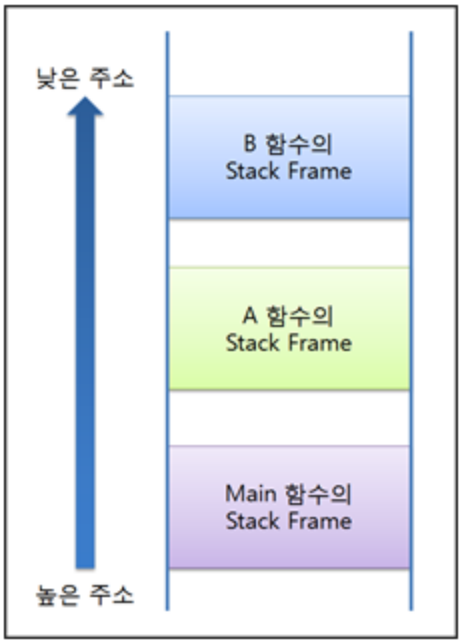
Calling Convention

[기본적인 개념]

1. 개요
- 함수 호출 규약(Calling Convention)이란, 함수를 호출하는 방식에 대한 약속

2. Stack Frame
- 함수를 호출 할 때 상위에서 진행하던 함수의 정보를 저장하고 인자를 전달하기 위해 Stack Frame이라는 구조를 사용한다.



▲함수의 Stack Frame

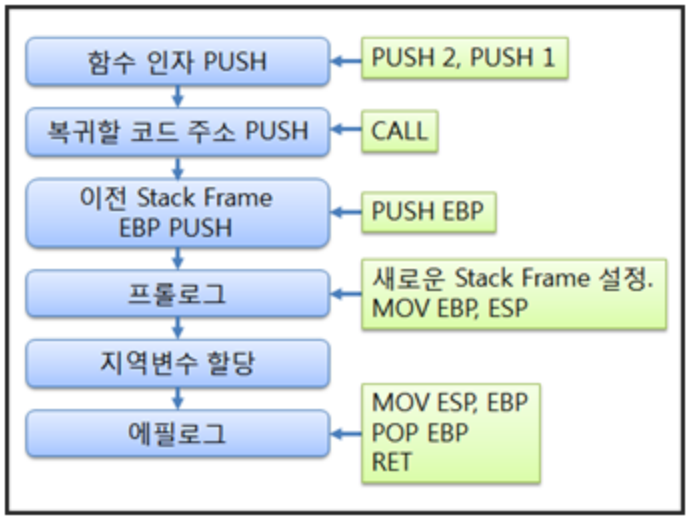
▲Stack Frame의 구조

Stack Frame은 함수의 호출 과정에서 호출되는 함수가 사용하기 위해 할당되는 Stack의 공간을 의미한다.

프로그램이 실행되면 가장 먼저 Main 함수의 Stack Frame을 할당한다. Main 함수가 A 함수를 호출하면서 A의 Stack Frame을 할당하고, 다시 A 함수 내에서 B함수를 호출하면 B의 Stack Frame을 할당한다.

Stack Frame에는 함수를 호출할 때 입력한 전달 인자, 함수가 종료될 때 복귀할 명령어의 주소(return address), 이전 Stack Frame의 EBP값을 저장하고, 지역변수를 저장하기 위한 공간을 할당한다.

5. 함수 호출 시 Stack의 변화 순서



4. 함수 호출 규약의 구분 방법

인자 전달	인자 전달의 순서 (왼쪽 인자부터 / 오른쪽 인자부터)
	인자 전달에 사용하는 매체 (Stack / 레지스터)
Stack Frame 정리 방법	Caller를 이용한 정리
	Callee를 이용한 정리

1) Caller : 호출자 (다른 함수를 호출 한 함수)
2) Callee : 호출된 자 (호출을 당한 함수)

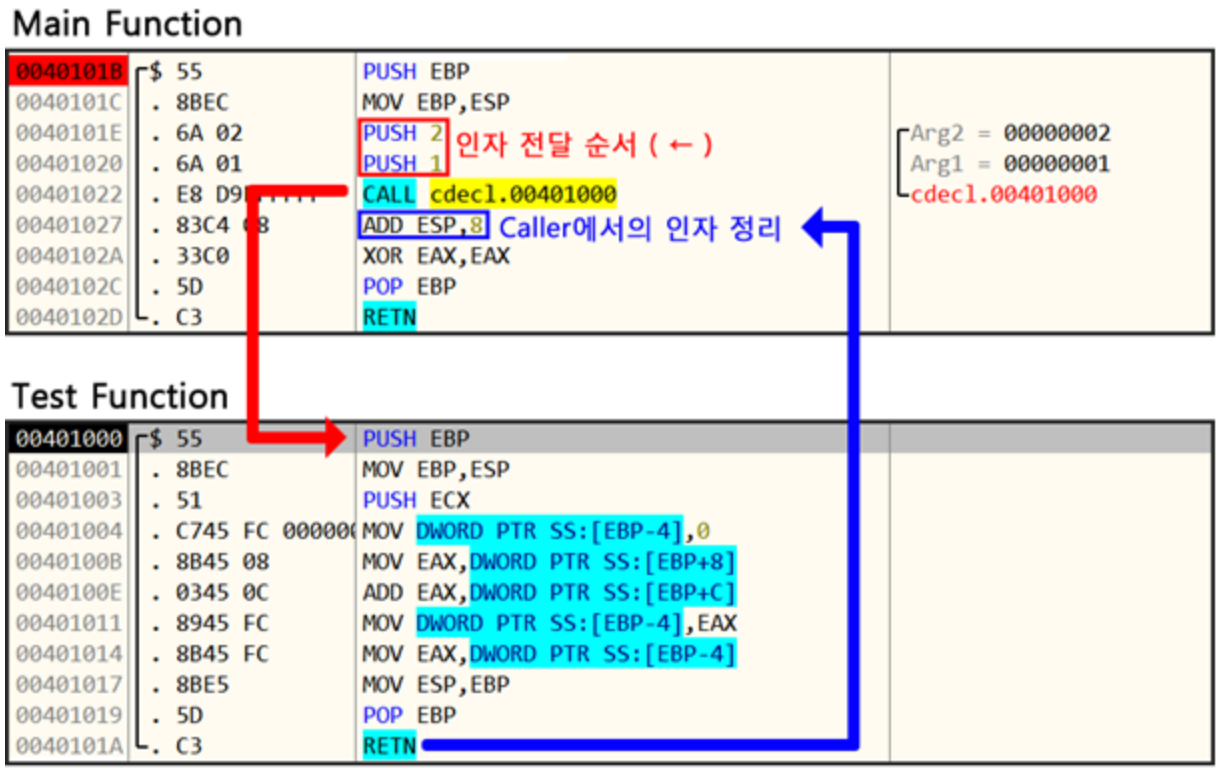
5. 함수 호출 규약의 종류

규약	인자 전달 순서	인자 전달 매체	Stack를 정리하는 함수
cdecl	←	Stack	Caller
stdcall	←	Stack	Callee
fastcall	←	레지스터 + Stack	Callee

1. cdecl

인자 전달 순서	가장 오른쪽 인자부터 전달한다. (←)
인자 전달 매체	Stack 을 사용한다.
Stack Frame 정리 방법	함수를 호출한 Caller가 인자를 정리한다.
C언어와 C++에서의 표준 함수 호출 규약이다. Caller가 인자를 정리하는 규약이므로, 가변인자를 사용할 수 있다.	

```
#include <stdio.h>
int __cdecl cdecl_Test(int a, int b)
{
  int nSum = 0;
  nSum = a + b;
  return nSum;
}

int main()
{
  cdecl_Test(1, 2);
  return 0;
}
```




```
Dump of file colosseum_operator.dll

File Type: DLL

  Section contains the following exports for colosseum_operato
r.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name
```

```
          1     0 000F105D Command = @ILT+28760(_Command)

  Summary

        1000 .00cfg
        7000 .data
        2000 .idata
        1000 .msvcjmc
       53000 .rdata
       11000 .reloc
        1000 .rsrc
      1F1000 .text
        E9000 .textbss
        1000 .tls
```

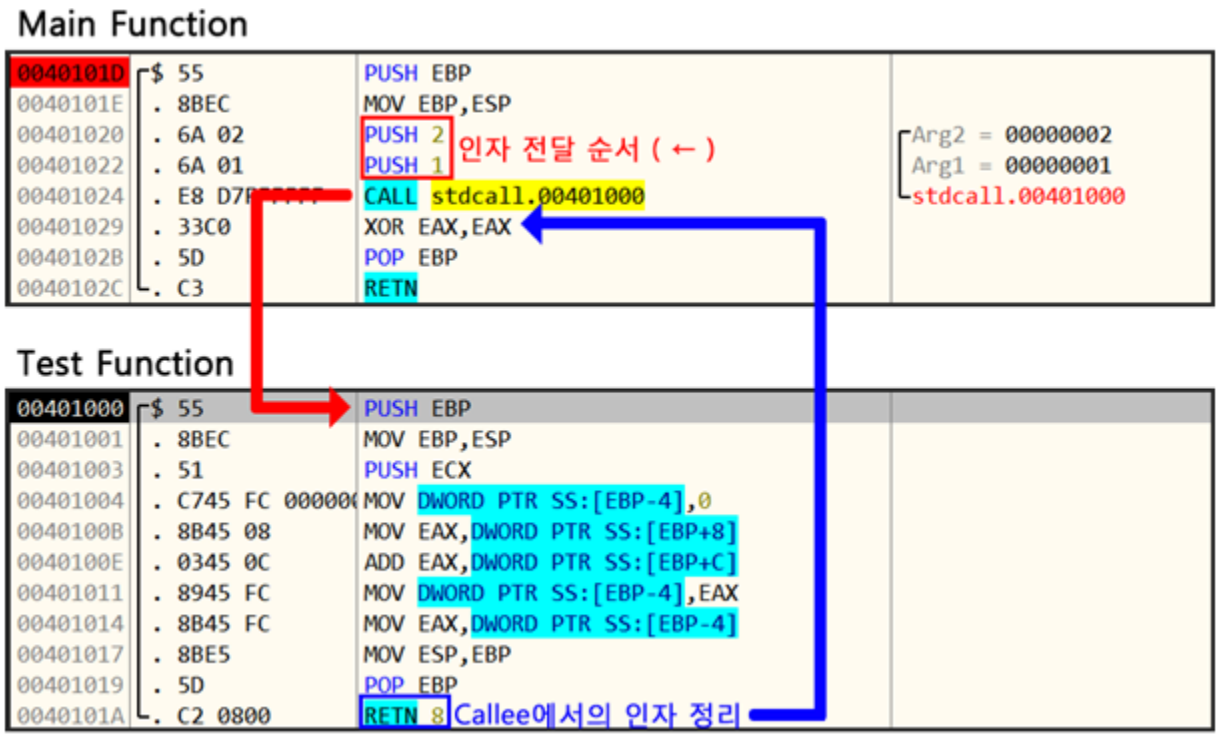
---

2. stdcall

인자 전달 순서	가장 오른쪽 인자부터 전달한다. (←)
인자 전달 매체	Stack 을 사용한다.
Stack Frame 정리 방법	호출을 당한 Callee가 함수를 종료하면서 인자를 정리한다.
Window API, Visual Basic에서 사용하는 표준 규약이다. 코드가 간결하지만 가변 인자를 사용할 수 없다.	

```
#include <stdio.h>
int __stdcall stdcall_Test(int a, int b)
{
  int nSum = 0;
  nSum = a + b;
  return nSum;
}
```

```
int main()
{
  stdCall_Test(1, 2);
  return 0;
}
```




```
Dump of file colosseum_operator.dll

File Type: DLL

  Section contains the following exports for colosseum_operato
r.dll

    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
```

```
           1 ordinal base
           1 number of functions
           1 number of names

    ordinal hint RVA      name

          1    0 000EC918 _Command@8 = @ILT+10515(_Command@8)

  Summary

        1000 .00cfg
        7000 .data
        2000 .idata
        1000 .msvcjmc
       53000 .rdata
       11000 .reloc
        1000 .rsrc
      1F1000 .text
        E9000 .textbss
        1000 .tls
```

2.1 stdcall 변환전 함수 이름 Export하기
DLL Source 아무 곳에나

```
#pragma comment(linker, "/export:Command=_Command@8")
```

```
Dump of file colosseum_operator.dll

File Type: DLL

  Section contains the following exports for colosseum_operato
r.dll
```

```
    00000000 characteristics
    FFFFFFFF time date stamp
        0.00 version
           1 ordinal base
           2 number of functions
           2 number of names

    ordinal hint RVA      name

          1    0 000EC918 Command = @ILT+10515(_Command@8)
          2    1 000EC918 _Command@8 = @ILT+10515(_Command@8)

  Summary

        1000 .00cfg
        7000 .data
        2000 .idata
        1000 .msvcjmc
       53000 .rdata
       11000 .reloc
        1000 .rsrc
       1F1000 .text
        E9000 .textbss
        1000 .tls
```

[Win32 DLL export 함수의 호출 규약]

1) x86 환경에서의 __cdecl, __stdcall에 대한 Name mangling

일반적으로 C/C++에서 별다르게 호출 규약을 지정하지 않고 다음과 같이 함수를 만든 후 Export 시키면

```
// C++ 헤더
__declspec(dllexport) int CDECL_Func(int value);

// C++ 구현 파일
__declspec(dllexport) int CDECL_Func(int value)
{
    printf("CDECL_Func: %d\n", value);
    return 42;
}
```

이는 호출 규약이 __cdecl로 정해진다.

```
// C++ 헤더
__declspec(dllexport) int __cdecl CDECL_Func(int value);

// C++ 구현 파일
__declspec(dllexport) int __cdecl CDECL_Func(int value)
{
    printf("CDECL_Func: %d\n", value);
    return 42;
}
```

위의 함수 이름(CDECL_Func)을 Visual Studio의 Native Tools CMD 창을 사용하여 확인(dumpbin /export Win32Project1.dll)해보면 "?CDECL_Func@@YAHH@Z" 처럼 나온다.

이렇게 개발자가 지정한 이름이 컴파일러의 필요에 따라 변경되는 것을 "name mangling"이라 한다.

"?CDECL_Func@@YAHH@Z" 문자열은 나름의 규칙을 가지는데 리턴 타입 및 각 인자에 따른 정보 등이 함축되어 있다. 따라서 해당 CMD창에서 (undname ?CDECL_Func@@YAHH@Z) 명령을 통하여 원래 함수의 Signature를 확인할 수 있다.

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC>undname ?CDECL_Func@@YAHH@Z
Microsoft (R) C++ Name Undecorator
```

```
Copyright (C) Microsoft Corporation. All rights reserved.


Undecoration of :- "?CDECL_Func@@YAHH@Z"
is :- "int __cdecl CDECL_Func(int)"
---
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC>undname
?CDECL_Func@@YA
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation. All rights reserved.


Undecoration of :- "?CDECL_Func@@YA"
is :- " ?? __cdecl CDECL_Func( ?? )"
---
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC>undname
?CDECL_Func@@YAH
Microsoft (R) C++ Name Undecorator
Copyright (C) Microsoft Corporation. All rights reserved.


Undecoration of :- "?CDECL_Func@@YAH"
is :- "int __cdecl CDECL_Func( ?? )"
```

C++의 export 함수 규칙이 이렇게 복잡한 것은 같은 이름의 함수 오버로딩과 가능 기능 때문이다. 문제는 이럴 경우 해당 DLL의 함수를 사용하는 측에서는 "?CDECL_Func@@YAHH@Z(value)"와 같은 방식으로 함수를 호출해야 하기 때문에 비효율적인 것을 알 수 있다.

즉, "@" 문자는 사용할 수 없으므로 EntryPoint 속성을 통해 이상한 이름을 직접 써줘야 한다.

```
using System.Runtime.InteropServices;
class Program
{
    [DllImport("Win32Project1.dll", EntryPoint = "?CDECL_Func@
@YAHH@Z")]
    internal static extern void CDECL_Func(int value);
```

```
    static void Main(string[] args)
    {
        CDECL_Func(5);
    }
}
```

이렇게 때문에 사용자하는 아래 키워드를 사용하여 export되는 함수들을 C 언어 시절처럼 단순한 이름으로 만드는 방법으로 감싸주어야 한다.

```
// C++ 선언부
#define Export_API extern "C" __declspec(dllexport)
Export_API int ExternC_CDECL_Func(int value);

// C++ 구현부
int ExternC_CDECL_Func(int value)
{
    printf("ExternC_CDECL_Func: %d\n", value);
    return 42;
}
```

이렇게 함으로써 C/C++ 컴파일러는 _cdecl 호출 규약을 유지하면서 이름을 함수 이름 그대로 내보내 줍니다. 또한 그럼으로써 사용하는 측에서 아래와 같이 쉽게 접근이 가능하다.

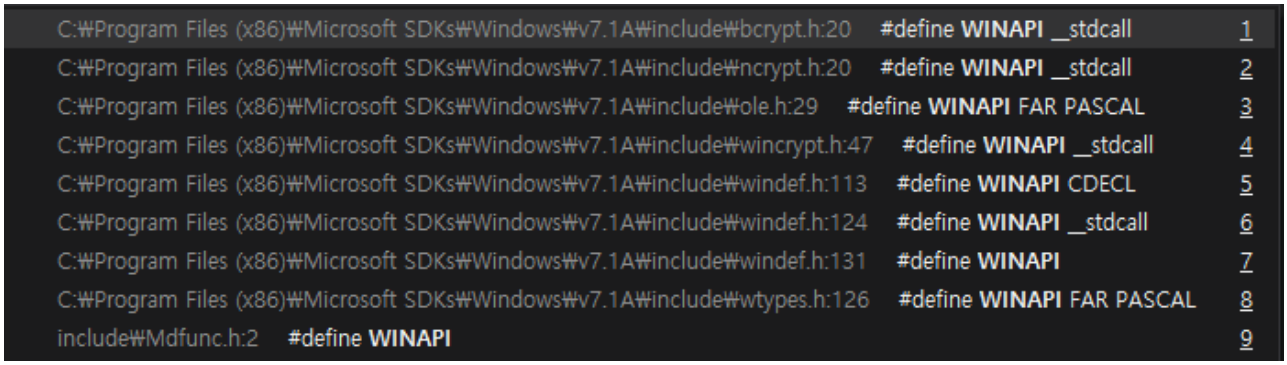
```
[DllImport("Win32Project1.dll")]
internal static extern void ExternC_CDECL_Func(int value);

static void Main(string[] args)
{
    ExternC_CDECL_Func(5);
}
```

Win32 API들은 _stdcall 호출 규약을 명시하고 있다.
즉, MS에서 제공하는 함수들을 보면 아래와 같이 WINAPI라고 선언되어 있는 것을 볼 수 있는데,

```
HANDLE WINAPI CreateEvent(
```

```
  _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
  _In_ BOOL bManualReset,
  _In_ BOOL bInitialState,
  _In_opt_ LPCTSTR lpName
);
```

아래와 같이 재정의 되어 있는 것을 알 수 있다.




```
// C++ 선언부
#define Export_API extern "C" __declspec(dllexport)
Export_API int __stdcall ExternC_STD_Func_Arg2(int value, int *pValue);

// C++ 구현부
int STD_Func(int value, int *pValue)
{
    printf("ExternC_STD_Func_Arg2: %d, %d\n", value, *pValue);
    return 42;
}
```

이렇게 하면 아래와 같이

```
_ExternC_STD_Func_Arg2@8
```

"_" 밑줄로 시작해 마지막은 '@' 글자와 함께 해당 함수의 인자로 요구되는 바이트 수가 포함됩니다.

- 4바이트 크기의 인자가 2개이므로 8바이트로 이렇게 됩니다. (4바이트가 아닌 인자 2개를 사용해도 4바이트 정렬이 되기 때문에 8로 됩니다.)

하지만 이런 이름 변경에 대해서는 C#에 알릴 필요는 없습니다. extern "C" + _cdecl이 적용된 경우와 동일하게 다음과 같이 해당 함수를 사용할 수 있습니다.

```
[DllImport("Win32Project1.dll", EntryPoint = "?STD_Func@@YGHH@Z")] // extern "C"가 없는 경우 직접 명시.
internal static extern int STD_Func(int value);


[DllImport("Win32Project1.dll")] // extern "C"가 있는 경우 함수 이름 그대로 사용.
internal static extern int ExternC_STD_Func(int value);


[DllImport("Win32Project1.dll")] // extern "C"가 있는 경우 함수 이름 그대로 사용.
internal unsafe static extern int ExternC_STD_Func_Arg2(int value, int *pValue);
```

이쯤 되면, export된 함수 이름만 보면 그것이 _cdecl인지, _stdcall인지 알 수 있습니다. 가령, 다음과 같은 이름이라면,

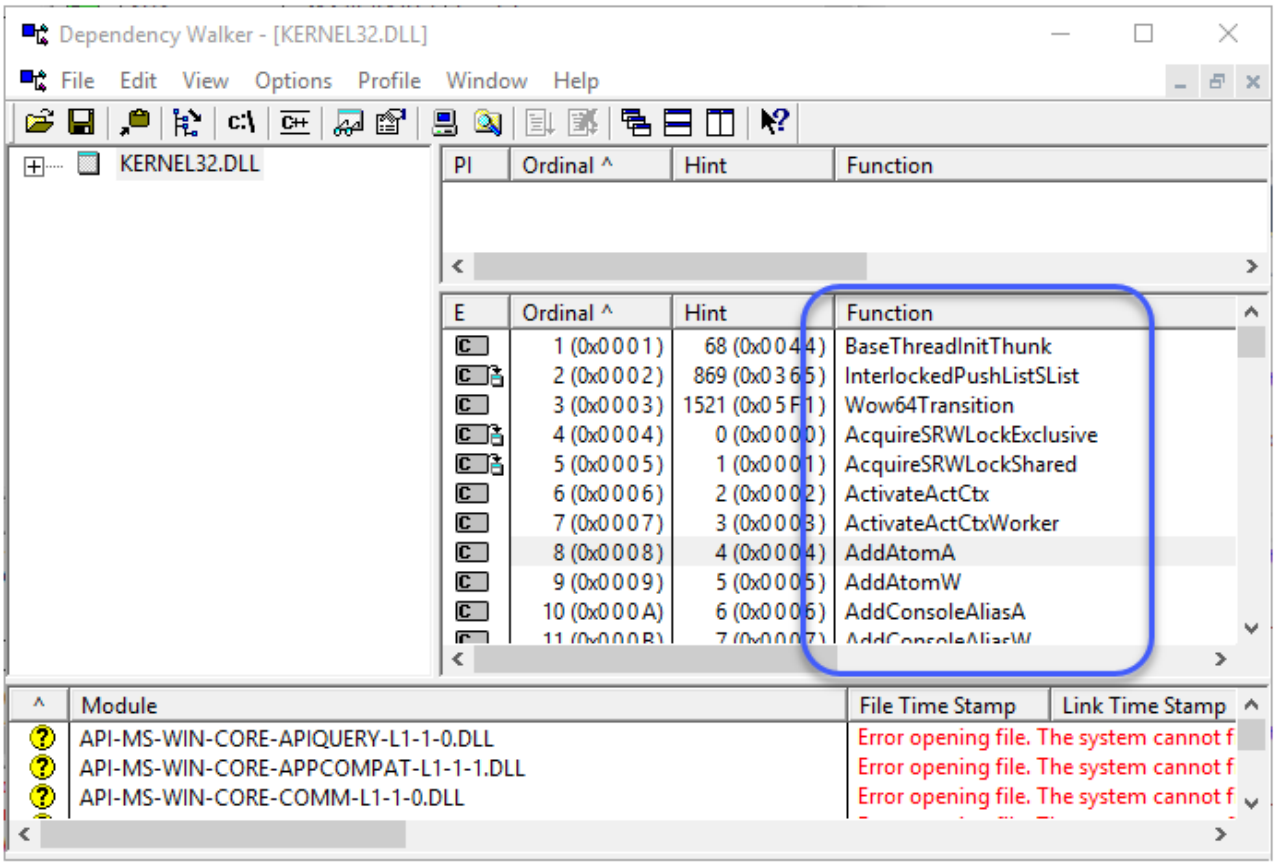
```
MyFunc
```

_cdecl 호출 규약에 extern "C"를 사용한 경우입니다. 반면 다음과 같이 구성된 이름이라면,

```
_MyFunc@12
```

인자가 3개인 _stdcall 호출 규약의 extern "C"가 적용된 경우입니다.

그런데, 아쉽게도 이런 규칙에 예외가 있습니다. 실제로 마이크로소프트의 kernel32.dll 등에 포함된 Win32 API들은 _stdcall 호출 규약을 따름에도 불구하고 마치 extern "C" + _cdecl 조합처럼 이름만 출력되는 형식입니다. 아래 그림은 kernel32.dll에서 export된 함수들의 목록을 depends.exe로 확인한 것입니다.



이유는 간단합니다. Visual C++의 경우 확장자가 ".def"인 "Module-Definition File"을 추가해 그 안에 export시키는 함수들과 그것들의 "ordinal number"를 지정할 수 있습니다. (ordinal number는 논외로 여기서는 다루지 않습니다.)

가령 다음과 같이 Source.def를 만들어 주면,

```
LIBRARY
    EXPORTS
        CDECL_Func_By_DEF
        ExternC_CDECL_Func_By_DEF
        STD_Func_By_DEF
        ExternC_STD_Func_By_DEF
```

여기 지정된 함수들은 C++ 헤더 파일에 호출 규약이나 extern "C"를 어떤 식으로 지정했든지 간에 상관없이,

```
__declspec(dllexport) int __cdecl CDECL_Func_By_DEF(int valu
e);
__declspec(dllexport) int __stdcall STD_Func_By_DEF(int valu
e);


extern "C"
{
    __declspec(dllexport) int __cdecl ExternC_CDECL_Func_By_DE
F(int value);
    __declspec(dllexport) int __stdcall ExternC_STD_Func_By_DE
F(int value);
}
```

export 함수의 이름 형식은 무조건 "함수 이름 자체"가 됩니다. 즉, 다음의 이름으로 export 가 됩니다.

```
CDECL_Func_By_DEF
ExternC_CDECL_Func_By_DEF
STD_Func_By_DEF
ExternC_STD_Func_By_DEF
```

이로 인해, export된 함수의 이름만 봐서는 그것이 _cdecl인지, _stdcall인지 단정 지을 수는 없습니다. 단지, "_" 밑줄과 "@바이트수"로 끝나는 형식이라면 _stdcall이 확실하다고 보시면 됩니다.

.def 파일에 지정된 함수의 경우, C#에서 별다른 설정 없이 이전과 동일하게 DllImport를 구성해 주면 됩니다.

```
[DllImport("Win32Project1.dll")]
internal static extern int CDECL_Func_By_DEF(int value);


[DllImport("Win32Project1.dll")]
```

```
internal static extern int ExternC_CDECL_Func_By_DEF(int valu
e);

[DllImport("Win32Project1.dll")]
internal static extern int STD_Func_By_DEF(int value);

[DllImport("Win32Project1.dll")]
internal static extern int ExternC_STD_Func_By_DEF(int value);
```

C#의 DllImport 특성은 기본 CallingConvention이 StdCall로 되어 있습니다.

```
DllImportAttribute Class
; https://msdn.microsoft.com/en-us/library/system.runtime.inte
ropservices.dllimportattribute(v=vs.110).aspx

DllImportAttribute.CallingConvention Field
; https://msdn.microsoft.com/en-us/library/system.runtime.inte
ropservices.dllimportattribute.callingconvention(v=vs.110).asp
x
```

```
The default value for the CallingConvention field is Winapi, w
hich in turn defaults to StdCall convention.
```

하지만, 이 글에서 설명한 모든 코드는 _cdecl, _stdcall에 상관없이 C#에서는 명시적인 CallingConvention 없이도 잘 실행됩니다.

그렇다고 문제가 없는 것은 아닌데, 프로젝트의 대상 Framework가 .NET 4.0 이상으로 지정된 경우 이를 Visual Studio에서 "F5 (Start Debugging)"로 실행시키면 _cdecl 호출 규약의 메서드를 호출한 이후마다 다음과 같은 디버거 오류 창이 뜨게 됩니다. (.NET 3.5 이하라면 오류 창이 뜨지 않습니다.)

```
Managed Debugging Assistant 'PInvokeStackImbalance' has detect
ed a problem in 'C:\Win32Project1\ConsoleApplication1\bin\x86
\Debug\ConsoleApplication1.exe'.
```

```
Additional information: A call to PInvoke function 'ConsoleApp
lication1!Program::ExternC_CDECL_Func' has unbalanced the stac
k. This is likely because the managed PInvoke signature does n
ot match the unmanaged target signature. Check that the callin
g convention and parameters of the PInvoke signature match the
target unmanaged signature.

If there is a handler for this exception, the program may be s
afely continued.
```

MDA(Managed Debugging Assistant) 예외 창에 대해서는 이전에도 몇 번 설명한 적이 있습니다.

```
문제 재현 - Managed Debugging Assistant 'DisconnectedContext' h
as detected a problem in '...'
; https://www.sysnet.pe.kr/2/0/10961

CallbackOnCollectedDelegate was detected
; https://www.sysnet.pe.kr/2/0/710

C++로 만든 DLL 을 C#에서 사용하기
; https://www.sysnet.pe.kr/2/0/11111
```

즉, Visual Studio가 잠재적인 버그 상황을 감지했을 때 띄워주는 대화창인데요. 이를 무시하면 해당 프로그램이 잘 동작할 수도 있지만 경우에 따라서 그렇지 않을 수도 있음을 미리 경고해 주는 것입니다. 따라서, 이 예외창이 뜨지만 실제로 "Ctrl + F5 (Start Without Debugging)"으로 실행했을 때 잘 동작한다고 해서 그냥 무시하고 지나가는 것은 좋지 않습니다.

위와 같은 경우, 결국 _cdecl 호출 규약에 대해 DllImport에서 명시적으로 CallingConvention을 설정해 주면 됩니다. 즉, 이 글의 예제 같은 경우에는 다음과 같은 DllImport 함수들이 모두 지정되어야 합니다.

```
[DllImport("Win32Project1.dll", EntryPoint = "?CDECL_Func@@YAH
H@Z", CallingConvention = CallingConvention.Cdecl)]
internal static extern int CDECL_Func(int value);

[DllImport("Win32Project1.dll", CallingConvention = CallingCon
vention.Cdecl)]
internal static extern int CDECL_Func_By_DEF(int value);

[DllImport("Win32Project1.dll", CallingConvention = CallingCon
vention.Cdecl)]
internal static extern int ExternC_CDECL_Func(int value);

[DllImport("Win32Project1.dll", CallingConvention = CallingCon
vention.Cdecl)]
internal static extern int ExternC_CDECL_Func_By_DEF(int valu
e);
```