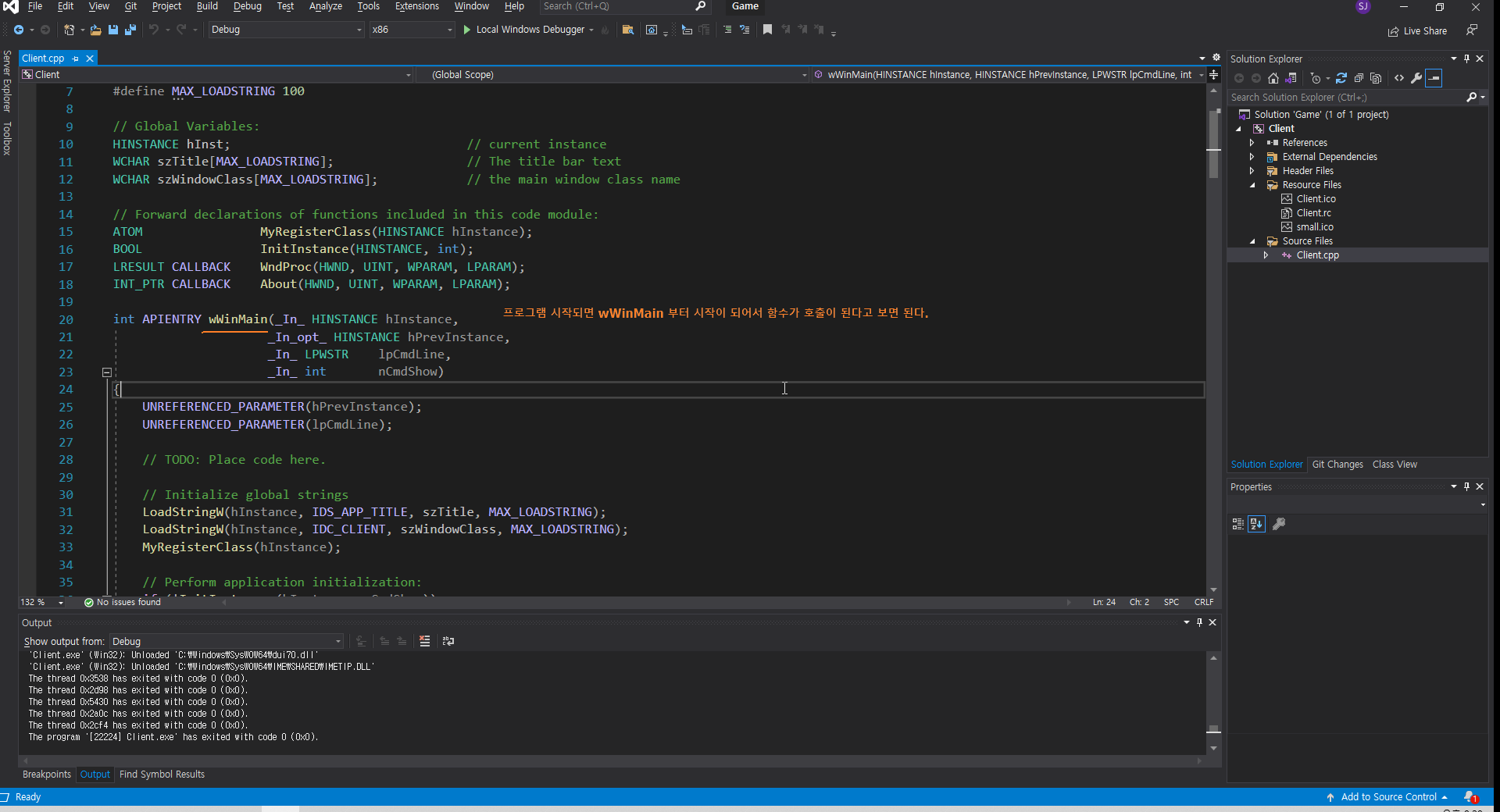
**DirectX12 초기화**

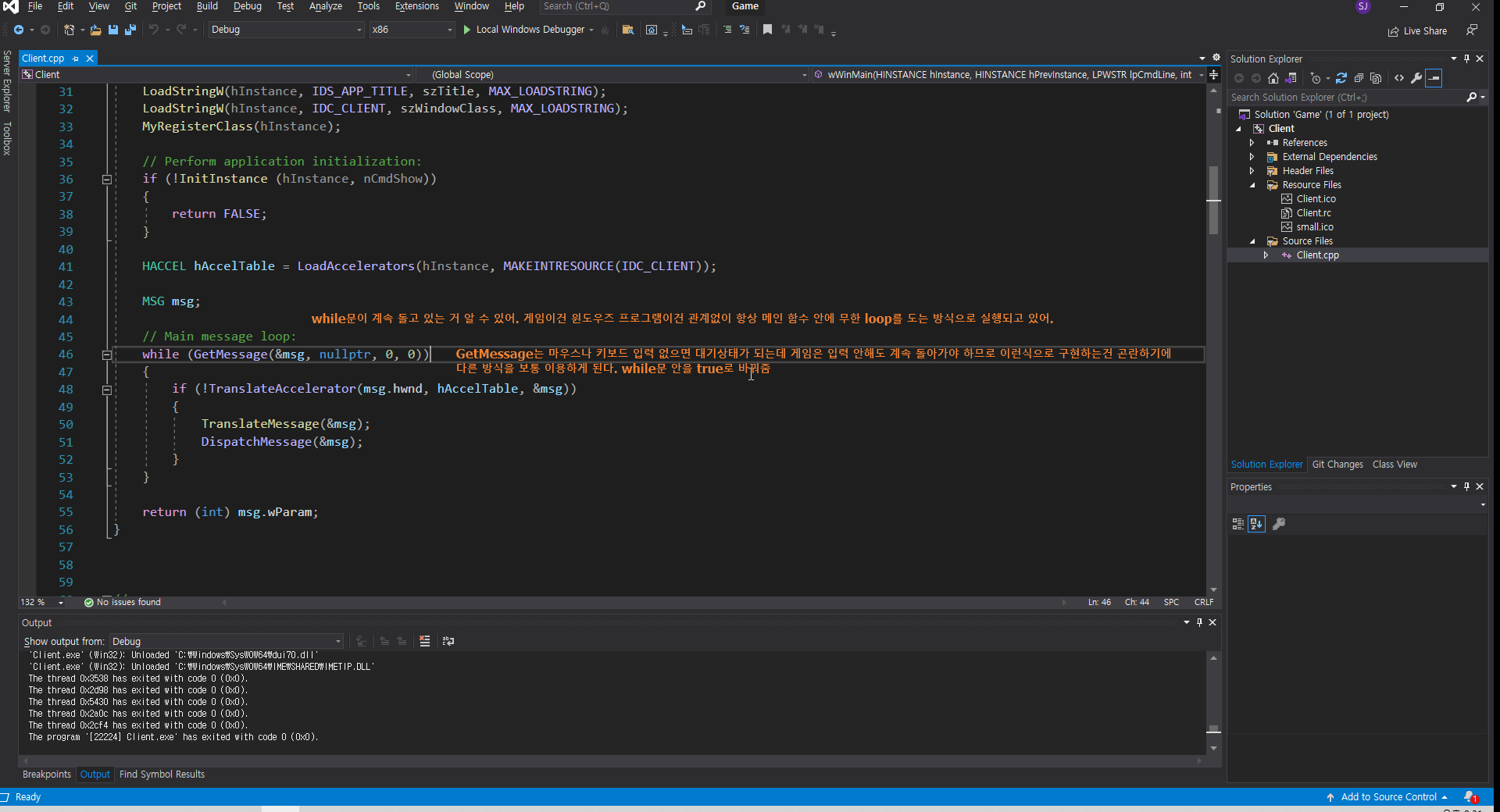
**1. 프로젝트 설정**

새프로젝트

Wnidows desktop Application 선택

윈도우즈 API를 이용해가지고 콘솔 프로그램과 다르게 일반적으로 사용하는 윈도우즈 프로그램이 만들어져 있는 거





client.cpp의 while문 안에 있는 PeekMessage()의 경우 인자가 굉장히 많은데 DirectX의 경우는 하나하나 의미를 찾아보는건 너무 힘든 일이야. 그냥 넘어가는게 좋아. 나중에 두번째, 세번째 공부를 할 때 관심이 생기면 인자를 찾아보면 되는거.

Precompile header라는 개념이 있어.

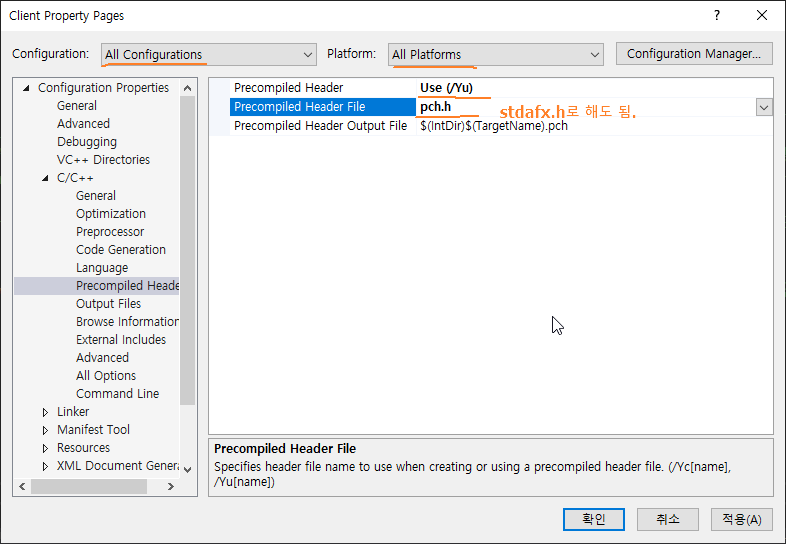
지금 여기서 어떤 vector라는 기능을 활용한다면 #include vector를 해줘야 하는데 그부분이 누락이 되었으니 당연히 vector를 활용할 수가 없어. 근데 자주 활용하는 기능들을 매번 include하는 것도 피곤한 일이 될 거야. 나중에 가면 DirectX에 관련된 라이브러리도 굉장히 많아질거고 기본적으로 vector와 같은 컨테이너들도 굉장히 많이 활용하고 싶어질텐데 매번 한땀한땀 인클루드 하는건 힘들거야. 그러니 공용으로 많이 활용할 거 같은 애들을 한방에 모아서 관리해주는게 좋은데 그걸 precompiled header라고 보면 됨.

편의성 외에도 미리 컴파일된 헤더라는 이름에서 알 수 있듯이 얘는 맨 처음에 미리 한번 공용으로 컴파일 된 다음에 컴파일된 그 애를 다들 꺼내서 사용하는 그런 개념이기 때문에 속도에도 이점이 있다.

visual studio버전에 따라 기본적으로 만들어지는 버전도 있고 안만들어지는 버전도 있어. pch.h라거나 stdafx.h라는 파일이 있으면 만들어진거고 그게 아니면 안만들어졌다고 보면 됨. 기본 설정에서 안만들어졌다고 해도 사용 못하는 건 아니고 설정값을 바꿔주면 됨.

Debug를 x64로 바꿔준다. 앞으로는 x86는 잊고 무조건 64비트 프로그램으로 만들어줄거기 때문에 x64로 만들어줬고,

Client 프로젝트 이름을 우클릭해서 속성에 들어가서 C/C++-> 미리 컴파일된 헤더로 가서 모든 구성, 모든 플랫폼, 미리 컴파일된 헤더 사용, 미리 컴파일된 헤더파일 pch.h로 해준다. stdafx.h로 해줘도 됨.



헤더파일 폴더의 내용들을 모두 소스파일 폴더로 옮기고 헤더파일 폴더는 삭제

소스파일 폴더 안에 새 filter를 추가해줘서 Utils라고 이름을 만들어주고, Game이라는 새filter를 더 추가해줌.

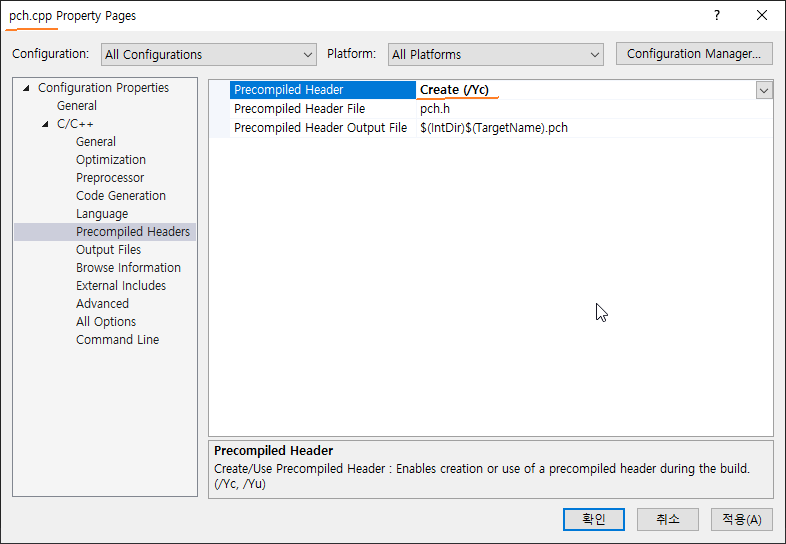
파일들 Utils로 다 옮겨주고

Utils에 클래스 추가 이름은 pch

pch.h에는 말 그대로 우리가 자주 활용을 할 애들을 몰빵해서 넣어준다고 보면 된다.

pch.cpp 우클릭해서 속성을 열고 미리 컴파일된 헤더를 만들기로 바꿔준다.

나중에는 익숙해지니 그냥 그런가보다 하고 넘어가면 됨.



이 상태에서 빌드 하면 에러 남. 미리 컴파일된 헤더를 사용한다고 했는데 client.cpp에서 이 미리 컴파일된 헤더를 사용 안하고 있으니까 발생한 문제. client.cpp 맨위로 가서 #include “pch.h”를 추가해주면 됨. 그럼 에러 없이 빌드가 됨.

이렇게 되면 내부에서 뜬급없이 vector를 활용했다거나 할 때도 아무런 위화감 없이 빌드가 통과됨을 알 수 있어. 즉 자주 활용이 되는 부분은 pch.h에 한방에 몰아서 해줬는데 이게 미리 컴파일이 되어서 한번만 컴파일한 다음에 그 기능들을 다시 사용할 때 다시 빌드하는게 아니라 미리 빌드해준 애들을 다 끌어서 한방에 활용할 수 있다는 장점이 생기게 됨.

Game필터쪽으로 가서 클래스 만들어주는데 Game이라는 이름의 클래스 만들어준다. convention 여러가지 있는데 일반적으로 class에 이름 자체 붙여주는 걸 선생님은 좋아해. 경우에 따라 게임 학원에서 C를 붙여 CGame이렇게 하는 경우도 많은데 팀마다 convention은 다르다.

이제 새로운 클래스엔 Game.cpp파일에 #include pch.h가 자동으로 추가되어 있는걸 볼 수 있어.

Init(), Update() 함수 만드는데 선언하고 구현부 자동으로 만들게 하는 단축키 있어. 선언부 이름 누른 상태에서 ctrl + .. 누르면 구현부 만들거냐는 선택 메뉴가 뜬다.

Client클래스 쪽은 윈도우즈 창을 만들고 걔를 렌더링하는 부분이 들어가 있는데 거기에 게임 코드를 넣어주면 헷갈리니까 깨끗한 마음으로 게임과 관련된 내용들은 별도의 클래스 Game에 빼줬다고 보면 됨. client.cpp에서는 무엇을 해야 되느냐. 말그대로 게임을 여기서 만들어줘서 초기화를 해준 다음에 main의 loop에서 update를 실행해주면 됨.

unique\_ptr<Game> game = make\_unique<Game>(); 로 Game을 만들어줌.

game->Init(); // 초기화를 해주고

game->Update();

루프를 한번 돌 때 마다 Update함수를 실행할거야. 어떤 게임이건 같은 방식으로 되어 있어.

빌드후 실행을 해보면

실행된 창에 파일, 도움말 부분이 마음에 안드니 제거해줄거야.

wcex.lpszMenuName = nullptr; 이렇게 nullptr로 바꿔치기 해주면 됨.

다시 실행해 보면 파일, 도움말이 사라져 있어.

엔진을 따로 별도의 라이브러리로 빼서 관리를 하게 될거야.라이브러리라는 개념이 들어가게 됨. #include로 쓴 <vector>나 <memory>도 결국 별도의 라이브러리야. 외부에서 만들어진 기능들을 끌어와서 사용하는 걸 말해. 왜 엔진 코드를 Client 코드에 끼워넣지 않고 별도의 프로젝트로 빼서 관리를 하느냐? 공통적인 기능들을 프로젝트 외부에 빼주면 많은 장점이 있어. 첫번째 게임 말고 또다른 게임을 만든다거나 할 때도 만약 Client프로젝트에 모든 코드가 들어가 있었다고 한다면 그 코드를 한땀한땀 복붙해서 다른 프로젝트에 넣어야해. 근데 그게 아니라 공용으로 관리하는 그런 라이브러리 프로젝트를 하나 만들어주게 되면 그 해당 코드들을 전부 재사용할 수 있는 큰 장점이 생기기 때문에 라이브러리로 관리한다. vector라는 코드가 모든 코드 자체를 한땀한땀 내부에서 사용했으면 딴 사람에게 건내줄 수 없었을 거야. 프로젝트 자체에 종속적이니까. 하지만 라이브러리로 빼주게 되면 필요할 때 마다 원하는 사람이 갖다 쓸 수 있다고 볼 수 있음.

추가->새프로젝트 누르면 지금까진 콘솔앱, 데스크톱 애플리케이션 같은걸 만들어줬지만 내려보면 라이브러리가 있어. DLL(동적 연결 라이브러리), 정적 라이브러리 이렇게 2개가 있어.

DLL – 실행 중인 여러 Windows 앱 간에 공유할 수 있는 .dll을 빌드하세요.

정적 – 다른 Windows 실행 파일 내에 패키지될 수 있는 .lib를 빌드하세요.

정적 라이브러리의 경우는 말 그대로 우리 프로젝트 자체에 같이 빨려가지고 합쳐져서 들어가는 개념. 노트북 안에 GPU가 내장되어 딸려오는 경우가 있고, 외장 GPU를 사서 실시간으로 노트북에 꽂아서 사용할 수 있어. 정적인 경우는 노트북 안에다가 부품을 꽂아가지고 같이 딸려오는 GPU가 되는 셈이고, 동적 연결 라이브러리는 프로그램이 실행되는 순간에 외부에 있는 애를 뭔가 꽂아서 사용하는 그런 개념이라고 볼 수 있어.

우린 기본적으로 정적 라이브러리로 만드는게 필요해. 이렇게 만들면 별도의 DLL파일을 관리할 필요 없이 그냥 우리 실행파일에 코드들이 딸려서 만들어지기 때문에 얘가 조금 더 관리하기가 편함. 정적 라이브러리로 생성. 프로젝트 이름은 Engine으로 한다.

pch.h가 자동으로 만들어져 있음. 파일들 소스파일 폴더로 옮겨주고 헤더파일 폴더는 삭제

신기하게도 Engine.cpp파일이 있지만 main 함수가 없다는걸 알 수 있어.

Engine을 시작프로젝트로 설정해서 실행해보면 바로 에러남을 알 수 있음. 얘는 실행하는 용도로 만들어진 프로젝트가 아니다라고 불평하고 있는 걸 볼 수 있어. exe 파일을 만들어서 실행하는 애가 아니라 다른 애한테 딸려가는 부품으로 만들고 있다고 볼 수 있어. 빌드만 하는건 위화감 없이 통과됨.

파일 탐색기에서 만들어진 결과를 살펴보면 C:\Users\Jang\Documents\Directx12\Game\x64\Debug 폴더에 보면 Engine.lib 파일이 만들어져 있는 것을 볼 수 있어. 이걸 딴애가 갖다 써서 그 부품을 조립해서 실행파일을 만들 때 얘를 참고한다. 클라이언트는 exe 파일 응용프로그램으로 되어 있는데 얘는 마지막이 lib 라이브러리 파일 이라는걸 알 수 있어.

source 폴더 삭제, engine.cpp 도 필요 없으니 삭제

framework.h의 내용을 pch.h에 붙여 넣고 framework.h는 삭제

Engine프로젝트에 Engine, Util, Resource 필터 생성

pch 파일도 Utils에 옮겨주고 Source File 폴더도 날려줌.

Util필터에 EnginePch라는 클래스 만들어줌.

pch.h에 **#include** "EnginePch.h" **포함 시켜줌.**

EnginePch클래스를 만든 이유는 나중에 클라이언트 쪽에서도 활용이 될 얘들을 한곳에 묶어서 관리해주기 위해 만들어줬다고 보면 됨. 각종 include, lib, typedef넣어 줄건데 한땀한땀 타이핑해도 되고 결과물을 강의에 올려 놓을테니 다운 받아서 복붙을 하는 형태로 따라와도 됨. 한땀한땀 치기에는 너무 양이 많다 보니까. 쳐도 말리진 않음.

그리고 DirectX에 관련된 기능들도 추가로 넣어주게 될거야.

d3dx12.h이건 마이크로소프트에서 만든거긴 한데 비공식적으로 딸려오는 라이브러리라서 따로 추가해줘야 함.이건 선생님 폴더에 있는거 Engine 프로젝트 폴더에 복붙해준다. 그리고 드래그&드롭으로 Utils 필터에 넣어주면 된다. 인터넷에도 올라와있는 파일임.

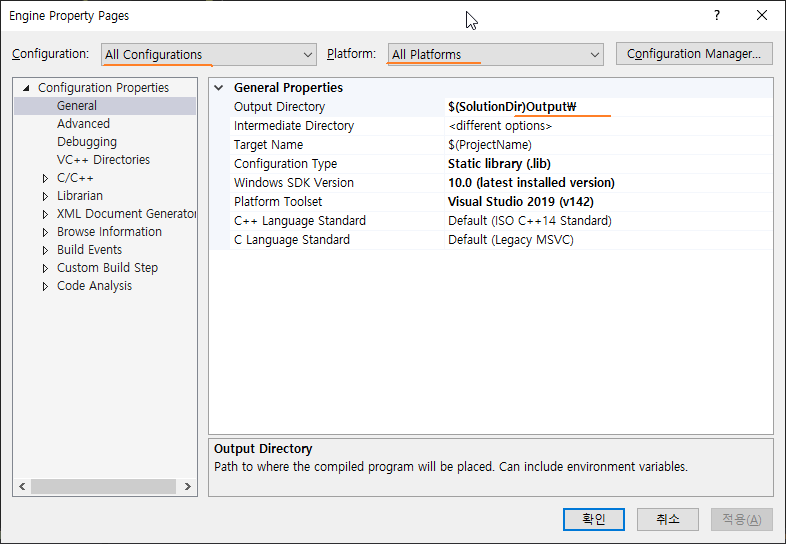
타이핑을 굉장히 많이 하게 될건데 그런걸 편리하게 할 수 있도록 자기가 편리하게 라이브러리를 만들어서 마소에서 제공을 해준거라고 볼 수 있어. 왜 비공식적인지 모르겠지만 편리한 기능들이 많이 들어가있어.

나중에 추가해도 되지만 하나하나씩 추가하면 에러가 너무 많이 날테니까 한번에 넣어본 것.

마지막으로 간단한 void HelloEngine(); 이라는 함수를 넣어줬어.

이 라이브러리 자체를 위에 있는 클라이언트에서 활용할 수 있는지를 체크해보기 위해 테스트 삼아 만들어본 함수임.

빌드를 하는데 파일들이 Game/x64 폴더에 뜨는데 마음에 안들어. debug에도 만들어지고 분산되어 만들어지는게 마음에 안드니 Game 폴더 안에 Output이라는 새폴더를 만들어줌. 최종 결과물은 이 Output에 저장이 되게끔 만들어줄거야. Engine 프로젝트 우클릭해서 속성에 들어감.

이렇게 수정해주고 Platform:을 x64로 하고 빌드 하면 잘 됨. 탐색기에서 보면 Output라는 폴더 안에 Engine.lib파일이 저장되어 있는 거 볼 수 있어.

탐색기에서 Game폴더 안에 Resources라는 폴더도 만들어줘.

엔진을 다룰 때는 폴더 구조를 어떻게 할지 정해줘야 하는데

Engine폴더에는 엔진코드가 들어갈거고,

Client폴더는 실제 클라이언트, 엔진을 활용한 게임 자체가 들어갈거고,

Output에는 최종 빌드 결과물 exe파일이 들어갈거고,

Resources는 나머지 리소스들이 들어갈거야.

이제 Client로 돌아가서 Client 쪽에서 엔진과 관련된 기능들을 끌어다쓰고 싶은 상황이 된거야.

Client에게 2가지를 알려줘야 하는데

header 파일들이 있는 경로와

lib 파일이 있는 경로를 알려줘야 한다.

현재는 Engine 폴더 안에 header 파일들이 있음. 경우에 따라 별도의 폴더에 있을수도 있음.

경로를 절대경로로 알려주기 보다는 프로젝트가 담긴 폴더를 다른 곳으로 옮길수도 있으니 상대경로로 만들어주는게 좋아. Game.sln 솔루션 파일을 기준으로 Engine으로 들어가면 header 파일이 있을 거고 Outpu t 파일에 들어가면 lib 파일이 있을거라고 경로 알려주면 됨.

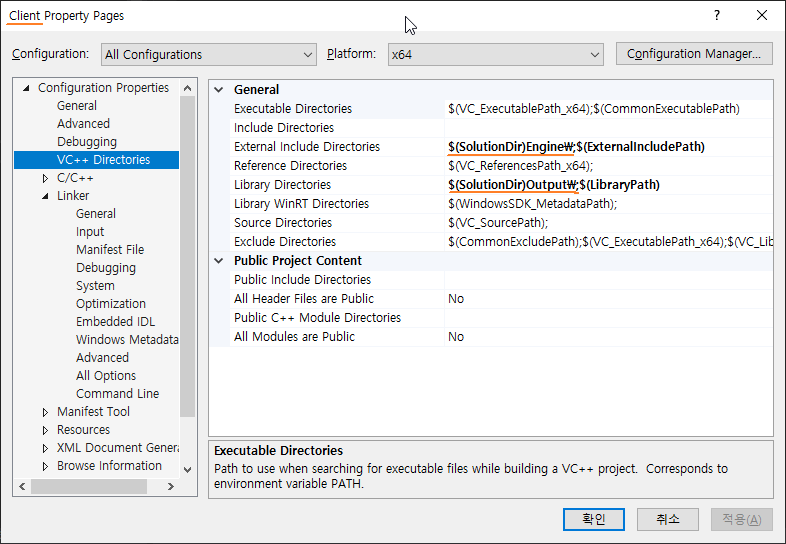
Client 프로젝트 우클릭해서 속성 들어가면

VC++ 디렉터리 에서 넣어주면 됨. 포함 디렉터리(External Include Directories)에는 header가 있는 경로를 알려주면 되는거고, 라이브러리 디렉터리(Library Directories)에는 lib가 있는 위치를 알려주면 됨.

포함 디렉터리(External Include Directories)에는 상대 경로로 $(SolutionDir)Engine\

까먹지 말고 적용을 매번마다 눌러 주는게 좋아.

라이브러리 디렉터리(Library Directories)에는 $(SolutionDir)Output\를 넣어준다.



header파일들, library를 어디서 찾아볼 것인지를 여기에 한줄을 추가한게 된거.

편집을 눌러 보면 기본적인 값들이 꽤 된다는 것을 알 수 있어. 지금까지 vector 같은걸 아무런 위화감 없이 쓸 수 있었던 것은 이런식으로 어디서 라이브러리들과 헤더 파일들을 searching 할 것인지 경로를 잡아놨기 때문에 가능했던 것이야. 거기다가 추가로 우리만의 경로 하나를 더 추가한 셈이 된거야.

여기까지 빌드해준다.

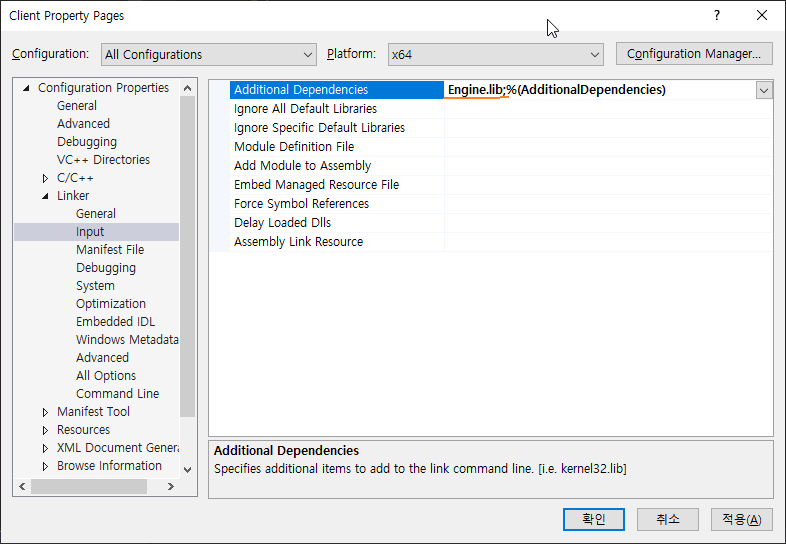
우리가 하고 싶은건 EnginePch 클래스에 있는 HelloEngine()을 사용하는게 목표. Game.cpp에 가서 Init()에 다가 HelloEngine()을 호줄하고 싶다 가정.

pch.h 파일에 **#include** "EnginePch.h" **을 추가해줌.**

빌드하면 에러가 뜨고 있어.

어떤 라이브러리 파일을 링크를 안해줬다는 문제임.

Client 프로젝트 우클릭해서 속성으로 가서 링커→입력→추가 종속성에 가서 Engine.lib를 넣어주면 됨.



빌드를 하면 문제 없이 빌드 됨을 알 수 있어.

매번 프로젝트의 속성에 가서 추가 종속성을 만들어주는 것도 귀찮아. 그러다 보니 이런식으로 추가 종속성에 넣어주지 않고 또 다른 방법이 하나 더 있어.

EnginePch.h에 보면 #pragma comment(lib, “d3d12”)이런 코드가 있었는데 얘가 그런 역할을 함. 속성에서 추가 하는게 아니라 코드를 이용해서 추가를 해주면 똑같이 동작을 한다.

pch.h에 #pragma comment(lib, "Engine.lib")를 추가해준다.

이건 전처리 단계라고 했어. 컴파일 이전에 전처리기가 이코드를 읽어가지고 아까 우리가 한 행동들을 그대로 해준다고 보면 됨. Engine.lib를 추가해주겠다고 인지를 한다고 보면 됨.

추가 종속성에 Engine.lib를 지워서 빌드를 해도 에러가 나지 않음을 볼 수 있다.

우클릭 속성에서 추가 종속성을 추가하는 걸 전처리기 이용해서 한방에 해줬다고 볼 수 있어.

Client 프로젝트를 시작 프로젝트로 설정한 다음에

HelloEngine()이 실행되는지 보기 위해 break point를 걸어둔 다음에 f5롤 실행하고 걸리면 f11을 누르면 HelloEngine()의 구현코드가 나온다. 이로서 다른 프로젝트에 있는 lib코드도 디버깅 할 수 있다는 걸 확일할 수 있어.

정리: 클라이언트 코드와 프로젝트와 코어를 담당하는 엔진 프로젝트를 분리를 해서 관리를 해서 이 엔진 라이브러리를 클라이언트에서 참조를 하게끔 우리가 뭔가를 열심히 연결시켜줬다고 보면 되는거고, 이게 처음에는 헷갈리지만 여러번 해보면 기억이 나게 됨.

client 의 pch.h에서

**#include** <vector>

#include <memory>

using namespace std; 삭제해도 무방해.

EnginePch에서 다 포함해줬기 때문에. 삭제해도 빌드 잘 됨을 알 수 있어.

이런 식으로 라이브러리와 클라이언트로 분할을 한 것까지 완료.