# 호출규약에 따른 어셈블리 코드의 변화

수원대학교 flag 지선호(kissmefox@gmail.com)

함수에 파라미터를 넘기기 위하여 caller 과 callee 사이에는 그 값을 어떤 방식으로 전달할지에 대한 사전약속이 필요합니다. 만약 사전에 어떠한 방식으로 인자값을 주고받을지 결정되어있지 않다면, 다른 종류의 컴파일러에서 생성된 Executable Binary 간의 호환성을 보장할 수 없고, 코드의 재사용과 라이브러리 참조가 주가 되는 현재의 시스템에서 큰 혼돈을 불러올 것입니다. 파라미터를 넘기기위해 공통적으로 스택 영역이 사용되지만(레지스터 영역이 사용되기도 함) 함수 호출이 끝난 후에 스택 영역에 저장되었던 파라미터의 처리는 caller 과 callee 둘 중의 하나가 처리해야 할 작업입니다. 여기서 크게 함수를 부른 곳에서 스택을 제거하는 C/C++ 방식과 호출된 함수가 스택을 제거하는 파스칼 방식의 두 가지로 구분을 할 수가 있습니다. visual c++ 에서는 5가지의 호출 규약을 사전에 정의하여 다른 컴파일러가 생성한 바이너리 간의 호환성을 지원해 주고 있습니다. 함수 호출 규약에 따라 어셈코드와 스택 영역이 어떻게 변화하는지 간단한 예제를 통해 살펴보겠습니다.

```
int __cdecl ExFunc1(int a, int b, int c)
{   return a+b+c;   }
int __stdcall ExFunc2(int a, int b, int c)
{   return a+b+c;   }
int __fastcall ExFunc3(int a,int b,int c)
{   return a+b+c;   }

void main()
{
    int a=1,b=2,c=3;

    ExFunc1(a,b,c);
    ExFunc2(a,b,c);
    ExFunc3(a,b,c);
}
```

main 함수에 break point 를 걸고 디버깅을 시작하여 디스어셈블된 화면과 레지스트리 상태창을 확인합니다. 프로그램의 디스어셈블된 코드는 다음과 같습니다.

```
004010F0    push        ebp  // main 함수 호출 이전의 base pointer 저장
004010F1    mov         ebp,esp // 새 base pointer 지정해줌
004010F3    sub         esp,4Ch // int a, int b, int c 변수 공간 설정
004010F6    push        ebx // 범용 레지스터의 내용을 스택에 저장
004010F7    push        esi
004010F8    push        edi
004010F9    lea         edi,[ebp-4Ch]          위에서 확보한 스택영역(4Ch) 의 경계에
004010FC    mov         ecx,13h                디버그 인터럽트(opcode 로 0xCC) 를 깔아
00401101    mov         eax,0CCCCCCCCh         놓아 스택의 범위를 넘어서면 디버거가 뜰 수
00401106    rep stos    dword ptr [edi]        있도록 함.
13:         int a=1,b=2,c=3;
00401108    mov         dword ptr [ebp-4],1
```

```
0040110F   mov          dword ptr [ebp-8],2
00401116   mov          dword ptr [ebp-0Ch],3
14:

15:        ExFunc1(a,b,c);  // __cdecl 방식
0040111D   mov          eax,dword ptr [ebp-0Ch]
00401120   push         eax
00401121   mov          ecx,dword ptr [ebp-8]
00401124   push         ecx
00401125   mov          edx,dword ptr [ebp-4]
00401128   push         edx
00401129   call         @ILT+10(ExFunc1) (0040100f)
0040112E   add          esp,0Ch
```

```
00401030   push         ebp
00401031   mov          ebp,esp
00401033   sub          esp,40h
00401036   push         ebx
00401037   push         esi
00401038   push         edi
00401039   lea          edi,[ebp-40h]
0040103C   mov          ecx,10h
00401041   mov          eax,0CCCCCCCCh
00401046   rep stos     dword ptr [edi]
00401048   mov          eax,dword ptr [ebp+8]
0040104B   add          eax,dword ptr [ebp+0Ch]
0040104E   add          eax,dword ptr [ebp+10h]
00401051   pop          edi
00401052   pop          esi
00401053   pop          ebx
00401054   mov          esp,ebp
00401056   pop          ebp
00401057   ret
```

```
16:        ExFunc2(a,b,c);  //  __stdcall 방식
00401131   mov          eax,dword ptr [ebp-0Ch]
00401134   push         eax
00401135   mov          ecx,dword ptr [ebp-8]
00401138   push         ecx
00401139   mov          edx,dword ptr [ebp-4]
0040113C   push         edx
0040113D   call         @ILT+5(ExFunc2) (0040100a)
```

```
00401070   push         ebp
00401071   mov          ebp,esp
00401073   sub          esp,40h
00401076   push         ebx
00401077   push         esi
00401078   push         edi
00401079   lea          edi,[ebp-40h]
0040107C   mov          ecx,10h
00401081   mov          eax,0CCCCCCCCh
00401086   rep stos     dword ptr [edi]
00401088   mov          eax,dword ptr [ebp+8]
0040108B   add          eax,dword ptr [ebp+0Ch]
0040108E   add          eax,dword ptr [ebp+10h]
00401091   pop          edi
00401092   pop          esi
00401093   pop          ebx
00401094   mov          esp,ebp
00401096   pop          ebp
00401097   ret          0Ch
```

```
17:        ExFunc3(a,b,c);  //  __fastcall 방식
00401142   mov          eax,dword ptr [ebp-0Ch]
00401145   push         eax
00401146   mov          edx,dword ptr [ebp-8]
00401149   mov          ecx,dword ptr [ebp-4]
0040114C   call         @ILT+0(ExFunc3) (00401005)
```

```
004010B0   push         ebp
004010B1   mov          ebp,esp
004010B3   sub          esp,48h
004010B6   push         ebx
004010B7   push         esi
004010B8   push         edi
004010B9   push         ecx
004010BA   lea          edi,[ebp-48h]
004010BD   mov          ecx,12h
004010C2   mov          eax,0CCCCCCCCh
004010C7   rep stos     dword ptr [edi]
004010C9   pop          ecx
004010CA   mov          dword ptr [ebp-8],edx
004010CD   mov          dword ptr [ebp-4],ecx
004010D0   mov          eax,dword ptr [ebp-4]
004010D3   add          eax,dword ptr [ebp-8]
004010D6   add          eax,dword ptr [ebp+8]
004010D9   pop          edi
004010DA   pop          esi
004010DB   pop          ebx
004010DC   mov          esp,ebp
004010DE   pop          ebp
004010DF   ret          4
```

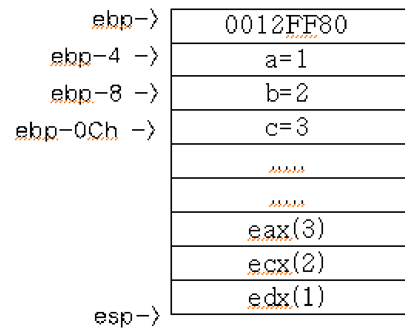
```
18:   }
00401151   pop          edi  // 이전 함수의 범용 레지스터 값 복구
00401152   pop          esi
00401153   pop          ebx
00401154   add          esp,4Ch  // main에서 사용된 stack 정리해줌
```

```
00401157    cmp         ebp,esp
00401159    call        __chkesp (00401180)  // 정리되지 않은 스택이 있나 check 하는 루틴
0040115E    mov         esp,ebp   // 함수 epilog
00401160    pop         ebp
00401161    ret
```

main 함수도 하나의 함수이기 때문에 동일한 prolog/ epilog 과정을 볼 수 있습니다. 메인 함수 내에서 호출되는 위의 세 함수들의 차이점을 확인해 보겠습니다. 먼저 __cdecl 방식의 Exfunc1함수가 호출되는 과정에 따른 스택의 변화를 살펴보겠습니다.

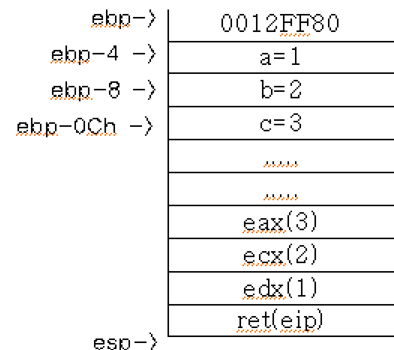
1. 호출된 함수가 인자를 받을 수 있도록 스택에 저장합니다. 저장되는 순서는 호출 규약에 따라 달라질 수 있고 위의 __cdecl 방식은 오른쪽에서 왼쪽으로 인자값이 저장되는걸 확인할 수 있습니다.

```
15:         ExFunc1(a,b,c);   // __cdecl 방식
0040111D    mov         eax,dword ptr [ebp-0Ch]
00401120    push        eax
00401121    mov         ecx,dword ptr [ebp-8]
00401124    push        ecx
00401125    mov         edx,dword ptr [ebp-4]
00401128    push        edx
```




2. 인자값이 스택에 저장된 후 call 명령으로 함수를 호출합니다. call 명령이 실행되면 함수 호출이 종료된 뒤 돌아와야 할 코드의 주소를 저장하기위해 스택에 eip 레지스터를 저장합니다.

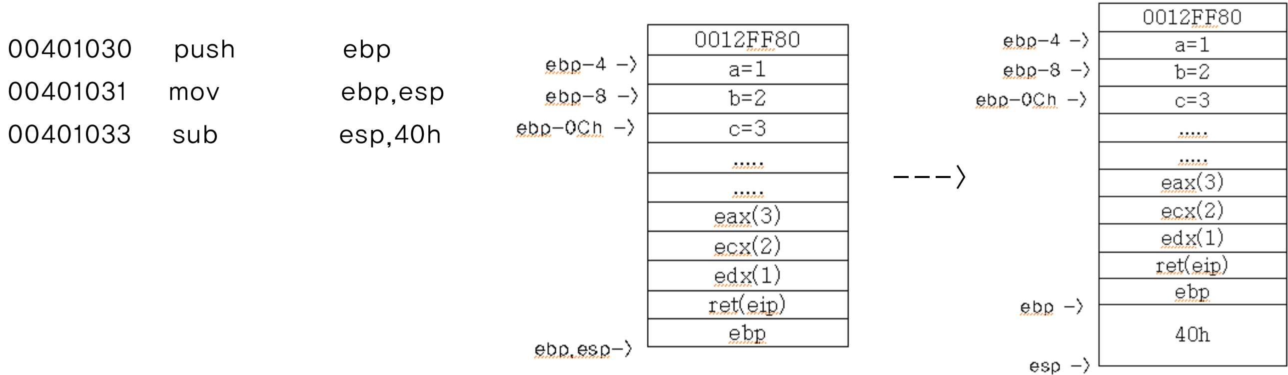
```
00401129    call        @ILT+10(ExFunc1) (0040100f)
```




3. 호출한 함수에서 사용될 스택 영역을 지정하기 위해 이전 base pointer 를 저장하고 현재시점의 esp 를 새로운 base pointer 로 설정합니다. 또 호출된 함수에서 사용할 지역변수를 저장할 공간을

스택에 할당합니다. 지역변수는 ebp 와 esp 사이의 공간을 이용하게 되며, ebp 를 기준으로 첫 번째 지역변수는 ebp-4, 두 번째 지역변수는 ebp-8 과 같은 식으로 참조하게 됩니다.

```
00401030    push    ebp
00401031    mov     ebp,esp
00401033    sub     esp,40h
```

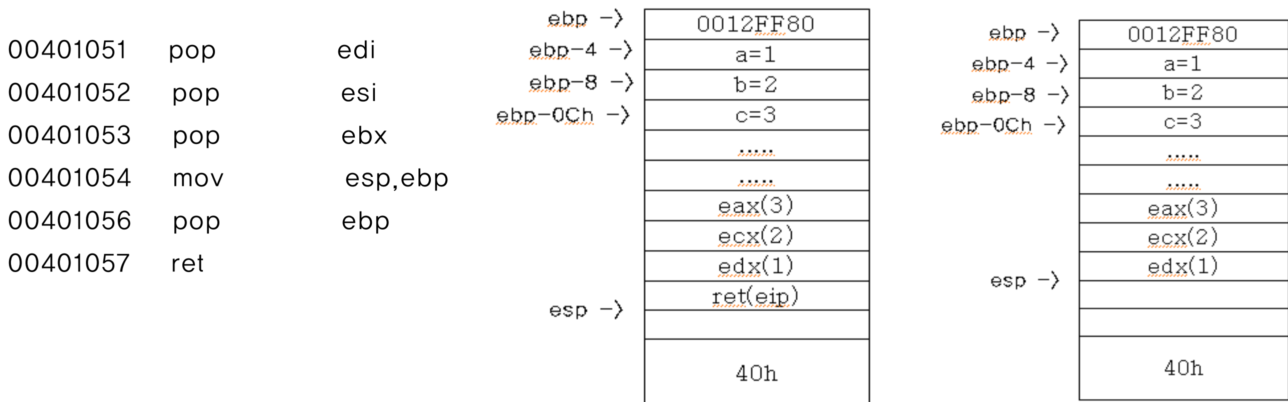



여기까지가 공통된 함수 호출 과정이고 이후에 호출된 함수에서 범용 레지스터를 사용하는 경우 안전하게 이전 함수의 실행 내역을 보존하기 위해 범용 레지스터의 내용도 스택에 저장하게 됩니다. 이 부분도 거의 공통적으로 함수 호출 과정에서 볼수 있는 코드입니다.

```
00401036    push    ebx
00401037    push    esi
00401038    push    edi
```

이제 함수의 실행에 필요한 모든 준비가 끝나고 필요한 연산을 수행하게 됩니다. 연산이 끝난 후에는 스택의 LIFO(Last in First Out) 성질대로 스택에 저장되어 있던 이전 함수의 실행 내역을 pop 하고 함수에 진입할 시점의 스택 포인터를 복구하고 이전 함수의 base pointer 를 복구합니다. 마지막 ret 명령으로 스택에 저장해 둔 리턴 주소를 꺼내어 eip에 로드합니다. 이제 호출된 함수에서 빠져나와 본래의 호출한 함수로 돌아온 상태가 되었습니다.

```
00401051    pop     edi
00401052    pop     esi
00401053    pop     ebx
00401054    mov     esp,ebp
00401056    pop     ebp
00401057    ret
```




다음으로 함수 호출에 사용된 인자를 정리해야 본래의 스택 상태로 돌아갈 수 있습니다. __cdecl 방식은 호출한 함수가 사용한 인자를 정리하게 규정되었으므로 main 함수에서 호출한 함수의 인자를 정리하게 됩니다. 간단하게 esp 레지스터에 사용한 인자의 개수만큼을 더하여 스택포인터를 이동시키는 방식으로 정리를 합니다.

```
0040112E    add          esp,0Ch
```

| | |
|---|---|
| ebp -> | 0012FF80 |
| ebp-4 -> | a=1 |
| ebp-8 -> | b=2 |
| ebp-0Ch -> | c=3 |
| | ..... |
| esp-> | ..... |
| | eax(3) |
| | ecx(2) |
| | edx(1) |
| | |
| | |
| | 40h |

__cdecl 방식의 호출 규약을 정리하면 다음과 같습니다.

| | |
|---|---|
| __cdecl 규약 default 함수 | C/C++ Global Function, Global Static Function, Static Member Function |
| 인자 전달 순서 | 오른쪽 인자 -> 왼쪽 인자 |
| return 값 전달 | eax 레지스터 |
| 특징 | 호출한 함수가 호출된 함수에게 넘겨준 인자값을 정리하기 때문에 호출되는 함수의 인자가 가변적이어도 상관없음. |

다음은 __stdcall 방식의 ExFunc2 함수의 어셈블리 코드입니다.

```
00401070   push       ebp
00401071   mov        ebp,esp
00401073   sub        esp,40h
00401076   push       ebx
00401077   push       esi
00401078   push       edi
00401079   lea        edi,[ebp-40h]
0040107C   mov        ecx,10h
00401081   mov        eax,0CCCCCCCCh
00401086   rep stos   dword ptr [edi]
00401088   mov        eax,dword ptr [ebp+8]
0040108B   add        eax,dword ptr [ebp+0Ch]
0040108E   add        eax,dword ptr [ebp+10h]
00401091   pop        edi
00401092   pop        esi
00401093   pop        ebx
00401094   mov        esp,ebp
00401096   pop        ebp
00401097   ret        0Ch     // 사용한 인자 스택에서 정리
```

(main에서 인자 전달)

```
16:        ExFunc2(a,b,c);  //  __stdcall 방식
00401131   mov        eax,dword ptr [ebp-0Ch]
00401134   push       eax
00401135   mov        ecx,dword ptr [ebp-8]
00401138   push       ecx
00401139   mov        edx,dword ptr [ebp-4]
0040113C   push       edx
0040113D   call       @ILT+5(ExFunc2) (0040100a)
```

코드의 의미와 스택의 변화는 위에서 설명되었으므로 생략합니다.
마지막 ret 0Ch 구문은 add esp, 0Ch 구문처럼 사용한 인자를 스택에서 제거하는 역할을 수행하게 됩니다. 호출된 함수에서 인자값 처리를 하기 때문에 main 에서는 호출된 이후에 인자값 정리에 관한 명

령어는 볼 수 없습니다.

__stdcall 방식의 호출 규약을 정리하면 다음과 같습니다.

| 사용되는 곳 | 윈도우 API 표준 호출 규약 (#define WINAPI __stdcall<br>APIENTRY __stdcall<br>CALLBACK __stdcall<br>PASCAL __stdcall<br>WSAAPI __stdcall ) |
|---|---|
| 인자 전달 순서 | 오른쪽 인자 -> 왼쪽 인자 |
| return 값 전달 | eax 레지스터 |
| 특징 | 호출된 함수가 인자값을 정리하기 때문에 스택에서 제거해야 할 인자의 개수를 정확히 알아야 함.(가변 인자 사용불가)<br>함수의 독립성이 뛰어나고 스택 제거 루틴이 호출된 함수측에 있기 때문에 빌드된 바이너리의 크기가 작아짐 |

다음은 __fastcall 방식의 ExFunc3 함수의 어셈블리 코드입니다.

```
004010B0   push      ebp
004010B1   mov       ebp,esp
004010B3   sub       esp,48h // ecx , edx 레지스터로 넘어온 인자값을 지역변수에 넣어준 뒤에 연산을 수행하기 때문에
                     위의 두 함수보다 8byte 더 잡히게 됨. 디버그 모드가 아닌 릴리즈 모드로 컴파일 시에는
                     바로 레지스터에서 연산 수행함.
004010B6   push      ebx
004010B7   push      esi
004010B8   push      edi
004010B9   push      ecx
004010BA   lea       edi,[ebp-48h]
004010BD   mov       ecx,12h
004010C2   mov       eax,0CCCCCCCCh
004010C7   rep stos  dword ptr [edi]
004010C9   pop       ecx
004010CA   mov       dword ptr [ebp-8],edx
004010CD   mov       dword ptr [ebp-4],ecx
004010D0   mov       eax,dword ptr [ebp-4]
004010D3   add       eax,dword ptr [ebp-8]
004010D6   add       eax,dword ptr [ebp+8]
004010D9   pop       edi
004010DA   pop       esi
004010DB   pop       ebx
004010DC   mov       esp,ebp
004010DE   pop       ebp
004010DF   ret       4  // 사용한 인자 스택에서 정리
```

(main에서 인자 전달)

```
17:        ExFunc3(a,b,c);  //  __fastcall 방식
00401142   mov       eax,dword ptr [ebp-0Ch]
00401145   push      eax
00401146   mov       edx,dword ptr [ebp-8]
00401149   mov       ecx,dword ptr [ebp-4]
0040114C   call      @ILT+0(ExFunc3) (00401005)
```

앞의 두 개의 함수와 다르게 main 함수에서 인자를 넘길 때 앞 쪽 두 개의 인자를 ecx , edx 레지스터에 넣어서 넘기는 것을 볼 수 있습니다. 세 번째 인자는 stack 으로 전달되었습니다. __stdcall 방식과 같이 호출된 곳에서 사용한 인자를 제거하므로 ret 4 구문에서 사용된 스택을 제거하게 됩니다. 하나의 인자값만 stack 으로 전달되었기 때문에 4byte 만 add 해주면 스택이 정리가 될 것입니다.

__fastcall 방식의 호출 규약을 정리하면 다음과 같습니다.

| 사용되는 곳 | VxD 시스템 함수들, Win32 유저모드에서는 거의 사용되지 않음 |
| --- | --- |
| 인자 전달 순서 | 처음 두 개의 DWORD 이하의 크기를 가지는 인자는 ECX와 EDX레지스터로 전달, 나머지 인자는 오른쪽에서 왼쪽으로 스택을 통해 전달 |
| return 값 전달 | eax 레지스터 |
| 특징 | 레지스터 호출규약이라고도 함. 메모리보다 상대적으로 빠른 레지스터를 이용하므로 낮은 호출 비용을 기대할 수 있음. 그러나 레지스터는 CPU에 종속적인 저장매체이기 때문에 호환성 문제로 인해 현재는 거의 사용되지 않음. |

위의 방식들 이외에 __thiscall 과 naked 호출 규약이 있습니다. 역시 간단한 예제로 속성을 살펴보겠습니다.

```
class CallingTest{
public:
    int TestCall(int a,int b,int c)
    {   return a+b+c;   }
};

void main()
{
    CallingTest Call;
    int a=1,b=2,c=3;
    Call.TestCall(a,b,c);
}
```

(__thiscall 사용 예제)

__thiscall 은 C++의 클래스 멤버 함수 호출 규약으로만 사용되는 방식입니다. 때문에 직접적인 함수 호출 규약으로 사용할 순 없고, 멤버 함수를 호출할 때 디폴트로 정해진 규약입니다.
main 함수의 어셈블리 코드는 다음과 같습니다.

```
0040D570   push        ebp
0040D571   mov         ebp,esp
0040D573   sub         esp,50h // CallingTest Call , int a , int b, int c
0040D576   push        ebx
0040D577   push        esi
0040D578   push        edi
0040D579   lea         edi,[ebp-50h]
0040D57C   mov         ecx,14h
0040D581   mov         eax,0CCCCCCCCh
0040D586   rep stos    dword ptr [edi]
9:         CallingTest Call;
10:        int a=1,b=2,c=3;
0040D588   mov         dword ptr [ebp-8],1
0040D58F   mov         dword ptr [ebp-0Ch],2
0040D596   mov         dword ptr [ebp-10h],3
11:        Call.TestCall(a,b,c);
0040D59D   mov         eax,dword ptr [ebp-10h]
```

```
0040D5A0    push        eax
0040D5A1    mov         ecx,dword ptr [ebp-0Ch]
0040D5A4    push        ecx
0040D5A5    mov         edx,dword ptr [ebp-8]
0040D5A8    push        edx
0040D5A9    lea         ecx,[ebp-4] // this 포인터 전달
0040D5AC    call        @ILT+20(CallingTest::TestCall) (00401019)
12:    }
0040D5B1    pop         edi
0040D5B2    pop         esi
0040D5B3    pop         ebx
0040D5B4    add         esp,50h
0040D5B7    cmp         ebp,esp
0040D5B9    call        __chkesp (00401180)
0040D5BE    mov         esp,ebp
0040D5C0    pop         ebp
0040D5C1    ret
```

```
00401070    push        ebp
00401071    mov         ebp,esp
00401073    sub         esp,44h
00401076    push        ebx
00401077    push        esi
00401078    push        edi
00401079    push        ecx
0040107A    lea         edi,[ebp-44h]
0040107D    mov         ecx,11h
00401082    mov         eax,0CCCCCCCCh
00401087    rep stos    dword ptr [edi]
00401089    pop         ecx
0040108A    mov         dword ptr [ebp-4],ecx
0040108D    mov         eax,dword ptr [ebp+8]
00401090    add         eax,dword ptr [ebp+0Ch]
00401093    add         eax,dword ptr [ebp+10h]
00401096    pop         edi
00401097    pop         esi
00401098    pop         ebx
00401099    mov         esp,ebp
0040109B    pop         ebp
0040109C    ret         0Ch
```

위에서 살펴본 호출 규약과 다른 점이라면 호출된 멤버 함수에 인자를 전달할 때 ecx 레지스터에 클래스 인스턴스 자신을 가리키는 포인터를 같이 전달하는 걸 볼수 있습니다. 이것이 바로 this 포인터이고 ecx 레지스터로 전달된 this 포인터 덕분에 멤버함수는 자기가 언제 어디에서 호출되었다 하더라도 호출자가 어떤 인스턴스인지를 정확히 알 수 있는 것입니다.
호출된 함수의 어셈블리 코드를 보면 ecx 레지스터로 전달받은 this 포인터를 지역변수 공간에 할당하는 것을 볼 수 있지만 디버그 모드로 컴파일되어 생긴 큰 의미 없는 코드인 것 같습니다, 마지막에 리턴 구문에서 __stdcall 방식처럼 호출된 함수가 인자를 정리하는 것을 확인할 수 있습니다.

__thiscall 방식의 호출 규약을 정리하면 다음과 같습니다.

| 사용되는 곳 | C++ 클래스 멤버 함수 , 직접적인 함수 호출 규약 사용 불가 |
|---|---|
| 인자 전달 순서 | 오른쪽 -> 왼쪽 , ecx 레지스터로 클래스 포인터 전달 |
| return 값 전달 | eax 레지스터 |
| 특징 | 디폴트 규약으로 __thiscall 방식이 사용되지만 직접 지정할 경우 다른 호출 규약을 사용할수도 있음. 다른 호출 규약으로 선언될 경우 첫 번째 인자로 this 포인터가 전달됨. |

마지막으로 naked 호출 규약을 살펴보겠습니다. 모든 함수는 함수에서 사용될 스택프레임을 설정하는 prolog / epilog 과정을 가지게 됩니다. 하지만 함수 선언부 앞에 __declspec(naked) 를 붙여주게 되면 컴파일러는 함수의 prolog / epilog 과정을 생성하지 않으므로 사용자가 직접 제작하여 좀 더 로우레벨 수준의 코딩을 할 수 있습니다. 이 속성은 함수의 타입을 정의하는 것이 아니라 단지 prolog/epilog 구문만 생략되는 것이기 때문에 함수 호출 규약이라기보다 함수가 구현되는 방법의 문제라고 말할 수 있습니다. 따라서 위에서 살펴본 호출 규약에 상관없이 함수 앞에 추가를 해 주면 됩니다.

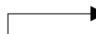
간단한 예제를 살펴보겠습니다.

```
__declspec(naked)int ExFunc(int a,int b,int c){
    __asm{
        push ebp
        mov ebp,esp
        sub esp,__LOCAL_SIZE
    }
    {
        int d=0;
        d=a+b+c;
        __asm{mov eax,d};
    }
    __asm{
        mov esp,ebp
        pop ebp
        ret
    }
}

void main()
{
    int a=1,b=2,c=3;
    ExFunc(a,b,c);
}
```

코드에서 __LOCAL_SIZE 심볼은 스택 프레임에 로컬 변수 공간을 할당하는데 사용되는 심볼입니다. 컴파일러에서 사용자 정의된 모든 로컬 변수와 컴파일러가 생성하는 임시 변수들의 총 바이트 수로 __LOCAL_SIZE 값을 정하게 됩니다.
디스어셈블된 코드를 보겠습니다.

```
00401050   push      ebp
00401051   mov       ebp,esp
00401053   sub       esp,4Ch
00401056   push      ebx
00401057   push      esi
00401058   push      edi
00401059   lea       edi,[ebp-4Ch]
0040105C   mov       ecx,13h
00401061   mov       eax,0CCCCCCCCh
00401066   rep stos  dword ptr [edi]
21:        int a=1,b=2,c=3;
00401068   mov       dword ptr [ebp-4],1
0040106F   mov       dword ptr [ebp-8],2
00401076   mov       dword ptr [ebp-0Ch],3
22:        ExFunc(a,b,c);
0040107D   mov       eax,dword ptr [ebp-0Ch]
00401080   push      eax
00401081   mov       ecx,dword ptr [ebp-8]
00401084   push      ecx
00401085   mov       edx,dword ptr [ebp-4]
00401088   push      edx
00401089   call      @ILT+5(ExFunc) (0040100a)
0040108E   add       esp,0Ch // 인자값 정리
23:   }
00401091   pop       edi
```



```
1:     __declspec(naked)int ExFunc(int a,int b,int c){
00401020   push        ebp
2:         __asm{
3:             push ebp
4:             mov ebp,esp
00401021   mov         ebp,esp
5:             sub esp,__LOCAL_SIZE
00401023   sub         esp,44h
6:         }
7:         {
8:             int d=0;
00401026   mov         dword ptr [d],0
9:             d=a+b+c;
0040102D   mov         eax,dword ptr [ebp+8]
00401030   add         eax,dword ptr [ebp+0Ch]
00401033   add         eax,dword ptr [ebp+10h]
00401036   mov         dword ptr [d],eax
10:            __asm{mov eax,d};
00401039   mov         eax,dword ptr [d]
11:        }
12:        __asm{
13:            mov esp,ebp
0040103C   mov         esp,ebp
14:            pop ebp
0040103E   pop         ebp
15:            ret
0040103F   ret
```

```
00401092    pop         esi
00401093    pop         ebx
00401094    add         esp,4Ch
00401097    cmp         ebp,esp
00401099    call        __chkesp (004010c0)
0040109E    mov         esp,ebp
004010A0    pop         ebp
004010A1    ret
```

main 함수에서 인자를 제거하는 것을 보면 특별한 호출 규약 없이 함수가 선언되었기 때문에 C언어 디폴트 호출 규약인 __cdecl 방식으로 선언이 된 것을 알 수 있습니다. ExFunc 함수의 어셈블리 코드를 보면 사용자가 제작한대로 prolog/epilog 코드가 생성된 것을 확인할 수 있습니다. Naked 속성 함수에서는 return 구문을 사용할 수 없기 때문에 인라인 어셈블로 eax 에 리턴값을 넣어줘야 합니다. 위에서는 함수내에서 연산작업을 수행하다가 자연스럽게 eax 레지스터에 결과값이 저장이 되어서 불필요하게 같은 작업을 수행하는 코드가 두 번 들어갔습니다

naked 방식의 호출 규약을 정리하면 다음과 같습니다.

| | |
|---|---|
| 사용되는 곳 | 가상 장치 드라이버 제작, Hooking 이나 injection 관련 코딩, 실행속도를 최적화하기 위해 C의 일정 코드를 어셈블리어로 바꿀 때 |
| 인자 전달 순서 | 선언되는 호출 규약에 따름 |
| return 값 전달 | 없음 |
| 특징 | 이 속성은 함수의 타입을 정의하는 것이 아니라 단지 prolog/epilog 구문만 생략되는 것이기 때문에 함수 호출 규약이라기보다 함수가 구현되는 방법의 문제라고 말할 수 있음 |