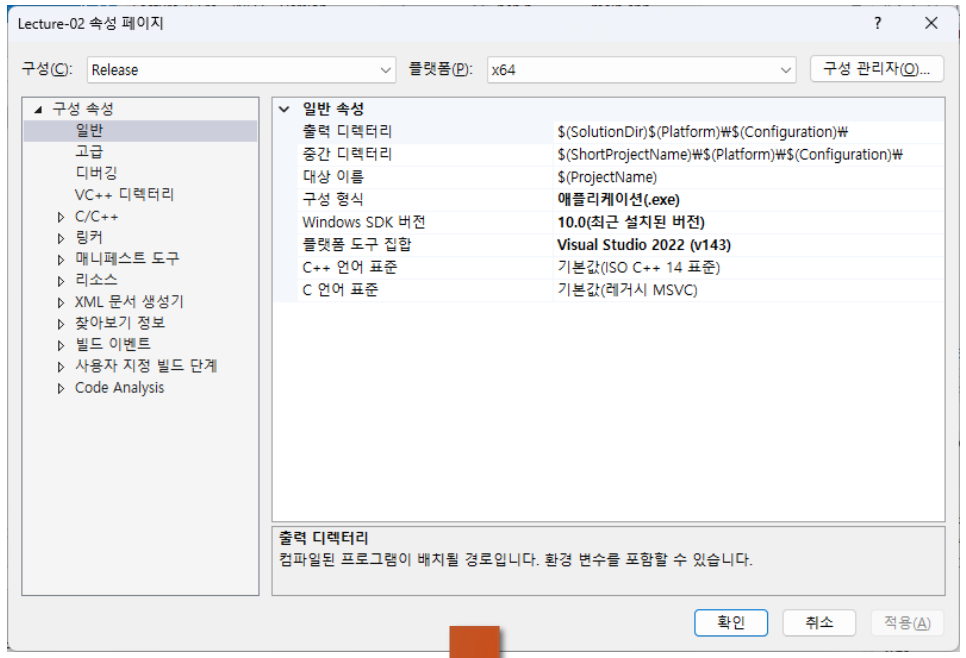
소스 관리 빌드된 산출물 경로 관리하기

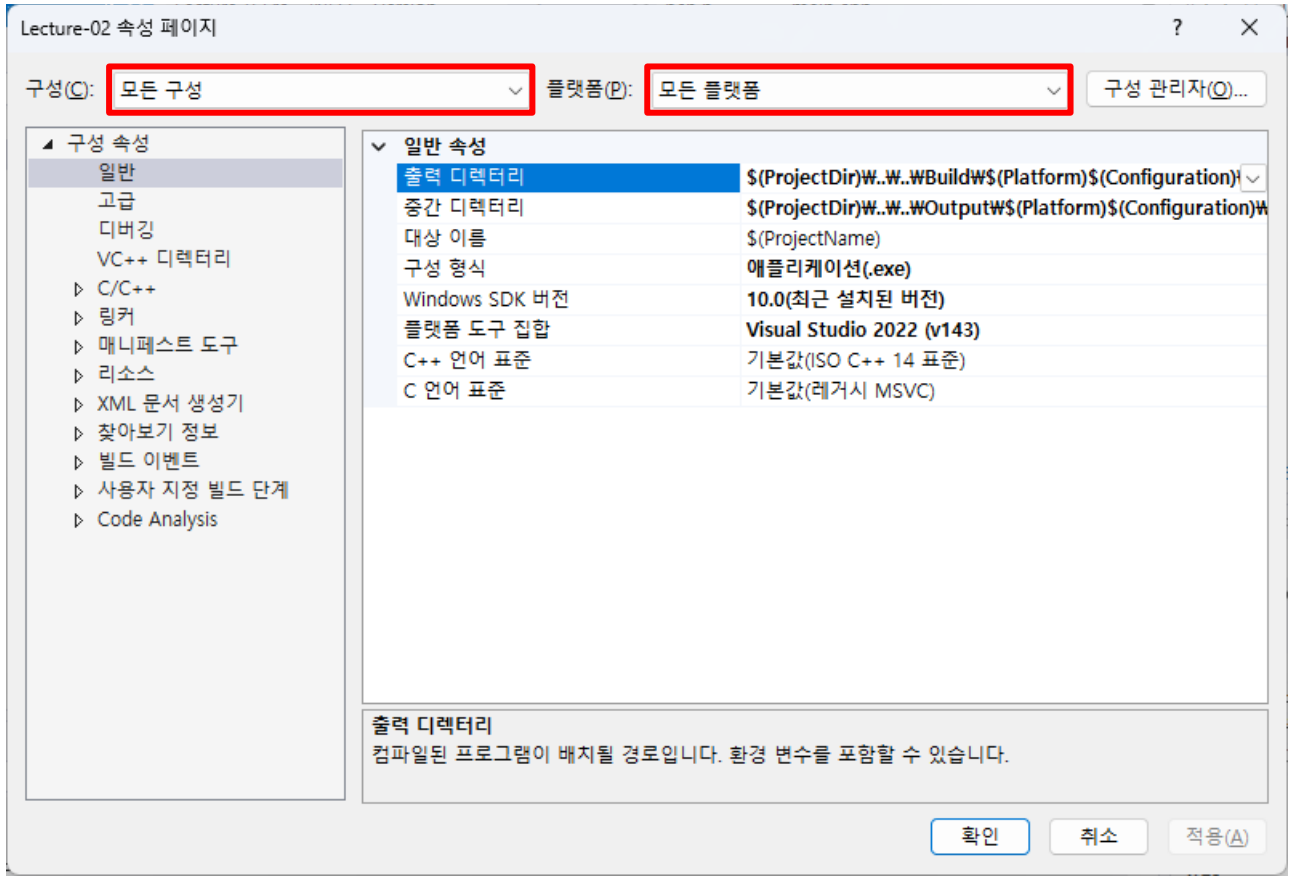
비주얼 스튜디오에서 [프로젝트 속성]을 띄워보면 다음과 같은 화면이 나타납니다.

[구성 속성] -> [일반] 탭에서 출력 디렉터리와 중간 디렉터리를 각각 다음과 같이 수정하면 됩니다.

이때 주의 사항은 반드시 ***[모든 구성]***과 ***[모든 플랫폼]***을 선택해서 한 번에 변경하세요.

출력 디렉터리	$(ProjectDir)\..\..\Build\$(Platform)$(Configuration)\
중간 디렉터리	$(ProjectDir)\..\..\Output\$(Platform)$(Configuration)\$(ProjectName)\



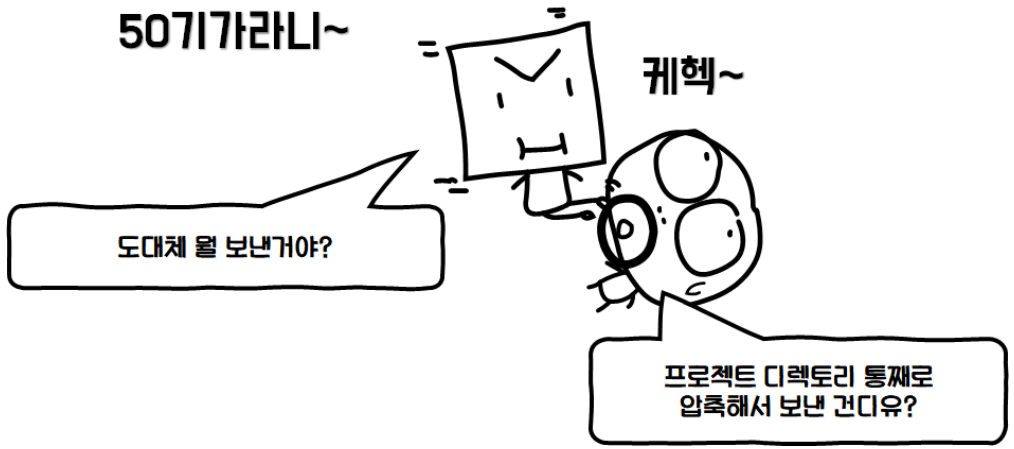


소스코드를 과제로 제출할 때 주의하기

위 그림을 보면 다양한 형식의 파일이 있습니다. 거기다 .vs라는 숨겨진 디렉터리도 눈에 띕니다. 이 중에서 우리가 관리해야 할 파일은 무엇일까요? 결론부터 말하자면 다음과 같습니다.

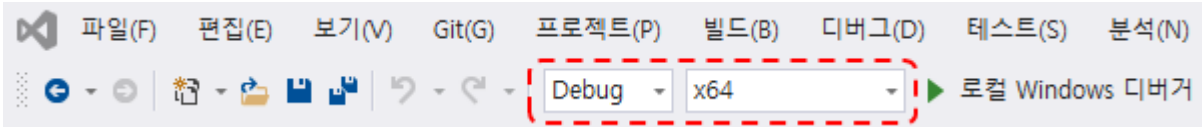
구분	이름	필요	설명
모든 소스코드	*.h, *.cpp, *.hpp, *.c	O	C++ 및 C언어 소스파일
VC 프로젝트 파일	*.vcxproj	O	Visual Studio 프로젝트 파일
VC 프로젝트 파일필터	*.vcxproj.filters	O	Visual Studio 프로젝트 파일트리 폴더 구분정보
VC 프로젝트 유저설정	*.vcxproj.user	X	Visual Studio 프로젝트 런타임 실행 정보
VC 솔루션 파일	*.sln	O	Visual Studio 프로젝트 묶음 파일
VC 캐시 디렉터리	.vc	X	Visual Studio 가 사용하는 임시 디렉터리
CMake 설정 파일	CMakeLists.txt	O	리눅스나 맥에서 빌드할 때 사용하는 스크립트

필요한 파일만 추려서 보관합니다.
GIT이든 ZIP이든 말이죠.
특히 .vs 디렉토리에는 수GB의 불필요한 데이터를 포함하니 더 주의하세요.



개발툴 익히기

비주얼 스튜디오 상단 메뉴바에 보면 다음과 같은 선택창이 있습니다

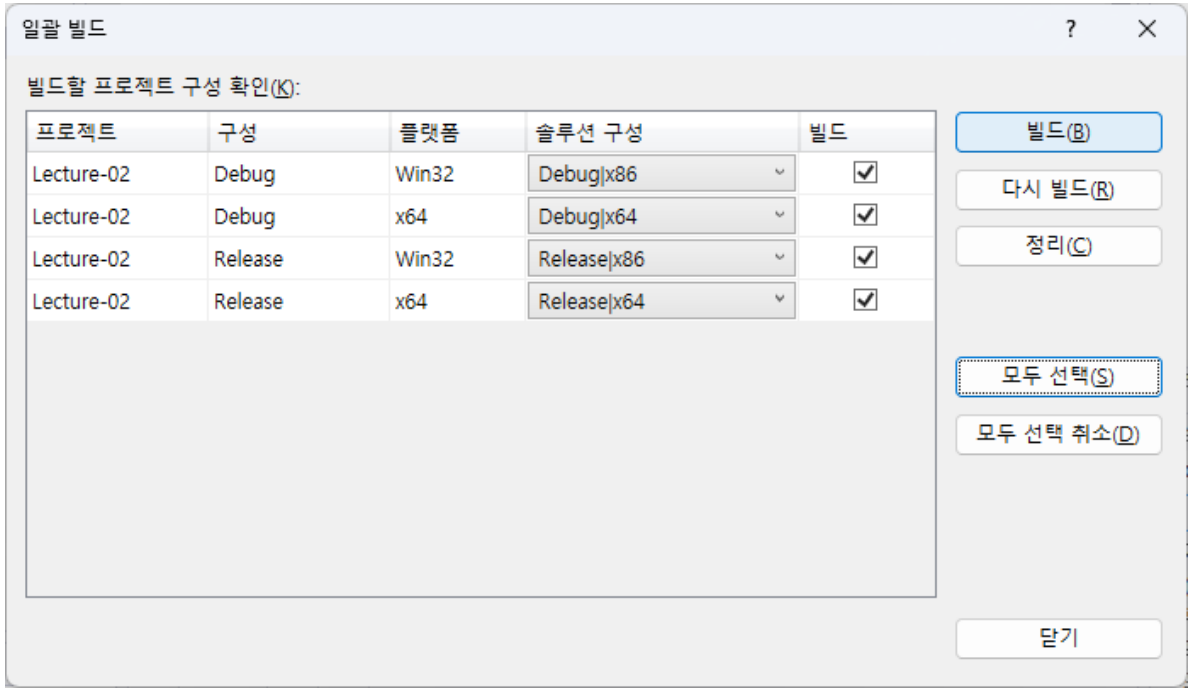


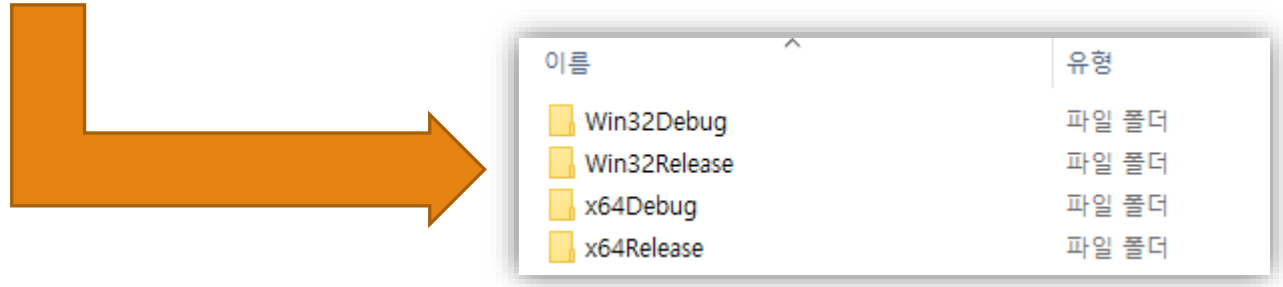
구성 옵션	설명
Debug	디버깅 정보가 포함된 바이너리 빌드
Release	실행에 필요한 정보만 포함한 바이너리 빌드

플랫폼 옵션	설명
Win32	32비트 CPU에서 구동가능한 바이너리 빌드
x86	Win32와 같음, x64 네이밍와 통일하기 위함
x64	64비트 CPU에서 구동가능한 바이너리 빌드

	Debug	Release
Win32	Win32Debug	Win32Release
x64	x64Debug	x64Release

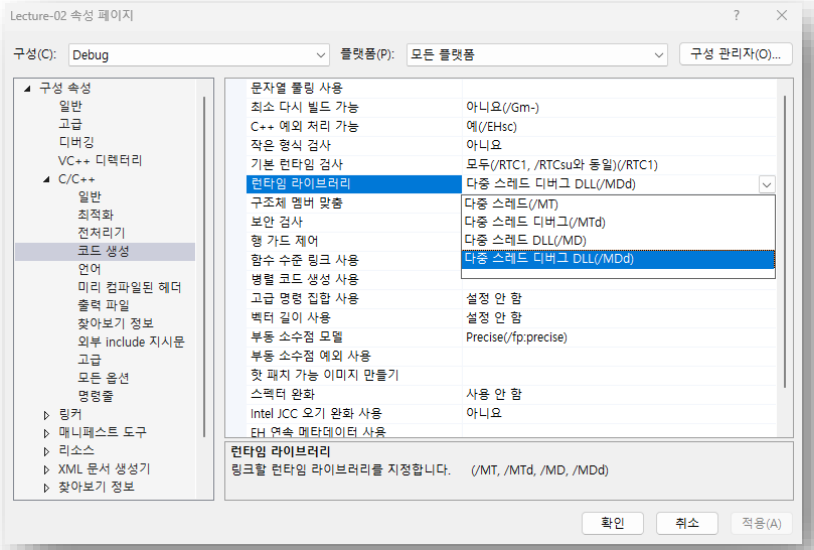
여러 빌드 옵션을 동시에 빌드하는 기능이 있습니다. [메뉴] -> [빌드] -> [일괄 빌드]를 선택하면 다음과 같은 창이 나옵니다.





당초 계획인 8개가 아닌 4개만 나왔네요. 곧 늘어나게 될 겁니다.

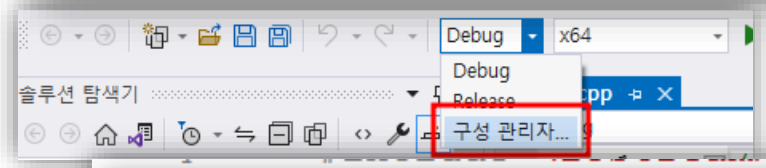
[속성] -> [C/C++] -> [코드 생성] -> [런타임 라이브러리]에 가보면 생소한 용어들이 나옵니다.

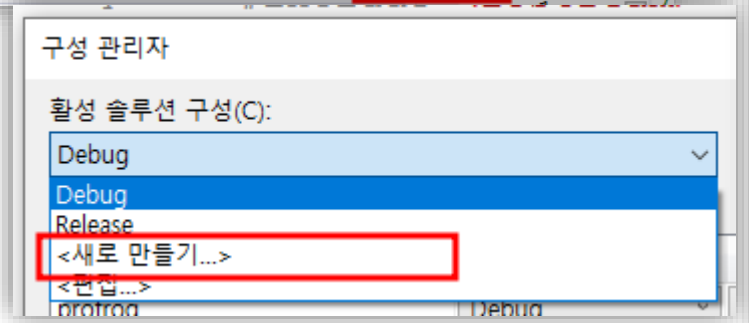


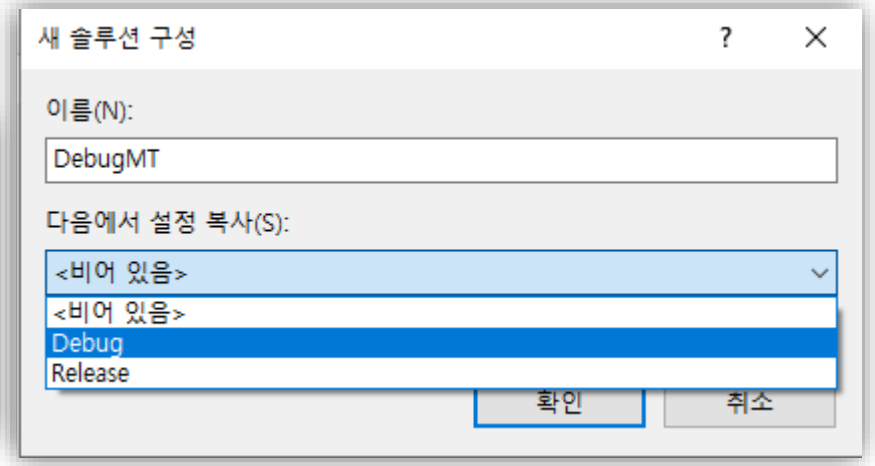
	Debug	Release
다중 스레드 DLL	MDd	MD
다중 스레드	MTd	MT

기본으로 생성된 Debug와 Release는 각각 MDd와 MD 방식으로 빌드됩니다.

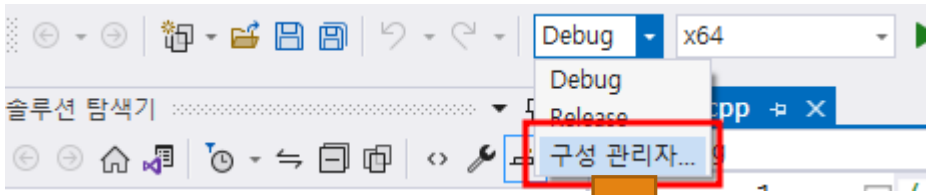
여기에 MTd와 MT 방식의 새로운 설정을 추가하여 4가지 방식의 빌드를 모두 구성해보겠습니다.



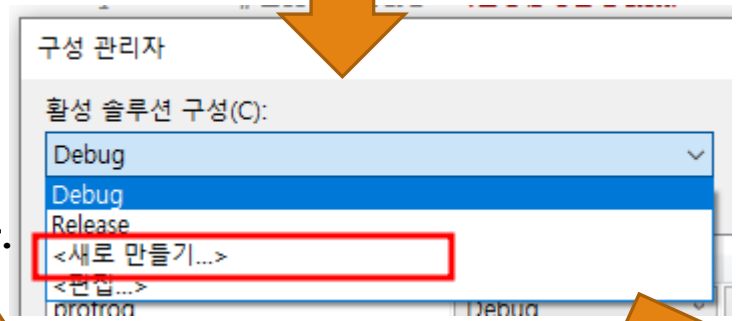




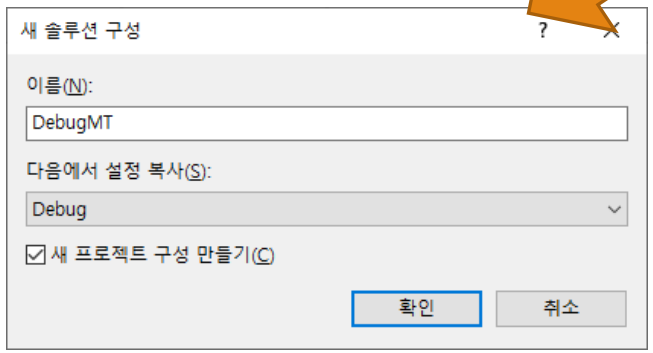
이제 DebugMT와 ReleaseMT 옵션을 추가합니다.



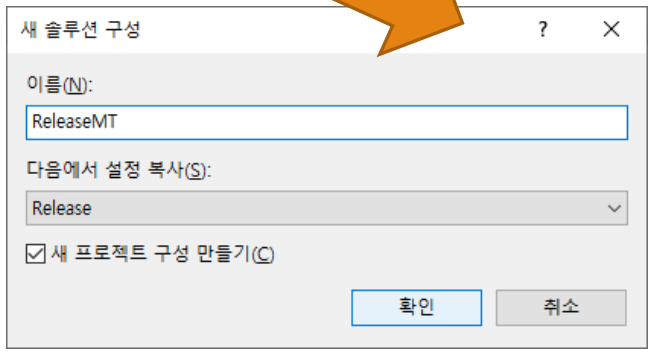
구성 관리자의 새로 만들기를 눌러서
DebugMT와 ReleaseMT를 각각
Debug와 Release로부터 복사해 만듭니다.



(1)

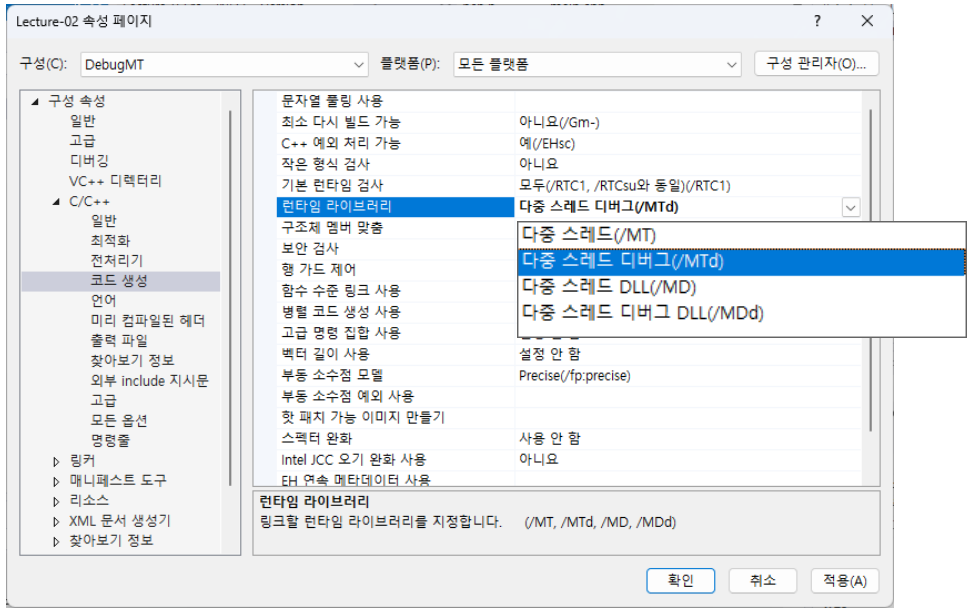


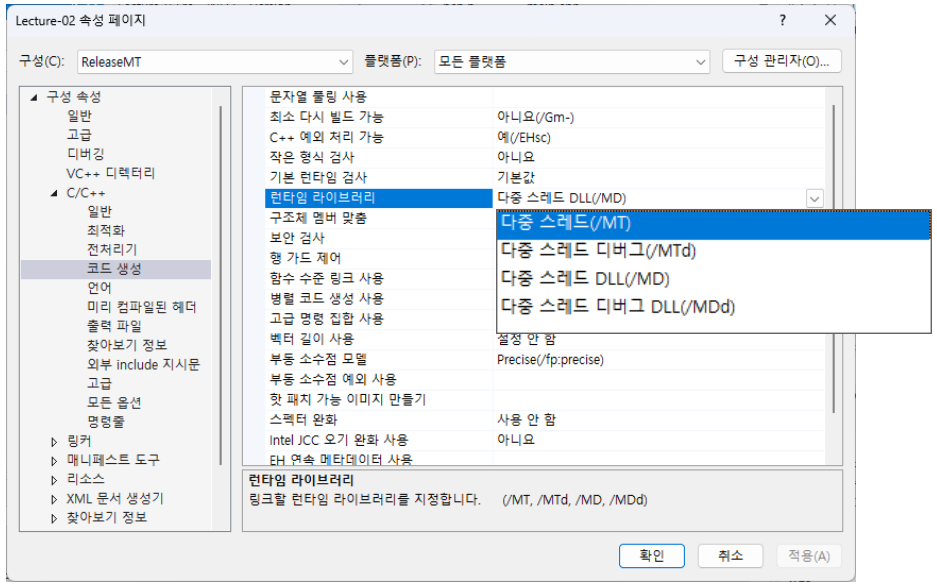
(2)



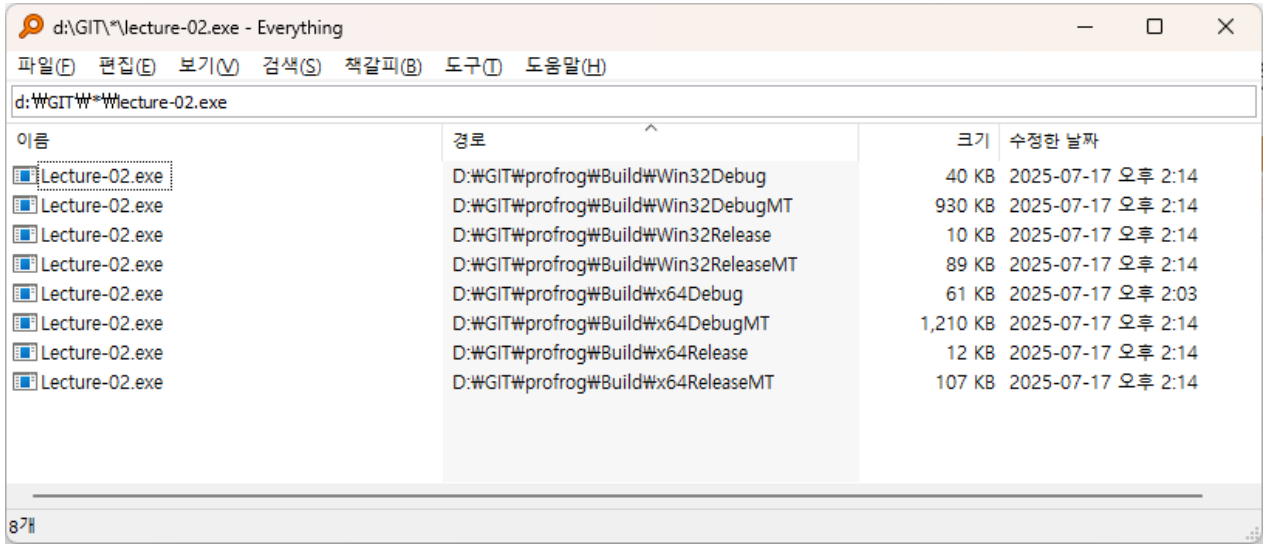
그후 DebugMT와 ReleaseMT 의 런타임 라이브러리 형식을

각각 MTd와 MT로 변경해 줍니다.





이제 ***[일괄 빌드]***를 수행하면 다음과 같이 8종류의 바이너리가 생성됩니다.



같은 소스 코드를 빌드 옵션만 다르게 주어 빌드했는데도 크기가 다양하게 나왔습니다.

x86	MD	MT
Debug	40 KB	833KB
Release	10 KB	101 KB

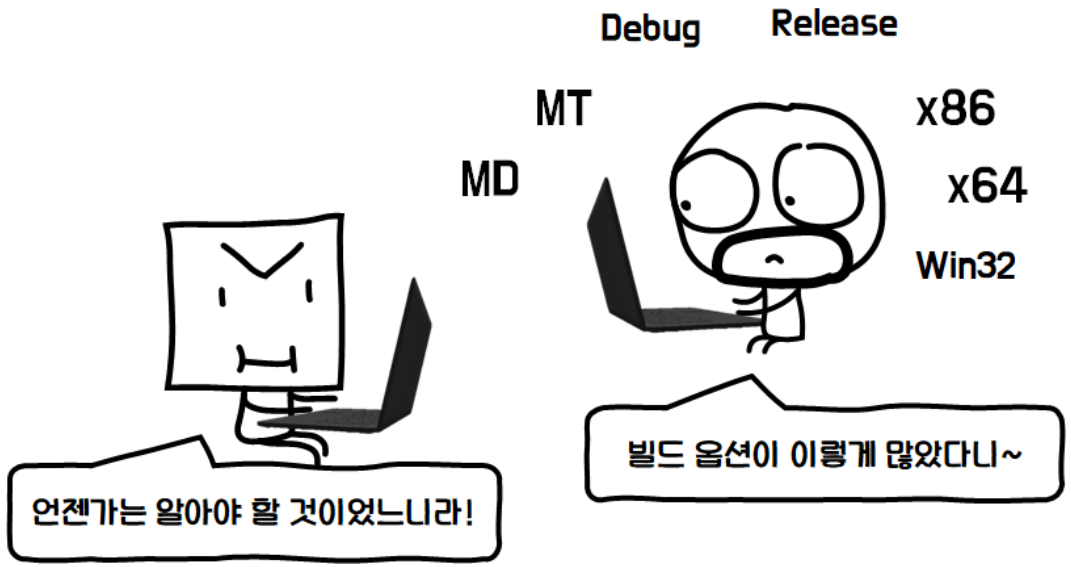
x64	MD	MT
Debug	61 KB	1,078 KB
Release	12 KB	119 KB

각 빌드 설정에 따른 크기를 비교할 수 있겠습니까? 추측해 보세요.

시험문제로 낼 수도 있습니다.

디버깅 정보가 추가되는 이유로 Debug 모드는 크기가 더 큽니다. 또 MD보다는 MT가 훨씬 큰 것을 알 수 있습니다. 이제 둘의 차이점을 알아보겠습니다.

런타임 라이브러리	설명	장점	단점
다중 스레드 DLL	시스템 API 함수를 DLL에서 가져오도록 빌드	파일 크기가 작음	재배포 패키지 설치 필요
다중 스레드	시스템 API 함수를 정적으로 링크해서 빌드	단독으로 실행가능	파일 크기가 큼



외부 라이브러리
CPPCORE 사용하기

CPPCORE 란 무엇인가

CPPCORE를 개발한 저는 Windows, CensOS, 우분투, MAC OS X, 안드로이드, 아이폰, 임베디드 등 다양한 운영체제와 장치에서 오랫동안(나이와 비례) 프로젝트 경험이 있었습니다. 같은 기능을 수행하는 알고리즘을 운영체제가 다르다고 비슷하게 여러 번 구현하는 일이 잦았는데 솔직히 어렵다기 보다 지겨웠습니다.

구체적인 예로, 영상과 음성을 다루는 DVR 솔루션을 개발할 때의 일입니다. 동영상 파일 mp4을 화면에 뿌려주는 과정에서 mp4 파일을 읽고 파싱하고 소켓으로 전송하는 과정을 윈도우와 안드로이드폰과 아이폰에 각각 구현해야 했습니다. 처음에는 혼자 다 해보다가 비슷한 반복적인 일이 너무 괴로웠습니다. 그래서 플랫폼별 개발자를 따로 두어 각자 개발하게 하였는데 문제는 각자의 개발속도와 방법이 다르고 한 곳에서 발생한 버그는 다른 곳에서도 비슷하게 발생하는 등의 관리 문제가 있었습니다.

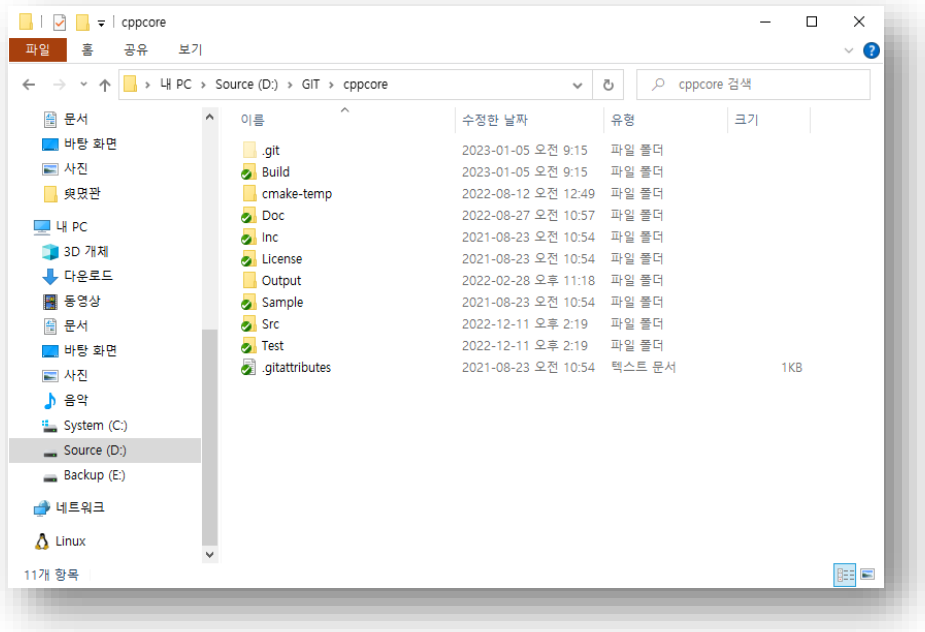
이 문제를 풀기 위해서 한 번 작성한 C++ 코드는 어떤 장치와 운영체제에서도 그대로 사용할 수 있는 방법을 고안했습니다. 이것을 크로스플랫폼 개발 기술이라고 부릅니다. 먼저 가장 문제가 되는 시스템 고유의 API를 공통적으로 사용할 수 있게 래핑하였습니다. 이후 공통 API를 바탕으로 문자열이나 파일, 메모리, 데이터, 네트워크 등을 더 쉽게 다루기 위한 유틸리티 함수들을 덧붙여 나갔습니다.

결과적으로 공통적이며 재사용가능한 코드들이 누적되었고, 향후 개발하는 모든 프로젝트에 적용하여 사용하고 있습니다. 실제로 코드의 생산성과 질이 수십 배는 올랐습니다. 어느 정도 안정성이 확보되었다고 생각한 시점부터 오픈소스로 공개하여 필요한 사람들이 쓸 수 있도록 강의와 책을 저술했습니다. 하지만 제 개인적인 노하우가 담긴 코드들이다 보니 아주 친절하지는 않습니다. 제대로 공부해서 쓸 사람만 쓰라는 생각이라.

1. 다음 저장소를 git clone명령으로 내려 받음

git clone https://github.com/profrog-jeon/cppcore.git

가급적 D:\GIT\cppcore 경로에 생성하도록 할 것



2. 빌드하여 윈도우용 cppcore.lib 및 리눅스용 libcppcore.a을 생성

윈도우용 빌드	cppcore\Build\BuildAll.bat 파일 실행 만일 빌드에 실패하면? - BuildAll.bat 파일을 열어서 MSBUILD 경로를 내 PC에 설치된 곳으로 변경(Everything으로 MSBuild.exe 파일의 위치를 찾음) SET MSBUILD="C:\Program Files\Microsoft Visual Studio\2022\Community\MSBuild\Current\Bin\MSBuild.exe"
유닉스 계열 빌드	Git/cppcore$ **mkdir cmake-temp** Git/cppcore$ **cd cmake-temp** Git/cppcore/cmake-temp$ **cmake ../Src && cmake --build .**

3. 내 프로젝트에 가져와서 빌드에 성공시킴

다음 코드를 준비합니다.

<pch.h>

```
#pragma once

#include "../../../cppcore/Inc/cppcore.h"

using namespace core;
```

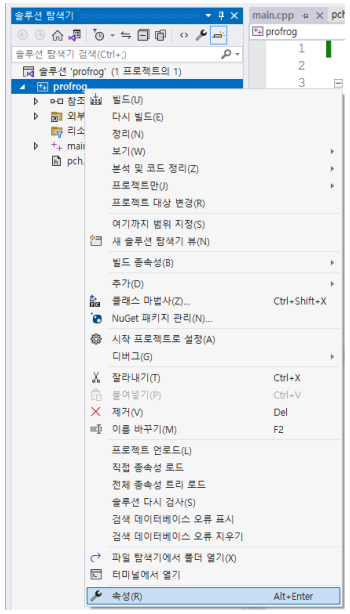
<main.cpp>

```
#include "pch.h"

int main()
{
   HANDLE hFile = CreateFile(TEXT("d:/report.txt"), GENERIC_READ_, OPEN_EXISTING_, 0);
   if (NULL == hFile)
      return -1;

   char szBuffer[10000+1];
   DWORD dwReadSize = 0;
   ReadFile(hFile, szBuffer, 10000, &dwReadSize);

   szBuffer[dwReadSize] = 0;
   printf("%s\n", szBuffer);

   CloseFile(hFile);
   return 0;
}
```

아마 컴파일에는 성공해도 링크에 실패할 겁니다.

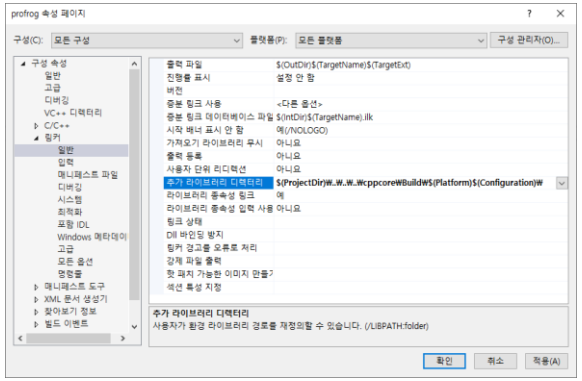
Cppcore 라이브러리를 링크 목록에 포함하지 않았으니까요.

```
빌드 시작...
1>------ 빌드 시작: 프로젝트: profrog, 구성: Debug x64 ------
1>main.obj : error LNK2019: "void * __cdecl core::CreateFileW(wchar_t const *,unsigned long,enum
core::E_FILE_DISPOSITION,unsigned long,unsigned long,void *)"
(?CreateFileW@core@@YAPEAXPEB_WKW4E_FILE_DISPOSITION@1@KKPEAX@Z)main 함수에서 참조되는 확인할 수 없는 외부 기호
1>main.obj : error LNK2019: "bool __cdecl core::ReadFile(void *,void *,unsigned long,unsigned long *)"
(?ReadFile@core@@YA_NPEAX0KPEAK@Z)main 함수에서 참조되는 확인할 수 없는 외부 기호
1>main.obj : error LNK2019: "void __cdecl core::CloseFile(void *)" (?CloseFile@core@@YAXPEAX@Z)main 함수에서 참조되는
확인할 수 없는 외부 기호
1>D:₩GITW profrog₩Src₩profrog₩x64₩Debug₩profrog.exe : fatal error LNK1120: 3개의 확인할 수 없는 외부 참조입니다.
1>"profrog.vcxproj" 프로젝트를 빌드했습니다. - 실패
========== 빌드: 0개 성공, 1개 실패, 0개 최신 상태, 0개 건너뜀 ==========
========== 00:00.630 경과 ==========
```



[프로젝트 속성] > [링커] > [일반] > [추가 라이브러리 디렉터리] 에 다음을 적어줍니다.

$(ProjectDir)\..\..\..\cppcore\Build\$(Platform)$(Configuration)\



여전히 링크에 실패할 겁니다. 경로만 지정했지 라이브러리를 지정하지 않았거든요.

```
빌드 시작...
1>------ 빌드 시작: 프로젝트: profrog, 구성: Debug x64 ------
1>main.obj : error LNK2019: "void * __cdecl core::CreateFileW(wchar_t const *,unsigned long,enum
core::E_FILE_DISPOSITION,unsigned long,unsigned long,void *)"
(?CreateFileW@core@@YAPEAXPEB_WKW4E_FILE_DISPOSITION@1@KKPEAX@Z)main 함수에서 참조되는 확인할 수 없는 외부 기호
1>main.obj : error LNK2019: "bool __cdecl core::ReadFile(void *,void *,unsigned long,unsigned long *)"
(?ReadFile@core@@YA_NPEAXOKPEAK@Z)main 함수에서 참조되는 확인할 수 없는 외부 기호
1>main.obj : error LNK2019: "void __cdecl core::CloseFile(void *)" (?CloseFile@core@@YAXPEAX@Z)main 함수에서 참조되는
확인할 수 없는 외부 기호
1>D:₩GITW프로그₩Src₩프로그x64₩Debug₩profrog.exe : fatal error LNK1120: 3개의 확인할 수 없는 외부 참조입니다.
1>"profrog.vcxproj" 프로젝트를 빌드했습니다. - 실패
========== 빌드: 0개 성공, 1개 실패, 0개 최신 상태, 0개 건너뜀 ==========
========== 00:00.630 경과 ==========
```

pch.h 파일에 다음 코드를 추가합니다.

```
#pragma comment(lib, "cppcore.lib")
```

<pch.h>

```
#pragma once

#include "../../../cppcore/Inc/cppcore.h"

using namespace core;

#pragma comment(lib, "cppcore.lib")
```

```
빌드 시작...
1>------ 빌드 시작: 프로젝트: profrog, 구성: Debug x64 ------
1>main.cpp
1>profrog.vcxproj -> D:₩GITW프로그₩Src₩프로그x64₩Debug₩profrog.exe
========== 빌드: 1개 성공, 0개 실패, 0개 최신 상태, 0개 건너뜀 ==========
========== 00:01.390 경과 ==========
```

고생 많으셨습니다!