

윈도우 프로그래밍 테크닉

호출(calling)의 예술

목차

목차.....	1
License.....	1
소개.....	1
연재 가이드.....	1
필자소개.....	2
필자 메모.....	2
Introduction.....	2
함수 호출 규약(calling convention).....	3
cdecl 호출 규약.....	4
stdcall 호출 규약.....	4
fastcall 호출 규약.....	5
thiscall 호출 규약.....	6
x64 호출 규약.....	7
액티베이션 레코드(Activation record).....	8
스택 프레임(Stack Frame).....	9
콜백 함수.....	10
콜백 클래스, 인터페이스.....	11
동적 콜백 멤버 함수.....	12
도전 과제.....	15
참고자료.....	15

License

Copyright © 2007, 신영진

이 문서는 Creative Commons 라이선스를 따릅니다.

<http://creativecommons.org/licenses/by-nc-nd/2.0/kr>

소개

cdecl, stdcall, fastcall, thiscall, x64 호출 규약에 대해서 알아보고, 함수를 호출하는 과정에서 스택이 어떻게 이용되는지 살펴본다. 이러한 지식을 토대로 컨텍스트 정보를 넘겨받지 못하도록 설계된 API에 클래스 멤버 함수를 콜백으로 전달하는 방법에 대해서 알아본다.

연재 가이드

운영체제: 윈도우 2000/XP

개발도구: Visual Studio 2005

기초지식: C/C++, Assembly, Win32 API

응용분야: 콜백을 이용하는 프로그램

필자소개

신영진 pop@jiniya.net, <http://www.jiniya.net>

시스템 프로그래밍에 관심이 많으며 다수의 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. 최근에는 SpiderMonkey를 임베딩 시켜서 사용하는 것에 관심이 많다.

필자 메모

요즘의 프로그래밍 언어나 시스템들은 개발자에게 고도로 추상화된 환경을 제공한다. 화면에 점을 찍는 작업을 생각해 보자. DOS 시절에는 그 단순한 작업을 하기 위해서 VGA의 구조와 원리를 공부해야 했었다. 하지만 지금은 DC의 핸들을 얻고, SetPixel을 호출하면 모든 작업이 끝난다. 점을 찍기 위해 VGA 구조를 공부하는 사람은 아무도 없다.

추상화는 개발자들에게 생산성 향상이라는 놀라운 선물을 안겨 주었다. 하지만 동시에 추상화는 개발자들이 하부 시스템에서 벌어지는 일들을 정확하게 볼 수 없도록 만들었다. 개발자들에게 양날의 검인 셈이다. 추상화로 잃어버린 시력을 되찾기 위해서는 추상화의 단계 밑에 있는 것들을 직접 살펴보고 이해해야 한다. 물론 이 과정은 지루하고, 시간이 오래 걸리며, 당장 지금의 작업에 도움이 안 되는 것처럼 보인다. 하지만 이 과정 끝에서 생각해 본다면 잃는 것 보다는 얻는 것이 더 많음을 느끼게 될 것이다.

Introduction

언젠가 필자는 <리스트 1>과 같은 코드에 대한 질문을 받은 적이 있다. 멤버 함수를 콜백으로 넘기는 예제다. 필자는 당연히 Proc 함수를 static으로 만들고 param으로 finder 포인터를 전달하라고 했다. 이것이 일반적인 해결책이다.

리스트 1 클래스의 멤버 함수를 콜백으로 넘기는 코드

```
class CFindWindowByPid
{
private:
    DWORD m_pid;
    std::vector<HANDLE> m_windows;

public:
    CFindWindowByPid(DWORD pid)
    {
        m_pid = pid;
    }

    BOOL CALLBACK Proc(HWND hwnd, LPARAM param)
    {
        DWORD pid;
        GetWindowThreadProcessId(hwnd, &pid);
        if(pid == m_pid)
            m_windows.push_back(hwnd);

        return TRUE;
    }
};
```

```

    }
};

int main()
{
    CFindWindowByPid finder(10);
    EnumWindows(finder.Proc, 0);
    return 0;
}

```

최근에 필자는 앞서 소개한 방법이 통하지 않는 무개념의 API를 하나 알게 되었다. EnumSystemCodePages란 API가 그것이다. 통상적으로 콜백을 지원하는 함수는 콜백 함수 포인터와 컨텍스트 정보를 전달할 포인터를 같이 인자로 받는다. 그런데 이 API는 컨텍스트 정보를 전달할 어떤 인자도 가지고 있지 않다. 이 API를 사용하기 위해서 우리가 할 수 있는 모든 일은 전역 변수를 쓰는 것이다. 아래 함수 원형이 그것을 말해준다. 콜백 함수인 EnumCodePagesProc으로 넘어오는 인자는 단 하나 코드 페이지 정보뿐이다.

```

BOOL EnumSystemCodePages(CODEPAGE_ENUMPROC lpCodePageEnumProc, DWORD dwFlags);
BOOL CALLBACK EnumCodePagesProc(LPTSTR lpCodePageString);

```

이런 잘못 설계된 콜백 함수들이 API에만 있는 것은 아니다. 오래 전 필자가 작업했던 한 해외 백신 엔진의 SDK도 이런 문제점을 가지고 있었다. 컨텍스트 정보를 전달할 방법이 전혀 없었고, 그 때 필자는 무척 곤란한 상황을 겪었다.

그렇다면 우리는 저런 API를 만날 때마다 전역 변수를 사용할 수 밖에 없을까? 좀 더 우아하게 해결할 수 있는 방법은 없을까? 약간의 어셈블리 지식과 호출 규약에 대한 지식이 있다면 답을 찾는 것이 그리 어렵진 않다. 호출 규약의 의미와 종류별 특징에 대해서 살펴보고, 최종적으로 어셈블리를 통해서 문제를 해결하는 방법에 대해서 알아보자.

함수 호출 규약(calling convention)

이 세상 모든 통신은 송신자, 수신자, 약속이라는 세 가지 구성 요소로 이루어진다. 함수를 호출하는 것도 코드 내부의 작은 통신이라 할 수 있다. 따라서 여기에서 앞서 말한 세 가지 구성요소가 존재한다. 송신자는 함수를 호출하는 곳이고, 수신자는 호출되는 함수가 된다. 마지막으로 약속에 해당하는 것이 호출 규약이다.

함수 호출과 관련된 약속의 주된 내용은 인자의 전달 방법, 전달 순서, 정리 방법, 리턴 값의 전달 경로다. 이런 약속 방법에 따라 다양한 호출 방법이 있다. Visual C++에서는 그 중에서도 cdecl, stdcall, fastcall, thiscall이라는 네 가지 방법을 지원한다. 함수 호출 규약을 지정하는 방법은 간단하다. 함수명과 리턴 값 사이에 호출 규약을 명시해주면 된다. 아래에 나와 있는 func1은 stdcall 호출 규약을 사용하는 함수가 된다. 호출 규약 앞에 언더스코어(_)를 두 개 붙인 것이 예약어로 사용된다.

```
void __stdcall func1();
```

다양한 호출 규약을 지원하는 것은 용도에 맞게 선택해서 쓸 수 있는 장점이 있다. 반면에 종류가 다양함으로 인해서 복잡하고 디버깅이 힘들어 지는 원인이 된다. 또한 호출 규약이 다를 경우 에러가 발생하기도 한다. 이러한 점 때문에 x64 환경에서는 새로운 호출 규약으로 단일화 시켰다. 따라서 Visual C++에서 사용할 수 있는 호출 규약에는 총 다섯 종류가 있는 셈이다. 각각의 호출 규약에 대해서 좀 더 자세히 살펴 보도록 하자.

cdecl 호출 규약

cdecl은 C 표준 함수 호출 규약이다. 특별한 지시자가 없으면 모든 C 함수는 기본적으로 cdecl 호출 규약을 사용한다. 파라미터는 오른쪽에서 왼쪽 순서로 스택을 사용해서 전달된다. 스택을 통해 전달한 파라미터는 호출한 곳에서 정리한다. <리스트 2>과 <리스트 3>에 간단한 cdecl 함수와 그것을 호출하는 어셈블리 코드가 나와있다.

리스트 2 cdecl 함수

```
extern "C" int __cdecl CdeclFunc(int a, int b, int c)
{
    printf("%d %d %d\n", a, b, c);
}
```

<리스트 3>를 살펴보자. 파라미터가 오른쪽에서 왼쪽 순서로 스택으로 전달되기 때문에 가장 먼저 push되는 3이 c에 해당한다. 2는 b에, 1은 a에 해당한다. 호출한 곳에서 스택을 정리해야 하기 때문에 자신이 push한 것들을 제거해 주어야 한다. pop을 통해서 비울 수 있지만 그렇게 할 경우에는 호출할 때의 들어간 인자만큼 pop을 해 주어야 하기 때문에 비효율적이다. 보통은 <리스트 3>에 나타난 것과 같이 esp를 직접 조작해서 스택을 비운다.

리스트 3 CdeclFunc를 호출하는 어셈블리 코드

```
push 3
push 2
push 1
call CdeclFunc
add esp, 12
```

cdecl 규약의 가장 큰 특징은 가변 인자를 지원한다는 것이다. 가변 인자 함수란 파라미터를 정해진 개수가 아닌 가변적으로 전달하는 것을 말한다. printf가 대표적인 함수다. cdecl 호출 규약이 가변 인자 함수를 지원할 수 있는 이유는 호출한 곳에서 스택을 정리하기 때문이다. 스택에 인자를 넣은 곳에서 제거 하기 때문에 자신이 인자를 얼마나 전달했는지 정확하게 알 수 있고, 그 정보를 토대로 정확하게 삭제할 수 있는 것이다.

stdcall 호출 규약

stdcall은 윈도우의 표준 함수 호출 규약이다. 특별한 표기가 없는 한 대부분의 API는 모두 이 호출 규약을 사용한다. stdcall은 cdecl과 마찬가지로 스택을 통해서 오른쪽에서 왼쪽 순서로 파라미터를 전달한다. 단지 차이가 있다면 cdecl은 호출한 곳에서 스택을 정리하지만 stdcall은 호출을 당한 함수 내부에서 스택을 정리한다는 점이다. <리스트 4>에는 간단한 stdcall 함수가 <리스트 5>에는 그것을 호출하는 어셈블리 코드가 나와있다.

리스트 4 stdcall 함수

```
extern "C" int __stdcall StdcallFunc(int a, int b, int c)
{
    printf("%d %d %d\n", a, b, c);
}
```

Visual C++의 경우 stdcall을 사용하는 함수에 대해서는 <리스트 5>에 나타난 것과 같이 함수 이름을 장식한다. 앞쪽에 언더스코어를 붙이고, 뒤쪽에 @와 함께 인자의 바이트 수를 적어준다. StdcallFunc는 4바이트 인자 세 개를 받기 때문에 12가 붙는다.

리스트 5 StdcallFunc를 호출하는 어셈블리 코드

```
push 3
push 2
push 1
call _StdcallFunc@12
```

<리스트 6>에 StdcallFunc의 어셈블리 리스트가 나와있다. stdcall 호출 규약에서는 스택 정리를 함수 내부에서 한다고 했다. 함수 마지막에 있는 ret 12가 스택을 정리하는 역할을 한다. x86 어셈블리에서는 ret 다음에 숫자를 적어주면 그 만큼 스택을 비운 다음 리턴한다. 이러한 특징 때문에 stdcall 함수는 cdecl에 비해 두 가지 장점을 가진다. 수행 속도와 코드 크기가 그것이다. ret, add 명령어를 수행하는 것보다 ret 명령어를 한번 수행하는 것이 속도가 더 빠르다. 그리고 함수를 호출할 때마다 매번 add 명령어가 붙지 않기 때문에 프로그램의 전체 크기도 줄일 수 있다.

리스트 6 StdcallFunc 어셈블리 리스트

```
PUBLIC _StdcallFunc@12
_TEXT SEGMENT
_a$ = 8
_b$ = 12
_c$ = 16
_StdcallFunc@12 PROC NEAR
    push    ebp
    mov    ebp, esp

    ... 종략 ...

    pop    ebp
    ret    12
_StdcallFunc@12 ENDP
_TEXT ENDS
```

fastcall 호출 규약

fastcall은 이름처럼 빠른 실행을 위한 호출 규약이다. fastcall이 빠른 이유는 파라미터의 일부를 레지스터를 사용해서 전달하기 때문이다. x86 계열에서는 일반적으로 ecx, edx로 두 개의 파라미터를 전달하고, 나머지는 스택으로 전달한다. 파라미터는 오른쪽에서 왼쪽으로 전달되고, 스택 정리는 호출되는 함수 내부에서 한다.

리스트 7 fastcall 함수

```
extern "C" int __fastcall FastcallFunc(int a, int b, int c)
{
    printf("%d %d %d\n", a, b, c);
}
```

<리스트 8>에 보이는 것처럼 fastcall 함수도 이름 장식이 된다. stdcall과 다른 점은 앞쪽에 언더스코어(_)가 아닌 @가 붙는다는 점이다. 앞 쪽 두 개의 파라미터가 ecx와 edx를 통해 전달되는 것을 볼 수 있다.

리스트 8 FastcallFunc를 호출하는 어셈블리 코드

```
push 3
mov edx, 2
mov ecx, 1
call @FastcallFunc@12
```

불행하게도 fastcall 함수가 이름만큼 굉장히 빠르지는 않다. 인자의 개수가 두 개를 넘어서면 다른 호출 규약과 마찬가지로 스택을 사용하고, 인자의 개수가 두 개 이하라고 하더라도 함수가 복잡할 경우에는 인자 값을 다시 스택에 저장해야 하기 때문이다. fastcall 함수가 속도적인 측면에서 이득을 볼 수 있는 경우는 인자의 개수가 두 개 이하인 간단한 함수에서이다.

thiscall 호출 규약

thiscall은 C++의 멤버 함수를 위한 호출 규약이다. 기본적인 원칙은 stdcall과 동일하고, 추가적으로 ecx를 통해서 this 포인터를 전달한다는 특징이 있다. 멤버 함수에 특별한 호출 규약을 지정하지 않으면 thiscall이 사용된다. <리스트 9>에는 간단한 thiscall 함수가, <리스트 10>에는 그것을 호출하는 어셈블리 코드가 나와있다.

리스트 9 thiscall 함수

```
class CCallConv
{
public:
    int ThisCall(int a, int b, int c);
};

int CCallConv::ThisCall(int a, int b, int c)
{
    return printf("%d %d %d\n", a, b, c);
}
```

리스트 10 ThisCall 함수를 호출하는 어셈블리 코드

```
push 3
push 2
push 1
lea ecx, conv
call CCallConv::ThisCall
```

그렇다면 멤버함수에 다른 호출 규약을 지정하면 어떻게 될까? 그냥 지정된 호출 규약이 사용된다. 이 경우에 this 포인터는 첫 번째 인자로 전달된다.

지금까지 언급한 함수 호출 규약 별 특징이 <표 1>에 나와있다. “인자 전달 순서나 리턴 값을 다르게 처리하는 호출 규약도 있나요?”라고 물어보시는 분들이 종종 있다. 인자를 왼쪽에서 오른쪽으로 전달하는 방식은 볼랜드의 호출 규약 쪽에 많이 있다. 또한 과거 Visual C++이 지원했던 pascal 호출 규약도 왼쪽에서 오른쪽으로 전달한다. 그리고 EAX를 사용하지 않는 호출 규약에 대

해서는 필자도 아직까지 들어 본적이 없다. 보다 많은 호출 규약에 관한 정보를 알고 싶다면 참고 자료에 있는 함수 호출 규약에 관한 위키 페이지를 참고하자.

표 1 함수 호출 규약 별 특징

호출 규약	인자 전달 순서	인자 전달 방법	인자 파괴 위치	리턴 값	특징
cdecl	오른쪽에서 왼쪽	스택	호출한 곳	EAX, fp0(부동 소수)	가변 인자를 지원한다.
stdcall	오른쪽에서 왼쪽	스택	호출된 곳	EAX, fp0(부동 소수)	Windows 표준 호출 규약이다.
fastcall	오른쪽에서 왼쪽	레지스터, 스택	호출된 곳	EAX, fp0(부동 소수)	처음 두 개의 인자를 ECX, EDX를 통해 전달하기 때문에 빠르다.
thiscall	오른쪽에서 왼쪽	레지스터, 스택	호출된 곳	EAX, fp0(부동 소수)	ECX를 통해 this 포인터를 전달한다.

x64 호출 규약

64비트 환경으로 넘어오면서 함수의 호출 규약도 크게 변경되었다. 가장 큰 변화라면 앞서 소개한 네 가지 호출 규약을 통일해서 단일 호출 규약으로 만들었다는 점이다. 64비트 환경에서는 앞서 소개한 호출 규약을 지정하는 지시자인 `_cdecl`, `_stdcall`, `_fastcall`, `_thiscall`은 모두 무시되고, 64비트의 호출 규약을 사용하는 함수로 컴파일된다. 64비트 호출 규약은 앞서 소개한 `_fastcall` 호출 규약과 유사하다. 단지 동작 방식이 그것보다는 다소 복잡하다는 점이 특징이다.

64비트 호출 규약은 파라미터를 전달하기 위해서 지정된 네 개의 레지스터와 스택을 사용한다. RCX, RDX, R8D, R9D 레지스터를 앞 쪽 네 개의 인자를 위해서 사용한다. 인자가 실수인 경우에는 XMM0, XMM1, XMM2, XMM3이 사용된다. 네 개를 넘어서는 인자들은 `_fastcall`과 마찬가지로 스택을 통해서 전달된다. <리스트 11>에 이러한 특성이 잘 나와있다.

리스트 11 64비트 호출 규약을 사용하는 함수들

```
// a는 RCX, b는 RDX, c는 R8, d는 R9, e는 스택을 통해 전달된다.
func1(int a, int b, int c, int d, int e);

// a는 XMM0, b는 XMM1, c는 XMM2, d는 XMM3, e는 스택을 통해 전달된다.
func2(float a, double b, float c, double d, float e);

// a는 RCX, b는 XMM1, c는 R8, d는 XMM3를 통해 전달된다.
func3(int a, double b, int c, float d);

// a는 RCX를 통해 전달된다.
func4(int a);
```

64비트 호출 규약이 `_fastcall`과 다른 점은 레지스터를 통해 전달하는 변수에 대한 것까지 스택 공간을 할당해야 한다는 점이다. <리스트 11>의 함수에서 `func1`은 40(5*8)바이트의 스택 공간을

필요로 하고, func3은 32(4*8)바이트의 스택 공간을 필요로 한다. 여기에 덧붙여 인자의 개수가 네 개 이하인 함수에 대해서는 모두 기본적으로 32바이트의 공간을 할당해야 한다. 따라서 func4 함수에 필요한 스택 공간도 32바이트가 된다. 레지스터에 저장된 값을 보관할 스택을 별도로 두는 이유는 함수 내부에서 해당 레지스터를 사용하고자 할 때, 값을 보관하기 쉽도록 하기 위해서다.

리턴 값이 64비트에 저장될 수 있는 경우에는 RAX를 통해 반환되며, 실수 타입인 경우에는 XMM0를 통해서 반환된다. 만약 리턴 값을 64비트에 저장할 수 없다면 호출하는 곳에서 해당 리턴 값에 대한 포인터를 첫 번째 인자로 전달해야 한다.

스택 정리는 호출한 곳에서 한다. 그런데 특이한 점은 스택 정리를 함수 호출을 할 때마다 하지 않는다는 것이다. 컴파일러는 해당 지역에서 호출되는 함수 중에 가장 많은 인자를 필요로 하는 함수에 맞추어 스택을 할당한 다음 해당 스택 공간을 지속적으로 활용한다. 그리고 함수가 끝나기 직전에 한번만 스택을 정리해 준다. 스택 포인터(RSP)를 조작하는 일이 빈번하지 않기 때문에 근소한 속도 향상을 가져온다고 할 수 있다. <리스트 12>는 앞서 나온 func1과 func4를 호출하는 코드로 이러한 특징을 잘 보여준다.

리스트 12 func1과 func4를 호출하는 어셈블리 코드

```
main PROC
    sub     rsp, 40

; func1(1,2,3,4);
    mov     r9d, 4
    mov     r8d, 3
    mov     edx, 2
    mov     ecx, 1
    call    func1

; func4(1);
    mov     ecx, 1
    call    func4

; return 0;
    xor     eax, eax

    add     rsp, 40
    ret     0
main ENDP
```

액티베이션 레코드(Activation record)

스택에 파라미터와 복귀 주소 등의 함수 호출과 관련된 정보가 저장된 것을 액티베이션 레코드라 부른다. 아래 코드를 호출하는 과정을 생각해 보자.

```
void func2(int) { func3(); }
void func1(int, int, int) { func2(1); }
int main() { func1(1,2,3); return 0; }
```

<그림 1>에 각 함수 호출에 따른 액티베이션 레코드 구조가 나와있다. main에서 func1을 호출하는 과정이 가장 왼쪽 그림이다. 파라미터가 오른쪽에서 왼쪽 순서대로 스택에 저장되고, call func1이 수행되는 순간 func1이 리턴 됐을 때 복귀해야 하는 주소가 스택에 저장된다. 다음으로 두 번째 그림은 func2가 호출되는 과정을, 세 번째 그림은 func3이 호출되는 과정을 보여준다. 각 함수

가 리턴 하면 쌓여있는 액티베이션 레코드가 하나씩 사라진다. 이러한 식으로 스택에 함수 호출 과정이 기록되기 때문에, 디버깅 과정 중에 손쉽게 호출 스택을 추적할 수 있다.

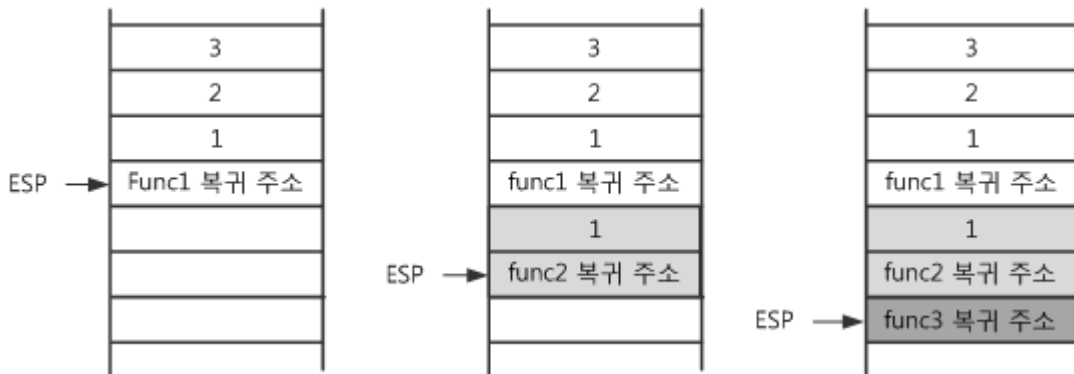


그림 1 함수 호출에 따른 스택의 변화

버퍼 오버플로와 같은 공격이 위험한 이유는 이러한 액티베이션 레코드를 손상 시키기 때문이다. 복귀 주소를 넘어서 기록할 경우 복귀 주소 값이 변경되고, 리턴하면 엉뚱한 주소로 점프한다.

스택 프레임(Stack Frame)

실제로 고급 언어로 작성되는 함수의 경우 앞서 소개한 함수와 같이 단순한 경우는 거의 없다. 대부분의 경우 함수 내부에서 사용하기 위한 지역 변수가 존재한다. 이러한 지역 변수를 효과적으로 관리하기 위해서 함수에 진입하면 해당 함수는 스택 프레임을 생성한다. 스택 프레임은 해당 함수가 지역 변수를 손쉽게 참조할 수 있게 도와 주고, 또한 디버깅을 용이하게 만들어 준다.

```
void func(int a, int b) { int c, d; }
```

위 함수에 대한 스택 프레임이 <그림 2>에 나와있다. EBP가 프레임 포인터가 된다. 그림과 같이 스택 프레임을 구성할 경우 지역 변수를 EBP+4, EBP+8과 같이 손쉽게 참조할 수 있다. 또한 저장된 이전 EBP를 토대로 이전 위치의 스택 프레임으로 거슬러 올라갈 수 있다.

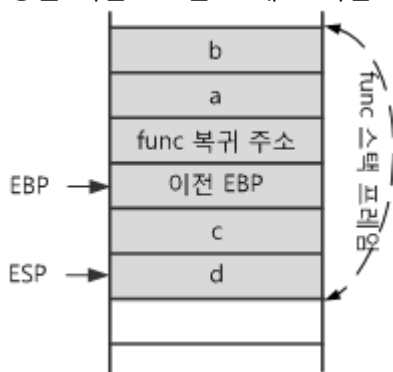


그림 2 func의 스택 프레임

함수 시작부에서 스택 프레임을 구성하는 코드를 프롤로그라고 하고 함수 끝에서 스택 프레임을 제거하는 부분을 에필로그라고 한다. func 함수에 대한 프롤로그와 에필로그가 <리스트 13>에 나와 있다. 프롤로그는 ebp를 저장하고 스택 공간을 확보하는 일을 한다. 에필로그는 확보된 스택

공간을 제거하고, ebp를 복구하는 것이 주된 일이다.

리스트 13 func의 프로로그와 에필로그

```
; 프로로그
push ebp
move ebp, esp
sub esp, 8

; 에필로그
move esp, ebp
pop ebp
ret
```

일반적인 C/C++ 함수들의 경우 이러한 스택 프레임을 컴파일러가 알아서 작성해 준다. 하지만 가끔 이러한 프로로그, 에필로그 코드를 직접 작성해야 하는 경우가 있다. 대표적인 예가 인터럽트 함수다. 인터럽트 함수의 경우 ret으로 리턴하지 않고, iret으로 리턴 해야 하기 때문이다. 이럴 때 도움이 되는 것이 naked 함수다. naked 함수는 컴파일러가 스택 프레임을 생성하지 않는 함수다. 모든 일을 직접 알아서 처리해 주어야 한다. naked 함수는 아래와 같이 __declspec(naked) 속성을 지정해서 간단하게 만들 수 있다.

```
void __declspec(naked) func() { _asm ret }
```

콜백 함수

콜백 함수란 말 그대로 나중에 불러주는 함수다. 호출된 함수 내부에서 구한 정보를 외부에서 임의대로 처리하기 위해서 많이 사용하는 방식이다. 윈도우를 열거하는 EnumWindows 함수를 생각해 보자. EnumWindows는 윈도우를 열거해서 각 윈도우 핸들을 구한다. 그 핸들을 어떻게 처리해야 할지는 EnumWindows에서 알 수 없다. 그 정보를 요청한 외부에서만 알 수 있다. 이 경우에 두 가지 접근 방법이 생긴다. 구한 윈도우 목록을 리스트 형태로 저장해서 외부로 리턴해 주거나, 콜백 함수를 사용해서 정보에 대한 처리를 외부에서 하도록 만드는 것이다. 보통의 경우 효율성 관 유연성 관점에서 콜백 함수가 이익이기 때문에 콜백을 많이 사용한다.

콜백 함수를 디자인할 때에는 세 가지에 신경을 써야 한다. 첫째, 콜백 함수의 인자로 사용자 정의 파라미터를 포함시킨다. 콜백을 사용하는 입장에서는 이 파라미터가 없다면 컨텍스트를 관리할 방법이 없다. 둘째, 콜백을 호출하는 함수를 콜백 함수를 통해 제어할 수 있도록 해야 한다. 보통 콜백 함수의 리턴 값으로 제어한다. 리턴 값이 FALSE인 경우 콜백을 호출하는 본체 함수도 리턴하도록 디자인한다. 셋째, 콜백 함수로 전달되는 정보는 핵심만 포함하도록 한다. 앞서 소개한 EnumWindows의 경우 윈도우 핸들만 인자로 전달한다. 나머지 정보들(윈도우 캡션 명, 클래스 명 등)은 윈도우 핸들로부터 구할 수 있기 때문이다. 전달하는 정보가 많아지면 함수 호출 오버헤드만 늘어나고, 그 정보를 사용하지 않을 경우는 고스란히 낭비되기 때문이다. <리스트 15>와 <리스트 16>에 전형적인 콜백 함수의 구조와 그것을 호출하는 코드가 나와있다.

리스트 14 전형적인 콜백 함수의 구조

```
typedef BOOL (CALLBACK *ENUMFILE_PROC)(LPCTSTR filename, LPARAM param);

BOOL EnumFiles(LPCTSTR dir, ENUMFILE_PROC proc, LPARAM param)
```

```

{
    while(...)
        if(!proc(filename, param))
            return FALSE;

    return TRUE;
}

```

리스트 15 콜백 함수를 호출하는 코드

```

BOOL CALLBACK MyEnumFileProc(LPCTSTR filename, LPARAM param)
{
    LPCTSTR dest = (LPCTSTR) param;

    if(_tcscmp(filename, dest) == 0)
    {
        printf("find\n");
        return FALSE;
    }

    return TRUE;
}

EnumFiles(_T("c:\\"), MyEnumFileProc, (LPARAM) _T("config.sys"));

```

콜백 클래스, 인터페이스

콜백 함수의 경우 C언어의 부족한 데이터 표현 형식에서 비롯된 것이다. C++에서는 향상된 여러 가지 언어적 표현 기법이 존재하기 때문에 콜백 함수 보다는 그것을 클래스로 포장한 인터페이스 클래스를 사용하는 것이 더 좋다. 인터페이스 클래스를 사용할 때의 장점은 데이터와 콜백 함수가 합쳐진다는 데 있다. 콜백 함수에 따른 구조체를 만들지 않아도 되기 때문에 네이밍 비용을 낮출 수 있고, 구조체 포인터를 LPARAM으로 변환하고 그것을 다시 복호화하는 저 수준의 코드를 작성하지 않아도 된다.

이러한 인터페이스 클래스는 OOP의 상속과 다형성을 사용하면 손쉽게 구현할 수 있다. <리스트 16>는 앞서 작성한 EnumFiles의 콜백 함수 버전을 인터페이스 클래스 구조로 변경한 내용을 담고 있다.

리스트 16 전형적인 인터페이스 클래스

```

class CEnumFileProc
{
public:
    virtual ~CEnumFileProc() {}
    virtual BOOL Invoke(LPCTSTR filename) = 0;
};

CMyEnumFileProc : public CEnumFileProc
{
public:
    virtual BOOL Invoke(LPCTSTR filename)
    {
        printf("%s\n", filename);
    }
};

BOOL EnumFiles(LPCTSTR dir, CEnumFileProc &proc)
{
    while(...)
        if(!proc.Invoke(filename))
            return FALSE;

    return TRUE;
}

```

}

동적 콜백 멤버 함수

자 이제 우리가 처음에 제기 했던 문제를 해결할 수 있는 이론적 배경은 모두 갖추었다. 조금 더 필요한 게 있다면 어셈블리 지식이라 할 수 있다. 여기에 사용되는 어셈블리는 쉽기 때문에 아주 기초적인 지식만 있으면 이해하는데 문제가 없다.

일단 이해를 쉽게 하기 위해서 결과 코드부터 보도록 하자. 우리가 문제를 해결한 코드는 <리스트 17>과 같은 형태가 될 것이다. 클래스 멤버 함수를 stdcall 호출 규약을 가지는 콜백 함수로 맵핑하는 템플릿 클래스를 작성하는 것이 핵심이다.

리스트 17 결과 코드

```
class CTestCodepage
{
public:
    BOOL OnCodePagesProc(LPTSTR lpCodePageString)
    {
        std::cout << lpCodePageString << std::endl;
        return true;
    }
};

int main()
{
    CTestCodepage asdf;
    CDynamicCallback<CODEPAGE_ENUMPROC> bb(&asdf, &CTestCodepage::OnCodePagesProc);
    EnumSystemCodePages(bb, CP_INSTALLED);
    return 0;
}
```

<리스트 17>에서 EnumSystemCodePages가 bb를 호출하면 <리스트 18>에 나와있는 DynamicCallbackOpCodes 구조체로 점프한다. 전체 함수를 어셈블리로 만드는 일은 복잡하기 때문에 이 함수는 간단하게 중간 함수로 점프하는 역할과 함께 중간 함수에서 사용해야 할 데이터를 포함하고 있다. offset은 이 구조체의 시작 번지부터 중간 함수까지의 상대 주소를 담고 있다. _this에는 &asdf가, _func에는 &CTestCodepage::OnCodePagesProc가 저장된다(<리스트 17> 참고).

리스트 18 DynamicCallbackOpCodes

```
#pragma pack(push, 1)
struct DynamicCallbackOpCodes
{
    unsigned char tag; // e8: CALL에 해당하는 OPCODE
    LONG_PTR offset; // 멤버 함수로 점프할 징검 다리 함수 주소 오프셋
    LONG_PTR _this; // 클래스 인스턴스 포인터
    LONG_PTR _func; // 실제 호출할 멤버 함수 포인터
};
#pragma pack(pop)
```

<리스트 19>에 나와있는 StdDynamicJumpProc이 멤버 함수를 호출해 주는 중간 함수다. 간단한 어셈블리이기 때문에 호출 흐름만 알면 쉽게 이해할 수 있다. 지금까지 설명한 함수의 호출 순서가 <그림 3>에 나와있다. EnumSystemCodePages에서 bb를 호출하면 할당된 메모리 블록인 DynamicCallbackOpCodes를 호출한다(1번 단계). DyanmicCallbackOpCodes는 다시 어셈블리로 작

성된 중간 함수인 StdDyanamicJmpProc을 호출한다(2번 단계). StdDyanmicJmpProc에서는 DynamicCallbackOpCodes에서 넘겨준 정보를 바탕으로 실제 클래스 함수로 점프한다(3번 단계).

리스트 19 StdDynamicJmpProc

```
static __declspec( naked ) int StdDynamicJmpProc()
{
    _asm
    {
        POP ECX
        MOV EAX, DWORD PTR [ECX + 4] // 실제 호출될 함수 주소
        MOV ECX, [ECX] // this 포인터
        JMP EAX
    }
}
```

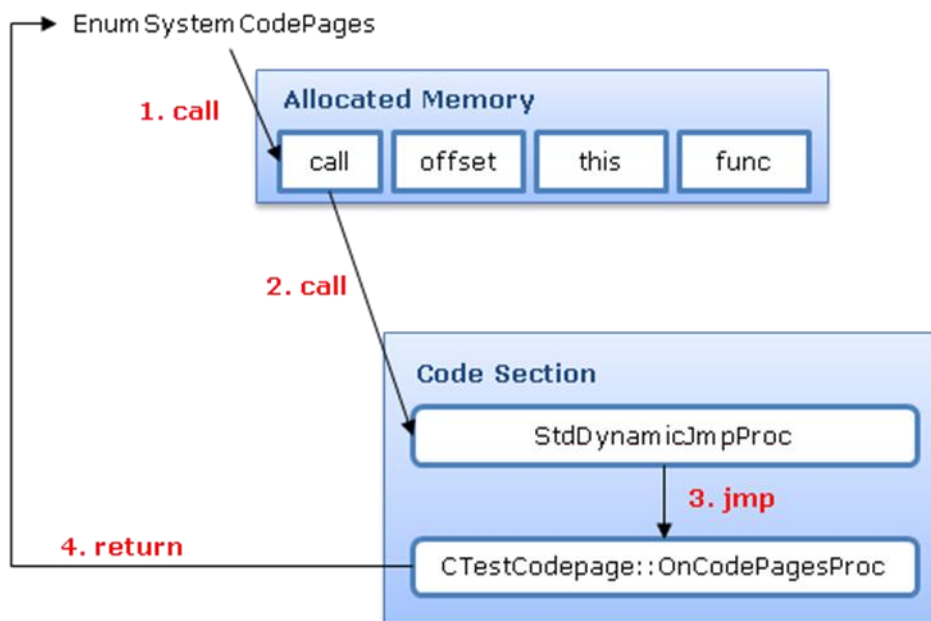


그림 3 함수 흐름도

각 단계별로 스택의 구조가 <그림 4>에 나와있다. 왼쪽 그림은 1번 단계의 스택을 오른쪽 그림은 2번 단계의 스택을 보여주고 있다. 1번 단계에서 스택에는 콜백 함수의 인자인 LPCTSTR과 EnumSystemCodePages로 리턴하는 주소만을 담고 있다. 여기서 2번 단계의 호출이 일어나면 DyanmicCallbackOpCodes로 리턴하는 주소가 추가된다. 이 스택의 상태에서 <리스트 19>에 나타난 코드가 수행된다.

POP ECX를 하면 ECX에 DyanmicCallbackOpCodes의 리턴 주소가 ECX에 불러진다. 그리고 스택에서는 해당 리턴 주소가 사라진다. 이 리턴 주소는 DyanmicCallbackOpCodes에서 call 명령이 수행된 다음 번지를 가리키고 있기 때문에 this가 시작하는 지점이 된다. 다음 단계는 EAX에 ECX+4에 있는 내용을 옮기는 과정이다. ECX+4에는 호출될 함수 주소가 저장되어 있다. 그리고 ECX에 ECX에 들어 있는 내용인 this 포인터의 주소를 불러 들인다. 앞서 thiscall의 경우 this 포인터를 ECX를 통해 전달한다고 배웠다. 최종적으로 EAX에 저장된 주소로 점프를 한다. 여기서 호출이 아닌 점프를 한다는 점에 유의해야 한다. call을 하면 스택에 다시 리턴 주소가 추가되고 해당 함수로

넘어간다. 따라서 함수 호출시의 스택 구조가 이상하게 되어 버린다.

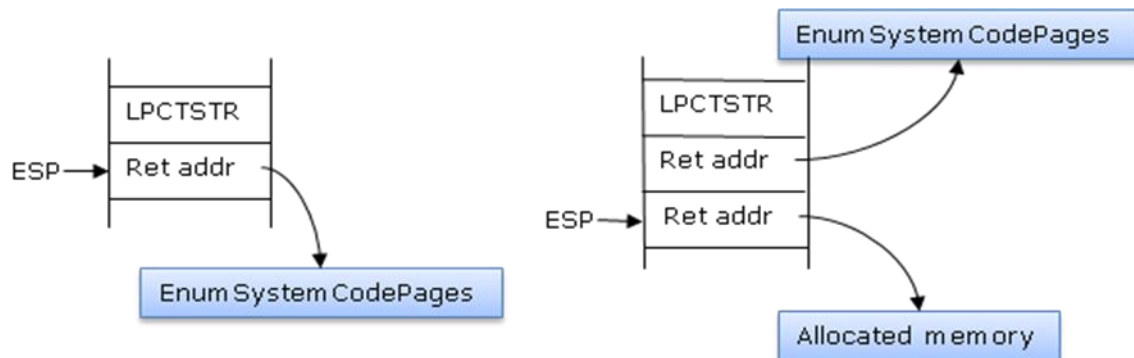


그림 4 함수 호출에 따른 스택 프레임

<리스트 20>에 이 모든 작업을 해주는 코드가 나와있다. 몇 가지 매직 넘버를 제외하면 별로 이해하기 힘든 내용은 없을 것이다. CalcJmpOffset은 StdDynamicJmpProc을 호출하기 위한 상대주소를 계산하는 함수다. Dest가 StdDynamicJmpProc가 되고, Src가 DyanmicCallbackOpCodes가 된다. 5를 더해주는 이유는 리턴 주소가 call 명령의 다음 번지가 되어야 하고, call 명령은 총 5바이트로 구성되기 때문이다. DyanmicCallbackOpCodes의 tag를 0xE8로 채우는 이유는 x86 어셈블러에서 call에 해당하는 명령어 코드(op-code)가 0xE8이기 때문이다.

리스트 20 멤버 함수를 stdcall 형태로 맵핑해 주는 함수

```
template <typename TStdcallType>
class CDynamicCallback
{
private:
    LONG_PTR m_pThis;
    LONG_PTR m_pFunc;
    TStdcallType m_pStdFunc;

    static LONG_PTR CalcJmpOffset(LONG_PTR Src, LONG_PTR Dest)
    {
        return Dest - (Src + 5);
    }

    void MakeCode()
    {
        if (m_pStdFunc) ::VirtualFree(m_pStdFunc, 0, MEM_RELEASE);
        m_pStdFunc = (TStdcallType)::VirtualAlloc(NULL, sizeof(DynamicCallbackOpCodes),
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        DynamicCallbackOpCodes * p = (DynamicCallbackOpCodes *)m_pStdFunc;
        p->_func = *(LONG_PTR *)&m_pFunc;
        p->_this = (LONG_PTR)m_pThis;
        p->tag = 0xE8;
        p->offset = CalcJmpOffset((LONG_PTR)p, (LONG_PTR)StdDynamicJmpProc);
    }

public:
    CDynamicCallback() {}

    template<typename T1, typename T2>
    CDynamicCallback(T1 pClassAddress, T2 pClassMemberFunctionAddress)
    {
        Assign(pClassAddress, pClassMemberFunctionAddress);
    }

    template<typename T1, typename T2>
    void Assign(T1 pClassAddress, T2 pClassMemberFunctionAddress)
```

```
{
    STATIC_ASSERT(util::type_trait::is_pointer<T1>::value);
    STATIC_ASSERT(util::type_trait::is_member_function_pointer<T2>::value);
    m_pFunc = *(LONG_PTR *)&pClassMemberFunctionAddress;
    m_pThis = (LONG_PTR)pClassAddress;
    m_pStdFunc = NULL;
    MakeCode();
}

~CDynamicCallback()
{
    ::VirtualFree(m_pStdFunc, 0, MEM_RELEASE);
}

inline operator TStdcallType()
{
    return m_pStdFunc;
}

inline TStdcallType operator()()
{
    return m_pStdFunc;
}
};
```

도전 과제

<리스트 20>에 나온 템플릿 클래스는 필자가 직접 작성한 것은 아니고, Notepad2 소스 코드에서 발췌한 내용이다. 코드를 처음 읽었을 때, 필자는 괜찮은 아이디어라고 생각했다. 하지만 누구나 알고 있듯이 VirtualAlloc, VirtualFree 등의 작업은 부하가 많은 일이고 작은 메모리 할당에는 적합하지 않다. 물론 코드의 원 저작자의 의도는 최대한 안전한 방법을 택한 것이라고 생각된다. 하지만 필자는 이 코드에서 VirtualAlloc, VirtualFree를 제거한다면 좀 더 쓸모 있는 코드가 될 거라고 생각한다. 이 코드의 다른 한 가지 단점은 32비트 환경에서만 사용할 수 있다는 점이다.

이번 시간에 배운 지식들을 토대로 VirtualAlloc, VirtualFree가 없는 버전을 제작해 보자. 또한 아울러서 64비트 버전은 어떻게 만들 수 있을지 고민해 보도록 하자.

참고자료

Debugging Applications

John Robins저, Microsoft Press

Assembly Language for Intel-Based Computers (5/E)

KIP R. IRVINE, Prentice Hall

x86 함수 호출 규약 위키 페이지

http://en.wikipedia.org/wiki/X86_calling_conventions

x64 호출 규약

[http://msdn2.microsoft.com/en-us/library/7kcdt6fy\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/7kcdt6fy(VS.80).aspx)

notepad2 홈페이지

<http://sourceforge.net/projects/notepad2/>