부록

1. 라이브러리 사용하기

라이브러리는 쉽게 말해서 미리 짜인 코드를 의미합니다. C언어를 배울 때에도 printf나 scanf같은 함수들은 여러분이 작성한 것이 아닌 이미 다른 사람이 만들어둔 것이었죠. 이런 것들이 라이브러리입니다. 라이브러리는 동적 라이브러리Dynamic Library와 정적 라이브러리Static Library 2가지로 나뉩니다. 동적 라이브러리는 프로그램의 실행시간에 읽어들여서 사용하지만, 정적 라이브러리는 컴파일 시간에 읽어들여서 프로그램에 포함시킵니다. 동적 라이브러리는 OS마다 확장자가 다릅니다. Windows 에서는 .dll, Linux 계열 OS에서는 .so 확장자를 사용합니다. 반면 Windows 에서 정적 라이브러리는 .lib 확장자를 사용합니다.

부록에 이번 장에서는 이 라이브러리를 사용하는 방법만을 간단하게 알아보도록 하겠습니다. 다른 사람이 만든 소스 코드를 가져와 사용하는 상황을 가정했을 때, 아래와 같은 과정으로 프로그램을 생성하게 됩니다.

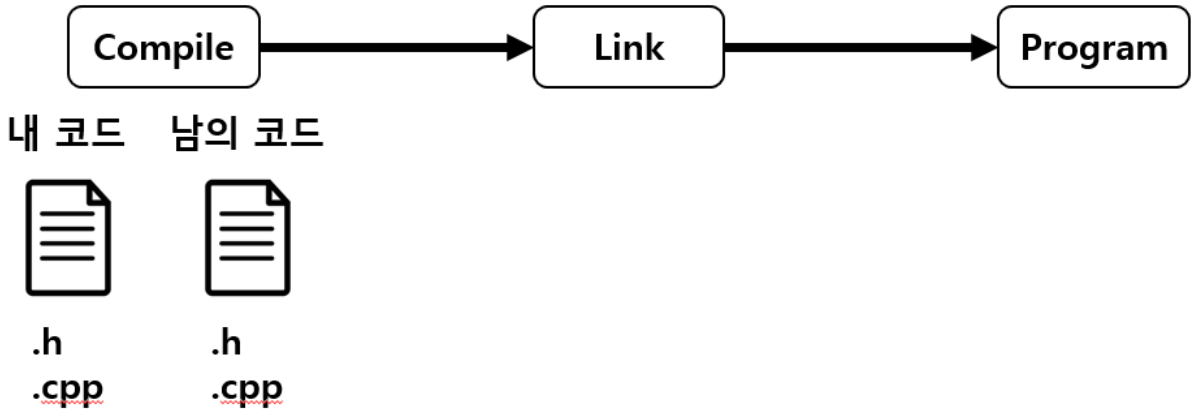


그림 1 배포된 소스 코드를 사용할 때의 과정

다른 사람의 소스 코드를 그대로 가져다가 사용하기 때문에 입맛에 따라 수정할 수도 있고[1], 32비트나 64비트와 같은 시스템 아키텍처의 영향을 받지도 않습니다[2]. 하지만 배포자가 소스 코드를 노출시키고 싶지 않다면 현명한 방법은 아닙니다. 또한 컴파일을 사용자가 직접 해야 하기 때문에 소스의 양이 크다면 그만큼 컴파일 시간이 늘어납니다.

---

[1] 물론 허락을 받아야 하겠죠? 시간이 나신다면 GPL과 같은 허가권에 대해서도 찾아보세요!

[2] 하지만 구현하기 나름입니다.

그림 2 배포된 소스 코드를 사용할 때의 과정

반면에 정적 라이브러리는 소스 코드를 미리 컴파일 한 것입니다. 따라서 링크과정에서 미리 컴파일 된 코드가 프로그램에 추가되기 때문에 시간과 성능을 절약할 수 있습니다. 다만 소스 수정이 불가능하고, 플랫폼 종속적이라는 한계는 존재합니다. 예를 들어 32비트에서 소스 코드를 정적 라이브러리로 변환했다면 64비트에서 사용할 수 없습니다. 그래서 이런 때에는 다양한 플랫폼에 맞춰서 배포해야 합니다.

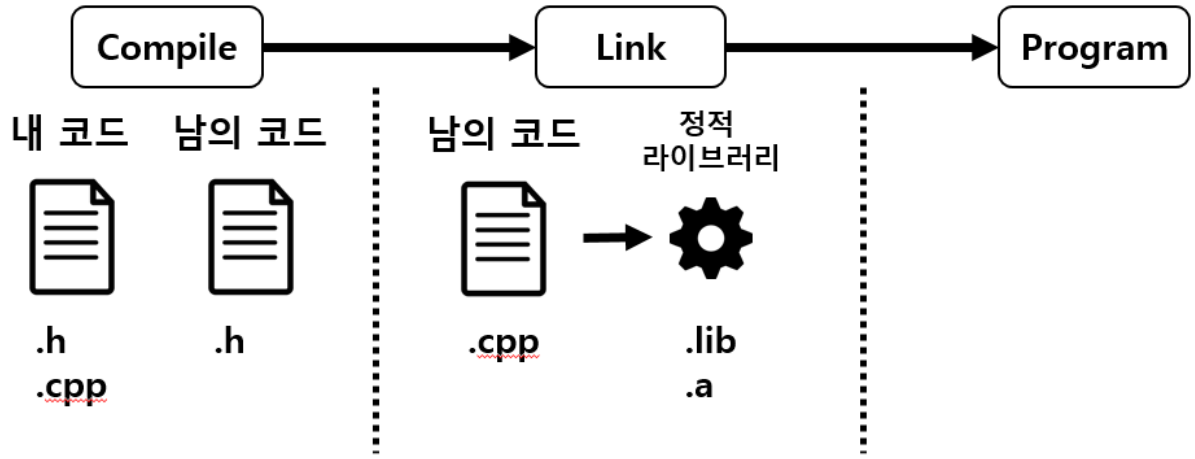


그림 3 정적 라이브러리를 사용할 때의 과정

동적 라이브러리는 프로그램을 실행할 때 DLL과 같은 파일에서 컴파일 된 소스코드를 읽어오는 것입니다. 따라서 DLL에서 제공하는 기능에 버그가 있다면 응용 프로그램을 고치지 않고 DLL 파일만을 교체하면 됩니다. 또한 여러 개의 프로그램이 같은 DLL을 공유 함으로서 메모리를 절약할 수도 있습니다. 다만 너무 많은 DLL을 사용한 DLL이 다른 DLL을 계속해서 요구하면서 미치듯이 DLL오류가 나는 경우가 있는데 이를 흔히 DLL지옥이라고 합니다. 또한 DLL파일을 바꿔서 악의적인 코드를 실행하게 할 수도 있습니다.[3]

[3] 이를 DLL Injection 이라고 합니다.

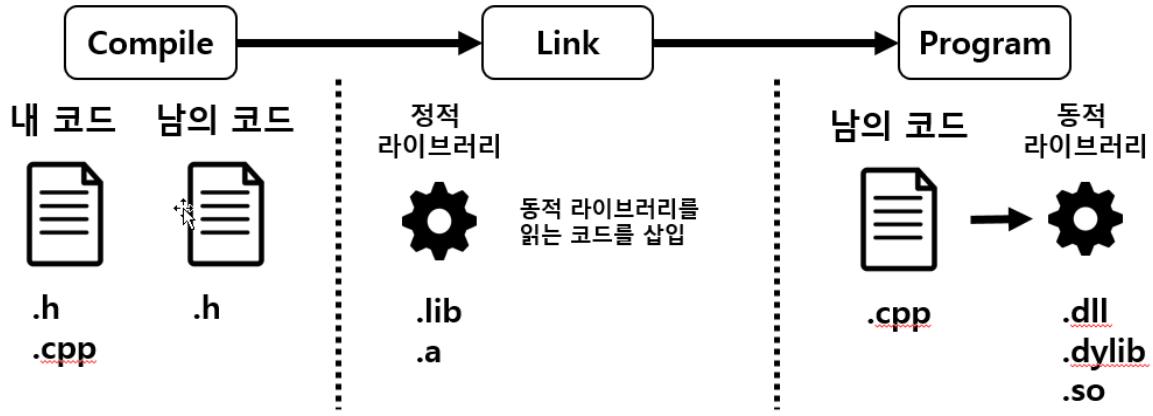


그림 4 동적 라이브러리를 사용할 때의 과정

원래는 동적 라이브러리 파일에서 직접 소스 코드를 읽어와야 하지만, 대부분은 이를 정적 라이브러리 파일에서 대신해주는 기능을 채택하고 있습니다. 따라서 동적 라이브러리는 개발자에게 대부분 정적 라이브러리와 함께 배포됩니다.

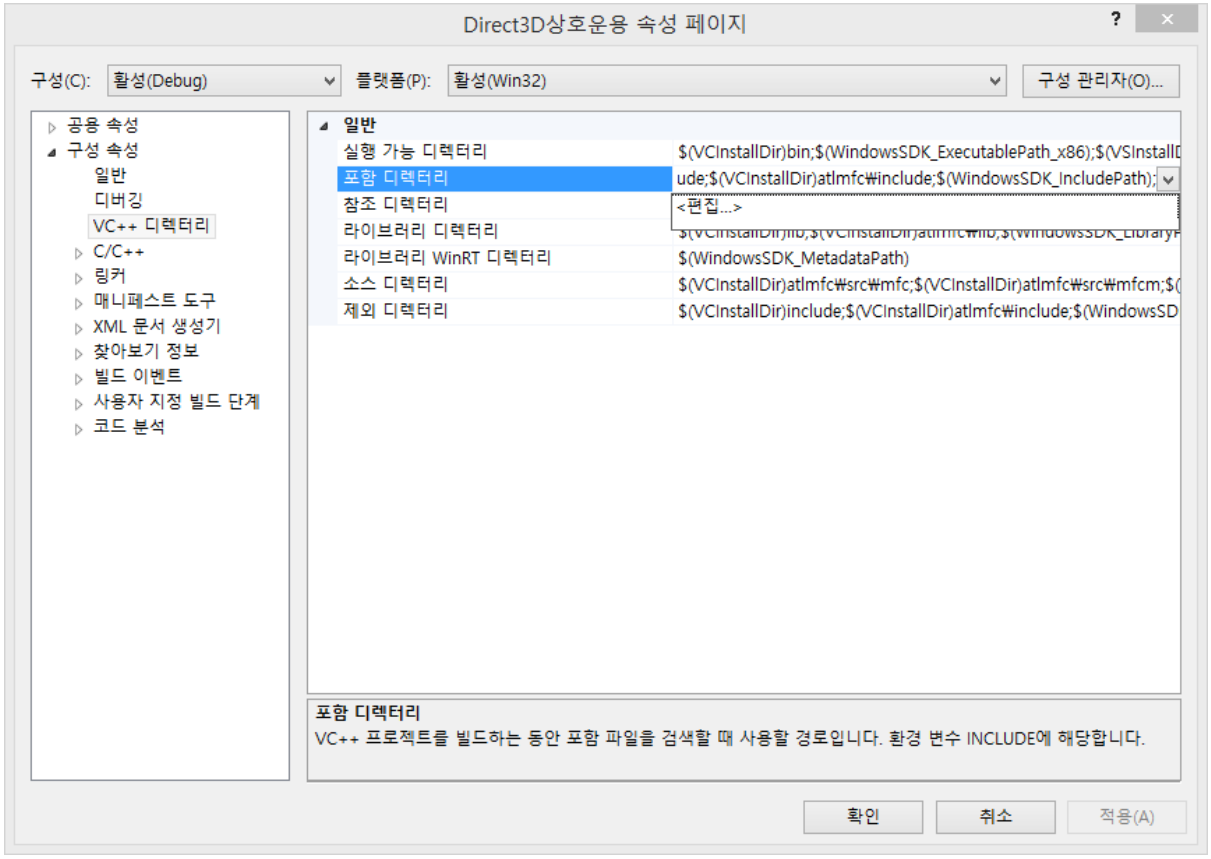
배포 방식	장점	단점
소스 코드	플랫폼 독립적이다 소스 코드가 공개되어 수정할 수 있다	컴파일 시간이 소요된다 소스 코드가 노출된다(장점이자 단점)
정적 라이브러리	컴파일 속도가 빠르다 소스 코드가 공개되지 않는다	플랫폼 종속적
동적 라이브러리	응용 프로그램의 크기가 줄어든다	플랫폼 종속적 DLL 지옥 보안 문제

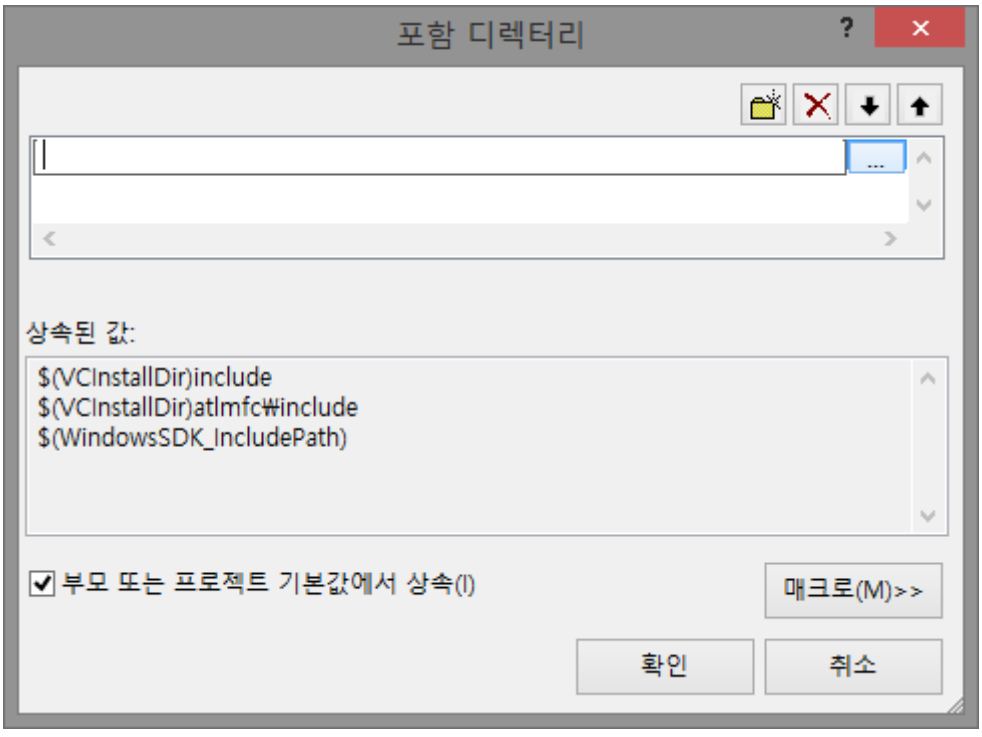
표 1 라이브러리 배포 방식에 따른 장단점

정적 라이브러리를 링크하려면 컴파일러에 설정을 해주어야 합니다. Visual Studio 2012를 기준으로 설명을 드리도록 하겠습니다. 프로젝트 – 설정 혹은 솔루션 탐색기의 프로젝트명을 우클릭하신 뒤의 속성을 클릭하면 [프로젝트명] 속성 페이지 창이 뜹니다.

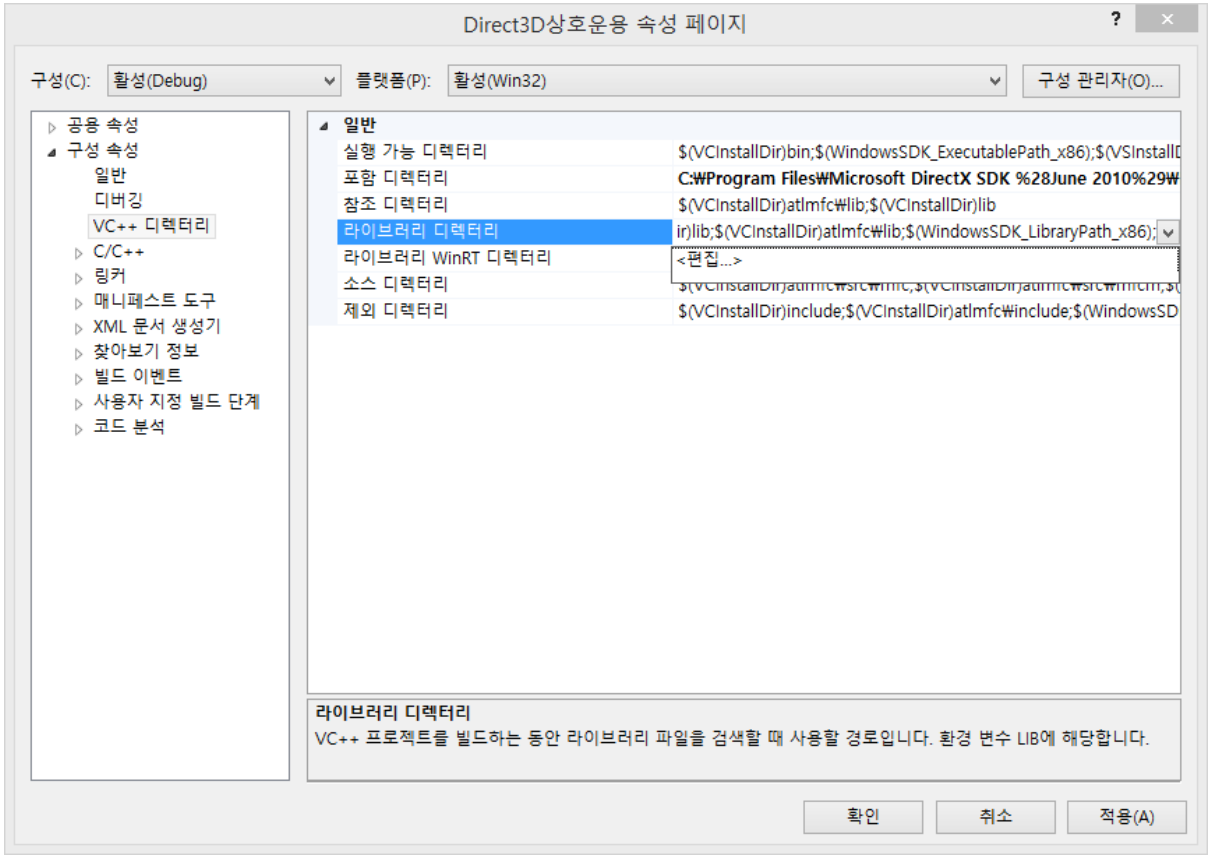
포함할 헤더 파일과 정적 라이브러리 파일이 존재하는 디렉터리 경로를 지정해 주어야만 컴파일러가 찾아낼 수 있습니다.[4] 왼쪽 리스트에서 구성 속성 – VC++ 디렉터리에서 포함 디렉터리를 편집해 경로들을 추가해 봅시다.

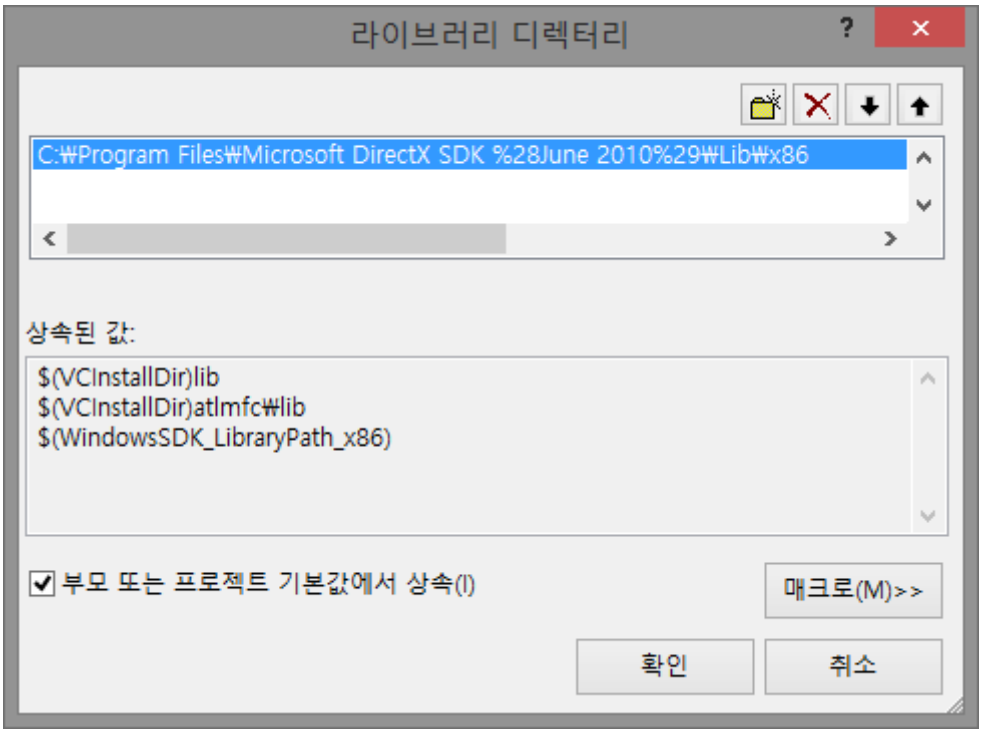
[4] 혹은 프로젝트 폴더 안에 넣어두면 됩니다. #include <>로 포함할 경우에는 무조건 위 방식을 사용해야 합니다.

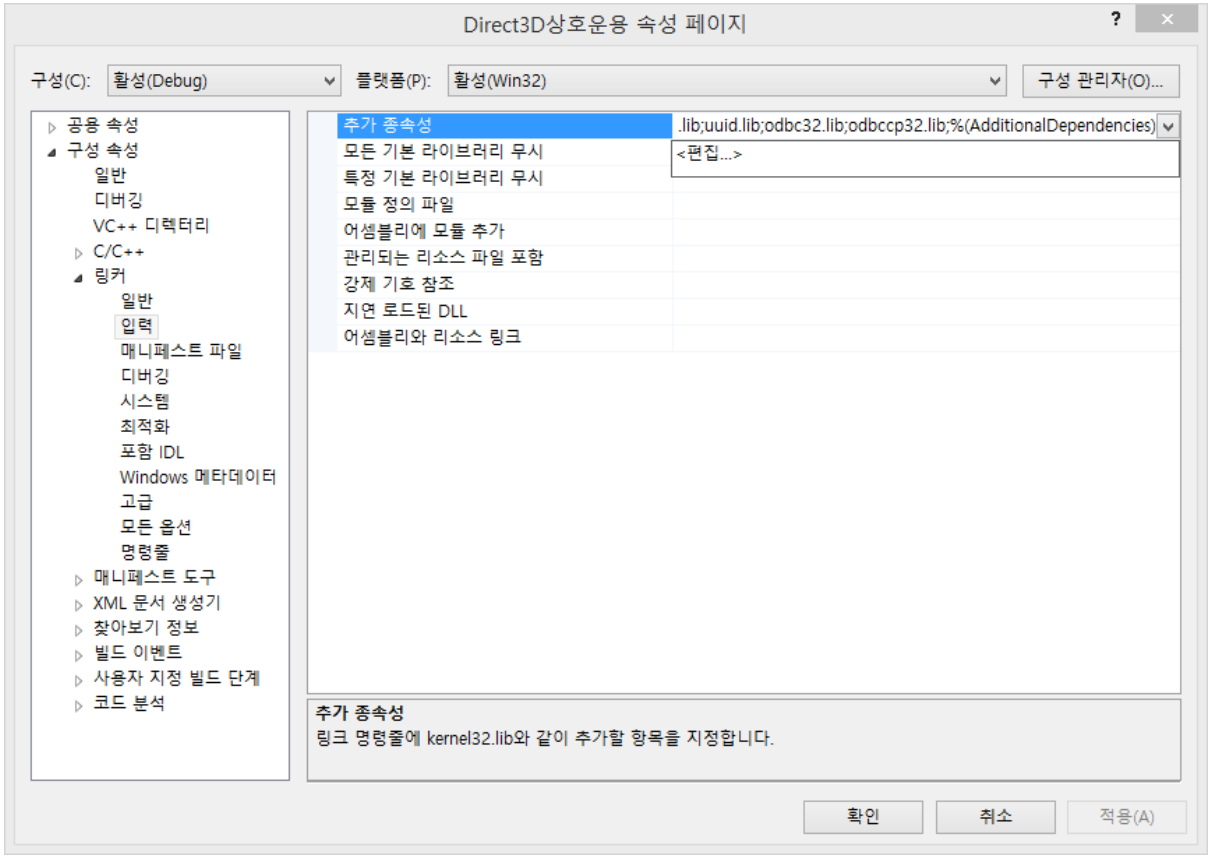




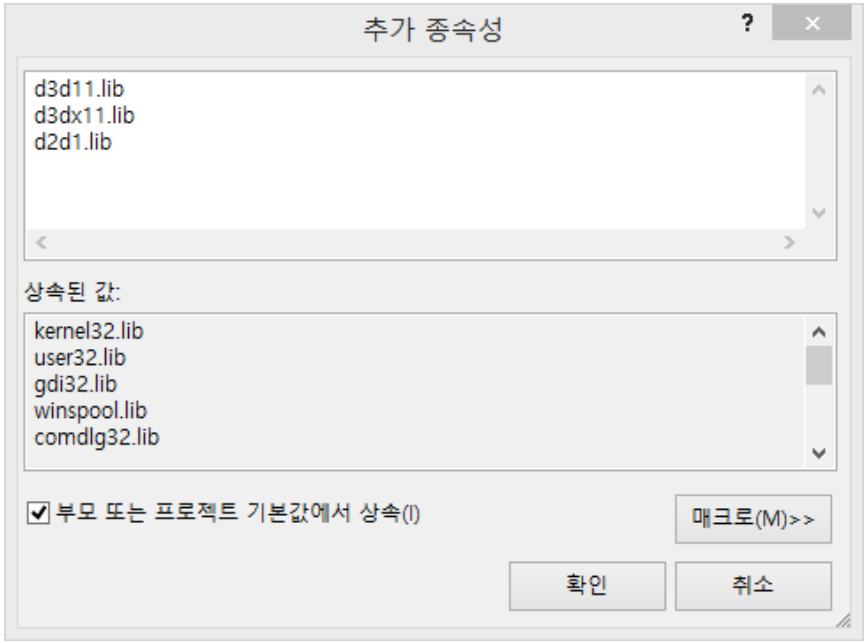
링크할 정적 라이브러리의 경로는 라이브러리 디렉터리에 추가해 줍시다.


Direct3D상호운용 속성 페이지
구성(C): 활성(Debug)
플랫폼(P): 활성(Win32)
구성 관리자(O)...
공용 속성
구성 속성
일반
디버깅
VC++ 디렉터리
C/C++
링커
매니페스트 도구
XML 문서 생성기
찾아보기 정보
빌드 이벤트
사용자 지정 빌드 단계
코드 분석
일반
실행 가능 디렉터리
$(VCInstallDir)bin;$(WindowsSDK_ExecutablePath_x86);$(VSInstallD
포함 디렉터리
C:₩Program Files₩Microsoft DirectX SDK %28June 2010%29₩
참조 디렉터리
$(VCInstallDir)atlmfc₩lib;$(VCInstallDir)lib
라이브러리 디렉터리
ir)lib;$(VCInstallDir)atlmfc₩lib;$(WindowsSDK_LibraryPath_x86);
라이브러리 WinRT 디렉터리
<편집...>
소스 디렉터리
제외 디렉터리
$(VCInstallDir)include;$(VCInstallDir)atlmfc₩include;$(WindowsSD
라이브러리 디렉터리
VC++ 프로젝트를 빌드하는 동안 라이브러리 파일을 검색할 때 사용할 경로입니다. 환경 변수 LIB에 해당합니다.
확인
취소
적용(A)


라이브러리 디렉터리
C:₩Program Files₩Microsoft DirectX SDK %28June 2010%29₩Lib₩x86
상속된 값:
$(VCInstallDir)lib
$(VCInstallDir)atlmfc₩lib
$(WindowsSDK_LibraryPath_x86)
부모 또는 프로젝트 기본값에서 상속(I)
매크로(M)>>
확인
취소



링크할 라이브러리는 구성 속성 – 링커 – 입력의 추가 종속성에 들어가 정적 라이브러리 이름들을 아래처럼 입력하시면 됩니다.



2. COM

COM[Component Object Model] 이란 Windows 환경에서 여러 언어나 소프트웨어서 기능을 함께 사용하기 위한 바이너리 코드의 표준입니다. Windows에서 프로그래밍을 하신다면 대부분 COM을 활용해야 합니다. 여기서는 COM에 대한 깊고 자세한 내용까지 다루지는 않겠습니다. 다만 저희가 이 자습서에서 알려드리는 DirectX 역시 COM을 따르기 때문에 이에 대한 이해를 돕고자 내용을 추가하였습니다. 간략하게 몇 가지 특징들만 설명 드리겠습니다. 첫 번째로, 그들은 IUnknown 클래스를 상속합니다. 그리고 IUnknown에 정의된 3가지 메소드들을 구현하게 됩니다.

```
ULONG AddRef()

ULONG Release()

HRESULT QueryInterface(const IID& iid, void**)
```

COM에서 순수한 추상 클래스를 인터페이스[Interface] 라고 하며, 클래스 이름 앞에 I를 붙입니다. 그들은 헤더파일 unknwn.h 안에 있는 IUnknown을 상속하는 하나의 인터페이스 클래스를 만든 뒤, 다시 상속하여 구현하는 클래스를 만든 다음, 그 클래스의 객체를 생성하는 팩토리 함수를 제공하여 실제 구현된 객체를 얻게 해줍니다. 왜 이렇게 복잡한 과정을 거치는지는 아래의 그림을 통해 알 수 있습니다.

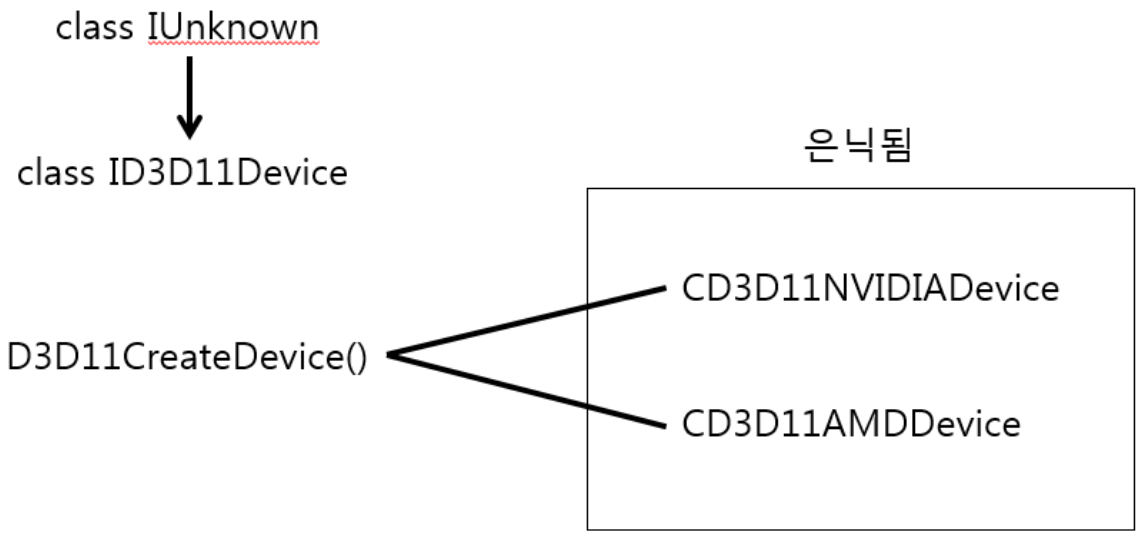


그림 5. 은닉된 ID3D11Device 구현 클래스들

ID3D11Device는 실제로 Direct3D 11에 존재하는 인터페이스이다. 이 인터페이스를 구현하는 2

가지 클래스가 있다고 가정해 봅시다.[5] CD3D11NVIDIADevice 클래스는 NVIDIA 그래픽카드를 위한 ID3D11Device 인터페이스의 구현이고, CD3D11AMDDevice 클래스는 AMD 그래픽카드를 위한 ID3D11Device 클래스의 구현입니다. D3D11CreateDevice() 함수는 자동적으로 시스템 사양을 알아보고 그에 알맞은 클래스의 객체를 ID3D11Device 인터페이스의 형태로 반환해 줍니다. 이 클래스가 NVIDIA 클래스인지, AMD 클래스인지는 알 필요가 없고, 알 방법도 없습니다.

COM의 모든 클래스들은 각각을 식별하는 GUID(Global Unique IDentifier)을 가지고 있습니다. GUID는 128비트의 고유 식별 정수입니다. 인터페이스의 GUID를 IID(Interface IDentifier)라고 합니다. GUID는 GUID라는 클래스로 Rpcdce.h 안에 선언되어 있습니다. IID는 typedef로 GUID와 같은 클래스이입니다. IUnknown의 QueryInterface 함수는 이렇게 은닉된 클래스 객체를 다른 클래스의 객체로 캐스팅한 후 반환해주는 역할을 합니다. 이를 인터페이스 질의(Query Interface)라고 합니다. 예를 들어 봅시다. 디버깅을 도와주는 ID3D11Debug라는 인터페이스가 있는데, ID3D11Device 인터페이스는 ID3D11Debug를 상속하지 않습니다. 그러나 은닉된 클래스가 이를 구현하거나(Is-A) 소유한다면(Has-A) 그 클래스에서 ID3D11Debug 타입의 객체를 얻을 수 있어야 합니다. 이럴 때 QueryInterface 메서드는 이를 얻어오는 데 도움을 줍니다. 아래 예를 봅시다.[6]

```
class CD3D11NVIDIADeviceWithDebug
      : public ID3D11Device,
      public ID3D11Debug,
{
      ...
      HRESULT QueryInterface(...)
      ...
};
HRESULT CD3D11NVIDIADeviceWithDebug::
      QueryInterface (
      const IID& iid,
      void **ppOut
      )
{
      ...
      if(iid == IID_ID3D11Debug) {
            *ppOut = (ID3D11Debug*)this;
            return S_OK;
```

[5] 실제로는 개발자 말고는 알 수 없습니다.

[6] 예시 코드일 뿐 실제 구현 방식과는 다를 수 있습니다.

```
	}
	...
}
```

```
ID3D11Debug *debug = 0;
ID3D11Device *device = ...
HRESULT result =
	device->QueryInterface (
		IID_ID3D11Debug,
		(void**)&debug
	);
```

아래의 예에서, QueryInterface 메서드를 통해서 ID3D11Debug 클래스 객체를 얻고자 한다고 해봅시다.. 저 ID3D11Device 객체는 실제로는 ID3D11Debug 클래스를 상속하는 CD3D11NVIDIADeviceWithDebug 클래스이기 때문에 결과를 인자로 온 ppOut에 저장한 뒤, 성공을 의미하는 S_OK를 반환합니다. 다양한 에러 코드가 있는데, 만약 저 클래스가 ID3D11Debug 인터페이스를 구현하지 않는다면, E_NOINTERFACE를 반환하여 객체와는 관련이 없음을 나타낼 수 있습니다. 일반적인 에러 코드는 E_FAIL(단순 실패)입니다.

AddRef 메서드와 Release 메서드는 참조 카운팅(Reference Counting)을 위해서 존재합니다. 참조 카운팅이란, 객체를 소유하고 있는 주체들의 수를 저장하였다가, 소유자들의 수가 0이 될 경우 객체를 소거합니다. 이 때 소유자들의 수를 참조 수(Reference Count)라고 한다. AddRef는 참조 수를 늘리는 함수이고, Release는 참조 수를 1줄이는 함수입니다. 두 함수 모두 새로운 참조 수를 반환합니다.

이 세가지 기능들 간편하게 사용할 수 있게 하는 스마트 포인터 클래스가 바로 atlbase.h 에 정의된 CComPtr 클래스입니다. 이 클래스는 참조 카운팅을 자동화 해주고, 템플릿을 통해서 인터페이스 질의를 좀 더 편리하고 유용하게 사용할 수 있으나 여기서는 설명하지 않겠습니다. 주의를 좀 주자면 &연산자가 오버로딩 되어 있는데, 이 때 소유한 인터페이스가 NULL이 아닐 경우 assert 처리가 되어 있어 디버깅 도중에 에러를 발생시킵니다[7]. 따라서 스마트 포인터가 가진 인터페이스를 얻을 땐 public 멤버인 p 에 접근하길 권장합니다.

COM은 아직까지도 윈도우에서 많이 사용되고 있습니다. 제 생각엔 COM을 잘 이해하고 있다면 고급 윈도우 개발자라고 해도 손색이 없을 것입니다.

---

[7] 이 점은 문제가 좀 있다

3. Direct3D11

Direct2D의 장점 중 하나는 Direct3D와의 상호운용Interoperability이 가능하다는 것입니다. 즉 Direct3D와 Direct2D를 함께 쓸 수 있습니다. 이는 상당히 유연하고 효과적인 기능입니다. 특히나 DirectWrite의 뛰어난 성능을 생각해 보았을 때 이는 굉장히 매력적입니다. 이번 장에서는 간단히 Direct3D11에서 Direct2D용 렌더타겟을 생성하는 예제만을 보여드리겠습니다. 물론 Direct3D10에서도 가능합니다.

크게 4가지로 요약할 수 있습니다.

1. D3D11CreateDevice 혹은 D3D11CreateDeviceAndSwapChain 함수에서 Flags 인자에 D3D11_CREATE_DEVICE_BGRA_SUPPORT를 추가합니다.
2. Direct2D로 그릴 텍스처를 QueryInterface메서드를 이용해 IDXGISurface 형식으로 캐스팅합니다. 이때 텍스처는 생성될 때 Bind 필드를 RENDER_TARGET_OUTPUT 타입으로 지정되어 있었어야 합니다. 만약 스왑체인의 버퍼라면 생성될 때에 DXGI_SWAP_CHAIN_DESC의 BufferUsage 필드가 DXGI_USAGE_RENDER_TARGET_OUTPUT여야 합니다.
3. Direct2D 팩토리의 CreateDxgiRenderTarget 메서드로 렌더타겟 객체를 생성합니다. 이때 D2D1_RENDER_TARGET_PROPERTIES 인자는 D2D1Helper 내의 함수의 기본 인자를 사용하면 안되며, Factory로 DPI를 얻어오고 D2D1_PIXEL_FORMAT 인자 역시 직접 지정해 주어야 합니다.

Direct3D11 With Direct2D 의 WinMain.cpp 중 일부

```
bool Setup(HWND hwnd)
{
	// 윈도우 크기 얻어오기
	RECT clientRect;
	GetClientRect(hwnd, &clientRect);

	// 스왑체인 생성 준비
	DXGI_SWAP_CHAIN_DESC desc = { 0 };
	desc.BufferCount = 1;
	desc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
	desc.BufferDesc.RefreshRate.Numerator = 60;
	desc.BufferDesc.RefreshRate.Denominator = 1;
	desc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
	desc.OutputWindow = hwnd;
	desc.Windowed = true;
	desc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
```

```
	desc.SampleDesc.Count = 1;

	// Direct3D11 장치와 문맥과 교환사슬 생성
	HRESULT hr = D3D11CreateDeviceAndSwapChain(
		nullptr,
		D3D_DRIVER_TYPE_REFERENCE,
		nullptr,
		D3D11_CREATE_DEVICE_BGRA_SUPPORT, // 필수!
		nullptr,
		0,
		D3D11_SDK_VERSION,
		&desc,
		&swapChain,
		&device,
		nullptr,
		&context
		);

	IDXGISurface *buffer = nullptr;
	ID2D1Factory *factory = nullptr;

	if (SUCCEEDED(hr))
	{
		// 스왑체인의 렌더타겟 버퍼를 IDXGISurface형식으로 얻어온다
		hr = swapChain->GetBuffer(0, IID_PPV_ARGS(&buffer));
	}
	if (SUCCEEDED(hr))
	{
		// D2D1팩토리 생성
		hr = D2D1CreateFactory(
			D2D1_FACTORY_TYPE_SINGLE_THREADED,
			&factory);
	}
	if (SUCCEEDED(hr))
	{
		float dpiX, dpiY;

		// 데스크톱 DPI를 얻어옵니다
		factory->GetDesktopDpi(&dpiX, &dpiY);

		hr = factory->CreateDxgiSurfaceRenderTarget(
			buffer,
			D2D1::RenderTargetProperties(
			D2D1_RENDER_TARGET_TYPE_DEFAULT,
			D2D1::PixelFormat(
```

```
                        desc.BufferDesc.Format,
                        D2D1_ALPHA_MODE_PREMULTIPLIED),
                    dpiX,
                    dpiY
                    ),
                    &rt
                    );
        }

        // 필요 없는 리소스는 해제
        SafeRelease(buffer);
        SafeRelease(factory);

        return SUCCEEDED(hr);
}
```

4. 스크럼

5. 형상관리도구

6. Doxygen

7. 단위 검사

8. 참조

Microsoft. (2014). "Direct2D and Direct3D Interoperability Overview". MSDN: http://msdn.microsoft.com/en-us/library/windows/desktop/dd370966(v=vs.85).aspx에서 검색됨

Wikipedia. (2014년 November월 30일). "Library (Computing)". Wikipedia: http://en.wikipedia.org/wiki/Library_(computing)에서 검색됨