16-1-다.호출 규약

출처: http://www.winapi.co.kr/clec/cpp2/16-1-3.htm

앞 항에서 Add 함수의 어셈블리 코드를 통해 스택 프레임의 실제 모양을 확인해 보았다. 인수는 뒤쪽부터 순서대로 전달하며 인수 전달에 사용한 스택 영역은 호출원이 정리했는데 이는 C/C++ 언어의 기본 호출 규약인 __cdecl 의 스택 프레임 모양일 뿐이다. 호출 규약이 바뀌면 스택 프레임의 모양과 관리 방법도 달라질 수 있다.

호출 규약은 호출원과 함수간의 약속이므로 양쪽이 다른 형태로 약속을 할 수도 있는 것이다. 그렇다면 __cdecl 이 아닌 다른 호출 규약은 어떻게 스택 프레임을 작성하는지 차이점을 분석해 보자. 호출 규약에 따라 인수를 전달하는 방법과 스택의 정리 책임, 함수의 이름을 작성하는 방법이 달라진다.

호출 규약	인수 전달	스택 정리	이름 규칙
__cdecl	오른쪽 먼저	호출원	_함수명
__stdcall	오른쪽 먼저	함수	_함수명@인수크기
__fastcall	ECX, EDX에 우선 전달 나머지는 오른쪽 먼저	함수	@함수명@인수크기
thiscall	오른쪽 먼저, this 포인터는 ecx 레지스터로 전달된다.	함수	C++ 이름 규칙을 따름.
naked	오른쪽 먼저	함수	없음

리턴값을 돌려 주는 방식도 호출 규약에 따라 달라질 수 있는데 다행히 현존하는 모든 호출 규약의 리턴 방식은 동일하다. 4 바이트의 값을 돌려줄 때는 eax 레지스터를 사용하며 8 바이트의 값을 리턴할 때는 edx:eax 레지스터 쌍을 사용한다. 8 바이트를 초과하는 큰 리턴값, 예를 들어 구조체 등은 임시 영역에 리턴할 값을 넣어 두고 그 포인터를 eax 에 리턴한다.

__stdcall

Add 함수의 호출 규약을 __stdcall 로 바꿔 보자. __stdcall 은 윈도우즈 API 함수들의 기본 호출 규약이며 비주얼 베이직도 이 호출 규약을 사용한다. __cdecl 과 인수를 전달하는 방법은 동일하되 인수 전달에 사용된 스택을 정리하는 주체가 호출원이 아니라 함수라는 점이 다르다. Add 함수의 호출 규약을 바꾸기 위해 다음과 같이 수정해 보자.

```
int __stdcall Add(int a, int b)
{
    int c,d,e;
    c=a+b;
    return c;
}
```

함수 이름앞에 __stdcall 키워드를 삽입하면 이 함수는 __stdcall 호출 규약을 사용한다. main 에서 함수를 호출하는 부분이 다음과 같이 변경된다.

```
push 2
push 1
call Add
result=eax
```

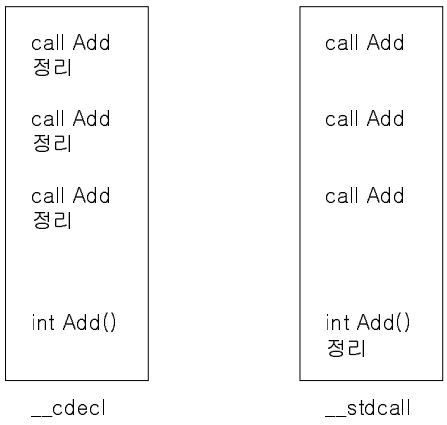
인수를 스택에 밀어 넣는 것과 인수를 푸시하는 순서는 동일하다. 단, call Add 다음에 add esp,8 코드가 없어 함수가 리턴된 후에 인수 전달에 사용한 스택을 복구하지 않는다는 점이 __cdecl 과 다르다. 인수 전달에 사용한 영역은 이제 Add 함수가 직접 정리한다. 이 함수의 접두, 본체는 __cdecl 과 동일하며 접미 부분이 다음과 같이 변경된다.

```
push ebp
....
ret 8
```

복귀 코드가 ret 에서 ret 8 로 바뀌었으며 복귀하면서 esp 를 8 만큼 증가시킨다. 이 코드에 의해 함수는 실행을 마치고 복귀함과 동시에 인수 영역을 해제한다. Add 함수 자신이 복귀하면서 스택을 정리하므로 호출원에서는 스택을 정리할 필요가 없다. 호출원은 인수를 순서대로 스택에 푸시한 후 함수만 호출하면 된다.

__cdecl 과의 차이점

__cdecl 과 __stdcall 의 가장 큰 차이점은 스택 정리 주체가 누구인가하는 점인데 사실 이 차이점이 컴파일된 결과 코드에 미치는 영향은 별로 없다. 스택 정리 주체와는 상관없이 스택은 항상 호출 전의 상태로 복구되며 프로그램의 동작도 완전히 동일하다. 실행 속도는 거의 차이가 없으며 프로그램의 크기는 비록 무시할만한 수준이기는 하지만 __stdcall 이 조금 더 작다. 왜냐하면 함수를 여러 번 호출하더라도 스택을 정리하는 코드는 함수 끝의 접미에 딱 한 번만 작성되기 때문이다. 반면 __cdecl 은 호출원이 스택을 정리하므로 호출할 때마다 정리 코드가 반복되어 프로그램 크기가 조금 더 커진다.



또 다른 중요한 차이점은 가변 인수 함수를 만들 수 있는가 아닌가 하는 점이다. __stdcall 은 함수가 직접 스택을 정리하기 때문에 가변 인수 함수를 지원하지 않는다. 함수 접미에 스택 정리 코드를 작성하려면 인수의 총 크기를 미리 알아야 하는데 가변 인수 함수는 전달되는 인수 개수가 가변이므로 이 크기가 고정적이지 않아 접미에서 스택을 직접 정리할 수 없다. 컴파일러가 접미의 ret n 명령에 대해 n 을 결정할 수 없는 것이다.

이에 비해 __cdecl 은 함수가 스택을 정리할 책임이 없으며 호출원이 함수를 부를 때마다 스택을 정리한다. 함수를 호출하는 쪽에서는 인수를 몇개나 전달했는지 알 수 있으므로 실제 전달한 인수 크기만큼 스택을 정리할 수 있다. 그래서 printf 나 scanf 같은 가변 인수를 지원하는 함수는 모두 __cdecl 호출 규약을 사용한다. 또한 윈도우즈 API 함수의 기본 호출 규약은 __stdcall 이지만 wsprintf 는 예외적으로 __cdecl 로 작성되어 있다.

호출 규약 중 호출원이 스택을 정리하는 것은 __cdecl 밖에 없으며 그래서 가변 인수를 지원할 수 있는 호출 규약도 __cdecl 이 유일하다. 가변 인수 함수를 만들려면 반드시 __cdecl 호출 규약을 사용해야 한다. 만약 가변 인수 함수를 __stdcall 로 작성하면 컴파일러는 이를 무시하고 __cdecl 로 강제로 바꾸어 버린다.

__fastcall

다음은 __fastcall 호출 규약을 테스트해 보자. 함수 정의부를 int __fastcall Add(int a, int b)로 수정하기만 하면 된다. 호출부의 코드는 다음과 같다.

```
mov edx,2
mov ecx,1
call Add
result=eax
```

__fastcall 은 인수 전달을 위해 edx, ecx 레지스터를 사용하는데 두 개의 인수를 차례대로 edx, ecx 에 대입했다. 만약 인수가 둘 이상이면 세 번째 이후의 인수는 __cdecl 과 마찬가지로 스택에 밀어 넣을 것이다. 인수 전달을 위해 스택을 쓰지 않고 레지스터를 우선적으로 사용하므로 인수 전달 속도가 빠르다는 이점이 있다. 함수의 코드는 다음처럼 작성된다.

```
push ebp
mov ebp,esp
sub esp,14h
mov [ebp-8],edx          // 첫 번째 인수를 지역변수로
mov [ebp-4],ecx          // 두 번째 인수를 지역변수로
mov eax,[ebp-4]
add eax,[ebp-8]
mov [ebp-0ch],eax            // c 는 세 번째 지역변수가 된다.
mov eax,[ebp-0ch]
mov esp,ebp
pop ebp
ret
```

edx, ecx 레지스터를 통해 전달받은 인수 둘을 순서대로 지역변수 영역에 복사한 후 사용하는데 어차피 인수도 지역변수의 일종이므로 이렇게 해도 별 상관이 없다. VC 는 fastcall 호출시 ecx, edx 로 인수를 넘기기는 하지만 이를 다시 스택의 지역변수로 만드는데 이렇게 되면 fastcall 을 하는 의미가 없다. 비주얼 C++은 fastcall 을 형식적으로만 지원할 뿐 fastcall 의 장점을 취하지는 않는데 이는 컴파일러 구현상 ecx, edx 레지스터가 꼭 필요하기 때문이다.

스택 정리는 함수가 하는데 Add 함수의 경우 인수가 두 개 뿐이므로 인수 전달을 위해 스택을 사용하지 않았으며 그래서 정리할 내용이 없다. 만약 인수가 세 개라면 제일 끝의 ret 는 ret 4 가 될 것이다. 레지스터는 스택보다 훨씬 더 빠르게 동작하기 때문에 __fastcall 은 이름대로 호출 속도가 빠르다. 대신 이식성에 불리하다는 단점이 있다. 이 호출 규약은 ecx, edx 레지스터를 사용하도록 되어 있는데 이 두 레지스터가 모든 CPU 에 공통적으로 존재하는 것이 아니기 때문이다. 그래서 윈도우즈 API 는 이 호출 규약을 지원하기는 하지만 사용하지는 않는다. 볼랜드의 델파이가 __fastcall 을 사용한다.

thiscall

thiscall 은 클래스의 멤버 함수에 대해서만 적용되는데 ecx 로 객체의 포인터(this)가 전달된다는 것이 특징이며 나머지 규칙은 __stdcall 과 동일하다. 예외적으로 가변 인수를 사용하는 멤버 함수는 __cdecl 로 작성되며 이때 this 는 스택의 제일 마지막에(그러므로 첫 번째 인수로) 전달된다.

이 호출 규약은 컴파일러가 멤버 함수에 대해서만 특별히 적용하는 것이므로 일반 함수에는 이 호출 규약을 적용할 수 없다. thiscall 은 이 호출 규약의 이름일 뿐 키워드가 아니기 때문에 함수 원형에 thiscall 이라고 쓸 수도 없다. 멤버 함수이기만 하면 컴파일러가 알아서 thiscall 호출 규약을 적용한다. 객체니 멤버 함수니 this 니 하는 것들은 C++편에서 배우게 될 것이다.

__naked

__naked 호출 규약은 컴파일러가 접두, 접미를 작성하지 않는 호출 규약이다. 스택 프레임의 상태 보존을 위해 컴파일러가 어떤 코드도 작성하지 않으므로 접두, 접미는 사용자가 직접 작성해야 한다. 스택은 어셈블리 수준에서만 다룰 수 있으므로 인라인 어셈블리를 사용해야 하며 제약점도 많기 때문에 일반적인 목적으로는 사용되지 않는다.

이 호출 규약이 반드시 필요한 경우는 C/C++이 아닌 언어에서 호출하는 함수를 작성할 때이다. 예를 들어 어셈블리에서는 인수 전달에 스택을 쓰지 않고 범용 레지스터만으로도 인수를 전달할 수 있다. 이런 경우는 C 컴파일러가 만들어주는 접두, 접미가 불필요하다. 또한 속도가 지극히 중요한 디바이스 드라이버를 작성할 때도 이 호출 규약을 사용한다. __naked 호출 규약을 사용하려면 함수의 정의부에 __declspec(naked)를 적어주면 된다.

여기서 알아본 호출 규약 외에도 __pascal, __fortran, __syscall 이라는 호출 규약이 있었으나 지금은 지원되지 않는다. 비주얼 C++은 과거와의 호환성을 위해 이 단어들을 키워드로 인정하기는 하지만 실제로 사용할 경우 에러로 처리한다. 이상으로 다섯 가지의 호출 규약에 대해 정리했는데 실제로 사용되고 사용자가 지정할 수 있는 호출 규약은 현실적으로 __cdecl, __stdcall 두 가지밖에 없는 셈이다.